KIPS Trans. Softw. and Data Eng.
Vol.12, No.1 pp.1~18
ISSN: 2287-5905 (Print), ISSN: 2734-0503 (Online)

https://doi.org/10.3745/KTSDE.2023.12.1.1

A Code Clustering Technique for Unifying Method Full Path of Reusable Cloned Code Sets of a Product Family

Kim Taeyoung† · Lee Jihyun†† · Kim Eunmi†††

ABSTRACT

Similar software is often developed with the Clone-And-Own (CAO) approach that copies and modifies existing artifacts. The CAO approach is considered as a bad practice because it makes maintenance difficult as the number of cloned products increases. Software product line engineering is a methodology that can solve the issue of the CAO approach by developing a product family through systematic reuse. Migrating product families that have been developed with the CAO approach to the product line engineering begins with finding, integrating, and building them as reusable assets. However, cloning occurs at various levels from directories to code lines, and their structures can be changed. This makes it difficult to build product line code base simply by finding clones. Successful migration thus requires unifying the source code's file path, class name, and method signature. This paper proposes a clustering method that identifies a set of similar codes scattered across product variants and some of their method full paths are different, so path unification is necessary. In order to show the effectiveness of the proposed method, we conducted an experiment using the Apo Games product line, which has evolved with the CAO approach. As a result, the average precision of clustering performed without preprocessing was 0.91 and the number of identified common clusters was 0, whereas our method showed 0.98 and 15 respectively.

Keywords : Clone-and-own Approach, Software Product Line Migration, Product Line Code Base, Code Clustering

제품군의 재사용 가능한 클론 코드의 메소드 경로 통일을 위한 코드 클러스터링 방법

김 태 영† · 이 지 현†† · 김 은 미†††

요 약

유사한 소프트웨어는 기존 산출물을 복제하고 수정하는 클론-앤-오운(clone-and-own, CAO) 방법으로 개발되곤 한다. 그러나 클론-앤-오운 방법은 복제된 제품의 수가 늘면서 유지보수를 어렵게 만들기 때문에 나쁜 프랙티스로 간주된다. 소프트웨어 제품라인 공학은 체계적인 재사용을 통해 소프트웨어 제품군을 개발하는 방법으로 클론-앤-오운 방법의 문제를 해결할 수 있다. CAO 방식으로 개발되어 온 제품패밀리를 제품라인 공학으로 마이그레이션하는 작업은 여러 소프트웨어 제품에서 클로닝된 부분들을 찾아 통합하고 재사용 가능한 자산으로 구축하는 것으로부터 시작된다. 그러나 클로닝이 디렉토리부터 코드 라인까지 다양한 수준에서 발생하고 그 과정에서 이들의 구조에 변경이 일어날 수 있어 단순하게 클로닝을 찾아내는 것만으로는 고품질의 제품라인 코드베이스를 구축하기 어렵다. 성공적인 마이그레이션을 위해서는 소스 코드들 사이의 클로닝 관계를 찾는 것 이외에도 소스 코드들의 파일 경로와 클래스 이름, 메소드 시그니처 등의 동일성을 확보는 작업이 선행되어야 한다. 이에 본 연구는 CAO 기반으로 개발된 제품들로부터 마이그레이션 대상 제품들을 선정한 후 제품들에 흩어져 있는 유사 코드 집합을 검출하여 메소드 경로의 통일이 필요한 대상을 식별하는 클러스터링 방법을 제안한다. 제안 방법의 효과를 보이기 위해 CAO 방식으로 진화해온 ApoGames 제품군에 제안 방법을 적용하여 실험을 진행하였다. 그 결과, 전처리 없이 수행된 파일의 상대 경로 기반 클러스터링 방법의 평균 정밀도는 0.91이며 식별된 공통 클러스터의 개수는 0개인 반면에 이 논문에서 제안하는 전처리와 함께 수행된 메소드 시그니처 기반 클러스터링 방법의 평균 정밀도는 0.98로 개선되었으며 식별된 공통 클러스터는 최대 15개까지 증가하였다.

키워드 : 클론앤오운 개발 방법, 소프트웨어 제품라인 마이그레이션, 제품라인 코드베이스, 코드 클러스터링

1. 서 론

기업들은 단일 소프트웨어 제품(이하 제품)보다는 다양한 고객들의 니즈를 충족시키고 시장 점유율을 확대하기 위한 방안으로 유사하지만 서로 다른 피처(feature) 집합을 갖는 여러 가지 제품들을 패밀리로 개발하고 관리한다. 클론-앤-오운(Clone-And-Own, CAO) 방식은 제품군을 쉽게 개발

※ 이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2020R1F1A1071650).
† 비 회 원 : 전북대학교 소프트웨어공학과 석사과정
†† 정 회 원 : 전북대학교 소프트웨어공학과 교수
††† 종신회원 : 호원대학교 컴퓨터·게임학과 교수
Manuscript Received : September 13, 2022
First Revision : October 25, 2022
Accepted : November 18, 2022
* Corresponding Author : Lee Jihyun(jihyun30@jbnu.ac.kr)

하는 방법으로 기존 소프트웨어 시스템과 유사한 새로운 제품 개발 시, 리거시 코드를 클로닝(cloning)하고 적절히 수정(modify)함으로써 새로운 제품을 개발하는 방법이다[1, 2]. 여기서 클로닝은 기존 제품의 구성 요소를 복제(copy)하여 새로운 제품의 구성 요소로서 붙여넣기(paste)하는 재사용 행위를 말한다. 비록 제품군 개발 초기에는 CAO 방식의 제품 개발이 효과적으로 쓰일 수 있지만, CAO 기반 제품 개발의 가장 큰 문제는 각각의 제품이 독립적으로 관리되기 때문에 제품의 수가 증가할수록 유지보수가 어렵다는 점이다.

소프트웨어 제품라인 공학(Software Product Line Engineering, SPLE)은 플랫폼 기반의 체계적인 재사용을 통해 관련 제품군의 신속한 개발 및 효율적인 유지보수를 지원하는 방법이다. CAO 방식으로 개발되어 온 제품군을 제품라인으로 마이그레이션하는 목적은 여러 제품에 클로닝되어 공통적으로 나타나는 부분들을 공유 가능한 자산으로 통합하여 플랫폼을 구축함으로써 높은 재사용성을 보장하는 동시에 CAO 방식의 개발로 인한 유지보수의 문제를 개선하는 것이다. 제품군 개발 과정에서 애드혹(ad-hoc)하게 적용되는 CAO 방식의 특성상 CAO 방식으로 개발된 제품들의 경우에는 요구사항 명세, 아키텍처 설계 명세 등 문서화가 충분하지 않은 경우가 많다. 이 경우 소스 코드로부터 마이그레이션을 진행하는 추출식 SPLE 방법을 적용한다[3].

CAO 방식으로부터 SPLE로 마이그레이션 할 때 주요 도전 과제는 공통성과 가변성을 추출하는 것이다[4-6]. 그래서 많은 기존 연구들이 CAO로 만들어진 제품군의 소스 코드가 주어지면 서로 다른 제품들 간의 코드 비교를 통해서 공통 부분과 가변 부분을 식별하는 작업을 수행한다. 그리고 제품으로부터 사용자 관점에서의 기능 단위인 피처 집합(feature set)을 식별하고, 각 피처를 관련된 공통/가변 부분의 코드와 연결 및 통합하여 재사용 가능한 플랫폼을 구축하게 된다. 결과적으로 이러한 작업들은 독립적으로 관리되던 제품군의 소스 파일들을 오버랩(overlap)함으로써 제품군의 제품들이 하나의 코드베이스로 통합 관리될 수 있도록 만든다.

기존 마이그레이션 연구에서 고려된 제품들은 주로 기존 제품 전체를 클로닝하여 가급적 기존 기능을 온전히 사용하면서 추가로 필요한 일부 기능을 확장하거나 기존 구현 부분의 일부를 수정하여 만들어진 제품들이다. 이와 같은 제품 수준에서의 클로닝은 재사용되는 특정 피처의 구현 코드가 동일 디렉토리 구조 및 파일로 정의되며 이들은 모든 제품에서 동일한 경로와 시그니처를 가지게 된다. 그래서 기존 마이그레이션 연구들은 제품들의 소스 코드를 오버랩하는 과정에서 나타나는 구현 부분에 대해서만 가변성을 식별하고 이를 효과적으로 처리하는 전략을 수립하여 적용하는 데에 집중하였다.

그러나 CAO 방식의 제품 개발 과정에서는 국소적으로 이루어지는 클로닝 또는 제품별 독립적으로 수행되는 유지보수 활동에 의해 이러한 경로 및 구현 부분이 복합적으로 변화하는 상황이 발생한다[6, 7]. 실제로 CAO 방식에서 클로닝을 통한 재사용은 기존 제품 전체 또는 일부 디렉토리, 파일, 클래스, 메소드, 라인 등 다양한 규모로 수행된다. 다시 말해, 동일한 라인이 서로 다른 메소드에서 클로닝되어 사용될 수 있고, 동일한 메소드가 서로 다른 클래스나 파일에서 클로닝되어 사용될 수 있다. 더 나아가 동일한 소스 파일이 두 제품에서 서로 다른 디렉토리 구조로 클로닝될 수도 있다. 또한, 클로닝을 통한 재사용은 제품 개발 초기뿐만 아니라 제품이 생성된 이후에 독립적으로 유지보수 되는 동안에도 발생한다. 이러한 상황들로 인해 제품 간 메소드 경로 부분에서 가변성이 발생하게 된다. 메소드 경로는 객체 생성 및 메소드 호출에 필요한 클래스의 이름 및 메소드의 시그니처뿐만 아니라 임포트(import)에 필요한 디렉토리 경로와 파일의 이름을 포함한다. 결국 메소드 경로의 가변성은 공통 자산으로 관리될 수 있는 소스 파일들을 가변 자산으로 관리되도록 만들어 관리 대상 자산의 개수 증가뿐만 아니라 향후 구축될 플랫폼의 재사용을 복잡하게 만드는 요인이 된다[8].

이 논문은 CAO 기반으로 개발된 서로 독립된 제품들로부터 기능적으로 유사한 제품들을 선별하여 마이그레이션 대상 제품들을 결정하고, 제품 간 서로 유사한 코드들을 식별하여 동일한 메소드 경로를 갖도록 수정함으로써 제품들의 구조적 동일성을 확보하는 방법을 제안한다. 메소드의 시그니처는 그 자체로 시스템이 제공하는 기능을 함축하는 레이블(label)인 동시에 메소드 경로를 구성하는 말단 요소로서 클로닝에 의한 메소드 경로의 가변성을 효과적으로 식별할 수 있는 수단이기 때문에 제안 방법에서 핵심적인 역할을 한다.

제안한 방법을 검증하기 위해서 자바 언어로 개발된 ApoGames의 20개의 제품에 제안 방법을 적용하였다[9]. 그 결과, 전처리 없이 수행된 파일의 상대 경로 기반 클러스터링 방법의 평균 정밀도는 0.91이며 식별된 공통 클러스터의 개수는 0개인 반면에 이 논문에서 제안하는 전처리와 함께 수행된 메소드 시그니처 기반 클러스터링 방법의 평균 정밀도는 0.98으로 개선되었으며 식별된 공통 클러스터는 15개로 증가하였다. 이로써 각 클러스터에 포함된 소스 코드들이 구조적으로 통일되었을 때 공통 코드 식별의 정확도가 향상됨을 확인하였다.

이 논문의 구성은 다음과 같다. 우선, 2장에서는 연구 배경으로 사례를 통해 코드 클러스터링의 목적과 필요성을 설명한다. 3장에서는 CAO 기반 제품 개발을 SPLE 기반 제품 개발로 마이그레이션 하는 과정에서 메소드 경로의 통일을 통한 제품들 간의 구조적 동일성 확보를 목표로 하는 클러스터링 방법을 제안한다. 4장에서는 간단한 예제로 제안 방법이 적용되는 과정을 설명한다. 5장에서는 제안한 방법을 평가하기 위한 실험 설계와 결과를 설명한다. 6장은 관련 연구를 소개하고, 마지막 7장에서는 이 논문의 결론과 향후 연구에 대하여 논의한다.

2. 연구 배경

마이그레이션을 통해 제품군의 소스 파일들을 하나의 코드베이스로 통합 관리하고자 할 때, 동일한 메소드 경로의 소스 코드에서 발생하는 가변성은 조건부 컴파일과 같은 어노테이션(annotation) 기법 등을 활용하여 통제될 수 있다. 그러나 어떠한 소스 코드가 임포트하는 다른 소스 파일이 제품마다 각기 다른 디렉토리, 파일 이름, 클래스 이름 등 서로 다른 메소드 경로를 갖는 경우에는 마이그레이션의 결과로 만들어진 코드베이스가 각각의 제품들의 기능을 온전히 지원하기 위해서는 이들의 코드가 서로 유사하더라도 기존 경로가 보존된 채로 코드베이스에 통합되고 여전히 독립적인 소스 파일로서 관리되어야 한다. 이러한 사실은 제품들 간에 메소드 경로의 공통성을 충분히 확보하는 것이 마이그레이션에 있어서 상당히 중요함을 의미한다.

Fig. 1은 CAO 기반으로 만들어진 제품들에 대하여 메소드 경로에 대한 가변성을 사전에 식별하여 정제한 경우와 그렇지 않은 경우의 마이그레이션 결과를 보여주는 예제이다. 그림 우측에 보이는 마이그레이션의 결과는 제품들의 소스 코드를 오버랩하여 생성된 클래스 다이어그램을 보여준다. Fig. 1(a)는 기존 제품들의 메소드 경로의 차이에 의해 나타나는 구조적 가변성이 필수적인 것으로 보고 이들을 온전히 유지한 채로 소스 코드를 오버랩한 결과이다. 반면 Fig. 1(b)는 메소드 시그니처를 고려하여 서로 유사성을 띤 제품과 그 클래스를 묶어 관련 소스 파일들이 동일한 메소드 경로를 갖도록 수정된 이후에 제품들의 소스 코드를 오버랩한 결과이

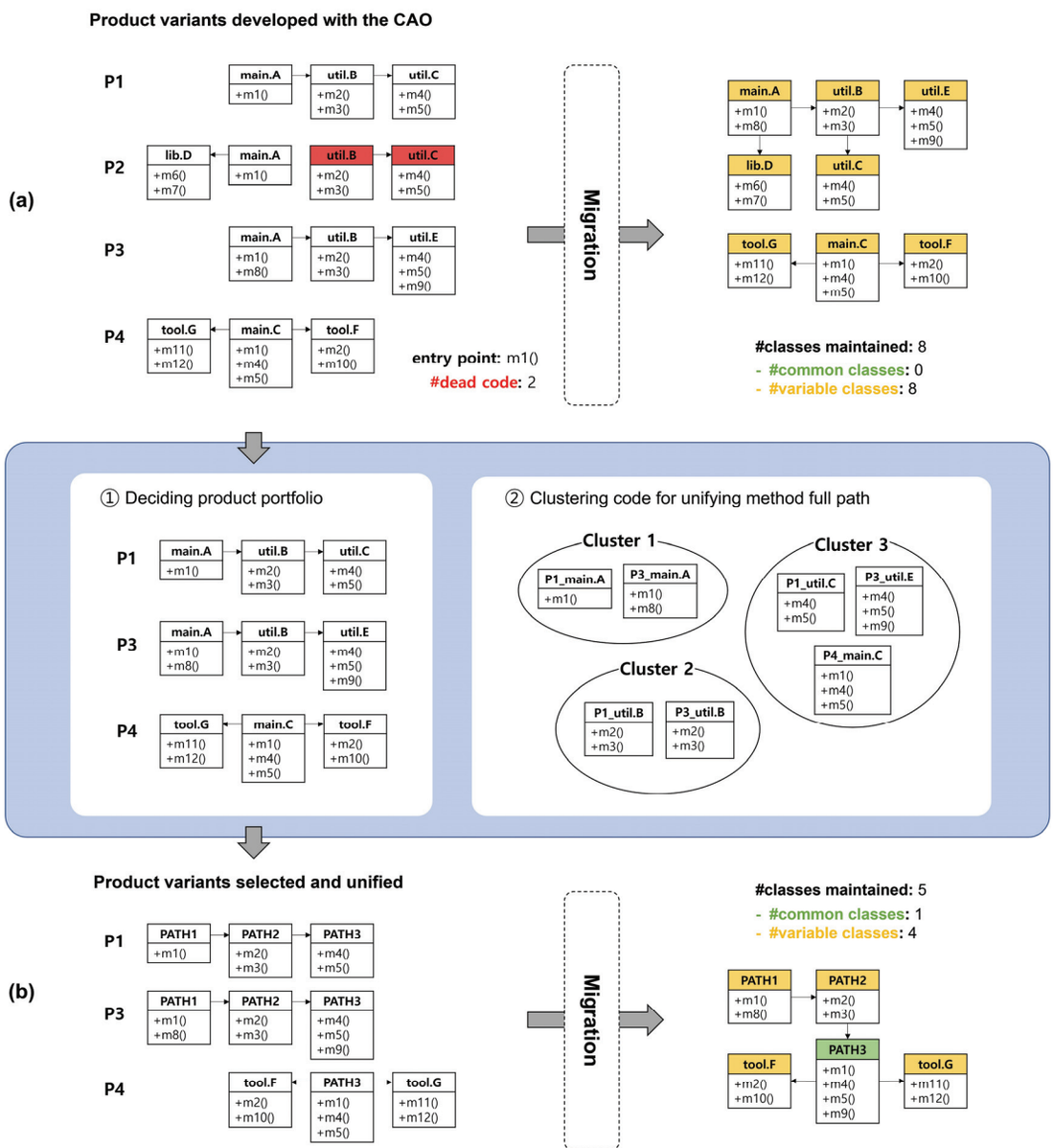

Fig. 1. Motivating Example

다. 결과를 통해 알 수 있듯이 기능적으로 유사한 정도가 적은 제품 P2를 마이그레이션 대상에서 제외하고, 클러스터로 식별된 소스 파일들의 메소드 경로를 통일한 이후에는 유지보수 및 관리 대상 클래스의 수가 8개에서 5개로 줄어들 뿐만 아니라 모든 제품에서 공통적으로 사용되는 클래스의 수가 0개에서 1개로 증가한 것을 확인할 수 있다. 여기서 모든 제품에서 재사용 가능한 공통 클래스의 존재는 플랫폼 구축을 위한 필수 조건이 된다. 그러므로 메소드 경로의 통일 대상을 식별하기 위해 수행되는 코드 클러스터링은 메소드 경로 차이로 인해 발생하는 불필요한 구조적 가변성을 식별하고 제거할 수 있도록 도움으로써 SPL 코드베이스와 플랫폼을 최적화하는 데에 기여한다.

3. 메소드 경로 통일을 위한 코드 클러스터링 방법

이 장에서는 CAO 기반으로 개발된 제품들로부터 기능적으로 유사한 제품들을 선별하여 마이그레이션 대상 제품들을 결정하고, 서로 다른 제품들의 소스 파일로부터 하위 경로가 동일하면서 서로의 소스 코드가 유사한 파일들을 묶어 각각의 클러스터로 분류하는 방법을 제안한다. 이렇게 식별된 클러스터의 내부 파일들이 동일한 메소드 경로를 갖도록 조정되었을 때, SPLE로의 마이그레이션이 효과적으로 수행될 토대가 만들어진다. Fig. 2는 이 논문이 제안하는 마이그레이션 제품 선정 및 제품 간 메소드 경로의 통일 대상을 결정하는 클러스터링 방법을 소개한다. 각 단계는 CAO 개발 방식의 특수성과 클러스터링 결과가 활용될 SPLE의 환경을 고려하여 구성되었다.

3.1 제품 최적화

제품 최적화 단계는 각 제품을 구성하는 소스 코드 중에서 제품의 실행과는 무관한 소스 코드를 찾아 제거한다. CAO 방식의 제품군 개발은 기존 제품으로부터 기능적으로 유사하여 재사용 가능하다고 판단되는 소스 코드를 복제한 뒤 필요에 따라 관련 코드를 추가, 수정, 삭제함으로써 새로운 제품을 개발한다. 이 과정에서 기존 제품으로부터 복제되었지만 새로 개발되는 제품에서는 사용되지 않는 소스 코드인 데드코드가 발생한다.

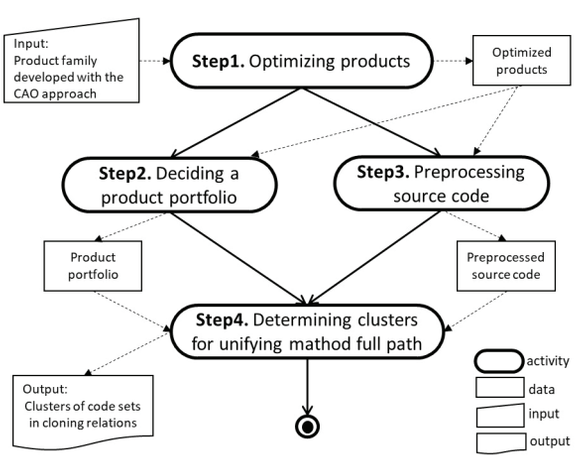

Fig. 2. Approach Overview

데드코드는 리팩토링의 대상임이 분명하지만, 그 유형에 따라 단일 제품 측면에서는 크게 문제가 되지 않을 수 있다. 반면 SPLE로 마이그레이션 하는 과정에서는 데드코드가 제품들 간의 공통성 및 가변성을 파악하는 데에 부정적인 영향을 미치게 된다. 공통 부분은 마이그레이션 대상 제품군에 속하는 모든 제품에서 항상 사용되는 부분으로 특정 도메인의 제품군이 기본적으로 제공해야 하는 기능과 관련된 부분임을 암시하며, 향후 플랫폼을 구성하고 플랫폼의 재사용 효과를 결정짓는 핵심 요소이다. 그러나 제거되지 않은 데드코드는 사실상 제품군의 일부 제품에서만 사용되는 가변 부분을 공통 부분으로 오인하도록 만드는 요인이 되며, 클로닝되었지만 더 이상 유지보수 되지 않는 데드코드와 지속적으로 유지보수 되는 소스 코드 사이에서 불필요한 가변 코드들이 추출되도록 만든다. 이러한 이유로 각 제품을 구성하는 소스 코드를 대상으로 제품의 실행과는 무관한 데드코드를 제거하는 작업이 수행되어야 한다.

제품과 무관한 코드를 식별하고 관련된 데드코드를 제거하는 작업은 다음 절차에 의해 수행된다:

a) 제품을 구성하는 모든 소스 파일을 파싱하여 소스 파일들 간의 참조 관계를 파악하고, 각 소스 파일을 정점으로 하며 참조 관계를 간선으로 하는 방향 그래프를 생성한다.
b) 프로그램 시작점에 해당하는 소스 파일을 시작 정점으로 하여 그래프 탐색을 수행하고, 시작 정점으로부터 도달 가능한 소스 파일과 도달 불가능한 소스 파일의 집합을 파악한다.
c) 그래프 탐색을 통해 도달 불가능한 소스 파일은 해당 제품에서 참조되지 않는 소스 파일이므로 그와 관련된 소스 코드를 데드코드로 간주하고 제거한다.

3.2 제품 포트폴리오 결정

이 단계는 CAO 방식으로 개발된 제품군 내에서 다른 제품들과 비교하여 기능적 유사도가 낮은 제품을 찾아 제거한다. 일반적으로 CAO 방식의 제품 개발은 제품군들 간의 재사용률 혹은 유사도를 일정 수준 이상으로 강제하거나 보장하지 않는다[9, 10]. 그러므로 새로 개발되는 제품에서 CAO 방식에 의해 기존 제품으로부터 재사용되는 소스 코드의 규모와 정도는 비결정적이다. 또한 클로닝된 소스 코드들은 서로 동기화되어 있지 않으며 이미 한 번 클로닝된 소스 파일이 수정 및 변경되어 다른 제품의 생성 과정에서 다시 클로닝될 수 있다[11]. 그 결과 CAO 방식으로 개발된 제품군이라 할지라도 기존 제품으로부터 클로닝된 소스 코드의 규모, 이미 클로닝된 소스 코드가 수정 및 변경되어 다시 클로닝된 횟수, 각 제품이 서로 독립적인 제품으로 유지보수된 기간에 따라 제품군들 간의 유사도는 점차 감소하게 된다.

SPLE는 하나의 제품군에 속하는 제품들이 서로 상당한 기능적 유사성을 보일 때, 공통적으로 사용되는 기능들을 토대로 공유 가능한 플랫폼을 구축하고 이를 유기적으로 관리 및 재사용함으로써 새로운 제품의 신속한 개발을 지원하는 개발 방법이다. 만일 다른 제품들과 비교하여 기능적인 유사도가 낮은 제품이 마이그레이션 대상 제품군에 포함된 경우, 모든 제품에서 재사용 가능한 공통 부분이 감소하고 가변성이 증가하여 플랫폼의 유지보수 및 재사용의 이점이 줄어들게 된다. 따라서 효율적인 제품라인을 구축하기 위해서는 제품군들 간의 기능적 유사도를 고려하여 적절한 마이그레이션 대상 제품들을 선별해야 할 필요가 있다.

우선, 마이그레이션의 대상 제품을 선정하기 위해서는 제품들 간의 유사도를 측정하기 위한 기준이 필요하다. 메소드는 프로그램 내에서 반복적으로 수행되는 일련의 작업에 대하여 독립성과 기능성을 부여하는 방법으로 프로그래밍 언어가 보편적으로 지원하는 재사용 단위이며, 메소드의 시그니처는 메소드의 이름과 파라미터의 타입 및 순서를 추출한 정보로 메소드 호출 시 대응 메소드의 고유성을 결정하는 요소인 동시에 제공하는 기능에 대한 함축적 표현이다. 또한 클로닝 이후에 메소드 시그니처의 변경은 그 메소드를 호출하여 사용하는 모든 관련 소스 파일들의 변경을 수반하게 된다. 그러므로 기존 제품의 구현을 재사용하여 비교적 간단하고 신속하게 새로운 제품을 개발하고자 하는 CAO 기반 제품 개발의 목적상 기존 제품과의 뚜렷한 기능적 차이에 의한 불가피한 변경이 아니라면 상당한 수정 노력을 요구하는 메소드 시그니처의 변경은 최소화된다. 이러한 이유로 이 논문에서는 제품들 간에 기능적으로 유사한 정도를 판단하는 수단으로 제품들이 포함하는 메소드와 시그니처 정보를 활용한다.

메소드 시그니처를 기반으로 제품군을 구성하는 특정 제품의 유사도를 정량화하는 목적은 공유 메소드로 구성된 제품이 특수 메소드로 구성된 제품보다 제품군과의 유사도가 더 높게 산정되도록 만들어 유사도가 낮은 제품을 마이그레이션 대상에서 제외함으로써 제품 포트폴리오를 구성하는 것이다. 여기서 공유 메소드란 어떠한 메소드 시그니처가 제품군을 구성하는 제품들 중 2개 이상의 제품에서 정의되어 사용된 메소드를 말하며, 특수 메소드란 어떠한 메소드 시그니처가 제품군을 구성하는 제품들 중 단 하나의 제품에서만 정의되어 사용된 메소드를 가리킨다. 메소드 시그니처를 기반으로 각 제품의 기능적 유사도를 측정하기 위하여 Equation (1)과 Equation (2)를 사용한다.

$$Similarity(p) = \sum_{s \in S_p} f(s) \tag{1}$$

$$f(s) = \begin{cases} -1, & |P_s| = 1 \\ (|P_s| - 1)^2, & |P_s| > 1 \end{cases} \tag{2}$$

$S_p = \{s \mid s$는제품p가정의하고있는메소드의시그너처$\}$,

$P_s = \{p \mid p$는메소드의시그너처s를포함하는제품$\}$■

특정 제품의 기능적 유사도는 해당 제품을 구성하는 모든 메소드의 시그니처에 대하여 이들 각각이 제품군에 속한 얼마나 많은 제품들에서 공통적으로 사용되는지에 따라 메소드 수준에서 개별적으로 수치화한 뒤, 이들을 합산하여 제품 수준의 유사도를 산출하게 된다.

우선, 위의 수식에 따르면 특정 메소드 시그니처가 제품군에 속하는 다른 n개의 제품에서 나타날 때, n의 제곱수 만큼 가중하여 합산한다. 이때 n의 상수 배가 아닌 제곱수 만큼 가중하여 정량화하는 이유는 유사도 역전 현상을 최소화하기 위함이다. 유사도 역전 현상이란 소수의 제품에서만 공유되는 메소드를 다량 포함하는 제품이 다수의 제품에서 사용되는 메소드를 포함하지 않더라도 제품군 내에서 가장 높은 유사도 값을 갖게 되는 현상을 말한다. 즉, 유사도를 측정하는 궁극적인 목적은 제품군을 구성하는 제품들이 다른 제품에서 재사용 가능한 요소들로 구성될 뿐만 아니라 가능하면 이들이 대부분의 제품에서 사용되어 플랫폼화 가능한 제품들을 선정하는 것인데, 유사도 역전 현상에 의해 이에 반하는 결과가 나타나게 된다. 이러한 현상을 최소화하고 다수의 제품에서 공유 가능한 메소드를 중심으로 제품의 유사도가 더 높게 측정되도록 공유 메소드에 대해서는 제곱 가중 방법을 적용한다.

반면, 특정 제품이 고유의 메소드를 많이 포함할수록 제품군에 속한 다른 제품들과의 유사도는 상대적으로 낮게 측정되는 것이 합리적이다. 따라서 단 하나의 특정 제품에서만 사용되는 특수 메소드에 대해서는 그 제품에 포함된 특수 메소드의 개수만큼 차감되어 제품의 유사도가 계산된다.

제품군을 구성하는 각 제품의 유사도를 측정하고 제품 포트폴리오를 결정하는 과정은 다음 절차에 의해 수행된다:

a) 제품군을 구성하는 모든 제품의 소스 코드를 파싱하여 제품별 메소드 시그니처를 추출한다.
b) 각 메소드 시그니처에 대하여 해당 메소드 시그니처가 정의된 제품의 개수를 파악한다.
c) 개별 제품에 대하여 제품을 구성하는 메소드 시그니처 집합에 유사도 정량화 수식을 적용하여 각 제품의 유사도를 측정한다.
d) 가장 낮은 유사도를 갖는 제품을 제품군에서 제외한다.
e) 과정 b-d를 반복적으로 수행하여 제품군으로부터 유사도가 가장 낮은 제품을 하나씩 단계적으로 제거한다.
f) 제품군의 구성에 따라 공통 메소드 시그니처의 개수와 가변 메소드 시그니처의 개수를 분석하여 마이그레이션할 제품군을 결정한다.

3.3 소스 코드 전처리

소스 코드 전처리 단계는 제품군을 구성하는 모든 소스 코드가 동일한 코딩 스타일로 표현되도록 만든다. 소스 코드는 추출식 마이그레이션에서 공통성과 가변성을 파악하기 위한 자료로 사용되며, 마이그레이션 결과로 생성되는 플랫폼에서 가변값과 그 가변값이 바인딩되는 가변점을 결정하는 데에 있어서 핵심 근거가 된다. 그러므로 메소드 경로를 통일할 대

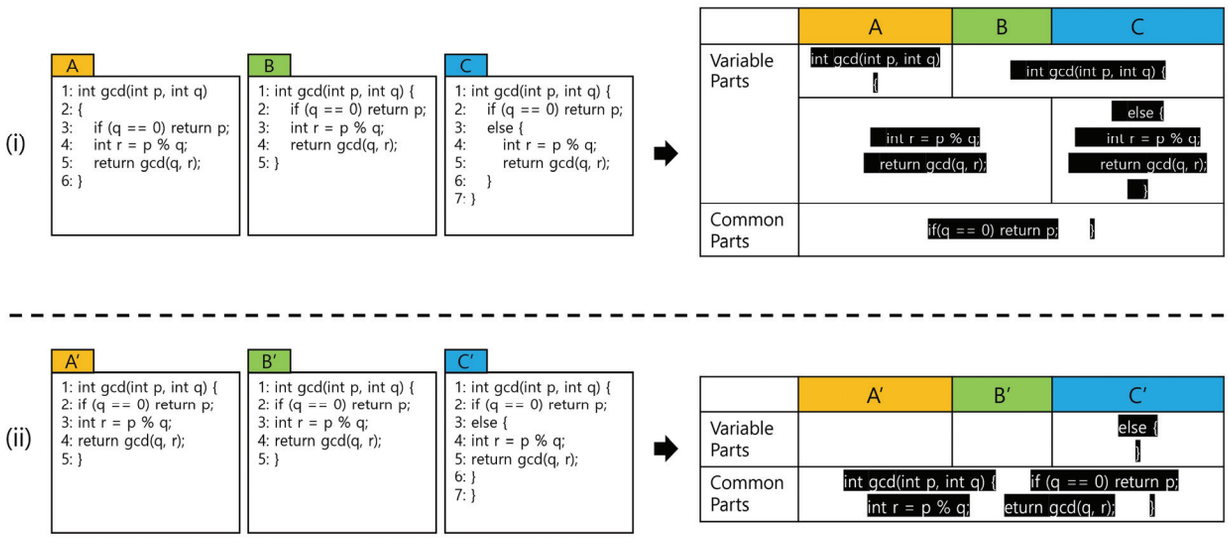

Fig. 3. Example Showing the Impacts of Coding Style Difference

상을 효과적으로 식별하기 위해서는 메소드 경로의 유사성뿐만 아니라 소스 코드의 공통성 또한 고려되어야 한다. 코딩 스타일을 통일시키는 작업은 소스 코드의 공통성 정도를 측정하는 작업이 보다 정확하게 수행될 수 있도록 만든다.

소스 코드의 공통성을 판단하기 위해 사용되는 일반적인 방법으로는 토큰 단위의 비교 방법과 라인 단위의 비교 방법이 있다. 토큰 단위의 비교 방법은 코딩 스타일에 그다지 영향을 받지 않는다는 큰 장점을 갖는다. 하지만 토큰 수준의 비교는 코드 그 자체에 대한 유사성을 보여줄 순 있어도 코드가 수행하는 작업에 대한 유사성을 보여주는 데에는 한계가 존재한다. 프로그래밍 언어가 기본적으로 제공하는 키워드(keyword)나 연산자(operator), 원시 타입(primitive type) 등은 많은 코드에서 반복적으로 사용되는 토큰들이며, 토큰 단위의 비교에서는 이러한 토큰에 의해 서로 다르게 동작하는 코드가 유사한 코드로 오인될 가능성이 비교적 높다. 이 논문에서는 구현 부분에 대한 라인 단위의 코드 비교를 통해 소스 코드의 공통성을 판단한다.

라인 단위의 공통성 및 가변성 식별 방법은 각 라인을 구성하는 토큰의 조합과 그 순서를 비교함으로써 소스 코드의 라인이 동일한지 판단하고, 얼마나 많은 라인들이 서로 다른 코드에서 공통적으로 나타나는지를 파악하여 공통성 정도를 산출하게 된다. 그러나 많은 프로그래밍 언어들이 코드를 작성함에 있어서 줄바꿈 문자의 위치나 공백 문자의 개수 등을 엄격하게 제한하지 않는다. 이는 특정 기능과 관련된 하나의 토큰 배열이 소스 코드로 작성될 때 수많은 형태로 표현될 수 있음을 의미한다. 그리고 이러한 특성은 사실상 동일한 코드가 라인 단위의 코드 비교 과정에서 서로 다른 코드로 식별되는 문제를 야기한다. 따라서 소스 코드의 공통성을 보다 정확하게 식별하기 위해서는 일련의 토큰들이 항상 동일한 형태의 소스 코드로 표현되도록 코딩 스타일을 일치시키는 작업이 수행될 필요가 있다.

Fig. 3은 서로 다른 코딩 스타일이 라인 단위의 공통성 식별에 미치는 영향을 보여주는 사례이다. 그림에서 보이는 6개의 코드는 최대 공약수를 구하는 유클리드 호제법에 대한 구현으로 모두 동일한 기능을 수행하는 코드이다. 그러나 Fig. 3(i)의 소스 파일 A, B, C는 서로 다른 코딩 스타일로 작성되었으며 그로 인해 라인 단위의 공통성 식별 결과에서 상당히 많은 라인들이 가변 부분으로 식별되는 것을 보여준다. 구체적인 예로, 소스 파일 A에서 1-2번 라인의 코드는 소스 파일 B의 1번 라인의 코드와 사실상 동일한 코드임에도 불구하고 중간에 삽입된 줄바꿈 문자에 의해 서로 다른 라인으로 식별되어 가변 부분(variable part)으로 분류되는 것을 볼 수 있다. 또한, 소스 파일 A의 4-5번 라인은 소스 파일 C의 4-5번 라인과 동일하지만 블록 영역을 명시할 목적으로 각 라인의 앞부분에 추가 삽입된 공백 문자에 의해 서로 동일하지 않은 라인으로 식별되어 가변 부분으로 분류되고 있다. 반면, Fig. 3(ii)의 소스 파일 A', B', C'은 Fig. 3(i)의 소스 파일 A, B, C를 토대로 코딩 스타일을 일치시키는 작업을 거쳐 토큰들이 재배치된 코드이며, 이 코드들을 가지고 라인 단위의 비교를 수행하게 되었을 때 비로소 최소한의 라인들이 가변 부분으로 정확하게 식별되는 것을 보여준다.

라인 단위의 비교가 효과적으로 수행될 수 있도록 소스 코드의 코딩 스타일을 일치시키는 작업은 다음 절차에 의해 수행된다:

a) 소스 코드로부터 소스 코드를 구성하는 토큰의 배열을 추출한다.
b) 토큰 사이에 공백 문자와 줄바꿈 문자를 삽입하는 단일 규칙을 세우고 추출된 토큰 배열에 적용한다.

3.4 코드 클러스터링

이 단계는 서로 다른 제품에 존재하지만 일부 동일한 경로를 포함하면서 구현 부분의 소스 코드가 상당히 유사한 소스

파일들을 식별하여 메소드 경로를 통일할 대상을 결정한다.

제품들의 소스 파일들로부터 메소드 경로를 통일할 대상들을 식별하는 작업은 메소드의 시그니처 정보를 활용한다. 또한, 통일 대상 소스 파일들을 선정 과정에서는 관련 소스 파일들의 코드들이 서로 얼마나 유사한지 고려한다. 동일한 경로를 갖더라도 코드의 높은 가변성은 재사용 및 유지보수를 어렵게 만드는 요인이 되므로 소스 코드의 공통성이 충분히 보장되지 않을 때에는 서로 독립인 소스 파일로서 관리되도록 하는 편이 낫기 때문이다. 이를 종합하면 메소드 경로를 통일하는 작업의 목표는 제품군을 구성하는 제품들이 가능한 동일한 경로를 갖도록 하되 메소드 경로가 통일될 소스 파일들의 코드가 일정 기준치 이상의 공통성을 갖도록 관련 소스 파일들을 그룹화하는 것이다.

Fig. 4는 메소드 경로 통일 대상으로 식별된 소스 파일들 간의 소스 코드에 대한 공통성을 계산하는 알고리즘(a)과 메소드 시그니처 정보를 기반으로 메소드 경로 통일 대상인 소스 파일들을 클러스터링하는 알고리즘(b)을 보여준다.

Fig. 4(a)는 특정 클러스터에 포함된 소스 파일들의 소스 코드들이 얼마나 유사한지 그 정도를 0에서 1사이의 수치로 정량화하는 알고리즘이다. 우선, 집합에 포함된 각각의 소스 파일의 코드를 라인 단위로 쪼개고(line 4), 해당 파일의 코드 라인 수를 누적하여 클러스터에 속한 모든 소스 파일들의 전체 코드 라인 수를 계산한다(line 5). 모든 소스 파일에 공통적으로 존재하는 라인과 그 개수를 식별하기에 앞서 효율적인 코드 비교를 위해 식별된 코드 라인들을 사전순으로 정렬한다(line 6). 현재 소스 파일이 첫 번째 파일이라면 비교 대상이 없으므로, 대상 코드의 모든 라인이 초기 공통 라인으로 설정되도록 초기화한다(line 7-8). 앞서 조사된 파일들에서 공통적으로 식별된 라인들과 현재 파일을 구성하는 라인들을 사전 순으로 비교하여 공통적으로 나타나는 코드 라인을 반복적으로 갱신함으로써 입력으로 주어진 집합의 모든 소스 파일에서 공통적으로 나타나는 코드와 그 라인의 수를 파악한다(line 10-24). 마지막으로 모든 소스 파일들을 구성하는 코드의 총 라인 수를 기준으로 모든 소스 파일에서 공통적으로 나타나는 코드의 라인 수가 차지하는 비율을 계산하여 소스 파일 간의 코드 공통성 정도를 정량화한다(line 30).

Fig. 4(b)는 각 소스 파일에 포함된 메소드의 시그니처들을 기반으로 클론 관계의 가능성이 있는 클래스의 집합을 클러스터 후보로 식별하고, 식별된 클러스터 후보들로부터 적절한 클론 클러스터를 선정하는 알고리즘이다. 우선 제품군의 모든 소스 파일로부터 사용된 메소드 시그니처 정보를 파악하고, 각 메소드 시그니처에 대하여 해당 메소드 시그니처가 사용된 소스 파일들을 묶어 클러스터 후보를 식별한다(line 1-11). 식별된 클러스터 후보는 경로 통일의 가능성이 다소 존재하는 소스 파일들의 묶음이며, 하나의 소스 파일이 여러 개의 클러스터 후보에 포함될 수 있다. 그러므로 어느 한 소스 파일에 대한 경로 통일 작업은 그 효과가 가장 큰 후보 집합에서 단 한 번만 처리될 수 있도록 제한되어야 한다.

(a) Measuring Commonality based on Sorted Lines of Code

Input
- **F**: a file list

Output
- **commonality**: a real number representing the rate of common lines

```
 1  entire ← create an empty list
 2  common ← create an empty list
 3  for all file ∈ F do
 4    L ← split all lines in the source code of the file
 5    add all lines of L to entire
 6    SL ← sort L in alphabetical order
 7    if common is empty then
 8      common ← SL
 9    else
10      i ← 1
11      j ← 1
12      temp ← create an empty list
13      while i ≤ |common| and j ≤ |SL| do
14        l1 ← get the i-th line in common
15        l2 ← get the j-th line in SL
16        if l1 precedes l2 in alphabetical order then
17          i ← i + 1
18        else if l2 precedes l1 in alphabetical order then
19          j ← i + 1
20        else
21          add l1 to temp
22          i ← i + 1
23          j ← i + 1
24        end
25      end
26      common ← temp
27    end
28  end
29
30  commonality ← (|file|*|common|) / |entire|
31
32  return commonality
```

(b) Clustering Cloned Source Files based on Method Signature

Input
- **P**: a product list
- **threshold**: a floating value for determining a clone set

Output
- **C**: a list of sets that consist of cloned source files

```
 1  S ← initialize a list of labeled sets
 2  for all product ∈ P do
 3    F ← get all source files of the product
 4    for all file ∈ F do
 5      M ← get all methods of the file
 6      for all method ∈ M do
 7        set ← get the set labeled the method's signature
 8        add file to set
 9      end
10    end
11  end
12
13  PQ[] ← initialize a priority queue array with size |P|
14  for all set ∈ S do
15    n ← get the number of products relevant to files in set
16    enqueue set to PQ[n]
17  end
18
19  C ← initialize a list of sets
20  visit ← initialize a set
21  while PQ is not empty do
22    top ← get the largest index value i that PQ[i] is not empty
23    set ← dequeue a set from PQ[top]
24    if set∩visit = ∅ then
25      commonality ← measure the similarity between files in the set
26      if commonality is above threshold then
27        add set to C
28        visit ← visit ∪ set
29      end
30    else
31      subset ← set – (set∩visit)
32      n ← get the number of products relevant to files in subset
33      enqueue subset to PQ[n]
34    end
35  end
36
37  return C
```

Fig. 4. Algorithms for Clustering and Commonality Index Calculation

경로 통일은 대상 소스 파일들의 코드 유사도가 보장되는 한 많은 수의 제품이 포함된 클러스터 후보의 소스 파일들이 동일한 경로를 갖도록 수정될 때 제품들 간의 구조적 동일성 확보의 효과가 가장 크다. 이러한 처리를 위해서 코드의 유사도에 따라 내림차순으로 정렬하는 우선순위 큐를 제품의 개수만큼 준비하고, 식별된 클러스터 후보들은 관련 제품의 개수에 따라 서로 다른 우선순위 큐에 삽입되도록 한다(line 13-17). 관련 제품의 수가 큰 클러스터 후보들로 구성된 우선순위 큐를 우선적으로 처리하며, 우선순위 큐로부터 추출된 클러스터 후보의 코드 공통성 정도가 임계치(threshold) 이상인 경우에는 해당 클러스터 후보를 클론 클러스터로 결정하고, 클론 클러스터와 관련된 소스 파일들은 경로 통일의 대상이 된다(line 22-30). 만약 어떠한 클러스터 후보가 이전에 경로 통일 대상으로 선정된 소스 파일을 포함하는 경우에는 이전에 경로 통일 대상으로 선정된 이력이 없는 소스 파일들을 선별하여 새로운 클러스터 후보로 재정의하고, 재정의한 후보 집합을 연관된 우선 순위 큐에 삽입함으로써 이후 작업을 계속한다(line 31-33).

4. 제안 방법의 적용 예시

이 장에서는 간단한 예제를 통해 앞서 제안한 방법이 적용되는 과정을 설명한다. Fig. 5는 Fig. 1에서 사용된 제품들의 소스 파일과 의사 코드이다. Fig. 6은 Fig. 5의 제품과 의사 코드를 대상으로 제안 방법을 적용할 때 각 단계에서 수행되는 작업과 그 결과를 보여주고 있다.

4.1 제품 최적화

제품 최적화 단계에서는 제품별 참조 그래프를 생성하고, 참조 그래프를 활용하여 데드코드를 식별해 제거한다. Fig. 6(a)는 Fig. 5의 각 제품에 대하여 소스 코드들 사이에 존재하는 참조 관계를 그래프로 표현하고, 프로그램 시작점으로부터 그래프 탐색을 통해 식별되는 활성코드와 데드코드를 보여주고 있다.

Fig. 6(a)에서 그래프의 파란색 노드는 해당 제품의 실행 중 어느 순간에는 반드시 참조되어 사용되는 코드를 의미하며, 흰색 노드는 해당 제품의 실행에 있어서 단 한 번도 사용되지 않는 데드코드를 나타낸다. 화살표는 참조 그래프에서 노드 간 참조 관계를 나타낸다. 실선 화살표는 임포트 문장에 의해 참조된 코드가 참조하는 쪽의 코드에서 어떠한 형태로든 사용되고 있음을 의미한다. 점선 화살표는 임포트 문장에 의해 참조는 되었지만 참조된 코드가 참조하는 코드 내에서 전혀 사용되지 않은 경우이다. 참조된 코드에 대한 사용 여부는 참조되는 코드의 식별자가 참조하는 코드 내에서 한 번 이상 사용되었는지 파악하여 결정된다.

예를 들어, Fig. 5에서 제품 P2는 제품 P1을 클로닝하여 개발된 제품으로 두 개의 소스 코드 util.B와 util.C를 포함하고 있다. 하지만 제품을 커스터마이징 하는 과정에서 새로운 코드 lib.D를 작성하였고 main.A에서 사용되던 util.B의 메소드 B.m2()를 D.m6()로 변경하였다. 그 결과 제품 P2의 main.A는 util.B를 참조하지만 사용은 하지 않는 형태를 띠게 된다. 이를 반영하여 Fig. 6(a)에서 제품 P2의 참조 그래프는 main.A에서 util.B로의 연결을 점선 화살표로 표현한다. 다른 한 편, 제품 P2의 main.A는 lib.D를 참조하고 또한 사용하므로 참조 그래프에서 main.A에서 lib.D로의 연결은 실선 화살표로 표현된다. 제품 실행 시 처음 호출되는 메소드를 m1()이라 가정한다면, 제품 P2의 참조 그래프는 시작 메소드 m1()이 포함된 main.A를 그래프 탐색의 시작 노드로 표현하게 된다. 이렇게 생성된 제품 P2의 참조 그래프에서 시작 노드로부터 시작하여 실선으로 연결된 노드들에 대한 그래프 탐색을 수행하면, 도달 가능한 노드는 main.A와 lib.D가 된다. 그리고 제품 P2에서 util.B와 util.C는 그래프 탐색으로 도달이 불가한 노드이므로 데드코드가 된다.

4.2 제품 포트폴리오 결정

제품 포트폴리오 결정 단계에서는 각 제품이 제품군의 다른 제품들과 기능적으로 얼마나 유사한지를 수치화하고, 기능적 유사도가 낮은 제품들을 마이그레이션 대상에서 제외한다. Fig. 6(b)는 제품의 기능적 유사도를 산출하는 과정과 제품 포트폴리오를 결정하는 과정을 보여준다.

Fig. 6(b-i)는 제품군을 구성하는 소스 코드의 메소드 시그니처 정보를 바탕으로 기능적 유사도를 계산하는 과정을 보이고 있다. 이 과정에서 첫 번째로 수행되는 작업은 제품군의 모든 소스 파일로부터 식별 가능한 모든 메소드 시그니처의

P1

./main/A.file
```
import util.B;
class A {
  m1() {
    B.m2();
    stmt11;
    stmt12;
    stmt13;
    stmt14;
  }
}
```

./util/B.file
```
import util.C;
class B {
  m2() {
    C.m5();
    stmt21;
    stmt22;
    stmt23;
  }
  m3() {
    stmt31;
    stmt32;
  }
}
```

./util/C.file
```
class C {
  m4() {
    stmt41;
    stmt42;
  }
  m5() {
    stmt51;
    stmt52;
    stmt53;
    stmt54;
  }
}
```

P2

./main/A.file
```
import util.B;
import lib.D;
class A {
  m1() {
    D.m6();
    stmt11;
    stmt12;
    stmt13;
    stmt14;
  }
}
```

./util/B.file
```
import util.C;
class B {
  m2() {
    C.m5();
    stmt21;
    stmt22;
    stmt23;
  }
  m3() {
    stmt31;
    stmt32;
  }
}
```

./util/C.file
```
class C {
  m4() {
    stmt41;
    stmt42;
  }
  m5() {
    stmt51;
    stmt52;
    stmt53;
    stmt54;
  }
}
```

./lib/D.file
```
class D {
  m6() {
    stmt61;
    stmt62;
    stmt63;
    stmt64;
  }
  m7() {
    stmt71;
    stmt72;
    stmt73;
  }
}
```

P3

./main/A.file
```
import util.B;
class A {
  m1() {
    B.m2();
    stmt11;
    stmt12;
    stmt13;
    stmt14;
  }
  m8() {
    stmt81;
  }
}
```

./util/B.file
```
import tool.E;
class B {
  m2() {
    H.m5();
    stmt21;
    stmt22;
    stmt23;
  }
  m3() {
    stmt31;
    stmt32;
    stmt33;
  }
}
```

./util/E.file
```
class E {
  m4() {
    stmt41;
    stmt42;
  }
  m5() {
    stmt51;
    stmt52;
    stmt53;
    stmt54;
  }
  m9() {
    stmt91;
    stmt92;
  }
}
```

P4

./main/C.file
```
import tool.F;
import tool.G;
class C {
  m1() {
    F.m2();
    stmt11;
    stmt12;
    G.m11();
  }
  m4() {
    stmt41;
    stmt42;
  }
  m5() {
    stmt51;
    stmt52;
    stmt53;
    stmt54;
  }
}
```

./tool/F.file
```
class F {
  m2() {
    stmt25;
    stmt26;
    stmt27;
  }
  m10() {
    stmt101;
    stmt102;
  }
}
```

./tool/G.file
```
class G {
  m11() {
    stmt111;
    stmt112;
  }
  m12() {
    stmt121;
    stmt122;
  }
}
```

Fig. 5. Products used in the Motivating Example

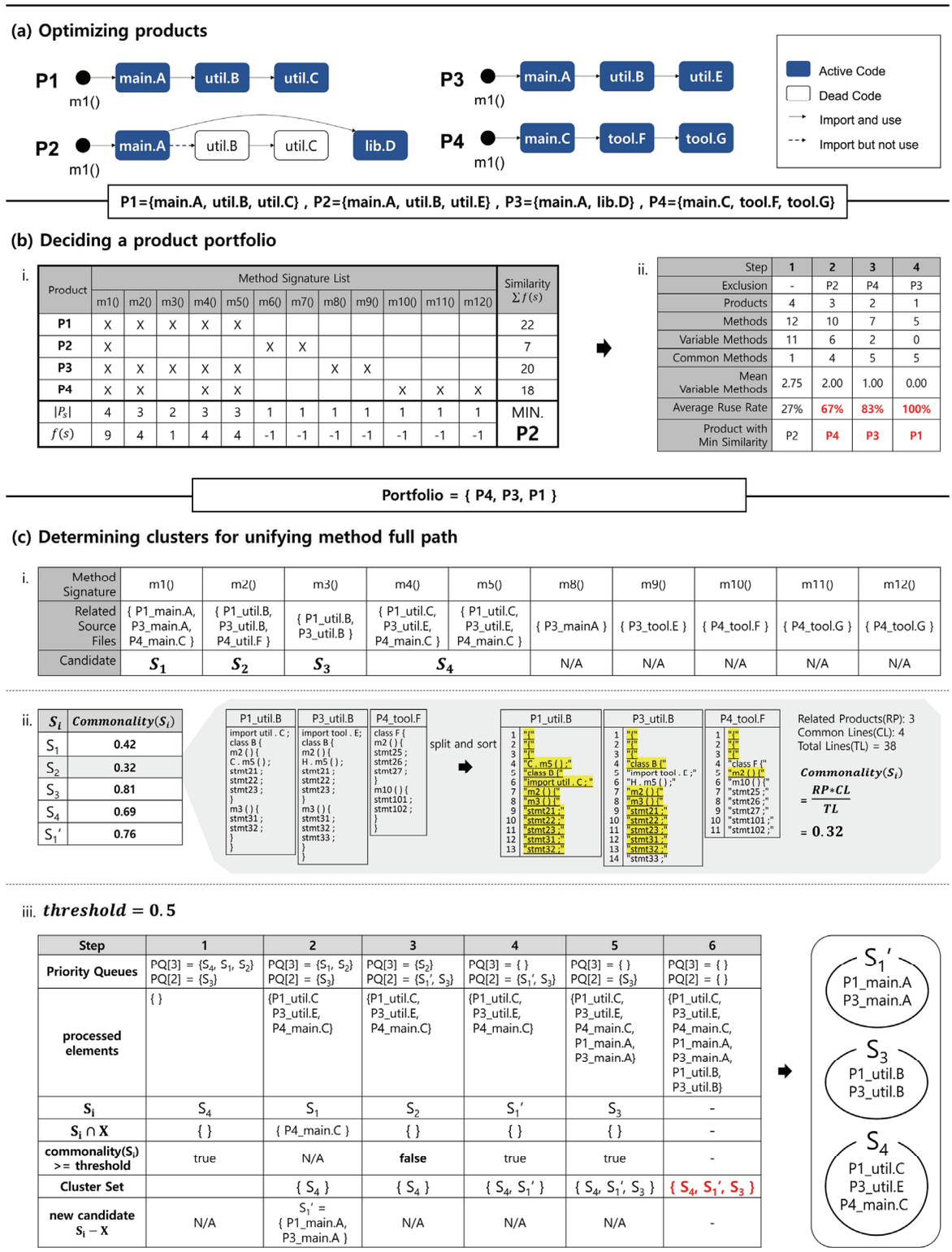

Fig. 6. Step-by-step Illustration of Our Method Using the Products in the Motivating Example

집합을 식별하고, 각 메소드 시그니처가 사용된 제품들을 파악하는 것이다. 식별 작업의 결과는 Fig. 6(b-i)의 표에서 제품의 행과 메소드 시그니처의 열이 교차하는 지점에 기호(X)를 사용하여 표현된다. 다음 작업으로는 메소드 시그니처 별로 메소드 시그니처가 사용된 제품의 개수를 파악하고, Equation (2)를 적용하여 메소드 시그니처 각각에 대한 가중치를 산정한다. Fig. 6(b-i)의 표에서 $|P_s|$ 행은 메소드 시그니처가 사용된 제품의 개수를 보여주고, $f(s)$ 행은 제안 방법에

서 정의한 Equation (2)에 의해 계산된 메소드 시그니처별 가중치 값을 보여준다. 예를 들어, m1()은 총 4개의 제품에서 사용되는 공유 메소드로 Equation (2)를 따르면 특정 제품 측면에서 자기 자신을 제외한 다른 제품에서 재사용되는 횟수를 제곱하여 유사도의 가중치가 계산되어야 하므로 3의 제곱만큼 가중되어 '+9'의 값을 갖게 된다. 반면, m6()는 단 하나의 제품에서만 사용되는 특수 메소드이므로 '-1'의 값을 갖게 된다. 마지막으로 각 제품의 기능적 유사도는 해당 제품을 구성하는 메소드 시그니처들의 가중치 값을 합산하여 계산되고, 이 값들은 Fig. 6(b-i)의 표에서 similarity 열에 표기되어 있다. 그 결과 제품군 내에서 기능적 유사도가 가장 낮은 제품은 기능적 유사도의 값이 7로 나타나는 제품 P2가 된다.

Fig. 6(b-ii)는 기능적 유사도를 측정하고 기능적 유사도가 가장 낮은 제품을 제거하는 과정을 단계적으로 반복하였을 때, 제품군을 구성하는 공통 메소드와 가변 메소드의 개수가 어떻게 변화하는지를 보여준다. 평균 가변 메소드의 개수는 총 가변 메소드의 개수를 총 제품의 수로 나눔으로써 계산된다. 그리고 평균 재사용률은 각 제품에서 제품을 구성하는 메소드들 중 공통 메소드가 평균적으로 차지하는 비율을 나타내는 값으로, 공통 메소드의 개수를 공통 메스드의 개수와 평균 가변 메소드의 개수를 합한 수로 나눔으로써 계산될 수 있다. 예를 들어, 어떠한 제품도 제외되지 않은 초기 제품군으로부터 식별 가능한 12개의 메소드 시그니처 중 공통 메소드 시그니처는 m1()으로 단 한 개뿐이며, 나머지 11개는 가변 메소드 시그니처에 해당한다. 그리고 초기 제품군은 구성하는 제품의 수는 4개이므로 각 제품마다 평균적으로 사용되는 가변 메소드의 개수는 2.75개가 된다. 따라서 제품군에 속한 제품의 평균 메소드의 개수는 공통 메소드의 개수와 평균 가변 메소드의 개수를 합한 3.75가 되며, 평균 재사용률은 공통 메소드의 개수를 평균 평균 메소드의 개수로 나누고 이를 백분율로 표현한 값인 27%가 된다. 이 예제에서는 제품 P2가 제품군으로부터 제외되었을 때, 산출되는 평균 재사용률 67%를 적절한 수준으로 보고 나머지 3개의 제품 P4, P3, P1을 포트폴리오로 결정하였다.

4.3 소스 코드 전처리

소스 코드 전처리 단계에서는 제품군을 구성하는 모든 소스 코드에 대하여 코딩 스타일을 일치시킴으로써 이후 단계인 클러스터링 과정에서 코드의 공통 부분과 가변 부분을 보다 정확하게 식별할 수 있도록 만든다. 제안 방법에서 코딩 스타일을 일치시키는 목적이 코드 가독성 향상에 있지는 않다. 이 예제에서는 매우 간단한 규칙과 문자열 처리를 통해 코딩 스타일을 일치시키는 작업을 수행하였으며, 코딩 스타일을 적용하는 규칙은 다음과 같다:

i) 연속된 토큰 사이에는 띄어쓰기를 한 번 추가하여 연결한다.

ii) 왼쪽 중괄호 다음에는 줄바꿈 문자를 추가한다.

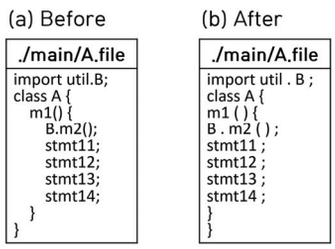

Fig. 7. Preprocessing Result

iii) 오른쪽 중괄호 또는 세미콜론 다음에는 줄바꿈 문자를 추가한다.

v) 마지막에 삽입되는 줄바꿈 문자는 제거한다.

Fig. 7은 제품 P1의 소스 코드를 대상으로 앞서 정의한 규칙에 따른 소스 코드 전처리 단계의 전과 후의 소스 코드를 보여준다.

4.4 코드 클러스터링

코드 클러스터링 단계는 클론 클러스터를 식별하는 단계이며, 클론 클러스터는 동일한 메소드 시그니처를 일부 포함하면서 서로의 소스 코드가 유사한 소스 파일들로 구성된다. 각 클론 클러스터에 속한 모든 소스 파일들에 대해서는 서로의 메소드 경로를 통일함으로써 클로닝 과정에서 발생하는 제품 간 구조적 차이를 줄일 수 있게 된다. Fig. 6(c)는 코드 클러스터링 과정에서 수행되는 작업과 결과를 보여준다.

Fig. 6(c-i)는 각 메소드 시그니처를 포함하는 연관 제품과 소스 코드를 파악하여 클러스터 후보로 식별하는 절차를 나타낸다. 예를 들어, 메소드 시그니처 m1()을 포함하는 소스 코드들은 제품 P1의 main.A와 제품 P3의 main.A, 제품 P4의 main.C이며 이들을 묶어 후보 클러스터 S_1 으로 정의한다. 메소드 시그니처 m4()와 m5()는 연관된 소스 파일의 집합이 서로 동일하므로 한 개의 후보 클러스터는 S_4 로 대응된다. 반면, 메소드 시그니처 m8()부터 m12()까지의 경우에는 해당 메소드 시그니처와 연관된 소스 파일이 단 하나이므로 메소드 경로를 일치시킬 필요가 없어 후보 클러스터에서 제외된다.

Fig. 6(c-ii)는 앞서 파악된 클러스터 후보에 대하여 클러스터를 구성하는 소스 코드 간 공통성을 정량화하는 과정과 결과를 나타낸다. 그리고 클러스터 후보의 공통성 계산은 소스 코드 전처리 단계에서 코딩 스타일이 통일된 소스 파일들을 토대로 수행된다. Fig. 6(c-ii)의 예시는 클러스터 후보 S_2 의 공통성 계산 과정을 보여준다. 우선 제품 P1의 소스 파일 util.B, 제품 P3의 소스 파일 util.B, 제품 P4의 소스 파일 tool.F의 전처리된 소스 코드를 가져오고 각 소스 코드를 라인 단위로 쪼개어 배열에 저장한 뒤, 각 배열에는 문자열에 대한 사전식 정렬이 수행된다. 그리고 각 배열을 순회하며 문자열 비교를 통해 공통 라인의 개수를 파악한다. 노란색으로 강조된 라인들은 왼쪽 소스 파일로부터 현재 소스 파일까지

공통으로 등장하는 라인들을 나타낸다. 그 결과 맨 끝에 위치한 제품 P4의 소스 파일 tool.F에 이르러 파악된 공통 라인의 개수는 4개이며 연관된 제품의 개수는 3개, 연관된 코드들의 총 라인 수는 38이 된다. 연관된 모든 코드에서 최소한 4개의 공통 라인은 항상 재사용 되므로 제품 전체로 보았을 때 공통 라인이 차지하는 총 라인의 수는 연관 제품의 수인 3을 곱한 12가 된다. 이를 바탕으로 전체 라인 측면에서 공통 라인이 차지하는 비율은 12를 38로 나눈 값인 약 0.32가 되고, 이 수치는 후보 클러스터 s_2의 공통성 정도를 나타낸다. 그리고 Fig. 6(c-ii)의 클러스터 후보 s_1'는 클론 클러스터를 결정하는 Fig. 6(c-iii) 과정에서 재정의되어 추가된 클러스터 후보이다.

Fig. 6(c-iii)는 후보 클러스터로부터 실제 메소드 경로의 통일 대상이 될 클론 클러스터를 선정하는 과정과 결과를 보여준다. 우선, 클론 클러스터를 선정하는 데에 있어서 보장되어야 하는 최소한의 공통성 정도를 임계치로 설정한다. 이 예제에서는 임계치를 0.5로 설정하였다. 식별된 후보 클러스터는 관련된 제품의 수에 따라서 서로 다른 우선순위 큐에 삽입되고, 삽입된 후보 클러스터는 우선순위 큐에서 공통성 정도가 높은 순으로 정렬된다. Fig. 6(c-iii)에서 'PQ[3]'는 3개의 제품과 연관된 클러스터 후보들이 속한 우선순위 큐를 말하며, 'PQ[2]'는 2개의 제품과 연관된 클러스터 후보들이 속한 우선순위 큐를 나타낸다. 우선순위 큐들로부터 클러스터 후보를 하나씩 추출하여 클론 클러스터의 여부를 평가할 때에는 연관된 제품의 수가 큰 우선순위 큐에 우선 접근하여 클러스터 후보를 추출한다. 따라서 후보 클러스터가 추출되는 순서는 일차적으로 클러스터와 연관된 제품이 많을수록, 이차적으로는 클러스터의 공통성 정도가 높을수록 앞서게 된다. Fig. 6(c-iii)에서 s_i의 행은 각 스텝에서 평가 대상이 되는 클러스터 후보를 나타내며, 클러스터 후보의 추출 및 평가 순서는 s_4, s_1, s_2, s_1', s_3 가 된다. 처리된 소스 파일의 집합 X는 앞선 스텝에서 클론 클러스터로 결정된 클러스터에 이미 한 번씩은 포함되었던 소스 코드를 나타낸다. 예제에서 첫 번째 스텝의 평가 대상은 클러스터 후보 s_4이다. 클러스터 후보 s_4는 클론 클러스터로 분류된 이력이 없는 파일들로만 구성되어 있으며 s_4의 공통성 수치는 0.69로 임계치 이상의 값을 가지므로 클론 클러스터로 결정되었다. 다른 한 편, 두 번째 스텝에서 평가 대상이 되는 클러스터 후보는 s_1이다. 그러나 클러스터 후보 s_1은 이전에 클론 클러스터로 분류된 이력을 가진 제품 P4의 소스 파일 main.A를 포함하고 있다. 이러한 경우에는 클러스터 후보 s_1으로부터 클론 클러스터로 분류된 이력이 없는 소스 파일들만 선별하여 새로운 클러스터 후보 s_1'를 재정의하고, 재정의된 클러스터 후보 s_1'은 관련 제품의 개수가 2개이므로 PQ[2]에 삽입하여 이후 클론 클러스터 결정 단계에서 관련 클러스터 후보가 재평가될 수 있도록 만든다.

결과적으로 Fig. 6의 모든 과정을 거쳐 식별된 클론 클러스터는 s_1', s_3, s_4이다. 그리고 클러스터 s_1', s_3, s_4에 속한 소스 파일들의 메소드 경로를 각각 PATH1, PATH2, PATH3로 통일한다면, 연구 배경에서 소개된 메소드 경로 통일 이후에 대한 동일 결과를 얻을 수 있게 된다.

5. 실험 및 결과

이 장에서는 CAO 기반으로 개발된 제품군에 제안 방법을 적용해봄으로써 메소드 경로 통일을 위한 코드 클러스터링 방법이 SPLE 마이그레이션 과정에서 유의미하게 작용하고 결과에 긍정적인 영향을 미치는지 실험을 통해서 평가한다.

5.1 실험 대상

ApoGames는 CAO 방식으로 개발된 20개의 자바 프로그램과 5개의 안드로이드 프로그램을 포함한다. 안드로이드 제품들은 자바 제품들에 비해 그 수가 적고 프로그램의 규모면에서도 작다. 자바 제품들은 2006년부터 2012년까지 총 7년에 걸쳐 개발된 제품들인 반면 안드로이드 제품들은 2012년부터 2013년까지 2년이라는 비교적 짧은 기간 동안 개발된 제품들이다. 메소드 경로의 차이에 의한 제품들 간의 구조적 가변성의 영향 정도와 제안 방법의 효과를 보다 분명하게 살펴보기 위해서 제품군을 구성하는 제품의 수가 많고 유지보수 기간이 긴 자바 프로그램을 대상으로 실험을 진행한다.

5.2 실험 결과 평가 방법

코드 측면에서 SPLE 마이그레이션의 목적은 클론 관계에 있는 소스 파일들을 식별하고 이들을 다수의 제품들이 재사용 가능한 형태로 통합하여 코드베이스와 플랫폼을 구축하는 것이다. 코드베이스는 제품마다 독립적으로 관리되던 소스 파일들이 효율적으로 유지보수될 수 있도록 지원하며, 플랫폼은 유사한 기능을 가진 다른 제품을 만들고자 할 때 신속한 제품 개발을 지원한다.

이 논문은 각기 다른 제품에 존재하지만 같은 메소드 시그니처를 포함하면서 서로 유사한 소스 코드들이 동일한 메소드 경로를 갖도록 수정함으로써 불필요한 구조적 가변성을 제거하여 마이그레이션이 효과적으로 수행될 수 있도록 그 기반을 마련하는 방법에 대해 소개하고 있다. 제안 방법에 의해 식별된 메소드 경로 통일의 대상 소스 파일들이 적절하게 선정되었는지 판단하기 위해서는 코드 클러스터링 과정에서 제품군의 소스 파일들이 이상적으로 분류되는 상황에 대한 구체적인 정의가 필요하다.

일반적으로 임의의 두 소스 코드 사이에 공통적으로 나타나는 요소가 서로의 코드 내에서 특정 임계치 이상의 비율을 차지할 때 두 소스 파일은 클론 관계에 있다고 정의한다. 그러나 이러한 정의를 셋 이상의 소스 코드들 사이에 적용하여 이들의 클론 관계를 판단하고 이들로부터 통합 대상을 결정하고자 할 때 이해가 상충하는 상황이 발생하게 된다. Fig. 8은 클론 관계에 있는 소스 파일들이 서로 다른 전략에 따라 달리 그룹화되었을 때 나타나는 결과를 간단한 예시를 통해 보여준다.

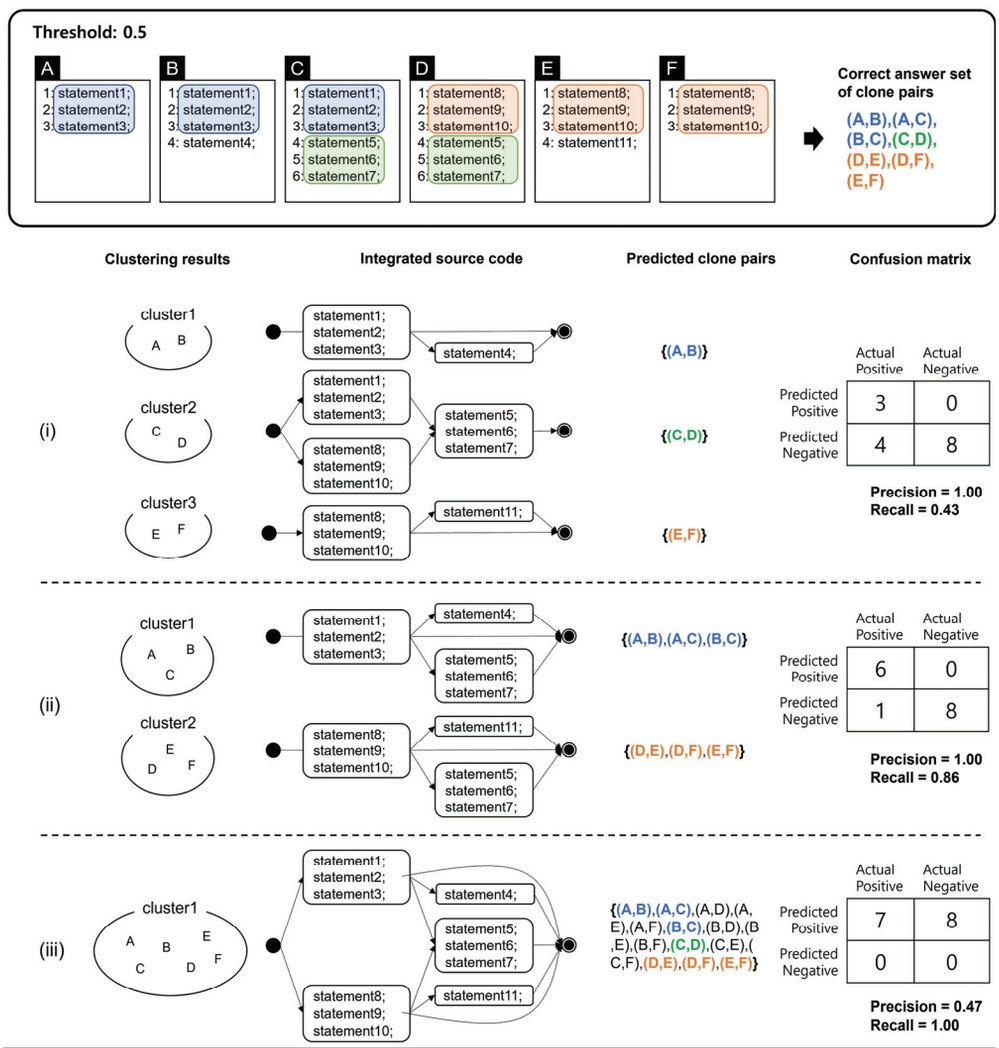

Fig. 8. Example of Integrated Source Code, Clone Pairs, and Confusion Matrix for Different Clustering Results

예제에서는 라인 단위의 비교를 통해 두 소스 파일에서 공통 라인으로 식별된 라인들이 서로의 코드에서 0.5의 임계치 즉, 절반 이상의 비중을 차지할 때 두 소스 파일이 클론 관계에 있다고 판단하고 이러한 클론 쌍들로부터 정답 집합을 결정한다. Fig. 8에서는 각 소스 파일에서 임계치를 초과하는 공통 라인의 집합을 서로 다른 색깔로 구분된 모서리가 둥근 사각형으로 표시하여 클론 관계에 있는 소스 파일들을 나타내고 있다. 이로써 식별된 클론 관계에 있는 소스 파일의 쌍은 (A,B), (A,C), (B,C), (C,D), (D,E), (D,F), (E,F)로 총 7개이며, 이들은 주어진 소스 파일들 간의 클론 관계에 대한 정답 집합이 된다.

Fig. 8의 (i), (ii), (iii)는 서로 다른 전략에 따라 클론 관계의 소스 파일들이 달리 그룹화된 결과를 나타내며, 통합된 소스 코드는 각 그룹의 소스 파일들이 통합되었을 때의 표현이다. 통합 대상으로서 결정된 각 클론 클러스터는 클러스터 내의 소스 파일들이 모두 클론 관계에 있다는 판단 하에 내려진 결정이므로, 각 클러스터 내의 소스 파일들로부터 조합 가능한 모든 쌍을 식별하고 이들을 예측된 클론 쌍으로 정의한다. 혼동 행렬은 클론 쌍의 정답 집합과 예측된 클론 쌍의 관계를 나타내며, 혼동 행렬을 기반으로 Equation (3)과 Equation (4)를 계산하면 클러스터링 결과에 대한 정밀도(precision)과 재현율(recall)을 산출할 수 있다.

$$Precision = \frac{|\ ActualPositive \cap PredictedPositive\ |}{|\ PredictedPositive\ |} \quad (3)$$

$$Recall = \frac{|\ ActualPositive \cap PredictedPositive\ |}{|\ ActualPositive\ |} \quad (4)$$

예를 들어, 클러스터링 전략 (ii)를 적용한 경우, 식별된 클러스터는 2개이며 첫 번째 클러스터에 속한 소스 파일 A, B, C로부터 조합 가능한 소스 파일의 쌍은 (A,B), (A,C), (B,C)로 3개이다. 마찬가지로 두 번째 클러스터로부터 얻을 수 있는 3개의 예측 클론 쌍은 (D,E), (D,F), (E,F)이 된다. 그 결과, 위의 수식을 적용하면 6개의 예측 클론 쌍은 모두 정답 집합에 포함된 클론 쌍이므로 전략 (ii)의 정밀도는 1.0이 되

Table 1. Metrics in Accordance with the Product Portfolio Decision

No.	excluded product	total products	total methods	variable methods	common methods	methods per product	average commonality	product with minimum similarity
1	-	20	3320	3292	28	193	15%	TutorVolley
2	TutorVolley	19	3286	3258	28	199	14%	ApoCheating
3	ApoCheating	18	3152	3117	35	208	17%	ApoDefence
4	ApoDefence	17	2850	2776	74	237	31%	ApoStarz
5	ApoStarz	16	2737	2658	79	245	32%	ApoSoccer
6	ApoSoccer	15	2290	2209	81	228	35%	ApoBot
7	ApoBot	14	2190	2097	93	243	38%	ApoSlitherLink
8	ApoSlitherLink	13	2062	1958	104	255	41%	ApoIcejump
9	ApoIcejump	12	1921	1810	111	262	42%	ApoMario
10	ApoMario	11	1624	1462	162	295	**55%**	**ApoCommando**
11	ApoCommando	10	1364	1199	165	285	**58%**	**ApoMarc**
12	ApoMarc	9	1317	1151	166	294	**56%**	**ApoSkunkman**
13	ApoSkunkman	8	1065	896	169	281	**60%**	**ApoSimpleSudoku**
14	ApoSimpleSudoku	7	1020	844	176	297	**59%**	**ApoPongBeat**
15	ApoPongBeat	6	966	788	178	309	**58%**	**ApoImp**
16	ApoImp	5	930	747	183	332	**55%**	**ApoSnake**
17	ApoSnake	4	880	671	209	377	**55%**	**ApoSimple**
18	ApoSimple	3	481	266	215	304	**71%**	**ApoIcarus**
19	ApoIcarus	2	406	113	293	350	**84%**	**ApoNotSoSimple**
20	ApoNotSoSimple	1	325	0	325	325	**100%**	**ApoRelax**

며, 정답 집합에 포함된 7개의 클론 쌍 중에서 6개만이 예측되었으므로 전략 (ii)의 재현율은 0.86이 된다.

코드 클러스터링의 효과는 독립적으로 관리해야 하는 소스 파일의 개수와 통합된 소스 코드의 재사용성에 의해 결정된다. 예제에서 클러스터링 결과 (i)는 통합된 소스 파일을 사용하는 과정에서 통제해야 할 코드의 가변성 부분이 적어 재사용성이 우수하지만, 결과 (ii)와 비교하면 중복 코드가 여러 소스 파일에 걸쳐 더 많이 보일 뿐만 아니라 통합 이후 관리해야 할 소스 파일의 개수 또한 더 많아 가장 좋은 결과로 보기엔 어렵다. 클러스터링 결과(iii)는 통합된 소스 파일의 개수가 1개로 나타나 가장 우수한 것처럼 보일 수 있지만, 재사용 과정에서 고려해야 할 부분이 많아 재사용의 복잡도가 가장 높으며 이러한 코드를 유지보수하는 것도 쉽지 않기 때문에 좋은 클러스터링 결과로 보기 어렵다. 반면, 클러스터링 결과 (ii)의 통합된 소스 코드는 복잡도가 높지 않으면서 2개의 소스 파일로 통합 관리될 수 있으므로 코드의 복잡도와 파일의 개수 두 측면을 종합적으로 고려할 때, 세 결과 중 가장 준수한 성능을 보여준다고 판단할 수 있다.

정밀도 및 재현율의 값과 통합된 소스 파일의 개수 및 재사용성을 고려한 이상적인 클러스터링의 결과 사이에서 나타나는 관계를 살펴보면 다음과 같이 정의할 수 있다. 정밀도가 높을수록 식별된 클러스터 내의 소스 파일들 대부분이 상호간 실제 클론 관계에 속하게 되어 코드 내부의 가변성이 낮은 경향을 띠지만, 소스 파일들이 소규모로 그룹화될 가능성이 크다. 반면 재현율이 높을수록 일부 소스 파일들 간에만 실제 클론 관계가 성립되더라도 서로가 간접적으로 연결되기만 한다면 이들 모두를 하나의 클러스터로 분류하게 되므로 소스 파일들이 대규모로 그룹화되어 더 적은 수의 소스 파일로 통합될 수 있지만, 모든 소스 파일 간 실제 클론 관계가 엄격하게 보장되지는 않기 때문에 코드의 가변성이 높아져 재사용을 복잡하게 만들 가능성이 높다. 이를 근거로 실험에 대한 평가는 제안 방법의 적용 전과 후의 정밀도와 재현율의 변화 정도를 통해 메소드 경로 통일이 마이그레이션에 긍정적으로 기여하는지 판단한다.

5.3 과정 및 결과

실험은 CAO 방식으로 개발된 ApoGames의 자바 프로그램을 대상으로 2장에서 소개한 제안 방법을 단계적으로 적용하여 수행하였다[9]. 본격적인 실험에 앞서 실험에 사용되는 ApoGames는 자바 언어로 개발된 제품들로 총 20개의 자바 프로그램들로 구성되어 있으며, 그중 7개의 제품은 원본 자바 소스 파일을 제공하지 않고 jar 파일만을 제공하고 있다. jar 파일로부터 자바 소스 파일을 추출하기 위해서 디컴파일링 (decompiling) 작업을 수행하였으며, 이 논문에서는 JAD 디컴파일러 (http://java-decompiler.github.io/)를 사용하여 원본 자바 소스 파일을 추출하였다.

1) 제품 최적화 단계

이 단계에서는 제품을 구성하는 소스 파일들 간의 참조 관계를 그래프로 표현하고 제품의 시작점을 포함하는 소스 파일로부터 그래프 탐색을 통해 도달 가능한 소스 파일과 그렇지 않은 소스 파일을 식별함으로써 각 제품에서 더 이상 사용되지 않는 데드코드를 찾아 제거하였다. 각 제품의 자바 소스 파일로부터 파일들 간의 참조 관계를 식별하고, 자바 프로그램의 시작점인 main() 함수가 정의된 소스 파일을 파악하기 위해 소스 코드를 파싱하는 작업이 요구된다. 실험에서 자바 코드의 파싱은 javaparser (https://javaparser.org/)를 활용하였다. 파싱된 소스 코드로부터 내부에서 사용된 식별자와 임포트된 소스 파일에 대한 정보를 수집하고, 임포트된 소스 파일에 정의된 식별자가 임포트하는 소스 파일의 코드에서 실제로 사용되는 경우 두 소스 파일을 잇는 참조 그래프를 생성하였다. 그리고 자바 언어는 하나의 프로젝트 내에서 둘 이상의 main() 함수의 정의를 허용하므로 이러한 경우에는 적절한 프로그램 시작점을 지정해 주어야 했다. 제품군을 구성하는 20개의 제품에 대하여 참조 그래프를 탐색하여 식별된 데드코드는 총 358개로, 각 제품에는 평균적으로 약 18개의 데드코드가 삽입되었음을 알 수 있었다. 그리고 각 제품에서 데드코드를 제거함으로써 제품군을 구성하는 소스 파일의 총 개수는 1318에서 933개로 줄어들었다.

2) 제품 포트폴리오 결정 단계

이 단계에서는 제품별 메소드 시그니처 정보를 토대로 제품군 내에서 기능적 유사도가 적은 제품, 즉 다른 제품의 기능들을 재사용하여 생성하기에는 복잡하거나 어려운 제품들을 식별하고 이들을 마이그레이션 대상에서 제외하였다. 메소드 시그니처는 파싱된 소스 코드로부터 메소드 이름과 파라미터의 타입, 파라미터의 순서, 추상 메소드 여부에 관한 정보 추출하여 정의되었다.

Equation (1)과 Equation (2)를 사용하여 산출되는 각 제품의 유사도 값은 해당 제품이 속한 제품군의 구성에 따라 영향을 받게 된다. 실험에서는 제품 최적화 단계가 효과적으로 수행될 수 있도록 제품군을 구성하는 각 제품의 유사도를 측정하고 유사도가 가장 낮은 제품을 제거하는 작업을 반복 점진적으로 수행하였다.

Table 1은 제품 포트폴리오 결정 단계의 수행 결과이다. 평균 공통성은 평균 메소드 시그니처 중에서 공통 메소드 시그니처가 차지하는 비율을 백분율로 나타낸 값이다. 최소 유사도를 갖는 제품은 현재 시점의 제품군 내에서 가장 낮은 기능적 유사도를 갖는 제품을 보여주므로, 표에서 최소 유사도를 갖는 제품의 열은 제품 최적화 과정을 거치면서 제품군으로부터 기능적 유사도가 낮은 제품들이 제외되는 순서를 보여준다. 10번째 시행 이후로는 평균 공통성의 변화가 크지 않으므로 안정화된 것으로 판단하여, 위에서부터 9개의 제품 TutorVolley, ApoCheating, ApoDefence, ApoStarz, ApoSoccer, ApoBot, ApoSlitherLink, ApoIcejump, ApoMario를 제외한 11개의 제품을 마이그레이션 대상으로 선정하여 이후 작업을 수행하였다.

3) 코드 전처리 단계

코드 전처리 단계는 제품군과 관련된 소스 코드가 소스 파일로 저장될 때 항상 동일한 스타일을 갖도록 함으로써 코드 클러스터링 단계의 공통성 계산 과정에서 공통 라인이 가변 라인으로 식별되는 상황을 최소화하기 위해 수행되었다. 실험에서는 javaparser가 제공하는 코드 정렬 기능을 활용하여 일차적으로 코딩 스타일을 일치시킨 다음, 주석문과 공백 라인을 제거하고 토큰을 하나 이상 포함하는 나머지 라인에 대해서는 각 라인의 시작과 끝 위치에 삽입된 공백 문자들을 제거하는 절차를 거쳐 코드 전처리가 수행되었다.

4) 코드 클러스터링

이 단계에서는 일부 메소드 시그니처가 동일하며 서로의 코드가 유사한 소스 파일들을 식별하여 메소드 경로의 통일 대상으로 분류하는 클러스터링 작업이 수행되었다. 우선 소스 파일을 파싱하여 모든 메소드 시그니처를 추출하였다. 그리고 식별된 각 메소드 시그니처에 대하여 해당 메소드 시그니처를 포함하는 파일들의 경로 및 관련 제품 정보를 수집하여 관련 소스 파일들의 집합을 생성하였다. 이렇게 생성된 소스 파일들의 집합들은 경로 통일의 후보 집합이 되며, 식별된 후보 집합은 서로소 관계가 아니므로 경로 통일 대상의 선정으로 이어지는 후보 집합의 평가 순서는 클러스터링의 결과에 상당한 영향을 미치게 된다. 그래서 일차적으로는 후보 집합과 연관된 제품의 개수가 많고, 이차적으로는 후보 집합을 구성하는 소스 파일 간 코드 공통성이 높은 후보 집합을 우선적으로 선택하여 평가를 진행하였다. 그리고 선택된 후보 집합의 코드 공통성이 임계치 이상일 때, 해당 후보 집합을 경로 통일 대상으로 결정하였다.

Table 2와 Table 3에서 전체 파일은 제품군을 구성하는 제품들에 포함된 자바 소스 파일의 개수를 말한다. 고유 파일은 단일 클러스터와 관련된 파일로 CAO 기반 제품군 생성 과정에서 클로닝을 통한 재사용과는 무관하다고 판단되는 파일들의 개수를 의미하며, 클론 파일은 공유 클러스터 및 공통 클러스터와 관련된 파일로 CAO 기반 제품 생성 과정에서 클로닝에 의해 생성되었다고 판단되는 파일들의 개수를 의미한다. 단일 클러스터, 공유 클러스터, 공통 클러스터는 각각 단 하나의 소스 파일로 구성된 클러스터, 모든 제품은 아니지만 둘 이상의 제품들로부터 수집된 소스 파일들로 구성된 클러스터, 모든 제품으로부터 수집된 소스 파일로 구성된 클러스터를 의미한다. 평균 공통성은 둘 이상의 소스 파일로 구성된 클러스터들의 평균적인 공통성 값을 나타내며 최소 공통성은 클러스터들 중에서 공통성이 가장 낮은 클러스터의 공통성 값을 나타낸다. 임계치는 메소드 경로의 통일 대상으로 선정된 클러스터가 보장해야 하는 최소한의 소스 파일 간 코드 공통성 정도를 의미한다.

Table 2. Clustering Performance without the Proposed Approach

products	total files	unique files	cloned files	total clusters	single clusters	shared clusters	common clusters	average commonality	minimum commonality
20	1318	730	588	785	730	55	0	0.863	0.028

Table 3. Clustering Performance with the Proposed Approach

products	total files	threshold	unique files	cloned files	total clusters	single clusters	shared clusters	common clusters	average commonality	minimum commonality
11	520	0.1	214	306	259	214	30	15	0.462	0.112
		0.2	264	256	306	264	28	14	0.598	0.211
		0.3	287	233	328	287	28	13	0.740	0.313
		0.4	296	224	334	296	25	13	0.795	0.408
		0.5	308	212	343	308	22	13	0.838	0.520
		0.6	315	205	348	315	20	13	0.863	0.632
		0.7	338	182	367	338	18	11	0.915	0.703
		0.8	348	172	377	348	19	10	0.945	0.806
		0.9	366	154	391	366	17	8	0.960	0.902
		1.0	452	68	464	452	9	3	1.000	1.000

Table 2는 CAO 기반으로 개발된 20개의 제품 전체를 대상으로 별도의 처리 없이 동일한 경로 즉, 동일한 디렉토리 경로 및 파일 이름을 갖는 소스 파일들을 클러스터링한 결과를 보여주며, 공통 클러스터가 단 하나도 식별되지 않았음을 확인 수 있다. 공통성 값은 0~1의 값을 나타내는데 최소 공통성 값이 0.028이라는 것은 서로의 코드가 상당히 다른 소스 파일들이 하나의 클러스터로 묶인 상황을 의미한다. 이러한 소스 파일들은 동일한 메소드 경로를 갖더라도 서로 유사하지 않은 코드이므로 각기 다른 메소드 경로를 갖도록 수정함으로써 코드베이스 내에서 독립적인 자산으로 분할 및 관리되는 것이 더 나은 선택이 될 수 있다.

반면, Table 3은 제안 방법을 적용하여 20개의 제품 중에서 관련성이 적다고 판단되는 9개의 제품을 제외한 11개의 제품을 대상으로 이 논문에서 제안한 클러스터링 방법을 적용하여 소스 파일들을 클러스터링한 결과를 보여준다. 그 결과 이전에는 식별되지 않던 공통 클러스터의 수가 제안 방법을 적용한 이후 15개로 상당히 증가한 것을 확인하였다. 또한, 식별된 클러스터들의 최소 공통성 역시 임계치 이상으로 유지되어, 식별된 모든 클러스터가 지정된 최소한의 코드 공통성을 보장하는 것을 확인할 수 있었다. 이러한 결과는 Table 2에서 보았던 0.028의 코드 공통성을 갖는 클러스터와 같이 부적절하다고 판단되는 클러스터들이 제안 방법에 의해 효과적으로 통제될 수 있음을 보여준다.

Table 4는 앞서 정의한 평가 지표인 정밀도와 재현율을 소스 코드 측면과 메소드 시그니처 측면에서 측정한 결과를 나타낸다. 임의의 두 파일에 대하여 두 파일로부터 식별된 동일 코드 라인이 각각의 소스 파일에서 임계치 이상의 비율을 차지하는 경우 두 파일의 쌍을 소스 코드 측면에서의 정답 집합을 구성하는 요소로 정의하였다. 다른 한 편, 임의의 두 소스 파일에 대하여 두 파일로부터 식별된 동일 메소드 시그니처가 각각의 소스 파일에서 임계치 이상의 비율을 차지하는 경우에는 두 파일의 쌍을 메소드 시그니처 측면에서의 정답 집합을 구성하는 요소로 정의하였다. 그리고 클러스터링 결과로 식별된 각각의 클러스터에 대하여 클러스터를 구성하는 소스 파일들로부터 조합 가능한 모든 소스 파일의 쌍을 예측 결과로 정의하였다. 정밀도와 재현율은 앞에서 정의한 정답 집합과 예측 결과를 바탕으로 Equation (3)과 Equation (4)에 의해 계산된 값을 나타낸다.

클러스터링 방법이 높은 정밀도를 보인다는 것은 클러스터에 포함된 대부분의 소스 파일들이 상호 간 높은 공통성을 띠며 서로 클론 관계에 있음을 나타낸다. 그러므로 클러스터링 방법이 높은 정밀도를 갖는 경우에는 클러스터에 속한 소스 파일들에 대하여 메소드 경로를 통일하는 작업을 수행함으로써 여러 제품에서 식별된 공통성이 높은 특정 소스 코드가 각 제품에서 사용될 때 항상 동일 구조를 갖는 요소로서 정의 및 사용되도록 강제하는 효과를 얻을 수 있다. 그리고 제품 간 공통 코드의 구조적 동일성을 확보하는 작업은 플랫폼 구축을 촉진한다.

클러스터링 방법이 높은 재현율을 갖는다는 것은 제품군에 속하는 임의의 두 제품에서 식별 가능한 클론 관계의 소스 파일 쌍이 클러스터링 결과로 파악된 클러스터로부터 대부분 식별될 수 있음을 의미한다. 그러나 재현율만 높은 경우에는 클러스터에 포함된 소스 파일들이 일부 또는 특정 소스 파일과의 관계에서만 높은 공통성을 보일 수 있다. 극단적인 예로, 제품군의 모든 소스 파일을 하나의 클러스터로 묶는 클러스터링 방법 있다면 이 클러스터링 방법의 재현율은 최대치

Table 4. Precision and Recall of Clustering Results

Threshold	20 Products				11 Products			
	Method Signature		Source Code		Method Signature		Source Code	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0.1	0.96	0.06	1.00	0.01	0.93	0.09	1.00	0.02
0.2	0.96	0.15	1.00	0.03	0.92	0.17	1.00	0.04
0.3	0.96	0.29	1.00	0.28	0.93	0.26	1.00	0.35
0.4	0.95	0.38	0.99	0.67	0.99	0.34	0.99	0.70
0.5	0.93	0.46	0.97	0.78	0.98	0.41	0.98	0.80
0.6	0.92	0.59	0.95	0.83	0.99	0.55	0.99	0.90
0.7	0.91	0.71	0.91	0.90	0.97	0.61	0.97	0.85
0.8	0.89	0.76	0.87	0.94	0.98	0.66	0.98	0.84
0.9	0.84	0.78	0.79	0.97	0.98	0.64	0.98	0.87
1.0	0.76	0.79	0.62	0.79	1.00	0.34	1.00	0.66
avg.	0.91	0.50	0.91	0.62	0.97	0.41	0.99	0.60

인 1이 된다. 재현율의 이러한 특징은 식별된 클러스터가 매우 낮은 코드 공통성을 갖게 되는 상황을 만들기도 한다.

Table 4에서 20개의 제품에 대한 결과 부분은 ApoGames 전 제품에 대하여 별다른 처리 없이 파일의 상대 경로 기반 클러스터링 방법을 적용한 결과로 동일한 디렉토리 경로 및 파일명을 갖는 파일들을 각각의 클러스터로 정의하였을 때의 정밀도와 재현율을 보여준다. 그리고 11개의 제품에 대한 결과 부분은 이 논문에서 제안하는 클러스터링 방법을 적용하여 클러스터를 식별하였을 때의 정밀도와 재현율을 보여준다.

제안 방법에 의해 메소드 시그니처 측면에서의 정밀도는 임계치에 따라 0.76~0.96이었던 값이 0.92~1.00으로 개선되었으며, 소스 코드 측면에서의 정밀도는 임계치에 따라 0.62~1.00이었던 값이 0.97~1.00으로 개선되었다. 이러한 결과는 제안 방법으로 식별된 클러스터가 높은 코드 공통성을 보장하므로 클러스터를 구성하는 파일들의 메소드 경로를 통일시킴으로써 제품들의 구조적 동일성을 확보할 수 있도록 돕는다.

반면, 메소드 시그니처 측면의 재현율은 임계치에 따라 0.06~0.79이었던 값이 0.09~0.66으로 감소하고, 소스 코드 측면에서의 재현율은 임계치에 따라 0.01~0.97이었던 값이 0.02~0.90으로 감소하였다. 그러나 파일의 상대 경로 기반 클러스터링 방법을 적용하였을 때 0.028의 코드 공통성을 갖는 클러스터가 식별되었으며, 코드의 공통성이 확보되지 않은 클러스터는 메소드 경로가 통일되더라도 구조적 동일성 확보로 인한 이점을 얻기 어렵다는 것을 앞서 확인하였다. 이는 클러스터링 방법의 재현율이 높더라도 높은 수준의 정밀도가 보장되지 않는다면 식별된 클러스터가 효과적으로 사용될 수 없음을 나타낸다.

실험 결과, CAO 기반으로 개발된 제품군일지라도 기능적으로 유사하지 않은 제품이 제품군에 포함될 수 있음을 확인하였다. 또한 클로닝 과정에서 공통 코드가 서로 다른 메소드 경로를 갖는 요소들로 재정의될 수 있음을 확인하였다. 그리고 CAO 기반 제품 개발 과정에서 발생하는 이슈들은 본격적인 마이그레이션에 앞서 기능적으로 관련성이 높은 제품들을 선정하고, 소스 파일의 코드 및 메소드 시그니처의 유사성을 고려하여 경로 통일 대상을 식별하는 일련의 작업을 통해 구조적 동일성을 확보함으로써 완화될 수 있음을 확인하였다.

6. 관련 연구

[12]는 클론을 공통 또는 공유된 부분을 지시하는 것으로 간주하고, 클론 검출이 PL로의 마이그레이션에 유용함을 Salion에서의 경험을 통해 확인해 주었다. 클론 검출 방법의 결과는 후에 클론들 중 변경없이 그대로 사용한 클론은 공통성으로, 변경된 클론은 가변성으로 분류하는데 사용될 수 있다. 제품군의 멤버들 간의 복제되거나 복제되어 수정된 코드들을 찾아야 한다. [13-17]과 같은 클론 검출 방법을 사용하는 연구들은 유사도 계산 방법들을 사용하여 클론쌍의 유사도를 계산하고 정한 임계값에 따라 공통성 및 가변성 의사결정을 수행한다. [16]은 세부적으로 정의된 비교 메트릭을 제공하고 있다. [16]과 [17]은 클론한 모델로부터 공통성과 가변성을 추출하고 있다. [18]과 [19]는 공통 코드 추출을 유용하게 하기 위한 부가적인 방법을 제안하고 있다. [16]은 CAO로 개발된 제품으로부터 제품라인으로 마이그레이션 과정에서 리팩토링을 수행하여 효과적으로 공통 코드를 추출하는 방법을 제시하고 있다. [17]은 최소-와이즈(minwise) 해싱 기술을 활용하여 소스 코드 집합을 입력으로 받아 유사한 파일과 구성 요소를 추출하는 코드 검색 방법을 제안한다. 이들 연구는 산출 결과가 공통 코드와 가변 코드의 비율 정보를 제공하는 수준에 그치고 있다.

반면, 피처 로케이션 방법은 피처 로케이션 이후 공통성과 가변성에 대한 판단을 하고 피처 모델을 구성한다. [5]는 사용된 가변성 메커니즘에 따라 수작업으로 가변 피처 코드를

추출하였다. [20]은 자동화된 피처 로케이션을 수행하고 그 결과로부터 입력으로 사용한 본래 제품들의 재구성이 가능한지 확인한다. 그러나 그 결과는 피처라기보다는 소스 코드들에 대한 그룹핑으로 볼 수 있다. 이들 그룹핑은 규모(granularity)가 모듈 단위로 이 두 유형의 연구는 모두 피처 모델 구축을 위한 시작점을 제공하긴 하지만 PL에서 사용할 플랫폼 산출물에 대한 구축을 지원하는 형태는 아니다.

추가적으로, CAO 방식으로 개발된 소스 코드로부터 아키텍처를 재구축하는 연구 또한 진행되어 왔다[21-23]. [21]은 But4Reuse와 CodeCompare 도구를 이용하여 피처를 추출한 후 수작업으로 피처 모델을 복원하였다. [22]는 제품라인 아키텍처 복원을 위한 메트릭과 제품들간 개념적 계층구조에 대한 제안이고, [23]은 소스 코드로부터 공통 및 가변 클래스를 포함하는 클래스 다이어그램을 복원한 연구이다. 이들 연구는 제품라인 아키텍처의 시작점이 될 수는 있지만, 이후 코드베이스 구축까지 별도의 많은 노력이 투입되어야 한다.

7. 결론 및 향후 연구

이 논문은 CAO 기반으로 개발된 제품들로부터 마이그레이션 대상 제품들을 선정한 후 제품들에 흩어져 있는 유사 코드 집합을 검출하여 메소드 경로의 통일이 필요한 대상을 식별하는 클러스터링 방법을 제안하였다.

실험 결과, 제안 방법을 적용하였을 때 공통 클러스터의 개수가 기존 0개에서 설정 임계치에 따라 3~15개로 증가하였다. 공통 클러스터의 존재와 그 개수는 모든 제품에서 재사용 가능한 플랫폼을 구축할 수 있는지에 대한 여부와 제품 개발 시 플랫폼의 재사용 효과를 결정하므로 그 의미가 크다. 또한, 제안 방법의 정밀도는 평균 0.98로 파일의 상대 경로 기반 클러스터링 방법의 정밀도가 평균 0.91인 것에 비해 개선된 수치를 보여주었다.

제안 방법은 앞서 수행된 실험에서 긍정적인 결과를 보여주었다. 그러나 이 실험에서 사용된 ApoGames 제품군이 CAO 기반의 제품 개발에서 발생 가능한 모든 경우와 상황을 대변할 수는 없다. 향후 연구로는 더 다양한 언어와 규모로 작성된 제품군에 제안한 방법을 적용하여 제안 방법이 일반적으로 적용 가능한 방법인지 평가할 계획이다. 더 나아가 식별된 공통 부분과 가변 부분을 토대로 하나의 공유 또는 공통 자산으로 통합하고 이를 토대로 소프트웨어 제품라인의 플랫폼을 구축하는 연구를 수행할 계획이다.

References

[1] C. W. Krueger and K. Jackson, "Requirements engineering for systems and software product lines," Product line management white paper, Dec. 2009. (http://www.biglever.com/extras/RE_for_SPL.pdf).

[2] R. Lapeña, M. Ballarin, and C. Cetina, "Towards clone-and-own support: Locating relevant methods in legacy products," in *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*, pp.194-203, 2016.

[3] J. Bosch, "Maturity and evolution in software product lines: Approaches, artefacts and organization," *The 2nd International Systems and Software Product Line Conference (SPLC), Lecture Notes in Computer Science*, Vol.2379, pp.247-262, 2002.

[4] V. Anwikar, R. Naik, A. Contractor, and H. Makkapati. "Domain-driven technique for functionality identification in source code," in *SIGSOFT Software Engineering Notes*, Vol.37, No.3, pp.1-8, 2012.

[5] E. Kuiter, J. Krüger, S. Krieter, T. Leich, and G. Saake, "Getting rid of clone-and-own: Moving to a software product line for temperature monitoring," in *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC)*, Vol.A, pp.1-11, 2018.

[6] M. Abbas, R. Jongeling, C. Lindskog, E.P. Enoiu, M. Saadatmand, and D. Sundmark, "Product line adoption in industry: An experience report from the railway domain," in *Proceedings of the 24th International Systems and Software Product Line Conference (SPLC)*, Vol.A, pp.14-24, 2020.

[7] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proceedings of 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pp.25-34, 2013.

[8] E. Ghabach, "Supporting clone-and-own in software product line," Doctoral thesis, Software Engineering, Université Côted'Azur, 2018.

[9] C. Lima, I. do Carmo Machado, E. S. de Almeida, and C. von Flach G Chavez, "Recovering the product line architecture of the apo-games," in *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC)*, pp.289-293, 2018.

[10] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: A framework and experience," in *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pp.101-110, 2013.

[11] N. Lodewijks, "Analysis of a clone-and-own industrial automation system: An exploratory study," in *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution (SATToSE)*, pp.1-12, 2017.

[12] I. D. Baxter and D. Churchett, "Using clone detection to manage a product line," Workshop on Industrial Experience with Product Line Approaches, pp.1-3, 2002.

[13] D. Faust and C. Verhoef, "Software product line migration and deployment," in *Software: Practice and Experience*, Vol.33, No.10, pp.933-955, 2003.
[14] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," in *Software Quality Journal*, Vol.17, No.4, pp.331-366, 2009.
[15] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pp.163-172, 2008.
[16] A. Schlie, S. Schulze, and I. Shaefer, "Recovering variability information from source code of clone-and-own software systems," in *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*, pp.1-9, 2020.
[17] A. Schlie, A. Knüppel, C. Shidl, and I. Shaefer, "Incremental feature model synthesis for clone-and-own software systems in MATLAB/Simulink," in *Proceedings of the 24th International Systems and Software Product Line Conference (SPLC)*, Vol.A, pp.53-64, 2020.
[18] W. Fenske, J. Meinicke, S. Schulze, and G. Saake, "Variant-preserving refactorings for migrating cloned products to a product line," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp.316-326, 2017.
[19] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue, "Source file set search for clone-and-own reuse analysis," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pp.257-268, 2017.
[20] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability extraction and modeling for product variants," *Software System Model*, Vol.16, No.4, pp.1179-1199, 2017.
[21] J. Debbiche, O, Lignell, J Krüger, and T. Berger, "Migrating Java-based apo-games into a composition-based software product line," in *Proceedings of the 23nd International Systems and Software Product Line Conference (SPLC)*, Vol.A, pp.98-102, 2019.
[22] C. 1, W. Assunção, J. Martinez, W. Mendonça, I. C Machado, and C. Chavez, "Product line architecture recovery with outlier filtering in software families: the apo-games case study," in *Journal of the Brazilian Computer Society*, Vol.25, No.7, pp.1-17, 2019.
[23] J. Lee, T. Kim, and S. Kang, "Recovering software product line architecture of product variants developed with the clone-and-own approach," in *Proceedings of the IEEE 44th International Conference on Computers, Software and Applications (COMPSAC)*, pp.985-990, 2020.

김 태 영

https://orcid.org/0000-0002-5002-9472
e-mail : wareengineer@gmail.com
2019년 전북대학교 소프트웨어공학과(학사)
2019년 ~ 현 재 전북대학교 소프트웨어공학과(석사과정)
관심분야 : Software Product Line & Architecture Reconstruction

이 지 현

https://orcid.org/0000-0003-4512-806X
e-mail : jihyun30@jbnu.ac.kr
1993년 전북대학교 정보통신공학과(학사)
2000년 전북대학교 전자계산교육(석사)
2005년 전북대학교 컴퓨터과학과(박사)
2016년 ~ 현 재 전북대학교 소프트웨어공학과 교수
관심분야 : Software Product Line & Architecture Reconstruction

김 은 미

https://orcid.org/0000-0003-2108-8809
e-mail : ekim@howon.ac.kr
1993년 전북대학교 전산통계학과(석사)
1997년 오사카대학교 정보공학과(박사)
1997년 ~ 현 재 호원대학교 컴퓨터·게임학과 교수
관심분야 : Software Quality Evaluation & Development Methodology