creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

함수 단위 중복코드 쌍을 자동으로 안전하게 고치는 방법

Automatic and Safe Refactoring of Function-Level Code Clones

2019 년 2 월

서울대학교 대학원

컴퓨터 공학부

양 준 모

공학석사학위논문

함수 단위 중복코드 쌍을 자동으로 안전하게 고치는 방법

Automatic and Safe Refactoring of Function-Level Code Clones

2019 년 2 월

서울대학교 대학원

컴퓨터 공학부

양 준 모

함수 단위 중복코드 쌍을 자동으로 안전하게 고치는 방법

Automatic and Safe Refactoring of Function-Level Code Clones

지도교수 이 광 근

이 논문을 공학석사학위논문으로 제출함

2018 년 11 월

서울대학교 대학원

컴퓨터 공학부

양 준 모

양준모의 석사학위논문을 인준함

2018 년 12 월

위 원 장	하순회	(인)
부위원장	이광근	(인)
위 원	이창건	(인)

요약

본 논문은 JAVA 소스코드를 대상으로 함수단위 코드중복 쌍을 찾아 자동으로 안전하게 고쳐주는 방법을 제안한다. 이 방법은 크게 두 단계로 구성된다. (i) 전체 소스코드 내에서 중복코드 탐지기를 이용하여 중복코드 쌍을 찾아낸다. (ii) 찾아낸 중복코드 쌍을 유형별로 분류한 뒤 알맞은 알고리즘을 통해 안전하게 고친다. (i)에서는 Deckard[2] 중복코드 탐지기를 활용한다. (ii)에서는 입력받은 함수단위 중복코드 쌍을 두가지 유형으로 나누어 처리한다. 첫번째 유형은 두 함수의 타입이 같을 때로 두 함수를 병합하여 처리하고, 두번째 유형은 타입이 다를 때로 내부 중복부분을 추출하여 처리하는 것이다. 이 방법의 실효성을 보이기 위해 함수단위 중복 리팩토링 도구 AutoRefactor를 설계 및 구현하였다. 구현한 도구로 회사 제품코드와 오픈소스 프로젝트에서 찾은 실제 중복을 없애기 위해 실험한 결과, 같은 파일 내 존재하는 함수단위 중복코드 6,082 LOC(55개쌍 중복) 중 3,976 LOC(30개쌍 중복)를 처리할 수 있었다.

주요어: 중복 코드 수정, 자동 리팩토링, 중복 함수 병합

학번: 2017-20635

목차

요약 **i**

목차 **ii**

그림 목차 **iv**

표 목차 **v**

제 1 장 서론 **1**

1.1 동기 1

1.2 해결 방안 1

1.3 결과 2

1.4 논문의 구성 2

제 2 장 중복코드 탐지 **3**

2.1 중복코드의 유형 3

2.2 중복 리팩토링의 유형 4

2.3 코드 중복의 정의 5

2.4 Deckard[6] 6

제 3 장 중복처리 알고리즘 **7**

3.1 두 중복함수의 타입이 같은 경우 7

3.1.1 함수 병합 7

3.2 두 중복함수의 타입이 다른 경우 9

3.2.1 함수 추출 9

제 4 장 중복처리 알고리즘 확장 **12**

4.1 함수 병합 : 처리방식 다양화 12

4.2 벤치마크 적용 예시 15

제 5 장 실험 결과 **17**

제 6 장 관련 연구 및 논의 **19**

제 7 장 결론 **20**

참고문헌 **21**

Abstract **23**

그림 목차

그림 2.1 두 중복 함수의 추상구문트리와 서로 다른 부분을 가진 서브트리 6

그림 3.1 두 중복 함수 f_1과 f_2의 병합 7
그림 3.2 두 중복 함수 f_1과 f_2의 병합 예시 8
그림 3.3 함수 중복 병합 알고리즘 9
그림 3.4 두 중복 함수 f_1과 f_2내 중복 추출 9
그림 3.5 두 중복 함수 f_1과 f_2내 중복 추출 예시 10
그림 3.6 함수 중복 추출 알고리즘 11

그림 4.1 두 중복함수 f_1과 f_2의 인자 전달방식 병합 12
그림 4.2 두 중복함수 f_1과 f_2의 인자 전달방식 병합 예시 13
그림 4.3 인자 전달방식 함수 병합 알고리즘 14
그림 4.4 여러라인 함수 이름 전달하기 예시 15
그림 4.5 Lvalue 제네릭 타입인자 전달하기 예시 16

표 목차

표 2.1 단위/위치별 중복 유형 4
표 2.2 깃허브 내 리팩토링 커밋 코드 조사 내용표 5

표 5.1 중복탐지 및 자동 리팩토링 실험결과 17

제 1 장 서론

1.1 동기

현업에서 인력 및 시간부족의 원인으로 체계적인 개발을 하지 못하여 중복코드가 발생한다. 중복코드가 존재할 경우 코드 간의 연결성이 보존되지 않아 잠재적인 위험성을 가진다.[1][3] 이는 회사에서 개발되는 소프트웨어 뿐만 아니라 임의의 사람이 참여할 수 있는 오픈소스 소프트웨어에서도 나타나는 현상이다.

이러한 발생 중복의 다수는 함수단위이다. 실제 코드의 사례조사와 오픈소스 버전관리 사이트 깃허브에서 개발자들의 커밋수를 조사한 결과, 함수단위 중복이 가장 많이 발생하는 것을 확인할 수 있었다.

그러나, 중복코드의 실재에도 불구하고 중복을 제거하는 리팩토링은 잘 이루어지지 못하고 있다. 오래된 프로그램 코드의 경우 개발자들이 잘 모르기 때문에 고치기 전 이를 공부하기 위한 비용이 들고, 사람이 고칠 때는 항상 실수할 가능성이 있기 때문에 개발자들은 현재 잘 동작하는 코드를 손대기 꺼려하기 때문이다.

1.2 해결 방안

이러한 문제를 해결하기 위하여 본 연구에서는 중복코드를 탐지하고 중복된 함수단위 코드쌍을 자동으로 안전하게 고쳐주는 효율적인 방법을 제안한다. 이 방법의 동작과정을 풀어 설명하면 다음과 같다.

1. 소스 코드를 입력으로 받아 소스 코드내 중복 부분을 탐지한다: 프로그램을 트리로 변환하고, 트리의 서브트리를 뽑아내어 각각 비교한다. 트리의 구조적 유사도를 조절하여 유사한 코드 모양을 찾을 수 있다. 이 과정은 잘 알려진 중복코드 탐지기 Deckard를 활용한다.

2. 탐지된 중복코드 쌍에 대해 두 가지 유형으로 분류한 뒤, 각 유형에 맞는

중복제거 알고리즘을 적용시킨다: 두 개의 함수단위 중복코드 쌍이 함수의 타입이 같은 경우 병합하고, 함수의 타입이 다른 경우 내부 중복부분을 추출한다.

1.3 결과

본 연구의 방법이 효율적이라는 것을 보이기 위해 중복코드 자동 리팩토링 도구 AutoRefactor를 설계 및 구현하였다. AutoRefactor가 실제 현업의 제품 소프트웨어 3개와 오픈소스 벤치마크 3개를 대상으로 중복코드를 탐지하고 올바르게 고치는 것을 확인하였다. 실험 결과, 발견한 함수단위 중복의 65%를 올바르게 처리할 수 있었다.

1.4 논문의 구성

본 논문의 구성은 다음과 같다. 2장에서는 본 연구에서 풀고자 하는 중복을 정의하고, 이를 탐지하는 방법을 설명한다. 3장에서는 찾은 함수단위 중복코드 쌍을 고치는 알고리즘에 대해 설명한다. 4장에서는 본 연구의 알고리즘을 확장하는 과정을 설명한다. 5장에서는 AutoRefactor의 실험결과를 설명한다. 6장에서는 관련된 연구를 소개하고 본 연구와 비교한다. 7장에서는 본 논문의 결론을 짓는다.

제 2 장 중복코드 탐지

이 장에서는 기존 코드에서 발생하는 중복의 유형, 중복 유형별 리팩토링 방법들, 본 연구에서 풀고자 하는 함수단위 중복의 정의, 그리고 중복코드를 자동으로 찾을 때 사용하는 중복코드 탐지기 Deckard에 대해 소개한다.

2.1 중복코드의 유형

중복코드의 유형은 해당 중복코드의 크기별, 분포별로 나눌 수 있다. 다음은 JAVA 프로그램에서 발생하는 중복코드의 유형별 구분이다.

1. 함수/문장 단위 분산된 중복 (C1) : 함수/문장 단위 중복코드가 서로 다른 클래스에 존재하는 경우이다. 각 클래스내 존재하는 속성값 지정/반환 함수의 중복, 형제 클래스간 존재하는 유사한 기능의 함수 중복구현, 그리고 단순한 기능을 재사용하기 위한 중복구현이 있다.

2. 함수/문장 단위 지역적인 중복 (C2) : 함수/문장 단위 중복코드가 같은 클래스 내 존재하는 경우이다. 기능의 재사용 및 확장을 위해 기존의 함수를 복사하여 변수의 타입/이름, 혹은 상수값을 바꾼 경우가 있다.

3. 클래스 단위 분산된 중복 (C3) : 클래스 단위 중복코드가 여러 파일에 걸쳐 존재하는 경우이다. 이 경우 해당하는 케이스는 예외 클래스(Exception Class), 테스트 클래스, 자료구조의 구현체와 같이 비슷한 코드 패턴을 가진 클래스의 중복이 있다.

4. 클래스 단위 지역적인 중복 (C4) : 클래스 단위 중복코드가 같은 클래스 내 존재하는 경우이다. 비슷한 기능을 하는 자료구조 구현체를 내부 클래스를 이용해 하나의 클래스 내에 중복구현한 케이스가 있다.

단위 및 위치	클래스 단위	함수 단위
같은 클래스	클래스 단위 내부 중복(C4)	함수/문장 단위 내부 중복 (C2)
다른 클래스	클래스 단위 분산 중복(C3)	함수/문장 단위 분산 중복 (C1)

표 2.1: 단위/위치별 중복 유형

본 논문에서 풀고자 하는 문제는 단위/위치별 중복 유형(표 2.1) 중 2번에 해당하는 함수 단위 지역 중복을 자동으로 해결하는 방법이다. 같은 파일 내 존재하는 함수 단위 중복 쌍이 AutoRefactor가 받는 입력에 해당된다.

2.2 중복 리팩토링의 유형

중복 코드를 제거하기 위해 고안된 리팩토링 방법들이 있다.[4] 다음은 리팩토링 방법들을 중복 코드의 단위별, 의도별로 구분한 것이다.

1. Extract method (EM) : 중복된 함수/기능 단위 코드조각을 함수로 따로 빼내는 리팩토링 방법이다. 함수/문장 단위 지역적인 중복(C2)이 적용대상이다.

2. Pull-up method (PM) : 중복된 함수단위 코드조각을 상위 클래스로 올리는 리팩토링 방법이다. 함수/문장 단위 분산된 중복(C1)이 적용대상이다.

3. Extract superclass (ES) : 중복된 클래스단위 코드조각을 묶어 상위 클래스를 만들어 올리는 리팩토링 방법이다. 클래스 단위 분산된 중복(C3), 클래스 단위 지역적인 중복(C4)이 적용대상이다.

4. Extract utility-class (EU) : 중복된 클래스단위 코드조각을 묶어 중복된 기능을 표현한 클래스를 만들어 올리는 리팩토링 방법이다. 클래스 단위 분산된 중복(C3), 클래스 단위 지역적인 중복(C4)이 적용대상이다.

표 2.2는 깃허브 Search에서 EM/PM/ES/EC 키워드를 포함하도록 검색한 결과이다. Extract class에 대한 쿼리는 Extract utility-class 리팩토링 방법이 일

리팩토링 유형	쿼리명	검색결과
Extract method (EM)	”Extract method” duplicate	3099건
Pull-up method (PM)	”Pull-up method” duplicate	22건
Extract superclass (ES)	”Extract super-class” duplicate	18건
Extract class (ES + EU)	”Extract class” duplicate	148건

표 2.2: 깃허브 내 리팩토링 커밋 코드 조사 내용표

반적으로 해당 기능을 가진 클래스 명을 커밋명으로 사용하기 때문에 그보다 상위 집합에 해당하는 Extract superclass까지 포함한 Extract class 쿼리 결과를 확인한 것이다. 원하는 리팩토링 커밋은 중복 코드들에 관한 것이므로, duplicate 키워드를 포함한 커밋들만을 모았다.

AutoRefactor가 풀고자 하는 문제는 Extract method (EM)에 해당하는 같은 파일내 함수단위 중복이다. 표 2.2에서 보듯이, 사람들은 함수 단위의 중복 처리인 EM 리팩토링을 많이 하고 있었다. 이는 함수 단위 중복이 실제로 많이 발생하고, 사람들이 이를 해결하기 위해 문제를 많이 겪고 있다는 것을 보여준다.

2.3 코드 중복의 정의

먼저 JAVA 프로그램의 추상구문트리(AST) 구조에 대해 설명한다. JAVA 프로그램의 AST는 클래스 단위, 인터페이스 단위 정의부 블록을 상위로 가지고 그 안에 멤버 함수, 변수들의 자식 노드들을 가진다. 함수 노드의 자식으로는 블록, 그리고 그 하위에 문장, 그리고 말단 이전의 표현식, 마지막으로 최말단의 변수/함수의 이름, 상수값, 연산자 등이 있다.

함수단위 중복이 있다는 것의 정의는 다음과 같다. 그림 2.1은 두 함수 f_1, f_2의 AST를 나타낸다. 함수단위 중복코드의 AST인 AST_1과 AST_2가 블록 단위의 구조는 같고, 그 두 AST간의 차이를 가진 서브트리들 중 i번째 트리인 $SubT_1^i$와 $SubT_2^i$는 문장 단위이면서, $SubT_1^i$와 와 $SubT_2^i$의 차이가 말단 노드의 상수 값,

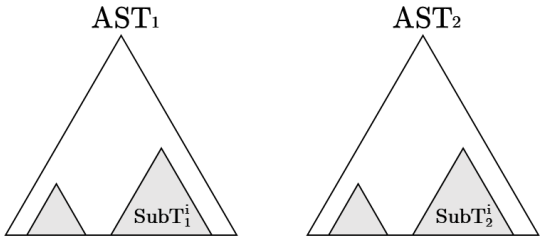

그림 2.1: 두 중복 함수의 추상구문트리와 서로 다른 부분을 가진 서브트리

변수/함수의 이름 등에만 있는 경우, 두 중복함수 쌍은 중복이다.

2.4 Deckard[6]

Deckard 중복코드 탐지기는 추상구문트리에서 동작하는 중복코드 탐지기이다. Deckard는 프로그램 트리를 벡터공간에 옮긴 뒤 벡터화된 트리끼리의 유클리드 거리를 계산하여 프로그램의 유사도를 계산한다. Deckard는 검사할 트리의 깊이, 그리고 중복쌍의 유사도 정도를 조절할 수 있으며, 구하고자 하는 중복의 크기 또한 조절이 가능하다. 또한, 큰 프로젝트 단위의 프로그램 집합도 빠른 시간내에 중복 탐지가 가능하다. AutoRefactor의 중복탐지 앞단은 Deckard의 힘을 빌려 중복코드 쌍들을 찾는다.

$$\text{Deckard} : (2^{\text{Program}} * \text{Params}) \rightarrow 2^{\text{CloneData}}$$

입출력 설명 Deckard의 입출력은 다음과 같다. 먼저 입력으로 2^{Program}(중복탐지 대상 프로그램 집합), Params(입력 파라미터들)을 받는다. 여기서 Params은 중복탐지의 단위, 유사도 등으로써 Deckard를 조절할 수 있는 인자 집합이다. 출력으로 $2^{\text{CloneData}}$(중복탐지 정보의 집합)을 낸다. CloneData는 중복 파일의 이름, 중복 라인, 중복의 크기를 포함한 정보이다.

제 3 장 중복처리 알고리즘

이 장에서는 Deckard를 이용하여 탐지한 중복 지점들에 대해 실제로 코드를 고치는 방법을 소개한다. AutoRefactor는 두 함수단위 중복쌍의 타입을 비교하여 유형별로 처리한다. 두 중복함수의 타입이 같으면 첫번째 알고리즘인 함수 병합 방법을 적용하고, 다르면 두번째 알고리즘인 함수 추출 방법을 적용한다. 여기서 타입이 같다는 의미는 두 함수의 반환타입, 인자들의 각 타입과 이름들, 그리고 던지는 예외까지 같은 경우를 의미한다.

3.1 두 중복함수의 타입이 같은 경우

3.1.1 함수 병합

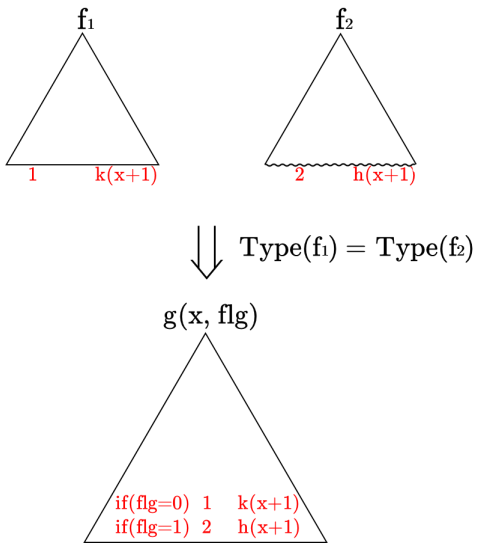

그림 3.1: 두 중복 함수 f_1과 f_2의 병합

두 중복함수의 타입이 같은 경우 함수 병합(그림 3.1)은 중복함수 f_1과 f_2를 병합된 함수 g로 만들어낸다. 중복부분은 하나로 합치고, 두 중복함수간 차이를

가진 부분을 모아 플래그 값을 이용해 분기로 나눈다.

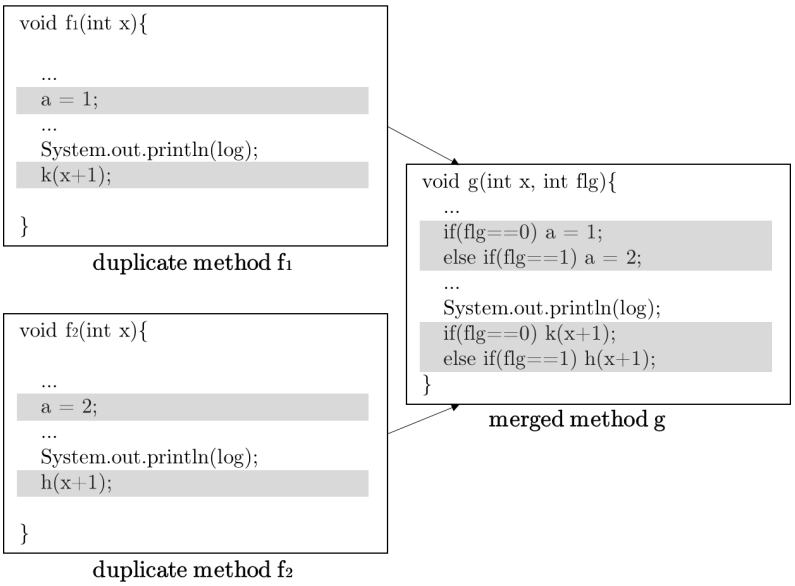

그림 3.2: 두 중복 함수 f_1과 f_2의 병합 예시

그림 3.2는 함수 f_1, f_2를 병합하는 예시이다. 두 중복함수 f_1, f_2내 차이가 존재하는 문장들에 분기가 삽입되고, 병합된 함수 g내에 실행흐름 선택자 flg에 따라 분기가 결정되어 실행된다.

함수 병합 알고리즘(그림 3.3)은 중복함수 F_1과 F_2를 받아 병합된 함수 G를 내놓는다. 두 중복함수의 차이가 존재하는 문장인 D를 모으고, D의 다른 토큰이 문장의 LHS에 해당되는 지 확인한다. 만약 맞을 경우, 본 알고리즘에서는 처리하지 못하여 종료한다. 아닐 경우, 새로 정의하는 병합함수 G에 인접한 문장들을 모아 플래그 값을 기준으로 분기하고, 바뀐 이름과 인자를 호출자들에게 적용한다.

호출자 이름 바꾸기 함수 병합은 두 중복함수의 정의를 바꾸기 때문에 기존의 호출자들에게 바뀐 이름과 인자를 전달해야 한다. 여기서, AutoRefactor는 전체 JAVA 프로젝트의 CallGraph를 이용한다. CallGraph는 호출자의 객체명과 함수명, 피호출자의 객체명과 함수명 정보의 리스트로 이루어져 있다. 두 함수의 호출자 정보를 모아 새로 바뀐 이름과 인자를 함께 호출자들에게 전달한다.

function MergeMethod (F_1, F_2);

Data: Duplicate Method F_1, F_2

Result: Merged Method G

$D \leftarrow$ getDiff(F_1, F_2)

if *hasLhsDiff* **then**

  abort;

else

  $G \leftarrow$ mergeDuplicateStmts(F_1, F_2)

  $G \leftarrow$ addFlgBranch(D)

  renameCallers;

end

그림 3.3: 함수 중복 병합 알고리즘

3.2 두 중복함수의 타입이 다른 경우

3.2.1 함수 추출

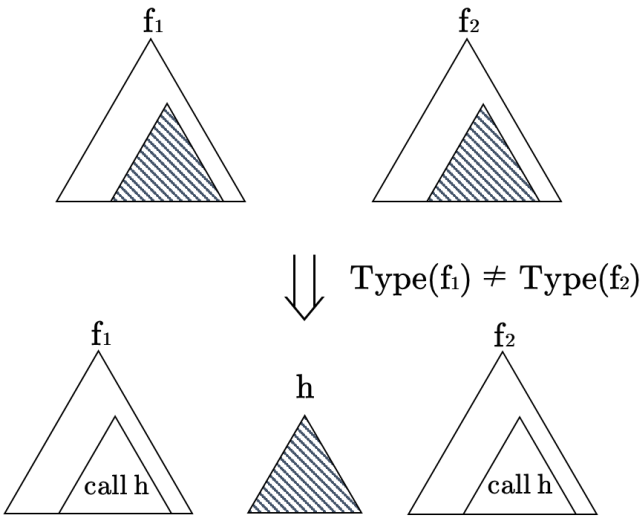

그림 3.4: 두 중복 함수 f_1과 f_2내 중복 추출

두 중복함수의 타입이 다른 경우 함수 추출(그림 3.4)은 중복함수 f_1과 f_2 내부의 가장 큰 중복 서브트리를 추출한다. 추출한 중복 서브트리를 새로운 함수 h로 정의하고, 중복이 존재하던 부분은 새로운 함수의 호출부로 대체한다. 두 중복함수의 타입이 같더라도 함수가 오버라이드된 경우 동일한 방법을 적용한다.

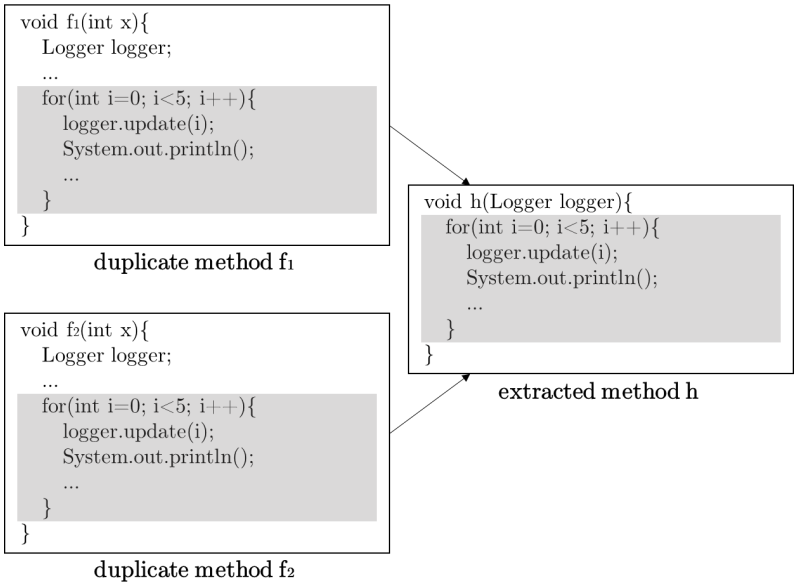

그림 3.5: 두 중복 함수 f_1과 f_2내 중복 추출 예시

그림 3.5는 함수 f_1과 f_2의 내부중복을 추출하는 예시이다. 두 중복함수 f_1, f_2 내 존재하는 가장 큰 블록인 for 반복문이 함수 h로 추출된다. 추출 시, 중복 블록 외부에서 정의된 Logger 변수를 함수 h에 인자로 전달해야 한다.

함수 추출 알고리즘(그림 3.6)은 중복함수 F_1과 F_2를 받아 추출된 함수 G를 내놓는다. 먼저, 두 중복함수 내 가장 큰 중복 서브트리 $SubT$를 찾는다. 찾은 $SubT$의 실행흐름 이전에 선언된 변수들 중 $SubT$내에 사용되는 변수들 $VarSet$을 모은다. 그리고 $SubT$의 실행흐름 이후에 영향받는 변수들 중 $VarSet$에 속하는 $SeVarSet$을 모은다. 여기서 추출한 서브트리 $SubT$를 새로운 함수로 정의할 때, 기존의 함수에서 $VarSet$을 전달받고 $SeVarSet$을 반환해야 한다. 마지막으로 기존 중복 서브트리 부분을 제거하고 새로운 함수 호출부로 대체한다.

<u>function ExtractMethod</u> (F_1, F_2);

Data: Duplicate Method F_1, F_2

Result: Extracted Method G

$SubT \leftarrow$ findLargestCommonSubtree(F_1, F_2)

if *!found* **then**

  abort;

else

  $VarSet \leftarrow$ fetchVarSet$(F_1, F_2, SubT)$

  $SeVarSet \leftarrow$ fetchSeVarSet$(F_1, F_2, SubT)$

  $G \leftarrow$ defineMethod$(SubT, VarSet, SeVarSet)$

  replaceMethodCall;

end

그림 3.6: 함수 중복 추출 알고리즘

제 4 장 중복처리 알고리즘 확장

이 장에서는 AutoRefactor의 중복처리 알고리즘을 확장하는 방법을 설명한다. 함수병합에서 인자로 전달하는 방식을 추가로 설명하고, 실제 현실세계의 프로그램에서 어떻게 적용될 수 있는지에 대해 소개한다.

4.1 함수 병합 : 처리방식 다양화

함수 병합 방법의 처리방식을 다양화할 수 있다. 병합시 두 중복함수 중 다른 상수 값과 타입, 함수의 이름을 인자로 전달하여 처리하는 것이다. 이는 몇몇 경우에서 분기를 나누는 기존 방식보다 고친 후 코드의 라인수를 더 줄인다. 그리고 중복의 차이를 직접 인자로 명시하여 가독성을 개선한다. 이 장에서는 함수 병합 알고리즘을 확장한 인자 전달 방식을 설명하고, 그 적용 예제를 보여준다.

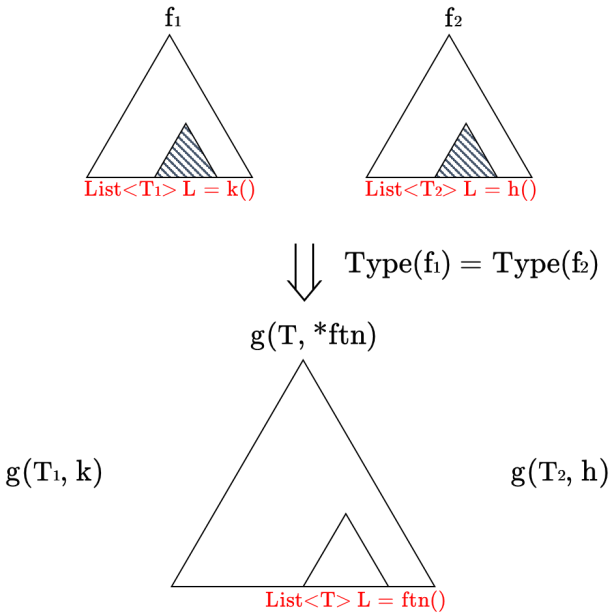

그림 4.1: 두 중복함수 f_1과 f_2의 인자 전달방식 병합

인자전달 방식 함수 병합(그림 4.1)은 중복함수 f_1과 f_2를 병합된 함수 g로 만들어낸다. 중복부분은 하나로 합치고, 두 중복함수간 다른 부분에 해당하는 상수값, 함수이름, 타입 부분을 비워놓고 인자로 받아 실행하게 한다.

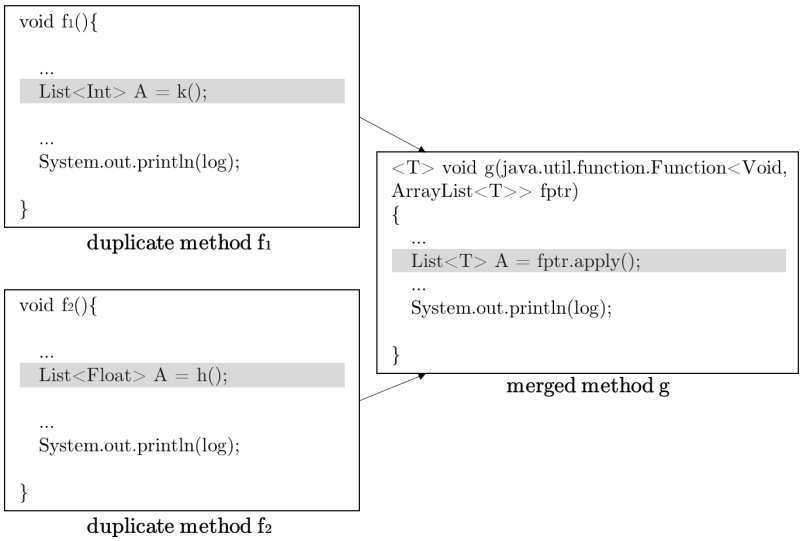

그림 4.2: 두 중복함수 f_1과 f_2의 인자 전달방식 병합 예시

그림 4.2는 인자전달 방식으로 실제 중복함수 쌍을 병합하는 예시이다. 두 중복함수 f_1, f_2의 차이가 존재하는 문장들 중, 다른 토큰들에 대해서 그 값을 비워놓고 인자로 받아 실행한다. 함수 이름의 경우 JAVA 1.8의 기능인 java.util.function을 사용하여 함수 오브젝트로 전달한다.

인자전달 방식 함수 병합 알고리즘(그림 4.3)은 중복함수 F_1과 F_2를 받아 병합된 함수 G를 내놓는다. 두 중복함수의 차이가 존재하는 문장인 D를 모으고, D 안에 있는 다른 차이가 나는 값 중 함수 병합시 인자로 빼낼 $Args$를 모은다. 여기서 전달하는 $Args$는 상수값, 타입, 함수의 이름이 될 수 있다. 타입의 경우, 제네릭의 내부 타입인자를 RHS의 함수 타입과 짝 맞추어 적용할 수 있다. 마지막으로 각 중복함수내 새로운 인자들과 바뀐 이름을 호출자들에게 적달한다.

```
function MergeMethodArgPass (F1, F2);
Data: Duplicate Method F1, F2
Result: Merged Method G
D ← getDiff(F1, F2)
Args ← fetchDiffTok(D);
if !argPassCond then
    abort;
else
    G ← mergeDuplicateStmts(F1, F2);
    if isTypePatch then
        G ← passArgAndType(D, Args);
    else
        G ← passArg(D, Args);
    end
    renameCallers;
end
```

그림 4.3: 인자 전달방식 함수 병합 알고리즘

함수병합 방식 비교 함수병합의 분기나누기 방식과 인자전달 방식을 비교했을 때 어떤 방식이 더 좋은지 판단하기 어렵다. 함수병합 분기나누기 방식은 함수 오브젝트를 쓰지 않기 때문에 어떤 함수가 불릴지를 미리 알 수 있고, 따라서 컴파일러 최적화가 더 가능할 수 있다. 그러나, JAVA 프로그램은 Just In Time 컴파일을 하기 때문에 어떻게 컴파일될 지 예측하기 힘들다. 실제로 두 방법을 비교했을 때 큰 차이가 없었다. 여러 테스트케이스에서 실험해본 결과, 평균 2% 내외 성능변화만 있음을 확인할 수 있었다.

4.2 벤치마크 적용 예시

기존의 함수 병합 알고리즘 대신 인자전달 방식으로 병합한 결과 좀 더 간결한 코드를 만들 수 있었다. 다음은 인자전달 방식을 적용한 실제 프로그램 예시이다.

여러라인 함수 이름 전달하기 중복사례 중 두 중복함수내 다른 함수 이름이 예시(그림 4.4)와 같이 여러라인에 걸쳐있는 경우가 있다. 이 경우, 함수의 이름만 인자로 빼내어 전달하면 병합된 코드를 좀 더 간결하게 만들 수 있다.

```
void put_h264_chroma_mc8_c( ){

    ...
    OP_PUT(dst_base[dst_offset + 0]);
    OP_PUT(dst_base[dst_offset + 1]);
    OP_PUT(dst_base[dst_offset + 2]);
    OP_PUT(dst_base[dst_offset + 3]);
    ...
}
```

```
void avg_h264_chroma_mc8_c( ){

    ...
    OP_AVG(dst_base[dst_offset + 0]);
    OP_AVG(dst_base[dst_offset + 1]);
    OP_AVG(dst_base[dst_offset + 2]);
    OP_AVG(dst_base[dst_offset + 3]);
    ...
}
```

그림 4.4: 여러라인 함수 이름 전달하기 예시

그림 4.4의 두 중복함수의 차이는 OP_PUT, OP_AVG 두 함수의 이름이고, 두 중복함수에는 호출되는 함수의 전달 인자까지 동일하다. 여기서 함수 내부의 중복된 인자부분은 유지하며 함수의 이름을 인자로 넘긴다. 실행시간에 OP_PUT, OP_AVG을 함수 오브젝트로 전달하여 실행할 수 있다.

Lvalue 제네릭 타입인자 전달하기 Lvalue의 제네릭의 타입인자가 다른 예시 (그림 4.5)가 있다. 인자전달 방식 함수 병합으로 이를 타입인자를 빼내어 처리할 수 있다. 이 경우에 Lvalue의 제네릭의 타입인자를 주는 RHS의 함수 타입을 함께 보고 인자로 타입과 함수 오브젝트를 전달한다. 실행시간에 전달되는 인자에 따라 함수에 맞춰 타입을 결정하고 실행될 수 있다.

```
ActionForward statsUserToner( ){

   ...
   List<User> slist = selectTopUser();

   request.setAttribute("FORM", form);
   request.setAttribute("SLIST", slist);

   return mapping.findForward("user");
}
```

```
ActionForward statsDeptToner( ){

   ...
   List<Dept> slist = selectTopDept();

   request.setAttribute("FORM", form);
   request.setAttribute("SLIST", slist);

   return mapping.findForward("dept");
}
```

그림 4.5: Lvalue 제네릭 타입인자 전달하기 예시

그림 4.5의 두 중복함수 차이는 LHS의 제네릭 타입인자 User, Dept와 해당 제네릭 타입의 객체를 반환하는 RHS의 함수 selectTopUser, selectTopDept와 마지막 라인의 스트링 상수값 "user", "dept"이다. 이 경우는 타입과 함께 타입을 결정하는 RHS의 함수를 전달하고, 마지막 문장의 스트링 상수 값을 함께 인자로 전달할 수 있다.

제 5 장 실험 결과

중복코드를 탐지하고 고치는 실험을 위해, 회사제품 프로젝트 3개와 오픈소스 벤치마크 3개, 총 6개의 벤치마크를 대상으로 실험하였다. 코드 중복을 탐지하기 위한 탐지기로 Deckard를 사용하였고, 중복탐지의 트리구조 유사도는 100%, 같은 파일 내 중복에 대해 실험하였다. 실험환경은 Ubuntu 16.04.5 LTS 시스템 및 Intel(R) Core(TM) i7 2.70GHz, 8GB RAM이며 최대 4개의 코어를 사용하였다.

Criteria / Bench	LOC	#DC(Pair/LOC)	#PC(Pair/LOC)(count)	Time
ePrint	215.37K	11P/1,496LOC	5P/800LOC(1/-/4)	3m29
jpcsp	198.98K	15P/1,694LOC	9P/1,272LOC(5/1/3)	3m57
open-lmis	147.86K	9P/520LOC	6P/289LOC(5/1/-)	5m27
Ironjacamar	136.18K	10P/1,127LOC	5P/749LOC(3/2/-)	1m51
dp-server	85.71K	5P/896LOC	3P/726LOC(2/1/-)	3m07
risk-view	49.67K	5P/349LOC	2P/140LOC(1/1/-)	1m40
Total	833.77K	55P/6,082LOC	30P/3,976LOC(17/6/7)	19m31

표 5.1: 중복탐지 및 자동 리팩토링 실험결과

표 5.1은 탐지된 중복코드를 AutoRefactor를 이용해 고친 결과이다. Bench는 벤치마크의 이름이다. Criteria는 표에서 나타낸 실험 기준들이다. LOC는 각 벤치마크의 전체 코드 줄 수이다. #DC(Detected Clones)는 각 벤치마크에서 탐지된 함수단위 중복코드 쌍의 갯수와 코드 줄 수를 나타낸다. #PC(Patched Clones)는 AutoRefactor이 각 벤치마크에서 탐지된 중복을 성공적으로 고친 중복코드 쌍의 갯수와 코드 줄 수를 나타낸다. (count)는 #PC에서 리팩토링의 유형별로 계수한 결과이다. (count)는 각각 함수병합 분기나누기로 처리된 중복, 함수추출로

처리된 중복, 함수병합 인자전달로 처리된 중복의 수를 나타낸다. Time은 각 벤치마크 별로 중복을 탐지하고 고치는 데에 걸린 시간의 합계이다. ePrint, dp-server, risk-view는 회사의 제품 프로그램이고, open-lmis[7], Ironjacamar[8], jpcsp[9]는 오픈소스 벤치마크 프로그램이다. Total은 전체 실험결과의 합계이다.

실험 결과, 전체 83만 라인의 프로그램에 대해서 중복을 탐지하고 고치는 데 19분 31초가 걸렸다. 탐지된 전체 중복 6,082 LOC(55개쌍 중복) 중 65%에 해당하는 3,976 LOC(30개쌍 중복)를 고칠 수 있었다.

안전성 확인 AutoRefactor가 중복코드를 고치고 난 후에도 프로그램 행동이 이전과 같다는 것을 확인하기 위해 수정 전후의 빌드와 테스트 결과를 비교하였다. Gradle 빌드 자동화 도구와 개발자들이 작성한 Junit 단위 테스트를 이용하여 코드 중복을 수정하기 이전 벤치마크에서 빌드와 테스트를 하고, 코드 중복을 수정한 이후 다시 빌드와 테스트를 확인하였다. 결과적으로, AutoRefactor가 중복을 수정하고 난 후에도 빌드와 테스트를 성공적으로 통과하는 것을 확인할 수 있었다.

제 6 장 관련 연구 및 논의

중복코드의 추적과 관리 IDE를 기반으로 한 중복코드의 추적과 관리에 대한 연구 [5]가 있다. 이 연구의 목적은 개발자 환경에서 중복코드를 추적하고 관리하고자 함에 가까운 것이라, 본 논문에서 목표로 하는 중복 자동 리팩토링과 방향이 다르다. 그러나 본 연구에서 다루는 중복에 대한 정의와 중복 관리에 대한 방법에서 아이디어를 얻을 수 있었다.

중복코드의 의존관계 측면에서 본 리팩토링 중복코드의 의존관계 측면에서 중복을 다루는 연구 [3]가 있다. 이 연구의 목적은 의존관계가 있는 중복 쌍을 정의하여 찾고 중복제거 리팩토링을 돕는 데 있다. 그러나, 중복코드의 의존관계와 그 연결관계에 좀 더 관심을 두며 실제로 코드 수준에서 중복을 고치는 방법에 대해서는 면밀히 다루지 않는다.

중복코드 처리의 한계 본 연구의 방법은 함수단위 중복코드 쌍에 대해서 가장 안전하게 처리할 수 있는 부분만을 수정한다. 구조적으로 동일한 두 중복함수의 차이가 말단 노드에 있고, 그 노드가 변수의 이름일 경우 이 중복함수는 처리하지 않는다. 이는 변수명을 짓는 개발자의 의도를 파악할 수 없기 때문이다. 최종적으로 중복코드를 처리하는 것은 개발자의 피드백이 필요할 것이다. 중복코드를 고치면서 정해줄 변수/함수의 이름 등을 입력받으면 상호작용을 통해 더 많은 중복을 고칠 수 있을 것이다.

제 7 장 결론

본 논문에서는 JAVA 소스코드를 대상으로 중복된 함수단위 코드쌍을 찾아 자동으로 안전하게 고쳐주는 방법을 제안한다. 이 방법은 전체 소스코드 내에서 Deckard 중복코드 탐지기를 이용하여 중복함수 쌍을 찾아내고 이를 분류한 뒤 중복함수 쌍의 타입에 따라 알맞은 알고리즘을 통해 고치는 것이다. 본 연구에서는 이 방법의 실효성을 보이기 위해 함수단위 중복 자동 리팩토링 도구 AutoRefactor를 설계 및 구현하였다. AutoRefactor는 실제 회사 제품코드와 오픈소스 프로젝트를 대상으로 같은 파일 내 존재하는 중복코드 쌍들을 모아 실험하여 전체 중복 6,082 LOC(55개쌍 중복) 중 3,976 LOC(30개쌍 중복)를 안전하게 처리할 수 있었다.

참고문헌

[1] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel and Stefan Wagner, "Do code clones matter?", in *2009 IEEE 31st International Conference on Software Engineering*, Vancouver, British Columbia, Canada, May 2009. pp. 485-495.

[2] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su and Stephane Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones", in *2007 IEEE 29th International Conference on Software Engineering*, Minneapolis, Minnesota, USA, May 2007. pp. 96-105.

[3] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue, "On refactoring support based on code clone dependency relation", in *2005 11th IEEE International Software Metrics Symposium*, Como, Lombardy, Italy, Sept 2005. pp. 10-16.

[4] Minhaz F. Zibran, Chanchal K. Roy, "A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring", in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, Williamsburg, Virgin Islands, USA, Sept 2011. pp. 105-114.

[5] Minhaz F. Zibran, Chanchal K. Roy, "Towards flexible code clone detection, management, and refactoring in IDE", in *2011 Proceedings of the 5th International Workshop on Software Clones*, Honolulu, Hawaii, USA, May 2011. pp. 75-76.

[6] Deckard from University of California / Singapore Management University, https://github.com/skyhover/Deckard

[7] Open Logistics Management Information System, OpenLMIS https://github.com/OpenLMIS/open-lmis

[8] Java EE Connector Architecture 1.7 Specification, Ironjacamar https://github.com/ironjacamar/ironjacamar

[9] PSP Emulator, Jpcsp https://github.com/jpcsp/jpcsp

Abstract

Automatic and Safe Refactoring of Function-Level Code Clones

Joonmo Yang
Department of Computer Science & Engineering
College of Engineering
The Graduate School
Seoul National University

This paper proposes a method to automatically find duplicated function-level code clone pairs and securely fix them for JAVA programs. This method consists of two major stages. (i) Find duplicated code pairs using code clone detector within the complete source code. (ii) Classify the located duplicated code pairs and fix them safely through the appropriate algorithm. Stage (i) utilizes the Deckard[2] code clone detector. In stage (ii), duplicated code pairs are divided into two classes. If two functions have same type, they are merged into a single function. If they don't have same type, internal duplicated code snippet is extracted. In order to demonstrate the effectiveness of this method, I designed and implemented function-level code clone refactoring tool, AutoRefactor. I experimented AutoRefactor with real company product codes and open source projects. Experimental results show that AutoRefactor can process 3,976 LOCs (30 clone pairs) out of 6,082 LOCs (55 clone pairs).

Keywords: Code Clone Refactoring, Automated Refactoring, Duplicate Code Merge

Student Number: 2017-20635