Journal of Convergence for Information Technology
Vol. 8. No. 2, pp. 91-103, 2018
ISSN 2586-1816
e-ISSN 2586-4440
DOI : https://doi.org/10.22156/CS4SMB.2018.8.2.091

임베디드 소프트웨어의 에너지 효율성과 언어 변환 지원을 위한 코드 리팩토링 기법 확장

남승우, 홍장의*
충북대학교 컴퓨터과학과

Extension of Code Refactoring Technique to Support Energy Efficiency and Language Conversion of Embedded Software

Seungwoo Nam, Jang-Eui Hong*
Department of Computer Science, Chungbuk National University

요 약 리팩토링은 기존 레거시 코드에 대한 품질을 확보하기 위한 공학적 기술로서, 프로그램의 기능은 변하지 않으면서 내부 구조를 개선하는 작업이다. 다양한 오픈 소스 소프트웨어가 재사용되면서, 기술적 이유 또는 시장 요구로 인하여 프로그래밍 언어 변환을 통한 소스 코드 재사용이 점진적으로 요구되고 있다. 이러한 상황에서 에너지 효율성을 고려하는 언어 변환 리팩토링 기법은 임베디드 소프트웨어 개발의 생산성은 물론 품질 향상을 위한 중요한 수단으로 여겨지고 있다. 본 논문에서는 기존에 제시된 에너지 절감형 리팩토링 기법에 추가하여 프로그래밍 언어의 문법 및 구조의 비교와 매핑을 통해 다른 언어로 변환하는 코드 리팩토링 기법을 제시하였다. 제안하는 리팩토링 기법의 활용은 소프트웨어 개발 언어의 환경 변화에 대처하고, 기존 코드의 재사용을 높임으로써 신속한 소프트웨어 개발 및 품질 향상을 통한 제품의 경쟁력 향상을 기대할 수 있다.

주제어 : 소스 코드 품질, 코드 리팩토링, 리팩토링 기법, 프로그래밍 언어 변환, 프로그래밍 패러다임

Abstract Refactoring is an engineering technique for securing the quality of existing legacy code, improving the internal structure without changing the functionality of the software. Along with the reuse of open source software, reuse of source code through programming language conversion is increasingly required due to technical or market requirements. In this situation, the refactoring technique including language conversion as well as energy efficiency is considered to be an important means for improving the productivity and the quality of embedded software development. This paper proposes a code refactoring technique that converts the grammar and structure of a programming language into those of a different language through comparison and mapping, in addition to the existing energy efficient refactoring technique. The use of the proposed refactoring technique can expect to improve the competitiveness of the product through rapid software development and quality improvement by coping with the environment change of the software development language and enhancing the reuse of the existing code.

Key Words : Source code quality, Code refactoring, Refactoring technique, Programming language conversion, Programming paradigm

*This research was supported by Next-Generation Information Computing Development Program through the NRF of Korea funded by the Ministry of Science, ICT (NRF-2017M3C4A7066479).
*Corresponding Author : Jang-Eui Hong(jehong@chungbuk.ac.kr)
Received April 03, 2018 Revised April 16, 2018
Accepted April 20, 2018 Published April 28, 2018

1. 서론

급격하게 변하는 비즈니스 환경에서 소프트웨어의 유지보수 문제는 비즈니스 운영의 핵심 문제로 자리잡고 있다. 사용자에게 제공하는 서비스의 수명주기는 짧아지고 있으며, 개발자는 사용자에게 제공할 서비스를 개발하기 위해 많은 노력을 하고 있다.

기존 레거시 시스템 기반 서비스 환경에서 새로운 서비스의 개발에 대한 요구의 결합은 기존 시스템에 대한 빈번한 개선과 기능 확장을 필요로 하고 있다[1]. 이러한 개선과 확장은 레거시 시스템의 소스 코드에 대한 지속적인 변경을 필연적으로 유발한다[2,3]. 따라서 레거시 시스템의 높은 소스 코드 품질은 신속하고 정확한 서비스 개발 및 기능을 반영하기 위한 시간 및 금전적 비용을 줄일 수 있게 하는 핵심 요소이다.

기존 레거시 시스템에 대한 소스 코드의 품질 확보를 위한 공학적 방법은 대표적으로 리팩토링 기술이 있다[4]. 리팩토링은 기존 사용자 서비스의 기능에 대한 변화 없이 소스 코드를 재구조화 하는 활동으로서, 소스 코드에 대한 가독성 및 이해성을 높이고, 새로운 변경을 용이하게 수행하기 위한 모듈화 구조의 소스 코드를 생성하는 작업이다[5,6].

과거 리팩토링과 관련된 연구들이 많이 제시되어 왔다. M. Flower는 소스 코드가 갖는 Code Smells을 정의하고, 이러한 Smells을 제거하기 위한 68여개의 소스 코드 리팩토링 기법을 제안하였다[7]. 제안된 기법을 지원하기 위한 다양한 도구들도 개발되어 왔었고, Hong et. al은 특정 시스템 분야에 맞는 소스 코드 품질 향상을 위한 리팩토링 기법 및 리팩토링 지원 도구들을 분석하여[8], 개발에 적용할 수 있도록 도와주는 연구가 진행되어 왔었다.

과거 다양한 리팩토링 기법들과 이 기법들을 지원하는 도구들이 개발되어 왔지만[9-14], 다른 프로그래밍 언어에 맞게 코드를 재구조화 시켜주는 리팩토링 기법은 제안되지 않았다. 오픈 소스 소프트웨어가 재사용 되면서 단순히 인터페이스의 재구조화를 위한 리팩토링이 의미있게 여겨져 왔지만[15], ICT 융합 분야에 대한 소프트웨어 개발이 증대되면서, 이러한 리팩토링이 재사용과 연결되어 프로그래밍 언어의 변환을 함께 요구하고 있다. 따라서 리팩토링 기법에 대하여 개발자들이 겪는 어려움은 다음과 같이 나열할 수 있다.

- 많은 재사용 가능한 코드가 존재하지만, 수정없이 재사용하기가 쉽지 않다.
- 재사용 가능한 오픈 소스를 찾는 경우에도 개발 언어와 다른 언어인 경우가 종종 있다.
- 기존의 언어 변환 기법이 존재하지만, 리팩토링을 통한 언어 변환이 자동화 되어 있지 않다.

이와 같은 문제들을 해결하기 위해 코드상에 Bad smell이 날 것 같은 소스 파일을 불러오면, 리팩토링 기법의 적용과 함께, 확장성을 고려한 다중 언어 변환을 지원해 다른 언어를 사용하는 시스템에서도 재구조화된 소스 코드를 사용할 수 있게 하고자 한다. 이를 통해 사용자는 보다 편리하게 재사용 가능한 소스 코드를 선택할 수 있으며, 또한 재사용의 향상으로 보다 빠르고 효과적으로 대상 시스템을 개발할 수 있는 장점을 제공받게 된다.

본 논문의 구성은 다음과 같다. 2장에서는 기존의 리팩토링 기법 및 프로그래밍 언어 변환에 대해 조사 분석하였으며, 3장에서는 언어 변환 리팩토링을 위한 데이터 모델과 언어 변환 리팩토링 방법을 제시한다. 4장에서는 결론 및 향후 연구에 대해 제시하였다.

2. 관련 연구

2.1 리팩토링 기법 분석

리팩토링에 대한 연구는 1990년 W. Opdyke와 R. Johnson의 연구 논문에서 처음 제시되었다[9]. 그 후 리팩토링 기법에 대한 구체적인 연구 결과가 1991년 W. Griswold의 논문에서 제시되었고, 객체지향 패러다임을 다룬 리팩토링 기법이 1992년 W. Opdyke의 논문에 등장하였다[9,10].

품질 특성을 고려한 리팩토링 기법 분류는 크게 3가지로 정리할 수 있다[8]. 각각의 목적들은 리팩토링을 통해서 얻고자 하는 효과가 궁극적으로 무엇인지에 따라 구분해 놓은 것이며 특정 시스템이 지향하는 품질요소에 따라 고려해야 될 사항이다.

(1) 유지 보수성 : 복잡하게 엉킨 소스 코드를 재구조화 하면서, 코드의 모듈화 특성을 향상시키는데 그 목적이 있다. 즉, 코드에 대한 이해성을 높여서 유

지보수성을 향상시키고자 하는 것이다.

(2) 성능: 소스 코드의 재구조화를 통해 코드에 대한 이해성을 높이는 것과 동시에 성능의 개선을 추구하는 연구들도 진행되었다.

(3) 소모전력: 임베디드 시스템이나 모바일 소프트웨어의 경우, 소모전력 절감의 품질 특성을 요구한다. 따라서 리팩토링을 통하여 소모 전력을 절감하고자 하는 연구들이 최근에 진행되었다.

이와 같은 소프트웨어의 품질 특성을 기반으로 한 기존의 수행된 연구들을 분류하면 Table 1과 같이 정리할 수 있다. Table 1에서 각 제안된 기법들은 하나의 품질 요소를 향상시킬 수 있다는 것을 보여준다. 특정 품질 요소를 목적으로 한 리팩토링 적용일 경우 제안된 해당 기법들을 사용하면 된다는 것을 알 수 있다.

Table 1. Classification of refactoring techniques based on quality factors[8]

Authors	Proposed Techniques	Quality
M. Fowler [7]	• Duplicated Code • Long Method • Large Class • Long Parameter List • Data Clump • Temporary Field • Incomplete Library Class • Message Chains, etc.	Maintainability
J. Garcia [11]	• Removing Global • Sequences & Structures • Parallel Library • Strategy Encapsulation • Avoiding Data Race, etc.	Performance
A. Verto [12]	• Parameter By Value • Self Assign • Mutual Exclusion OR • Switch Redundant Assign • Dead Local Store • Non Short Circuit, etc.	Energy Consumption
J. Hong [13]	• Complex Expressions • Common Sub-expressions • Tail Recursion • Loop Structure • Dead Code	

리팩토링 기법은 특정 패턴에 의해 적용되기 때문에 제시된 다양한 기법들을 활용하여 리팩토링 할 수 있다. M. Fowler는 각 기법의 적용 방법에 대한 패턴을 다음과 같이 4가지로 유형화하였다[7].

(1) Abstraction: 정보의 은닉과 일반화의 개념 적용을 적용한 유형

(2) Breaking: 기존의 코드를 더 작은 논리적 코드 조각으로 분리, 즉 외부 클래스 및 외부 메서드를 생성하는 유형

(3) Moving: 필드 또는 메서드를 다른 Class나 모듈로 위치를 옮겨 객체 간 상호작용으로 생기는 오버헤드 감소하는 유형

(4) Substitution: 타입 체킹 및 조건을 변경하여 유지보수성을 향상하는 유형

품질의 특성과 기법의 패턴에 의한 리팩토링 분류는 목적에 따라 적용할 수 있는 리팩토링 기법과 그 기법의 패턴에 대한 이해를 용이하게 한다.

2.2 언어 변환 연구

Lee의 연구는 절차지향 소프트웨어로부터 클래스와 상속성을 추출하기 위한 방법론을 제안하였는데[16], 객체지향 프로그램의 많은 클래스들에 대한 가능한 경우의 수를 생성하기 위해 새롭고 두드러진 방법론을 기술하였다. 이러한 방법론은 클래스들 간의 추상화와 각 단계별 유사도에 기반을 두고 있는데 첫 번째로 모든 가능한 경우의 클래스 후보군을 생성하고, 두 번째로, 그룹내의 클래스들 간의 유사도를 기반으로 하여, 클래스 후보군의 계층 구조의 상속성을 추출한다. 세 번째로 클래스 후보군의 클래스들의 집합과 영역 모델에서 같은 클래스 계층을 갖는 클래스 집합 사이의 유사도 뿐만 아니라 클래스 후보군과 영역 사이의 그룹 간의 유사도를 측정한다. 마지막으로 최고 또는 최적 클래스 후보군이 그룹과 계층 간의 유사도 비교에 의해 선택된다. 만약 그러한 그룹이 존재하지 않는다면 전체적으로 최적의 후보를 가진 완벽한 그룹을 생성하기 위해 각 그룹으로부터 부분적으로 최고 또는 최적의 클래스 후보들이 결합된다.

이 연구의 가장 중요한 이점중의 하나는 상속성을 가진 최고 또는 최적의 클래스 후보군을 선택하려는 재공학 전문가에게 포괄적이고 통합적인 환경을 제공하는 것이다.

3. 언어변환기반 리팩토링

3.1 언어 변환 고려사항

2.1절에서 품질 특성을 고려한 3가지 리팩토링 기법 분류를 소개하였다. 리팩토링으로부터 얻는 효과는 개발자의 궁극적인 목적에 따라 다르다. 어느 기법을 써서 재

구조화 시켰는 가는 개발팀의 소프트웨어 운영 방침에 따라 달라질 수 있다. 임베디드 시스템인 경우에는 성능, IoT 및 모바일 컴퓨팅에서는 전력 소모량에 초점을 둘 것이다[17]. 리팩토링을 지원하는 소프트웨어의 언어 변환 측면의 이점은 기존 시스템의 코드를 재구조화해 다른 언어를 이용하는 환경에서도 사용할 수 있도록 하는데 있다. 하지만 언어마다 지향점과 문법적 특징이 다르기 때문에 변환할 때 주의해야 한다.

프로그래밍 언어는 지향하는 관점에 따라 분류할 수 있는데 크게 절차지향 프로그래밍 언어와 객체지향 프로그래밍 언어로 분류할 수 있다. 절차지향 프로그래밍은 프로시저 호출의 개념을 기반으로 하는 구조화 프로그래밍에서 파생된 프로그래밍 패러다임으로 대표적인 예로 C언어가 있다[18]. 객체지향 프로그래밍은 객체 개념을 기반으로 하는 프로그래밍 패러다임으로[19], 필드 형태로 데이터를 포함할 수 있고 코드의 프로시저 형태를 메서드라고 한다. 대표적인 예로 C++, Java가 있다.

Table 2는 최근에 소프트웨어 개발에서 어떤 언어가 많이 쓰였는지에 대한 지표를 보여준다[20]. Java, C, C++ 순으로 가장 많이 쓰였기 때문에 이 3가지 프로그래밍 언어를 중심으로 문법적인 특징을 비교하여 언어 변환에 대한 구현 방향을 설명한다.

Table 2. Ranking of most used programming language

Mar 2018	Mar 2017	Language	Ratings
1	1	Java	14.94%
2	2	C	12.76%
3	3	C++	6.45%
4	5	Python	5.87%
5	4	C#	5.07%

3.1.1 C에서 C++로

C에서 C++ 변환에 고려해야 할 사항에 대해서 Lili Qiu는 다음과 같이 정의했다[21].

(1) 구조 변환: C 파일 각각을 C++ 클래스로 매핑할 것인지. 아니면 모든 C파일을 단일 C++ 클래스로 변환할 것인지에 대한 선택을 해야 한다. 전자일 경우에는 함수를 호출할 때마다 어떤 객체에 속하는지 알아야 하는 문제점이 있고, 후자일 경우에는 모든 전역변수와 함수는 구성원의 데이터로 가지고 있어 전역변수명이 겹치는 현상이 나타난다.

(2) 타입 변환: C++에서 암시적 형식 변환은 더 제한적이다. 즉, C에서의 암시적 타입 변환을 사전에 탐지하고 막아야 한다. 이 과정에서 모든 할당에는 유형 검사가 필요하다.

(3) 인수 형식: ANSI C를 만족한 상태에서 C++로 변환해야 한다. ANSI C가 아닌 경우 함수 파라미터의 공백은 임의의 수의 인수를 취할 수 있음을 나타내는 것이고 ANSI C인 경우에는 함수가 인수를 사용하지 않는다는 것을 나타낸다.

(4) 정적 변수: C에서 변수 앞에 static은 다른 파일에서 사용할 수 없게 한다. C++에서 static은 오직 복사본을 하나만 생성한다는 뜻을 가지고 있기 때문에 C에서 C++로 변환할 때 static의 의미를 혼동하면 안 된다.

(5) 구조체 멤버: C에서 struct는 필드만 멤버로 들어갈 수 있지만 C++에서 struct는 함수도 멤버로 들어갈 수 있기 때문에 C에서 C++로 변환할 때 C++에서 사용하는 struct 의미가 되지 않도록 주의해야 한다.

위의 (1)번에서 두 가지 방법 중 어떤 방법으로 변환할 것인지 결정하는 것에 따라 static 변수와 non-static 변수의 이름 충돌 문제들을 중점적으로 고려해야할 수 있다. Lili Qiu는 모든 C 파일을 단일 C++ 클래스로 변환한다는 전제하에 구현 방법을 제시하고 있다. 이러한 단일 C++ 클래스로 변환하는 것은 절차지향 프로그래밍 패러다임을 그대로 가져가기 때문에 객체지향 관점으로 리팩토링 할 수 없게 된다. 예를 들어, Large Class, Incomplete Library Class 등에 제안된 기법을 사용할 수 없다. 본 논문에서는 C 파일 각각을 C++ 클래스로 매핑 변환하여 리팩토링을 수행할 수 있도록 제시한다.

3.1.2 C++에서 Java로

C++에서 Java로의 변환은 다음과 같은 고려사항이 있다[21].

(1) 동적 할당: C++는 명시적으로 Deallocation을 해야 하지만 Java에서는 가비지 콜렉터가 자동적으로 메모리 반납 문제를 해결한다.

(2) 다중 상속: C++의 Abstract Class는 다중 상속을 허용하지만 Java의 Abstract Class는 다중 상속을 허용하지 않기 때문에 C++의 Abstract Class를 Java 언어로 변환할 때에는 interface로 변환해야 한다.

(3) 인수 전달: C++에서 primitive data를 제외한 사용자 정의 객체를 파라미터 상으로 전달하는 것은 Call by value에 해당되지만 Java에서는 Call by reference에 해당된다.

(4) 변수 초기화: C++에서 초기화 되지 않은 변수는 garbage value를 가지지면 Java의 모든 변수는 초기화 전에 default value를 가진다.

(5) 함수 프로토타입: C++는 함수 프로토타입에 throw 절이 없지만 Java에는 존재한다.

(6) 가상 함수: C++은 모든 가상 함수가 명시적으로 정의해야 하지만 Java에서는 모든 가상함수가 명시적으로 정의되지 않아도 된다.

(7) C++에서는 패키지에 대한 개념이 없기 때문에 #include는 정확히 동일하지는 않지만 대용품으로 사용될 수 있다.

(8) 라이브러리 : 모든 Java 내장 클래스에 해당하는 C++ 버전이 포함된 새로운 C++ 라이브러리를 생성해야 한다.

Java의 패키지, 가비지 콜렉터의 개념과 여러 문법적인 차이 등을 고려해주면 C에서 Java로 변환이 가능하지만 C에서 바로 Java로 언어 변환을 하는 것이 C에서 C++, C++에서 Java로 언어 변환을 하는 것보다 더 많은 문제를 해결해야 한다. C에서 Java로 변환할 때 중간에 C++로 변환한 후 Java로 변환하면 절차상 까다롭지만 언어 변환간 발생할 수 있는 오류의 발생 위험을 낮출 수 있다.

3.2 리팩토링 도구 분석

본 논문에서는 연구팀이 그 동안 수행하여 왔던 저전력을 지원하는 리팩토링 기법 도구를 기반으로 언어 변환을 지원하는 리팩토링 기법을 고려하였다[8,22].

Fig. 1은 현재 개발되어진 리팩토링 도구에 언어 변환 기능을 추가한 화면이다. Fig. 1의 하단의 메뉴에서 버튼 Energy 는 Original 소스 코드를 에너지 절감이 가능한 코드로 리팩토링하는 기능을 제공한다. 이 기능은 언어의 변환 없이 코드 구조에 대한 변환 기능을 제공한다. 또한 버튼 Halstead 는 원시 소스코드와 리팩토링 소스코드의 헬스테드 복잡도를 보여주는 팝업을 생성하는 기능이다. 리팩토링의 결과에 대하여 코드 복잡도를 보여줌으로써, 보다 효과적인 리팩토링이 이루어졌는지를 확인할 수 있다. 그리고 맨 좌측의 버튼 Conversion 이 언어 변환을 지원하는 기능이다.

Fig. 1과 Fig. 2는 언어 변환 지원 리팩토링을 수행할 툴의 구성과 실행 결과를 보여준다.

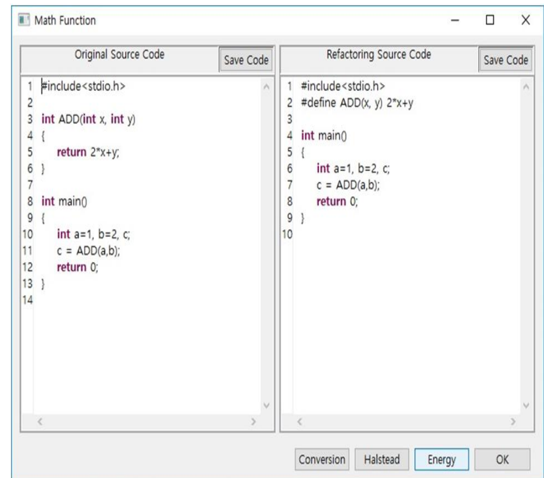

Fig. 1. Screen of newly added function Conversion

언어 변환을 지원하는 기능에 의해 생성되는 변환 코드의 예는 Fig. 2와 같다. Fig. 2는 C 언어 코드를 C++ 언어로 변환한 예를 보여준다. 다양한 언어 변환이 가능할 수 있겠지만, 핵심적으로 요구될 수 있는 C언어를 C++ 언어로 변환하는 기능을 구현하였다. Fig. 1의 버튼 Conversion에 의해 팝업창이 생성되도록 개발하였다.

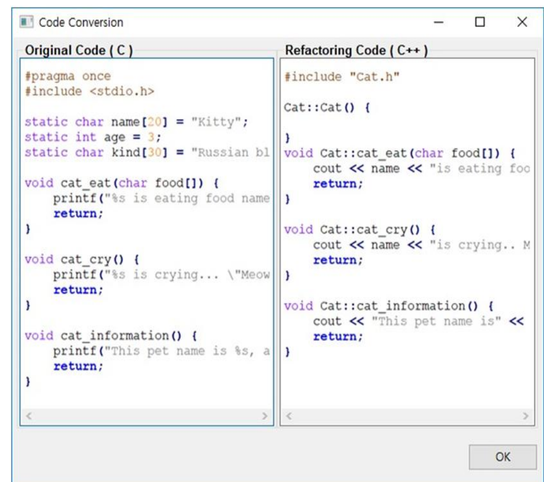

Fig. 2. An example of language converted refactoring

3.3 언어변환 지원기법

3.3.1 언어변환을 위한 오브젝트 정의

언어 변환 리팩토링을 수행하기 위해 고려해야 할 오브젝트들은 다음과 같다.

(1) Source Code: 리팩토링 대상이 되는 파일의 소스 코드에 해당되는 오브젝트이다. 프로그래밍 언어에 따라 다른 확장자를 가지는 file 형식으로 존재한다.

(2) General features: 소스 코드가 가지고 있는 프로그래밍 패러다임 구조의 특징을 나타낸 오브젝트이다. 한 쪽으로 언어 변환을 할 때 고려해야할 특징들을 자세하게 기술된 XML file 형식으로 존재한다. 예를 들어, 구조적 프로그래밍에서 객체지향 프로그래밍으로 리팩토링을 수행하여야 할 경우에 그에 맞는 특징들이 기술되어 있다.

(3) General grammar: 같은 프로그래밍 패러다임 언어의 문법을 공용으로 사용하기 위해 만든 중간 문법으로 XML file 형식으로 존재한다.

(4) Process index: 시스템이 언어 변환 리팩토링을 수행하는데 필요한 참조 정보를 속성으로 가지고 있는 오브젝트이다. 이 오브젝트가 가지고 있는 참조 정보들은 다음과 같다.

(a) Paradigm index: 소스 언어 또는 타겟 언어가 프로그래밍 패러다임 변환 지표의 어느 위치에 있는지를 나타낸다. Table 3은 절차지향 프로그래밍에서 객체지향 프로그래밍 사이의 프로그래밍 패러다임 변환 지표이다. +값이 클수록 객체지향에 가깝고 -값이 클수록 절차지향에 가깝다. 두 패러다임의 차이를 나타내는 Paradigm gap value는 Target paradigm index와 Base paradigm index의 차로 구할 수 있다.

Table 3. Paradigm index between procedural-oriented & object-oriented languages

Language	C	C++	Java, C#
Class Use(+1)	No	Yes	Yes
Pointer Use(-1)	Yes	Yes	No
Total	-1	0	+1

(b) Progress index: 시스템의 현 상태에서 얼마나 더 언어 변환 리팩토링을 수행해야 되는지 알 수 있다. 예를 들어 Progress index가 -1이고, Target이 되는 index가 +1이면 객체지향 패러다임을 가지는 구조로 두 번 수행한다.

(3) Target language: 언어 변환 리팩토링을 수행 후 최종적으로 어떤 언어 문법에 맞출 것인지 선택하는 데 필요하다.

언어 변환 지원 리팩토링을 수행할 때 문법뿐만 아니라 프로그래밍 패러다임에 맞는 구조로도 변환을 해야 한다. 문법은 프로그래밍 패러다임이 같은 언어일 경우, 공용으로 사용될 수 있는 중간 문법을 두어 변환할 수 있다. 중간 문법을 만들기 위해 소스 코드 오브젝트를 Fig. 3과 같이 하위 2개의 오브젝트를 가지도록 구성하였다.

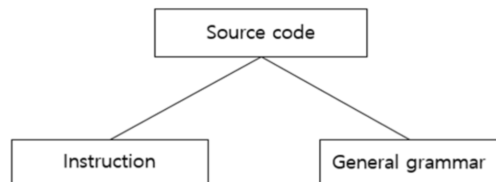

Fig. 3. Base object & sub object structure for grammar

Fig. 3에서 Instruction 오브젝트는 코드 라인별로 해당 명령어들을 어셈블리 코드 형태로 가지고 있으며, General grammar 오브젝트는 중간 문법으로, 같은 프로그래밍 패러다임 언어들이 공용으로 사용될 수 있는 언어의 문법들이 기술되어 있다.

3.3.2 언어변환 모델의 오브젝트 매핑

기존의 소스 코드 오브젝트를 Instruction, General grammar와 같은 하위 오브젝트로 의미를 매핑하려고 할 때 스키마의 구조를 바꿔 주어야 한다. 언어 변환 리팩토링을 수행하기 위한 모델의 스키마 구조 변경은 Philip이 정리한 방법을 이용하였다[23]. 기존 모델에 대한 스키마 변경 그리고 모델 오브젝트 간의 매핑은 Fig. 4와 같다.

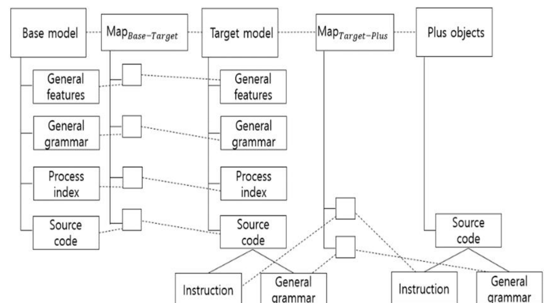

Fig. 4. Objects mapping in conversion model

Fig. 4의 Base model은 언어 변환 리팩토링을 수행하기 위한 기본 모델로 General features, General grammar, Process index, Source code 4개의 오브젝트를 가지고 있다.

Plus objects 모델은 Source code에서 부문별 리팩토링 수행과 중간 문법 사용을 위해 새롭게 추가한 2개의 하위 오브젝트를 가지고 있다. Target model 에서는 이렇게 추가된 2개의 하위 오브젝트를 반영하기 위해 Source code에 매핑 시킨 것을 보여준다. Fig. 5는 기존 소스 파일 정보를 담고 있는 모델의 스키마를 변경하는 과정을 보여준다.

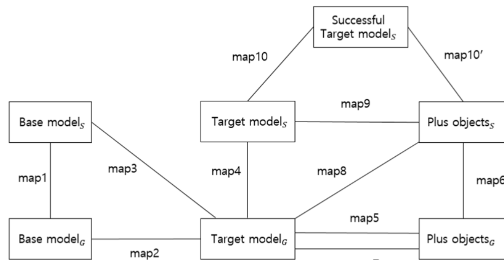

Fig. 5. Process of schema change & mapping

Fig. 5에 나타난 각 매핑 단계에 대한 설명은 다음과 같다.

(1) map1 - 기존 데이터 모델의 일반화 : 기본이 되는 Base model의 스키마를 수정하기 위해 메타 모델(Base model$_G$)로 일반화 시키는 작업이다.

(2) map2 - 일반화된 타겟 모델(Target model$_G$)과의 매치 : 오브젝트의 수정이 이루어진 모델과 비교하여 Base model에서 매핑할 수 있는 오브젝트들만 연결한다. 여기서 Target model은 스키마 구조는 완성이 되었지만 매핑이 완성되지 않은 불완전한 상태를 가진 모델이다.

(3) map3 - 기존 베이스 모델과 Target model$_G$와의 매핑 : 둘의 매핑에는 map1에 있는 각 오브젝트의 사본이 포함되며, 이 오브젝트의 모든 Base model 오브젝트는 여전히 Target model$_G$에 있다.

(4) map4-Target model$_G$의 구체화 : 기존 모델 구조에 수정된 스키마를 덮어 쓰지 않도록 Deep Copy를 수행하고 이것을 Specialization 한다.

(5) map5-Target model$_G$과 새롭게 추가된 스키마들만 오브젝트로 가지고 있는 Plus objects 모델과 매치한다.

(6) map6-일반화된 Plus objects 모델(Plus objects$_G$)을 타켓 모델의 특성을 포함하도록 구체화한다.

(7) map7- Plus objects$_G$와 Target model$_G$의 매치 : 새롭게 추가된 하위 오브젝트 2개를 각각 매핑 시켜주는 작업이다.

(8) map8-Plus objects$_S$와 Target model$_G$을 매핑한다.

(9) map9-Plus objects$_S$와 Target model$_S$을 매핑한다.

map 10, map10' 는 Target model$_S$, Plus objects$_S$를 통합한 후, 매핑 시켜 모든 오브젝트가 매핑 된 구체화된 Successful Target model을 만드는 작업이다. 해당 단계가 끝난 후 구체화된 Successful Target model에서 일반화된 타겟 모델로 한 번에 매핑 시켜줄 수 있는 map11을 Fig. 5과 같이 만들면 작업은 종료된다.

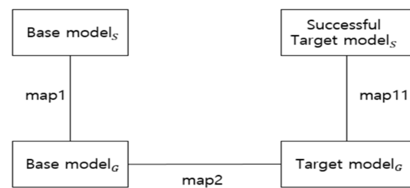

Fig. 6. Final mapping step for schema change

4. 언어변환 리팩토링 절차

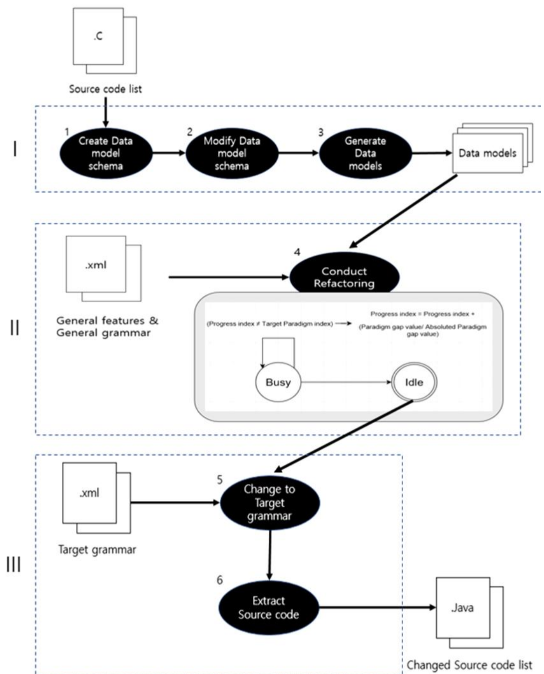

Fig. 7. Procedure for language-converted refactoring

3장에서 설명한 기법을 근간으로 새롭게 변경된 스키마 구조를 가지는 모델을 이용해 언어 변환 리팩토링을 수행하는 과정은 Fig. 7과 같다.

- 단계 I: Model schema change & Use changed model

(1) Create data model schema : 소스 코드 리스트와 Target language를 입력 받으면 소스 코드 및 언어 변환 리팩토링 진행 상태변수 등을 담고 있는 데이터 모델을 생성한다.

(2) Modify data model schema: 중간 문법을 이용하기 위해 데이터 모델의 스키마를 수정한다.

(3) Generate data models: 언어 변환 리팩토링을 수행하기 위한 데이터 모델을 인스턴스화 한다.

- 단계 II: Convert programming paradigm & conduct refactoring

(4) Conduct refactoring: 언어 변환 리팩토링은 독립된 상태의 집합으로 나타낼 수 있다. 각 상태에서 Progress index에 맞는 General features와 General grammar XML 파일을 불러온다. General features와 General grammar에 맞도록 구조와 문법을 변경한 후, 소스 코드에서 변경사항을 반영하고 다음 상태로 넘어간다.

(5) 다음 상태에서 Progress index 값이 Target Paradigm index와 같아지면 리팩토링을 종료한다.

- 단계 III: Change to Target grammar & Extract Source code

(6) Change to Target grammar: Target language에 따른 문법으로 변환한다(General grammar→Target grammar).

(7) Extract Source code: 데이터 모델에서 언어 변환 리팩토링이 반영된 소스 코드 리스트를 반환한다.

5. 사례 연구

3장에서 설명한 기법을 근간으로 실제 임베디드 소프트웨어 코드(IoT 기반의 대기 오염측정 시스템)를 가지고 사례 연구를 진행하였다. 먼저, 언어변환 및 리팩토링을 수행할 프로젝트 폴더의 코드파일 속성을 보고 Paradigm Index를 확인한다. 사례 연구에 사용한 실제 소스 코드는 C언어로 구성되어 있기 때문에 Paradigm Index 값이 -1이다. 목표로 하는 언어가 C++ 언어이기 때문에 Paradigm Index를 0으로 선정하고, 임베디드 소프트웨어에서 중요한 품질 요소인 성능을 위해 리팩토링의 목적을 성능 및 소모 전력으로 결정한다. 언어 변환 및 리팩토링을 위한 데이터 모델 스키마를 생성하기 전에 C파일의 구조변환 타입을 결정해야 한다. 객체지향 SW로 재공학하려는 목적이 특정 응용 영역에 맞게 SW 요소들을 싸는(Wrap) 것이 아닌 객체 모듈(Module) 형태로 재구성하려는 것일 때, 소스 파일 각각이 모듈화된 형태로 분류가 되어있으면 원시 파일인 C파일 각각을 C++클래스로 매핑하는 것이 좋다. 모든 통합된 C파일에서 C++클래스 및 상속성 추출 및 영역 전문가의 개입을 필요로 하지 않기 때문에 절차 상 간단하다. Fig. 8은 임베디드 소프트웨어의 모듈화된 소스 파일들을 보여준다. 크게 메인보드 제어, 하드웨어 모듈 제어, 통신으로 나눠고 세부적으로 해당 기능 및 역할에 따라 헤더파일과 소스 파일이 생성되어 있다.

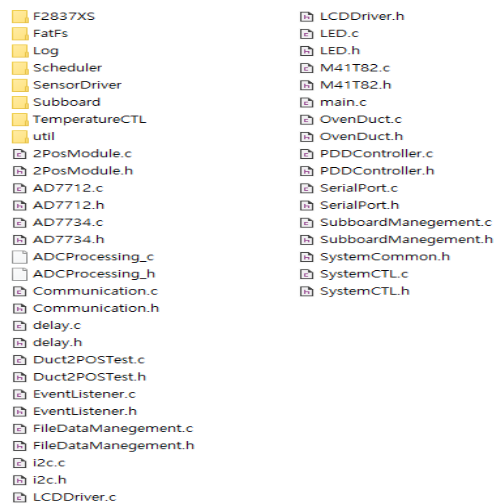

Fig. 8. Source files in embedded software project

많은 양의 소스 파일들에 대한 언어변환 및 리팩토링 결과를 모두 보여줄 수 없기 때문에 사례 연구에서는 main.c 파일을 기준으로 밀접한 관계를 가지는 소스 파일들의 범위를 언어변환 및 리팩토링의 대상으로 잡았다. 언어변환 및 리팩토링을 수행하기 위해서는 각 Paradigm Index마다 중간 문법을 선정해야 한다. 사례 연구에서는 Paradigm Index -1에 해당하는 중간 문법

은 ANSI C, Paradigm Index 0에 해당하는 중간 문법은 C++15로 선정했다. Base $model_S$과 Base $model_G$모델을 동일하게 만들기 위해 원시 소스 파일을 ANSI C 문법으로 작성하여 준비했고, Target $model_S$을 C++15로 선택하여 Target $model_S$과 Target $model_G$을 동일하게 하였다. 자연스럽게 Target model에 매핑을 위한 Plus $objects_S$ Plus $objects_G$도 동일해진다. 이러한 조건 하에서 언어변환 및 리팩토링을 위한 매핑 프로세스는 map2, map7, map10, map10', map11 순의 절차로 간소화된다.

map2는 프로그래밍 패러다임을 변화시키기 때문에 소스코드의 Instruction 형태 및 grammar를 많이 변형시킨다. 기존 C파일들을 C++클래스로 만드는 과정 및 클래스 간의 관계 설정 과정이 포함되어 있다. 사례 연구에서는 main.c 소스 파일에서 #include한 헤더파일간의 관계 설정, 헤더파일의 클래스화, main.c 소스 파일의 클래스화를 map2 과정에 포함시켰다. Fig. 9는 main.c 소스 파일에서 #include한 헤더파일의 목록들을 보여준다.

```
#include <stdio.h>
#include <string.h>
#include "SerialPort.h"
#include "LCDDriver.h"
#include "SystemCommon.h"
#include "communication.h"
:
#include "AD7734.h"
#include "./TemperatureCTL/TempOperation.h"
#include "OvenDuct.h"
#include "2PosModule.h"
#include "./TemperatureCTL/HeaterCTL.h"
:
#include "SubboardManegement.h"
#include "./SensorDriver/smba1x.h"
:
#include "LED.h"
#include "PDDController.h"
#include "Duct2POSTest.h"
#include "Subboard/ExtractorCTL.h"
#include "Subboard/MFCCTL.h"
#include "SystemCTL.h"
#include "EventListener.h"
:
#include "i2c.h"
#include "M41T82.h"
#include "Scheduler/schedule.h"
```

Fig. 9. Included header file in main.c file

헤더파일들을 하나씩 살펴본 결과 SystemCommon.h를 제외한 모든 헤더파일에서 SystemCommon.h 헤더파일을 #include 했다. 모든 다른 헤더파일과 관계를 가지는 SystemCommon.h 헤더파일은 구현파일(.c)이 존재하지 않고 #define 및 구조체, 전역변수만을 가지고 있어서 기존 항목들이 모두 클래스의 멤버 변수로 모두 변환된다는 것을 알 수 있다. map7에서 리팩토링 목적에 따라 수행될 수 있는 리팩토링 기법을 Plus objects를 통해 확인하고 Target model에 매핑하여 적용한다. 여기까지가 Fig. 7의 2단계의 4번째 액티비티인 언어변환 및 리팩토링 수행하기 위한 단계이다.

첫 번째 단계는 C파일을 C++클래스로 만드는 것이다. SystemCommon 클래스를 만드는 과정은 다음과 같다. #define이 선언되어 있으면 가지고 있는 구조체에서 #define으로 선언된 상수를 사용하고 있는지 확인해야 한다. #define으로 선언된 상수는 const static int로 대체할 수 있지만 전처리가 일반 값이 아닌 키워드 혹은 함수의 단위를 나타내는 것이면 대체할 수 없다. #define을 static const로 대체했을 때는 접근 지정자를 지정하는 것부터 이들을 수정 혹은 반환하는 멤버함수를 생성해야 하는지에 대한 문제점이 생겨 성능 및 소모 전력을 위한 리팩토링 목적과 어긋나는 방향으로 이끌 수 있다. 따라서 코드의 일부로 생성되지 않는 전처리기는 그대로 멤버변수로 두었다.

두 번째 단계는 구조체와 해당 클래스간의 관계를 파악하는 것이다. SystemCommon.h 헤더파일은 두 개의 구조체를 가지고 있는데 이들은 전처리 구문인 #define BOOL int를 멤버변수들의 자료형으로 사용하고 있다. SystemCommon 클래스의 전처리 구문을 사용하려면 두 개의 구조체는 SystemCommon 클래스의 nested class로 만들어져야 한다. nested class로 만드는 또 다른 이유는 에너지 효율을 발생시키는 'Change Reference to Value' 리팩토링 기법의 사용 때문이다. 관리가 애매한 클래스를 따로 만들어 호출하는 것보다 nested class로 가지고 있는 것이 에너지 효율 면에서 좋다. 리팩토링을 고려한 언어변환 및 클래스화의 결과는 Fig. 10과 같다.

마지막 단계는 다른 클래스와의 의미적 관계식별을 고려한 언어변환 및 리팩토링이다. 기존 SystemCommon.h 헤더파일은 자료형이 SystemStatus 인 extern 전역변수를 가지고 있고 이 전역변수는 main.c 에서 초기화 한다. 즉, SystemStatus 클래스는 main.c 에서만 값을 변경할 수 있다. C++클래스의 접근지정자가 private, public, protected 3가지가 있는데 이들 중 위의 조건을 만족하게 하는 접근지정자는 protected 이므로, 결국에는 SystemStatus 클래스와 main.c 에서 파생될 Main 클래스와의 관계는 상속이라는 것을 알 수 있다. 하지만 SystemStatus 클래스는 SystemCommon 클래스의 nested class이기 때문에 Main 클래스에서 상속을 할 수 없다. 최선책으로써,

Main 클래스는 SystemCommon 클래스를 상속하고 SystemCommon 에서 Main 객체에서만 변경을 가능하게 하는 protected 접근지정자를 가진 멤버함수를 제공한다. 이 과정에서 SystemStatus 클래스는 SystemCommon 에게 private 접근을 허용하는 friend 키워드를 사용한다. 사례 연구에 추가 적용된 에너지 효율성 및 성능을 위한 리팩토링은 'Inline Method', 'Inline Temp', 'Encapsulate Field', 'Encapsulate Collection' 이다. 'Inline Method'는 메소드가 하는 일이 단순하다면 메소드를 호출하는 것보다 리턴에 메소드 코드부분을 그대로 쓰는 것이고, 'Inline Temp'는 Temp 변수가 일회용이라면 객체 함수를 호출을 Temp 변수 대신에 사용하는 것이다. 나머지 두 리팩토링은 추가적인 멤버함수를 생성해 코드 라인을 길어지게 하지만 에너지 효율 면에서 좋은 리팩토링 방법이다.

```
class SystemCommon
{
    // primary defines
#define VERSION_INFO                         40
#define EOT                                  0x04
#define BOOL                                 int
#define FLADDR                               unsigned long
#define FALSE                                0
#define TRUE                                 1
#define MAIN_MODE_STANDBY                    0
    // main board defines
    /* ... */
    // operation data defines
    /* ... */
    // system state defines
    /* ... */
private:
    SystemCommon() : SystemStatus(), ControlModuleData() {};

public:
    // nested class 1
    class SystemStatus
    {
    public:
        BOOL iCurrentOperationMode;
        long iCurrentPressures[2];
        // variables
        /* ... */

    } SystemStatus;

    // nested class 2
    class ControlModuleData
    {
    public:
        BOOL bWaitModuleEnable;
        long iWaitModuleEnableCounter;
        // variables
        /* ... */
    } ControlModuleData;
};
```

Fig. 10. An intermediate process result code

'Encapsulate Field'는 public 멤버변수에 대한 접근을 private 멤버변수와 이를 get(), set() 하는 멤버함수를 public으로 두어 사용하는 것이다. 'Encapsulate Collection'은 배열 또는 리스트단위의 자료구조를 파라미터로 호출해서 get(), set() 하던 것을 읽는 것은 읽기 전용 키워드인 const를 붙여서 전체를 반환해 읽고 추가 및 제거하는 것은 크기 하나의 자료형 단위의 파라미터를 함수 호출에 사용하는 것이다.

Fig. 11-(a)과 Fig. 11-(b)는 최종적인 SystemCommon 클래스의 코드를 나타낸다. SystemCommon 클래스가 많은 클래스에서 참조되는 것을 Fig. 9에서 알 수 있었다. 즉, 데이터 값이 수정될 때 같은 객체에 반영 되어야 한다는 것을 의미하기 때문에 Singleton 패턴을 적용하였다.

클래스의 모든 멤버변수를 private로 지정했고 간단한 일을 하는 멤버 함수 앞에는 inline 키워드를 붙였다. SystemStatus 객체의 멤버변수에 대한 수정은 다른 객체에서 할 수 없도록 get 역할이 아닌 멤버함수는 private 접근지정자를 붙였다. 각 멤버변수에 대한 get(), set() 함수가 만들어진 것을 알 수 있으며 75번째 라인 long iCurrentPressures에 대한 리팩토링인 'Encapsulate Collection' 기법을 적용한 것을 119 ~138번째 라인에서 확인할 수 있다[2].

Fig. 12는 SystemCommon 클래스를 상속한 Main 클래스를 정의한 코드를 나타낸다. 226번째 라인에서 알 수 있듯이 SystemCommon 객체를 참조하기 위해서는 해당 객체에서 제공하는 static 함수를 이용하여야 하며 다른 모든 객체와 공통의 SystemCommon 을 사용하기 위함이다. Main 클래스와 SystemCommon 을 제외한 다른 모든 클래스들은 SystemCommon 을 참조 사용하는 의미적 관계만 존재한다. Fig. 13은 클래스간의 관계를 보여주는 클래스다이어그램을 나타낸다.

본 논문의 사례 연구는 C에서 C++로의 언어변환과 성능 및 에너지 효율성을 목적으로 하는 리팩토링의 수행결과를 보였다.

Table 4는 사례 연구에서 사용한 리팩토링으로 인한 에너지 효율의 효과를 보여준다[24]. 리팩토링 할 코드 구조가 많으면 많을수록 에너지 효율은 더욱 증대될 것이다. M. Fowler가 제시하는 다른 리팩토링 기법들은 주로 객체지향 프로그래밍에서 수행될 수 있는 것으로 원시 소스의 Paradigm Index가 0 또는 1인 경우에 사용할 수 있다.

```
6   class SystemCommon
7   {
8       // primary defines
9   #define VERSION_INFO                                    40
10  #define EOT                                             0x04
11  #define BOOL                                            int
12  #define FLADDR                                          unsigned long
13  #define FALSE                                           0
14  #define TRUE                                            1
15  #define MAIN_MODE_STANDBY                               0
16  // main board defines
17  /* ... */
42  // operation data defines
43  /* ... */
55  // system state defines
56  /* ... */
65  private:
66      SystemCommon() : SystemStatus(), ControlModuleData() {};
67
68  public:
69      // nested class 1
70      class SystemStatus
71      {
72          friend class SystemCommon;
73      private:
74          BOOL iCurrentOperationMode;
75          long iCurrentPressures[2];
76          // variables
77          /* ... */
114         inline void setICurrentOperationMode(BOOL _iCurrentOperationMode)
115         {
116             iCurrentOperationMode = _iCurrentOperationMode;
117             return;
118         }
119         inline void addICurrentPressure(long iCurrentPressure)
120         {
121             int index;
122             if (index = std::find(std::begin(iCurrentPressures), std::end(iCurrentPressures)
123                 , iCurrentPressure) == std::end(iCurrentPressures))
124             {
125                 iCurrentPressures[index] = iCurrentPressure;
126             }
127             return;
128         }
129         inline void removeICurrentPressure(long iCurrentPressure)
130         {
131             int index;
132             if (index = std::find(std::begin(iCurrentPressures), std::end(iCurrentPressures)
133                 , iCurrentPressure) == std::end(iCurrentPressures))
134             {
135                 iCurrentPressures[index] = 0;
136             }
137             return;
138         }
139     public:
140         inline BOOL getICurrentOperationMode()
141         {
142             return iCurrentOperationMode;
143         }
144         inline const long* getICurrentPressures()
145         {
146             return iCurrentPressures;
147         }
148
149     } SystemStatus;
150
```

Fig. 11-(a). Final SystemCommon class part 1

```
151     // nested class 2
152     class ControlModuleData
153     {
154         friend class SystemCommon;
155     private:
156         BOOL bWaitModuleEnable;
157         long iWaitModuleEnableCounter;
158         // variables
159         /* ... */
183     public:
184         inline void setBWaitModuleEnable(BOOL _bWaitModuleEnable)
185         {
186             bWaitModuleEnable = _bWaitModuleEnable;
187         }
188         inline BOOL getBWaitModuleEnable()
189         {
190             return bWaitModuleEnable;
191         }
192         inline void setIWaitModuleEnableCounter(long _iWaitModuleEnableCounter)
193         {
194             iWaitModuleEnableCounter = _iWaitModuleEnableCounter;
195         }
196         inline long getIWaitModuleEnableCounter()
197         {
198             return iWaitModuleEnableCounter;
199         }
200     } ControlModuleData;
201
202     inline static SystemCommon& getInstance()
203     {
204         static SystemCommon instance;
205         return instance;
206     }
207 protected:
208     inline void setICurrentOperationMode(BOOL _iCurrentOperationMode)
209     {
210         SystemStatus.setICurrentOperationMode(_iCurrentOperationMode);
211     }
212     inline void addICurrentPressure(long iCurrentPressure)
213     {
214         SystemStatus.addICurrentPressure(iCurrentPressure);
215     }
216     inline void removeICurrentPressure(long iCurrentPressure)
217     {
218         SystemStatus.removeICurrentPressure(iCurrentPressure);
219     }
220 };
```

Fig. 11-(b). Final SystemCommon class part 2

```
224 static class Main : public SystemCommon{
225 private:
226     SystemCommon systemCommon = SystemCommon::getInstance();
227     // object variables
228     /* ... */
229
230 public:
231     void initializeSystemParameters()
232     {
233         int iCount = 0;
234         setICurrentOperationMode(MAIN_MODE_STANDBY);
235         for (iCount = 0; iCount < 2; iCount++) {
236             addICurrentPressure(-50000);
237         }
238         // SystemCommon variables initialization
239         /* ... */
240     }
241     void LCD_Display()
242     {
243         // LCD control
244     }
245     void initializedSystem()
246     {
247         // initialize system
248     }
249     void initializeTimer0()
250     {
251         // initialize timer
252     }
253     BOOL checkGCMainStatusChange()
254     {
255         // check status change
256     }
257 };
```

Fig. 12. Final Main class

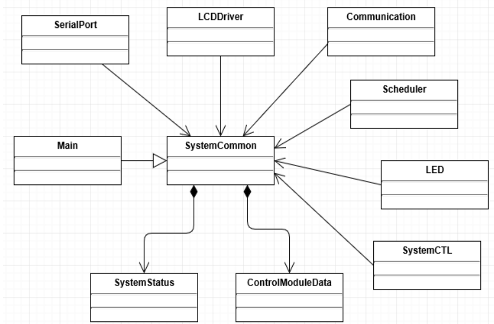

Fig. 13. A class diagram for SystemCommon

Table 4. Energy Efficiency for Code Refactoring Techniques

Refactoring Techniques	Consumption(μJ)		Effect Rate
	Origin	Refactor	
Inline Method	44.059	43.980	0.079
Inline Temp	43.599	43.394	0.205
Encapsulate Field	53.970	53.128	0.842
Encapsulate Collection	90.200	84.104	6.096
Change Reference to Value	55.372	53.685	1.687

6. 결론 및 향후 연구

리팩토링은 기존 코드의 품질을 향상시키는 공학적 기법이다. 비즈니스의 변화로 인해 기존 코드를 리팩토링으로 유지보수성을 좋게 하는 것뿐만 아니라 시장 요구로 프로그래밍 언어 변환이 함께 요구될 수 있다. 본 논문에서는 프로그래밍 패러다임을 고려한 언어 변환 리팩토링 기법을 제시하여 기존 리팩토링 연구에 대해 확장된 기능을 추가하였다. 또, 언어 변환 리팩토링의 융통성을 보장하기 위해 프로그래밍 패러다임별 구조의 특징 및 중간 문법을 모델 기반으로 제시하였다.

향후의 연구는 언어 변환 리팩토링과 관련한 패턴을 개발하여 빠르고 정확한 변환을 가능하게 하는 것이다. 오픈 소스로 제공되어 있는 소스코드들을 이용한 학습으로 구조적 틀을 완성하여 완성도 높은 자동 언어 변환 리팩토링을 제공할 수 있다.

REFERENCES

[1] A. Deursen1, P. Klint & C. Verhoef. (1999). Research Issues in the Renovation of Legacy Systems. *Proceedings of International Conference on Fundamental Approaches to Software Engineering*, 1-21.
DOI : 10.1007/978-3-540-49020-3_1

[2] A. Jatain & D. Gaur. (2015). Reengineering Techniques for Object Oriented Legacy Systems. *International Journal of Software Engineering & Its Applications,* 9(1), 35-52.
DOI : 10.1145/260303.260313

[3] G. Nascimento & C. Iochpe. (2009). A Method for Rewriting Legacy Systems using Business Process Management Technology. *Proceedings of the 11th International Conference on Enterprise Information Systems,* 1-6.

[4] P. Pirkelbauer, D. Dechev & B. Stroustrup. (2010). Source Code Rejuvenation is not Refactoring. *Proceedings of the 36th Conference on Current Trends in Theory & Practice of Computer Science(LNCS 5901),* 639-650.
DOI : 10.1007/978-3-642-11266-9_53

[5] E. Murphy-Hill, C. Parnin & A. P. Black. (2009). How We Refactor & How We Know it. *Proceedings of the 31st International Conference on Software Engineering(ICSE'09)*, 287-297.
DOI : 10.1109/icse.2009.5070529

[6] M. Kim, T. Zimmermann & N. Nagappan. (2012). A Field Study of Refactoring Challenges & Benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering(FSE'12),* 50, 1-11.
DOI : 10.1145/2393596.2393655

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke & D. Roberts. (2002). *Refactoring: Improving the Design of Existing Code.* San Francisco : Addison-Wesley.
DOI : 10.1007/3-540-45672-4_31

[8] D. Kim, Y. Jung, & J. Hong. (2016). Analysis of Refactoring Techniques & Tools for Source Code Quality Improvement. *Journal of Convergence for Information Technology, 6(4),* 137-150.
DOI : 10.22156/cs4smb.2016.6.4.137

[9] W. G. Griswold. (1991). Program Restructuring as an Aid to Software Maintenance. *Doctoral Dissertation, University of Washington, Seattle.*

[10] W. F. Opdyke. (1992). *Refactoring Object- Oriented Framework.* Doctoral Dissertation. University of Illinois at Urbana-Champaign, USA.

[11] J. D. Garcia & B. Stroustrup. (2016). Improving performance & maintainability through refactoring in C++. *Proceedings of the ACCU Conference*, 1-20.

[12] A. Vetro, L. Ardito, G. Procaccianti & M. Morisio. (2013). Definition, Implementation & Validation of Energy Code Smells: an exploratory study on an embedded system. *Proceedings of the Third International Conference on Smart Grids, Green Communications & IT Energy-aware Technologies*, 34-39.

[13] J. Lee, D. Kim, & J. Hong. (2016). Code Refactoring Techniques based on Energy Bad Smells for Reducing Energy Consumption. *Journal of KIPS transactions on software & data engineering, 5(5),* 209-220.
DOI : 10.3745/ktsde.2016.5.5.209

[14] S. Lee & H. Yoo. (2017). System Optimization Technique using Crosscutting Concern. *Journal of Digital Convergence, 15(3),* 181-186.
DOI : 10.14400/jdc.2017.15.3.181

[15] Y. S. Choi & J. E. Hong. (2017). Designing Software Architecture for Reusing Open Source Software. *Journal of Convergence for Information Technology, 7(2)*, 67-76.
DOI : 10.22156/cs4smb.2017.7.2.067

[16] J. Lee, Y. Lee & M. Lee. (2001). Extraction of Classes & Inheritance from Procedural Software. *Journal of KISS : Software & Applications, 28(9),* 612-628.

[17] J. P. Kim, D. H. Kim & J. E. Hong. (2011). Embedded software, Low-power software, Power reduction technique. *Journal of Convergence for Information Technology, 1(1)*, 55-65.

[18] Wikipedia. (2017). *Procedural Programming*. Wikipedia. https://en.wikipedia.org/wiki/Procedural_programming
[19] Wikipedia. (2017). *Object-oriented Programming*. Wikipedia. https://en.wikipedia.org/wiki/Object-oriented_programmng
[20] TIOBE. (2018). *TIOBE index for March 2018*. TIOBE. https://www.tiobe.com/tiobe-index/
[21] L. Qiu. (1999). *Programming Language Translation*. Doctoral Dissertation. Department of Computer Science, Cornell University, New York.
[22] D. Kim, J. Hong, I. Yoon & S. Lee. (2016). Code refactoring techniques for reducing energy consumption in embedded computing environment. *Cluster Computing Journal*, 1-17. DOI : 10.1007/s10586-016-0691-5
[23] P. A. Bernstein. (2003). Applying Model Management to Classical Meta Data Problems. *Proceedings of the 2003 CIDR Conference*, 1-12.
[24] J. J. Park, D. H. Kim & J. E. Hong. (2014). Analysis of Energy Efficiency for Code Refactoring Techniques. *Journal of KIPS transactions on software & data engineering, 3(3)*, 109-118. DOI : 10.3745/ktsde.2014.3.3.109

남 승 우(Nam, Seung woo) [학생회원]

▪ 2017년 2월 : 충북대학교 컴퓨터공학과(공학사)
▪ 2018년 1월 : University of Texas, Dallas, 방문 연구원
▪ 2017년 3월 ~ 현재 : 충북대학교 컴퓨터과학과 석사과정
▪ 관심분야 : 소프트웨어품질공학, 소프트웨어 안전성 분석, 사이버물리시스템, 소프트웨어 재사용
▪ E-Mail : swnam@selab.cbnu.ac.kr

홍 장 의(Hong, Jang Eui) [종신회원]

▪ 2001년 2월 : KAIST 박사(공학사)
▪ 2003년 10월 : 국방과학연구소 선임연구원
▪ 2004년 8월 : ㈜솔루션링크 부설 연구소장
▪ 2004년 9월 ~ 현재 : 충북대학교 소프트웨어학과 교수
▪ 관심분야 : 소프트웨어 품질, 요구사항 분석, 소프트웨어 안전, 소프트웨어 재사용
▪ E-Mail : jehong@chungbuk.ac.kr