韓國 컴퓨터情報學會 論文誌
第13卷 第4號, 2008. 7.

2008-13-4-1-1

임베디드 소프트웨어의 재사용성 향상을 위한 리엔지니어링 프레임워크

김 강 태*

Re-engineering framework for improving reusability of embedded software

kangtae kim*

요 약

대부분의 전자제품은 날로 다양한 소비자의 니즈로 인해 수많은 라인업을 보유하고 있다. 이에 대응하기 위해 보통 '베이스 모델'이라 불리는 초기 개발모델에서 각 상품화 과제의 특정 요구사항을 반영하여 파생개발을 한다. 제품 라인 업에 기반한 소프트웨어 라인 업이 형성되는 개발환경에서 '베이스 코드'의 구조와 그 구성요소의 품질은 향후 파생되는 여러 많은 제품의 생산성과 품질의 근간이 된다. '베이스 코드'는 최초 개발 후 여러 상품화를 거치면서 그 구조와 코드 자체에 수 많은 변경이 가해진다. 상품화 과제의 요구사양에 따라 변경되거나 추가되는 기능의 구현은 필수적 활동이며, 성능 개선 및 문제점 해결을 위한 구조 및 코드의 변경 역시 상품화를 통해 지속적으로 발생하는 유지관리 활동이다. 하지만 위와 같은 변경은 최초 설계 시 의도된 구조가 깨지거나 코드의 복잡도가 증가하는 등의 노쇠화 징후(Ageing symptom)로 나타나 유지관리에 어려움을 준다. 본 논문은 노쇠화된 베이스 코드의 상품화 적용 효율을 높이기 위해 재사용성, 유지보수성, 확장성 등의 비기능적 요소(Quality attribute)의 개선을 위한 절차와 기법으로 리엔지니어링 프레임워크를 제안한다.

Abstract

Most consumer electronics companies hold numerous line-ups to cope with divergent customer's needs. To cope with current situation, most products are derived from the 'base product' which is developed for brand new features with respect to the change requests. That is called derivation. After 'base code' is developed for newly introduced products, some modification will occur corresponding to the derivative product models. So, quality attributes of 'base code' affects quality and productivity of 'derived code'. But in the middle of continuous modification to 'base code', violation of architectural design decision and unauthorized or maybe unsophisticated change to source code willing to happen and thus it cause critical problem. Those code has 'aging symptom'both architectural and code level in nature. In this paper, we introduced reengineering framework which guide the procedure and tactics to find and fix 'aging symptom'for improvement on quality attribute of 'base code'.

▸ Keyword : 소프트웨어 리엔지니어링(Software re-engineering), 소프트웨어 유지보수(Software maintenance), 리팩토링(Refactoring), 리스트럭쳐링(Restructuring)

• 제1저자 : 김강태
• 접수일 : 2008. 3. 30, 심사일 : 2008. 6. 15, 심사완료일 : 2008. 7. 25.
* 삼성전자 책임연구원

Ⅰ. 서 론

소프트웨어 리엔지니어링은 기존 소프트웨의 구조 및 코드를 개선하기 위한 방법으로 기존 소프트웨어에 대한 유지보수성 향상을 위해 수행하는 일련의 작업이라는 것이 일반적인 정의이다[1]. 일련 프로세스는 다음과 같다. 1) 기존 소프트웨어의 추상화된 특성을 추출하기 위한 역공학(Reverse engineering) 기법, 2) 동일한 추상화 레벨을 가지고 해당 구조를 변경시키는 재구조화 (Restructuring) 기법, 3) 기존의 추상화 수준과 다른 수준으로 구조를 변경시키는 변경(Modification) 기법, 4) 새로운 구조를 반영하기 위한 신규 구현 (Re-implementing) 기법 이 그것이다[2][3].

임베디드 소프트웨어 영역에서 특히 리엔지니어링 기법이 필요한 이유는 임베디드 소프트웨어만의 독특한 개발환경에 기인한다. 소프트웨어 자체가 판매 유닛이 아니기 때문에 제품의 라인 업에 종속적일 수 밖에 없다. 특히 전자제품은 기본적인 기능 위주로 개발된'베이스 모델'을 지역적, 기능적으로 파생하기 때문에 개별 라인 업에 대한 표준화된 부품화 및 품질 확보가 향후 상품화의 생산성과 품질에 근간을 이룬다. 제품을 구성하는 표준화된 부품의 개념으로 임베디드 소프트웨어 역시 '베이스 모델'에 대응하는 '베이스 코드'가 먼저 개발되고 향후 파생 모델에 특화된 요구사양 및 하드웨어 사양을 반영하여 개발된다[4]. 또한 임베디드 소프트웨어는 그 특성 상 지속적인 최적화 이슈가 존재한다. 원가절감을 위한 메모리 사용량 최적화, 속도향상을 위한 성능 최적화 등이 그것이다[5][6]. 이와 같은 최적화 이슈와 제품 파생에 대한 신규 기능 개발에 대한 '베이스 코드'의 대응은 곧 코드의 수정을 의미한다. 즉, '베이스 코드'의 상품화 이력이 많아질수록 코드에 대한 수정이 크게 늘어나게 된다. 이에 대응하기 위한 적절한 관리기법과 통제수단이 미약한 경우 무분별한 코드 수정에 의해 초기에 설정된 아키텍쳐 설계와 구현전략이 상실된다.

본 논문에서는 잦은 수정에 의한 임베디드 소프트웨어의 설계 및 코드의 품질저하 현상을 검출하고 이를 개선하기 위한 방법으로 리엔지니어링 프레임워크를 제안한다. 목적은 노쇠화된 베이스 코드의 적용효율을 높이기 위해 재사용성을 기반으로 유지보수성과 확장성을 증대시키는 것에 있다[7].

본 논문의 구성은 다음과 같다. 2 절에서는 관련된 연구를 비교 분석하고, 3과 4절에서는 리엔지니어링 프로세스와 뷰를 제안한 후 5절을 통해 결론을 맺는다.

Ⅱ. 관련 연구

지금까지 소프트웨어 리엔지니어링에 대한 많은 선행연구들이 있었다. 본 논문이 다루고 있는 임베디드 소프트웨어 영역에서는 Madisetti[8]와 DelPrincipe[9]의 연구가 대표적이다. 위의 연구에서는 임베디드 소프트웨어의 특성을 기반으로 주로 하드웨어와의 종속관계나 리엔지니어링 후 성능을 주로 다루었으나 전반적인 프로세스 및 체계적인 뷰와 메트릭에 대한 정의가 부족하다. 코드 리팩토링에 대해서는 Fowler가 디자인과 연계된 코드의 품질 개선 측면에 대한 연구로 공헌했다[10]. 위의 논문들을 기반으로 보다 체계적인 관리 측면의 접근을 시도한 연구가 진행되었다. 이는 현실에서의 적용성을 높이기 위한 시도로 특히 Ganesan 등의 연구[11]는 실험적으로 약 20여 회 이상의 파생을 거친 코드는 생산성과 품질 측면에서 더 이상 'base code'로의 기능을 상실한다는 연구결과를 내놓기도 했다.

위에서 언급한 파생에 따른 문제점은 최초 설계에서 의도된 구조가 깨지거나 코드의 복잡도가 증가하는 등의 현상으로 나타난다. 그리고 이와 같은 현상에 대한 Vissagio의 연구[12]에서는 이를 '노쇠화 현상(Ageing symptom)'이라 정의하였다. 코드에 '노쇠화 징후'가 발생했을 때 적용할 수 있는 기법이 리엔지니어링 기법으로 아키텍쳐와 코드 측면에서의 개보수를 의미한다. '노쇠화 징후'는 아래와 같이 다섯가지 현상으로 정리될 수 있으며, 각 현상의 발생여부를 고려하여 리엔지니어링 기법의 적용여부를 판단해야 한다. 1)Pollution: 기능을 구현하는데 필요없거나 중복적인 컴포넌트 혹은 코드를 포함한다. 2)Embedded knowledge: 상품화에 따른 변경사항이 적절한 형태의 정보(문서화 등)로 반영되지 않는다. 3)Poor lexicon: 소프트웨어의 구조 및 컴포넌트 명이 더 이상 기능차원의 식별자 역할을 하지 못한다. 4) Coupling: 소프트웨어를 구성하는 컴포넌트들 간의 의존성이 높아져 데이터 및 호출관계를 관리하기 어렵다 5) Layered architectures: 기능 및 역할을 구분하는 소프트웨어 계층구조가 더 이상 그 역할을 수행하지 못한다.

위와 같은 기술이 화보되어 있더라도 실제적인 적용이 실패하는 원인에 대한 Bergey의 연구[13]는 프로세스와 기준을 중요한 성공요인으로 평가하고 있다. 따라서 본 논문은 위와 같은 관련연구를 바탕으로 적절한 리엔지니어링을 수행하기 위한 절차, 기법 그리고 실행의 기준이 되는 뷰와 메트릭을 통해 보다 실제적인 적용이 가능한 프레임워크를 제안한다.

III. 리엔지니어링 프레임워크

3 절에서는 본 논문에서 제안하는 리엔지니어링 프레임워크에 대해 논한다. 리엔지니어링 프레임워크는 절차로써의 리엔지니어링 프로세스, '노쇠화 현상'을 판단하기 위한 기준으로써 리엔지니어링 뷰, 이를 수행하기 위한 자동화 도구 등의 리엔지니어링 환경으로 구성된다.

3.1 리엔지니어링 프로세스

〈그림 1〉은 본 논문에서 제안하는 리엔지니어링 프로세스 개념도이다. 프로세스는 총 5개의 스텝과 9개의 액티비티로 구성되어 있다.

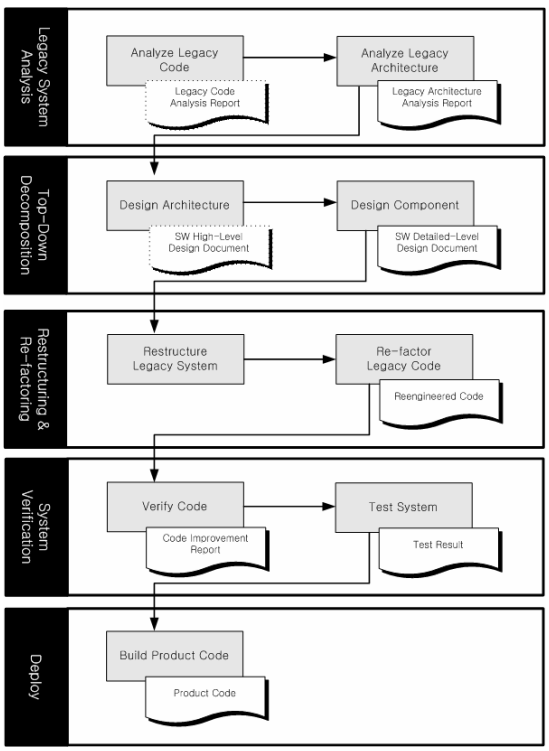

그림 1. 리엔지니어링 프로세스
Fig 1. Re-engineering Process

① Legacy system analysis

이 단계는 기존 소프트웨의 아키텍처와 소스 코드의 구현 상태를 분석하여 개선해야 할 문제점을 발굴하고 리엔지니어링의 목표를 수립하는 단계이다. 아키텍처 레벨과 코드 레벨의 분석을 병행한다. 코드의 '노쇠화 징후'를 최대한 정량적 기준으로 판단하기 위해 〈그림 2〉와 같이 의존도, 복잡도 등과 같은 여러 리엔지니어링 뷰를 활용한다.

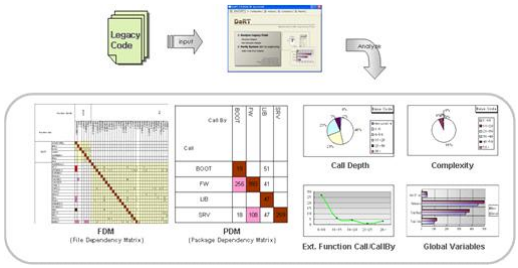

그림 2. 기존 시스템 분석
Fig 2. Legacy System Analysis

② Top-down decomposition

이 단계는 Legacy system analysis 단계에서 설정한 개선 목표에 따라 새로운 아키텍처를 설계하고, 후보 컴포넌트를 설계하는 단계이다. 여러 가지의 후보 아키텍처를 설계하고 비교 평가하는 방법이 도입될 수 있다. 〈그림 3〉은 아키텍처 설계 수준의 리엔지니어링 목표 수립의 예이다.

설계 목표	분류	설계 전략
재사용 환경 구축	공통	Core Library 설계, Core Library와 컴포넌트간의 Dependency를 줄일 수 있는 Abstract Layer 설계, Data Type을 정의하는 Base Library 구축 등을 Reference Architecture에 반영할 수 있는 전략 수립
재사용 컴포넌트 선정 및 설계	공통	재사용 환경 기반하에 재사용할 수 있는 컴포넌트 대상 선정 및 설계에 대한 전략 수립
표준 Specification 요구사항 반영	과제	과제에서 준수해야 하는 Specification의 요구사항을 만족할 수 있는 Reference Architecture 설계 전략 수립
Legacy System의 취약점 개선	과제	Legacy System Analysis 단계에서 발견된 취약점의 개선안을 Reference Architecture에 반영할 수 있는 전략 수립 (Reference Architecture는 Legacy System의 기능은 유지할 수 있어야 함)
비기능적 요구사항 반영	과제	Legacy System의 비기능적인 요구사항(성능, 안전성 등)을 반영할 수 있는 전략 수립

그림 3. 리엔지니어링 목표 수립
Fig 3. Establishing the Goal of Re-engineering

③ Restructuring & re-factoring

이 단계는 새롭게 설계된 아키텍처를 바탕으로 기존 코드를 재사용하기 용이한 구조로 재구조화하고, 코드를 개선하는 단계이다. Legacy system analysis 단계에서 정량적으로 분석된 코드의 현 상태, 특히 '노쇠화 징후'를 제거하는 활동이 수행된다.

④ System Verification

이 단계는 Re-structuring & re-factoring단계를 통해 개

선된 소프트웨어의 결과를 검증하고, 테스트를 수행하는 단계이다. 〈그림 4〉와 같이 각 뷰에 대해서 Legacy system analysis 단계의 분석결과에 대비하여 수행된다. Re-engineering은 재사용성 및 유지보수성(확장성, 변경용이성 등을 포함)등의 품질 특성(Quality attribute)을 목적으로 진행되기 때문에 임베디드 소프트웨어의 가장 중요한 품질 특성 중 하나인 성능 요구사양과의 상호위배 현상이 발생할 수 있다. 따라서 본 단계의 주요 목적은 분석을 통해 설정된 품질 특성의 요구사항을 재사용성/유지보수성과 성능 측면에서 상호검증 하는 것이다.

⑤ Deploy

이 단계는 상품화 적용을 위한 소프트웨어 릴리즈 단계이다.

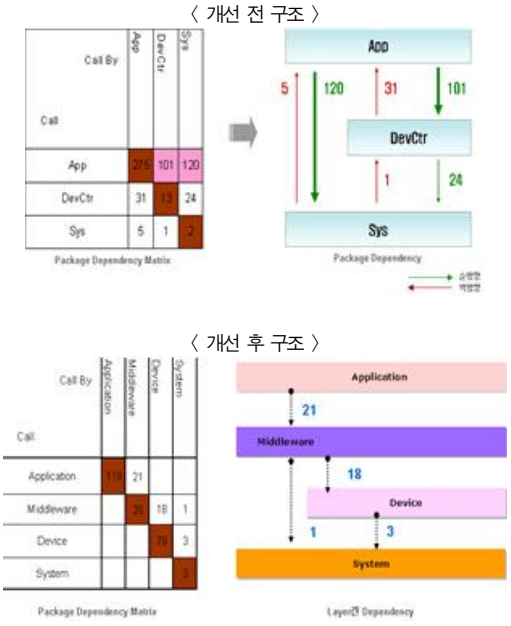

그림 4. 리엔지니어링 전후 비교
Fig 4. Verifying re-engineering results

3.2 리엔지니어링 뷰

본 논문에서는 리엔지니어링을 수행의 목적으로 노쇠화 현상에 대한 해결책에 대한 분류체계를 '리엔지니어링 뷰'로 정의한다. 〈표 1〉은 각 노쇠화 현상과 리엔지니어링 뷰 간의 연관관계를 나타낸다. 노쇠화 현상에 대한 분류기준은 Vissagio의 연구[10]를 기반으로 하였으며, 이에 따른 각각의 뷰와 메트릭은 본 논문에서 제안하는 바이다.

표 1. 노쇠화 현상에 따른 리엔지니어링 뷰
Table 1. Re-engineering views corresponding to aging symptom

Symptoms	Reengineering views
Pollution	4.3 Unused Function Elimination 4.6 Potential Defect
Embedded knowledge	4.5 Documentation
Poor lexicon	4.4 Naming Convention
Coupling	4.1 Module Dependency 4.2 Data Dependency
Layered architecture	4.1 Functionality Dependency 4.2 Data Dependency

각 '뷰'별로 〈표 2〉의 정량적 메트릭을 정의하여 리엔지니어링 활동을 통한 개선을 검증할 수 있도록 하였다.

표 2. 뷰에 따른 리엔지니어링 메트릭
Table 2. Re-engineering metrics corresponding to views

View	Formula
Functionality Dependency	$\frac{\sum ExternalPackageFunctionCall}{\sum FunctionCall}$
Data Dependency	$\frac{\sum ExternalPackageFunctionCall}{\sum FunctionCall}$
Unused Item	$\sum DecreasedUnusedLOC$
Naming Convention	Naming Convention Documentation
Documentation	Function Interface Automation Ratio
Potential Defect Elimination	Potential Defect Decrease Ratio
	$\sum DecreasedUnusedDeadCode$

본 논문에서 제시하는 메트릭은 상용툴인 'Understand C++', 'K7'등의 정적분석를 통해 원시데이터(raw data)를 추출한 후, 본 논문에서 제안하는 메트릭을 산출하는 별도의 데쉬보드 툴을 개발하여 적용하였다.

각 메트릭에 대한 상세한 설명은 4장에서 '뷰'를 기준으로 한 리엔지니어링 활동 설명을 통해서 논한다.

Ⅳ. 리엔지니어링 기준 : 뷰

4 절에서는 각각의 리엔지니어링 뷰에 대해 각 뷰 별로 필요성, 방법, 메트릭을 통해 상세 설명을 한다.

4.1 Functionality dependency

① 필요성

'베이스 코드'의 관리에 필수적인 요소 중 하나는 최초 설계에 의해 부여된 모듈의 독립성을 최대한 유지하는 것이다. 일반적인 개념의 모듈간 의존성에 대한 관리도 중요하나, 임베디드 소프트웨어의 특성 상 많은 주변장치를 제어하는 특징을 가지고 있기 때문에 이에 대한 변경이 매우 빈번히 발생한다. 하드웨어 제어부인 디바이스 드라이버는 상위의 어플리케이션이나 미들웨어에서 직접 접근하지 않는 구조로 유지하는 것이 재사용에 유리하다. 때문에 디바이스 드라이버로의 의존성을 메트릭으로 정의하고 이를 중점 관리하도록 한다.

② 기법

추상화 계층을 정의하여 디바이스 드리이버에 대한 독립성을 확보한다. 추상화 계층에는 HAL 및 OSAL이 있을 수 있다. 추상화 계층은 Coupling을 감소시키고 명확한 계층 구조를 통해 기능의 집중도를 높여 유지보수 및 재사용성을 향상시킨다. 여러 번의 상품화를 거치면서 각기 다른 개발자에 의해 개발된 베이스 코드는 디바이스 드라이버에서 미들웨어 및 어플리케이션을 호출하는 역 호출(Back call)과 어플리케이션에서 디바이스 드라이버를 직접 호출하는 건너뛰기 호출(Skip call)이 빈번하게 발생한다. 이러한 호출로 인해 의도하지 않은 문제가 발생하기도 하고 개발자의 유지보수 업무를 복잡하게 하여 생산성 저하의 원인이 되기도 한다[12]. 추상화 계층인 HAL, OSAL은 상위 계층의 구현을 수정하지 않은 채 추상화 계층만을 수정하여 대응 가능한 구조를 제공한다.

③ 지표

NOFCD (The Number of Function Call to the Device Driver Layer) : 디바이스 드라이버로의 직접 호출의 회수로 각각 역 호출과 건너뛰기 호출로 구성된다. 전체 LOC 비율로 하여 본 지표는 작을수록 의존성이 작으므로 궁극적으로는0을 목표로 한다.

4.2 Data dependency

① 필요성

보통 임베디드 소프트웨어에서는 시스템 상태값을 갖는 전역변수로 인해 데이터 의존성이 많이 생긴다. 전역변수는 다음과 같은 이유로 인해 많이 사용하게 된다. 1) 개발 편의성: 여러 다른 function 간 자료교환을 할 필요 없이 전역변수로 접근하면 프로그램 로직이 편해진다. 대부분의 경우 개발자는 일단 전역변수로 선언하여 데이터에 대한 접근성을 확보한 후 개발하려는 성향이 있다. 이에 대한 분석결과 평균 약 255의 전역변수는 실제 사용되지 않거나 지역변수로 전환해도 무방한 변수로 분석되었다. 2) 메모리 효율성: function 호출 수를 최적화하여 메모리 스택을 절약할 수 있다. 한정된 자원을 가진 개발환경에서는 필수적이다. 널리 알려진 바와 같이 무계획적인 전역변수 사용은 세마포어 문제 등 데이터 의존성에 의해 복잡도를 증가시킨다. 따라서 1)에 의한 전역변수는 기본적으로 제거하는 것이 목표이다. 또한 여러 번의 상품화 과정을 통해 코드가 수정되면서 사용하지 않거나 로컬화 할 수 있는 변수를 찾아내어 없애는 것도 포함된다.

② 기법

베이스 코드의 데이터 의존성을 분석 하여 전역변수 각각의 특성을 두 가지로 구분 한 후, 〈표 3〉의 기법을 활용한다. 1) 지역변수화 : 지역변수화가 가능한 것은 변수선언을 변경시킨다. 2) 사용되지 않는 전역변수는 삭제한다. 3) 전역변수 캡슐화: 해당 변수에 대한 접근을 인테피에스 함수화하여 처리한다. 4) Locking 기법: 세마포어 등의 기법을 적용한다.

③ 지표

Global Variable External Use Ratio (GVEUR): 전역변수가 정의된 모듈 이외의 모듈에서 해당 전역변수를 접근하는 비율로써 높을수록 의존성이 높아지며, 이 수치를 낮추는 것이 목표이다.

표 3. Data Dependency의 해결기법
Table 3. Solution to Data Dependency

필요성	기법
개발 편의성	1) 지역변수화 2) 미사용 전역변수 삭제
메모리 효율성	3) 전역변수 캡슐화 4) Locking 기법 적용

4.3 Unused element

① 필요성

베이스 코드가 변경 혹은 진화해감에 따라 새로운 신규 기능에 의한 코드의 추가가 발생한다. 이와 더불어 기존의 기능이 새로운 기능에 의해 대체되면서 기존 기능은 사용되지 않게 된다. 일반적으로 베이스 코드가 진화되면서 이렇게 사용되지 않는 구성요소는 점차 늘어나게 된다. 이렇게 사용하지 않는 구성요소는 유지보수 및 재사용에 악영향을 미칠 뿐이다. 따라서 이러한 요소를 찾아내어 제거하는 것이 필요하다. 이러한 구성요소에는 Unused variable, Unused function, Unused parameter, Unused data-type 등이 있다.

② 기법

일반적으로 사용되지 않는 구성요소들은 컴파일 시 제거되거나 포함되지 않기 때문에 기능구현 상으로는 큰 문제가 되지 않을 수 있다. 하지만 'Base code'로써 유지되어야 할 코드는 구조 및 코드 구현의 유지보수성이나 가독성 면에서 지속적으로 정비되고 관리될 필요가 있다. 코드에 대한 정적 분석을 통해 사용되지 않는 구성요소를 찾아낼 수 있고 검토를 통해 제거한다. Unused function의 경우, COTS component로부터 호출되는 의도적으로 공개된 Open API등이 제외되어야 할 검토 대상이다. Unused variable은 모두 삭제하도록 한다. 또, Unused parameter와 Data-type의 경우, 향후 사용을 고려하여 의도적으로 남겨놓은 것을 제외하고 삭제하도록 한다.

③ 지표

사용되지 않는 함수의 LOC를 기준으로 이 수치를 줄여가는 것을 목표로 한다.

1) Unused Function LOC (UFLOC): 사용되지 않는 함수가 차지하는 LOC의 비율을 나타낸다.
2) Decreased Unused Function LOC (DFLOC): 불필요한 함수로 삭제된 양의 LOC 비율을 나타낸다.

4.4 Potential defect elimination

① 필요성

임베디드 영역, 특히 전자업계의 소프트웨어는 디지털 융합 등 기술적 요인과 시장의 니즈에 부응하기 위해 소프트웨어의 사이즈와 복잡도가 날로 증가하고 있다. 또한 날로 치열해지는 경쟁 속에서 개발납기도 점점 줄어들고 있다. 이와 같은 이중고 속에서 품질개선에 대한 니즈는 지속적으로 늘고 있다. 리엔지니어링을 통해 코드를 정비할 때 잠재적 오류요소를 모두 발견하여 미리 방지한다면 상품화 시 품질에 투입되는 막대한 리소스를 절약할 수 있다. 잠재 문제점(Potential defect)은 향후 시장에서 문제가 될 가능성이 있는 것들을 말하며, 이를 일찍 발견하여 제거할수록 재작업에 대한 리소스를 절약하고 개발기간을 줄일 수 있다.

② 기법

잠재적 문제점은 정적분석 도구를 이용하여 검출한다. 잘못된 문제점도 검출될 수 있기 때문에 도출된 문제점은 별도의 분석과정을 통해 필터링되어야 한다. 잠재문제점은 심각도에 의해 분류되어 개선활동의 기준에 따라 처리한다.

③ 지표

잠재적 문제점의 개수(The Number of Potential Defects (NOPD))를 전체 LOC의 비율로 하여 지표로 정의한다.

4.5 Documentation

① 필요성

개발 정보 및 내역에 대한 문서화는 유지관리 측면에서 매우 중요하다. 그러나 실제 필드에서 개발자들은 문서화는 매우 부가적이며 이익이 수반되지 않는 활동으로 여겨져 설계 문서 작성 및 갱신이 잘 이루어지지 않는다. 초기 베이스 코드 개발 시 문서화가 잘 되었다 하더라도 여러 번의 상품화를 거치면서 변경된 코드는 이전의 문서화 내용과 일관성을 갖기 힘들다. 리엔지니어링은 이와 같이 설계와 구현 간 문서화 일관성을 확보하는 차원에서도 매우 유익한 활동이다.

② 기법

문서화 전략에 따른 작업이 필요하다. 기본설계와 상세설계 별로 리엔지니어링된 소프트웨어의 기본구조에서 모듈 간 의존성, 그리고 모듈 내 의존성, 데이터 의존성을 문서화 한다. 리엔지니어링의 특성 상 기존 코드를 기준으로 정비하는 경우가 많기 때문에 일반적인 개발방법에서의 설계 구현 순서로의 문서화와 달리 소스코드로부터 상세 설계문서(API 및 함수 설명서 등)를 생성하는 역공학적 문서 자동화 도구를 도입하여 문서작업의 생산성을 높일 수 있다.

③ 지표

아키텍쳐 문서화의 수준을 메트릭으로 정의하기는 어렵기 때문에 기본설계의 수준은 단순히 기본설계가 확보되었는가를 나타내는 메트릭을 정의한다. (High-Level Design Documentation (HDOC)) 상세설계는 전체 API 중 문서화된 부분의 비율인 CDOC(Code-Level Documentation Coverage)로 이 비율을 높이는 것을 목표로 한다.

V. 적용사례

본 논문에서 제안한 리엔지니어링 프레임워크를 디지털 멀티미디어 플레이어 도메인인 2개의 제품군(〈표 4〉)의 베이스 코드에 적용한 사례이다.

표 4. 적용과제 개요
Table 4. project information

Product Family	A	B
Product	DVD Recorder	Analog LCD TV
SLOC	268,037	79,649
Resources	6 people	3 people
Period	4.5 months	3 months
O/S	VxWorks	N/A

5.1 기존 베이스 코드 분석

리엔지니어링 프레임워크의 '뷰' 각 항목을 기준으로 베이스 코드를 분석한다.

① Functionality dependency

〈그림 5〉와 같이 프로젝트 B의 Application 영역에서 Video 모듈과 달리 Tuner 및 Audio모듈은 직접 하드웨어 디바이스 혹은 디바이스 드라이버로의 호출을 가지고 있다. 이러한 경우에 의존성이 발생한다. Dependency의 값은 5%(9/173)이다.

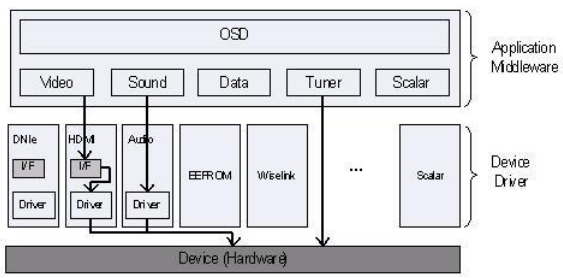

그림 5 과제 B의 아키텍쳐
Fig 5. Architecture of project B

② Data dependency

프로젝트 B에서 대부분의 모듈은 전역변수에 대한 읽기/쓰기를 모두 수행하고 있었다. GVEUR 수치는 35% 정도에 이르러 Data dependency가 매우 높은 상태였다.

③ Unused item

프로젝트 A에서 사용되지 않는 함수는 2121개로 약 26%에 이르렀고 LOC로는 74,417에 달했다. 소스코드는 개발된 지 7년이 넘은 오래된 코드로 여러 가지 사유로 이미 기능이 상실된 함수가 코드 상에 남아 있는 경우가 많았다.

④ Naming convention

프로젝트 B는 소스코드의 디렉토리 구조, 파일 및 함수명, 변수명에 대한 어떠한 룰과 규칙을 가지고 있지 않았다. 일부 함수명의 경우 'get', 'set'과 같은 Prefix를 통해서 그 기능을 추측할 수 있었으나 정형화된 룰에 의한 표기는 아니었다. 또한 여러 번의 상품화를 통해 초기에 정의된 기능이 없어진 채 완전히 다른 기능을 하면서 이름은 예전 함수 이름을 그대로 사용하는 경우도 빈번하였다.

5.2 Top-down decomposition

리엔지니어링 프레임워크의 '뷰' 각 항목을 기준으로 베이스 코드를 분석한다. 시스템 분석을 통해서 리엔지니어링의 목적이 식별되면, 이를 반영할 수 있는 새로운 아키텍처의 설계가 필요하다. 기존 아키텍처를 바탕으로 리엔지니어링 뷰에 따라 설정된 각 뷰 단위의 요구사양을 아키텍처 설계에 반영한다. 프로젝트 B의 경우Device driver와 상위 레이어 간 의존성을 제거하기 위해 〔그림 6〕과 같이 아키텍처 변경의 핵심을 DAL(Device Abstraction Layer) 설계에 두었다. 임베디드 소프트웨에서 가장 많은 변경은 역시 HW의 변경에 따른 Device driver의 교체이고 이에 의한 변경을 최소화하는 것이 아키텍처 설계의 목적이다. 실제적인 DAL의 적용은 기술적인 측면 뿐 아니라 Device driver 업체와의 API 표준화 활동과 같은 전략적 요소에 의해서 그 성공여부가 좌우된다. Re-engineering 과정의 직접적인 효과는 아니지만 이를 통해 DAL interface의 표준화 작업을 병행할 수 있으며, 실제적으로 효과적이다.

5.3 Re-structuring & re-factoring

수정된 아키텍처에 의해 구조 및 코드를 수정하는 작업이다.

1) Re-structuring

수정된 아키텍처를 반영할 수 있도록 코드 구조를 개선하는 작업이다. 논리적 구조로써의 아키텍처를 물리적 구조인 디렉토리 구조로 매핑시키되 최대한 1:1 대응 원칙을 고수할 수 있도록 설계한다. 기존 코드에 대한 디렉토리 구조 변경은 헤더 파일 등의 링크 정보에 대한 변경을 고려하여 진행해야 한다. 프로젝트 A, B의 경우 재구조화 작업으로 a) 논리적 구조와 물리적 구조의 매핑 b) DAL(Device Abstraction Layer)을 포함한 아키텍처 레이어 재정비 c) 전역변수 관리

를 위한DMS(Data Management System) 정의 d) 제품군의 공통/가변 기능을 반영하여 공통 모듈과 가변 모듈을 분리하여 설계하는 가변성 설계를 수행하였다.

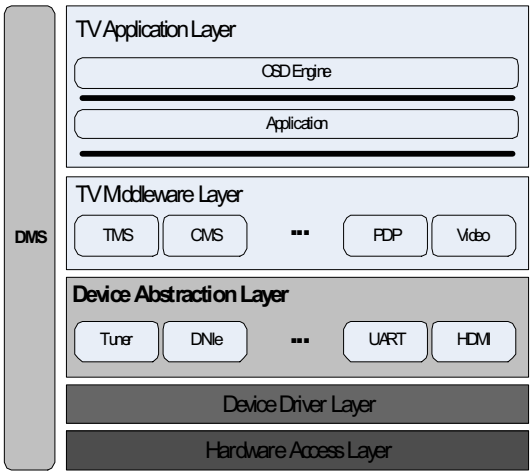

그림 6. 개선된 프로젝트 B의 아키텍쳐
Fig 6. refined architecture of project B

2) Re-factoring

수정된 아키텍처와 Re-structuring을 통해 결정된 설계 의도를 코드에 반영하는 작업이다. a) 전역변수의 static화 및 캡슐화 b) 미사용 아이템(함수, 변수 등)의 제거 c) Naming convention 적용 d) 잠재 문제점 제거 e) 소스 내의 스크립팅을 통한 설계 문서 역공학을 수행하였다.

5.4 System verification

4절에서 정의한 지표를 활용하여 지표값의 개선 여부를 통해 리엔지니어링 작업의 검증을 수행하였다. 〈표 5〉는 각 '뷰'에 대한 정량적 개선 수치이다.

표 5. 리엔진니어링 프레임워크 적용과제 개요
Table 5. project information

Metrics	Project A		Project B	
	Before	After	Before	After
NOFCD	2,239	174	173	66
GVEUR	57%	5%	35%	22%
UFLOC	74,417	2,161	8,517	5,917
DFLOC	-	72,256	-	2600
NCDOC	Not	Documented	Not	Documented
NCOR	-	100%	-	100%
HDOC	Not	Documented	Not	Documented
CDOC	Not	Documented	Not	Documented
NOPD	52 EA	0 EA	-	-

두 프로젝트 모두에서 가장 중점적으로 개선된 항목은 의존성으로 NOFCD와 GVEUR의 개선수치가 높은 것을 알 수 있다. 이는 시스템 전체 수준에서 의존성이 감소했음을 보인다. 특히 프로젝트 A의 경우 57%에 이르는 전역변수 의존성을 5%까지 줄였는데 이는 GVEUR 수치를 낮추기 위해 데이터 처리에 대한 아키텍처 자체를 변경하여 DMS(Data Management System)이라 불리는 별도의 Data 처리 모듈을 통해 데이터 종속성을 최소화시킴으로써 가능했다.

리엔지니어링을 통한 '베이스 코드'의 확보는 유지보수성 및 재사용성의 강화를 통해 직접적인 생산성 향상의 효과를 나타낸다. 실 예로 디지털 미디어 제품을 생산하는 여러 조직에서 〈표 6〉과 같은 실질적인 효과를 창출하였다.

표 6. 리엔지니어링 효과
Table 6. Return of re-engineering activities

항목	Product family A	Product family B
개선 내역	전역변수 20% 감소 Unused 함수 49% 감소 부품화 83% 증가 문서자동화 API 100%	M/W 부품간 의존성 62% 감소 코딩표준 100% 준수 부품화 33% 증가
경영 효과	M/M 65% 감소 (Lead time 30% 감소)	MM 54% 감소

VI. 결론 및 향후 연구

5절에서 제시된 바와 같이 본 논문에서 제시된 리엔지니어

링 프레임워크는 실제 과제를 통해서 그 효과가 입증되었다. 본 논문에서 제시하는 '리엔지니어링 뷰'를 통해 현 소프트웨어 시스템의 문제점을 정량적으로 분석하여 지표화할 수 있다. 지표 분석에 대한 기준으로써 '리엔지니어링 뷰'가 의미가 있다면 문제점에 대한 개선을 위한 절차로써 '리엔지니어링 프로세스'가 정의되어 리엔지니어링 프레임워크를 구성한다. 본 프레임워크의 적용 결과 구조적 개선을 위한 기준과 절차를 정의했다는데 의미가 있었으며, 실제 효과도 긍정적이었다. 다만, 각 지표에 대한 해석이 도메인에 따라 달라질 수 있기 때문에 이에 대한 가이드의 개발이 필요하며, 각 지표 별로 실제 개선작업에 적용할 수 있는 세부 기술을 연관하여 정리할 필요가 있었다. 따라서 향후 연구과제는 각 세부 지표별 분석 가이드의 개발과 개선 전략을 연계하는 리엔지니어링 프레임워크의 시스템화가 될 것이다.

참고문헌

〔1〕 R. S. Arnold, "A Road Map Guide to Software Reengineering Technology", in Software Reengineering, IEEE Computer Society Press, 1993.

〔2〕 E. J. Chikofsky, and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, 1990, pp. 13-17.

〔3〕 K. Periyasamy, and C. Mathew, "Paradigm Shift in Software Re-engineering: An Experience Report", IBM Centre for Advanced Studies Conference, 1996

〔4〕 Kangtae Kim, Hyungrok Kim, Woomok Kim, Building software product line form the legacy systems "experience in the digital audio & video domain", Proceedings of the 11th International Software Product Line Conference, 10-14 Sept., Kyoto, Japan, pp.171-180, 2007.

〔5〕 Y. Nakamoto, H. Takada, and K. Tamaru, "Embedded system technologies: Today and future", J inf Process Soc Japan, pp. 871-878, 1997

〔6〕 P. Kuvaja, J. Maansaari, V. Sepp änen, and J. Taramaa, "Specific requirements for assessing embedded product development", International Conference on Product Focused Software Process Improvement, pp. 68-85, 1999

〔7〕 Stan Jarzab다, "Software reengineering for reusability", Proceeding of COMPSAC 1993, IEEE, pp.100-106.

〔8〕 V. Madisetti et al., "Reengineering Legacy Embedded Systems", IEEE Design & Test of Computuers, 1999

〔9〕 M. DelPrincipe et al. "Reengineering: An Affordable Approach for Embedded Software Upgrade", Journal of Defense Software Engineering, 2001

〔10〕 M. Fowler, "Refactoring: Improve the Design of Existing Code", Addison Wesley, 2000

〔11〕 Dharmalingam Ganesan, Dirk Muthig, Kentaro Yoshimura, "Predicting Return-on-Investment for Product Line Generations", Proceeding of SPLC 2006, IEEE, 2006, pp. 13-22.

〔12〕 G. Visaggio, "Ageing of a Data Intensive Legacy System: Symptoms and Remedies", Journal of Software Maintenance and Evolution, vol. 13, no. 5, pp. 281-308, 2001

〔13〕 J. Bergey et al., "Why Reengineering Projects Fail", Technical Report CMU/SEI-99-TR-010, 1999

〔14〕 F. Buschmann et al., "Pattern-Oriented Software Architecture: A System of Patterns", Addison Wesley, 2001

저 자 소 개

김 강 태

2001 : 중앙대학교 대학원 컴퓨터공학과 박사(SE 전공)

2001~현재 : 삼성전자 SW연구소 SE팀 표준화파트장

관심분야 : SW Product Line, Re-engineering, CBD, Design Pattern, SPI