객체지향 기반의 Refactoring 프로세스
(Object-Oriented based Refactoring Process)

이 종 호 † 박 진 호 †† 류 성 열 †††
(Jong-Ho Lee) (Jin-Ho Park) (Sung-Yul Rhew)

요 약 기업에서는 급속한 컴퓨터 환경 변화 및 사용자 요구 증가 등의 요인들에 대응하기 위하여 많은 시간과 비용을 들여 기 개발되어 운영중인 시스템을 유지보수 한다. 하지만 대부분 임시적인 이러한 작업들은 많은 코드의 중복과 최적화 되지않은 시스템 구조를 산출하여 결국에는 전체적인 성능 저하를 가져오는 등의 문제를 발생시키게 된다. 또한 초기 개발 후 다른 개발자에 의해 작성된 코드는 개발관련 문서의 분실 및 부족, 기존 시스템 개발자의 부재 등의 문제로 코드의 이해와 재사용의 어려움 등의 한계점을 갖게 되었다. 이번 연구는 이러한 코드 재사용의 한계를 극복 하기 위하여 객체단위를 재사용 할 수 있는 객체지향 기반의 Refactoring 프로세스를 제시하고, 이를 D사에서 개발한 윈도우 시스템 개발 도구인 *D2D*에 적용한다. 또한 이의 과정에서 얻어진 시스템 성능 향상과 개발 및 유지보수 비용의 감소, 구조와 클래스들의 최적화 등의 실제적인 효과 증대 사례를 보인다.

Abstract The company invests its time and money for temporary maintenance to satisfy the fast change of the computer use environment and the user's demands.
Therefore, various problems occur including low performance because of duplication of codes and unstable structures from the restructuring and redevelopment.
Furthermore, if a developer, who did not participate in initial process of development, wrote additional program codes to upgrade or restructure, it would cause many problems such as lack or lose of development documentation, understanding of documentation and reuse of existing program language.
This study, Object-oriented Refactoring Process, suggests that the developers can reuse object unit to overcome the limit of the reusing code. In addition, this paper shows case study to verify our process by adjusting the project called "D2D", which is a case tool for developing windows system from Company D. Our works get positive results such as improvement of system performance, decreased cost of development and maintenance and optimizing structure and class.

1. 서 론

최근 인터넷 환경 등 전산환경의 급변과 사용자의 요구사항이 다양하게 증가됨에 따라 여러 개발자들에 의해 개발되어 사용되고 있는 시스템들을 짧은 기간에 대규모로 유지보수 하게 된다. 그러나 이러한 시스템들은 일반적으로 개발관련 문서의 분실 및 부족, 기존 시스템 개발자의 부재 등의 문제로 코드의 이해와 재사용의 어려움 등의 한계점을 갖게 됨에 따라 기능적으로나 구조적으로 많은 중복 소스코드를 갖게 되어 시스템의 성능 저하 요인으로 작용하고 있다. 또한 시스템 성능향상을 위한 유지보수 작업에서 이러한 문제점들을 해결하기 위하여 많은 비용과 시간이 소요된다[1].

재사용 단위 중 하나인 객체는 클래스로부터 생성되고, 클래스는 실세계에서 유사한 개체들이 공통적으로 필요로 하는 데이타와 이 데이타를 기반으로 수행되는 기능들을 정의한 소프트웨어 단위이다. 따라서 소프트웨어 재사용을 위해 기존의 함수 단위보다는 객체를 재사용 하는 것이 더 유용하게 된다. 하지만 기업의 소프트웨어 재사용은 자신들이 보유하고 있는 개발 문서나 해당 개발자의 경험을 통해 비체계적으로 이루어지고 있

† 정 회 원 : 숭실대학교 전자계산학과
jhlee@selab.soongsil.ac.kr
†† 비 회 원 : 숭실대학교 전자계산학과
jhpark@selab.soongsil.ac.kr
††† 종신회원 : 숭실대학교 컴퓨터학부 교수
syrhew@computing.soongsil.ac.kr
논문접수 : 2001년 1월 22일
심사완료 : 2001년 4월 30일

으며, 해당 개발자가 존재하지 않거나 개발 관련 문서 또는 소스 코드 등의 형상 관리가 체계적으로 이루어지지 않아 효과적인 소프트웨어의 재사용은 이루어지지 않고 있다[1,2].

따라서 재공학을 통하여 기 시스템의 구조와 기능을 분석하여, 문제점들을 해결하고, 운영중인 시스템의 소프트웨어를 재사용하여 성능향상을 시키는 것이 더욱 경제적이고 효율적인 방법일 것이다[3].

이번 연구에서는 재사용 방법 중의 하나인 Refactoring을 C++언어로 개발된 기 운영 시스템에 적용하여, 객체지향 개념의 Refactoring 프로세스를 제시하고, 이를 사용하여 구조의 최적화 및 시스템 수행 속도의 향상, 작업 과정의 이해도 향상과 중복 작업의 단순화 등의 성능 향상을 D사의 *D2D*에 적용하여 각 프로세스 단계별 수행 과정을 보인다.

2. 관련연구

2.1 Refactoring과 Restructuring

Refactoring이란 소프트웨어의 큰 변경 없이 내부적인 구조를 쉽게 이해하고, 경제적으로 수정할 수 있도록 만들어 주는 작업이다. 즉, 내부적인 요소들의 간단한 변경으로 시스템의 성능을 최적화시키는 작업을 말하며 기본적인 기능들의 변화 없이 성능을 향상시키고, 근본적인 원인 요소들만 찾아내서 변화시키기 때문에 작업이 용이하다[4,5].

실제적으로 시스템의 내부 구조의 개선이 이루어지지 않거나 외부 코드의 행위가 너무 오래 되었을 때에 시스템의 구성 요소들을 교환하여 성능을 향상시키거나, 기능적인 행동을 보장하면서 근본적인 원인들만 찾아내서 변화시킴으로써 더욱 더 향상된 시스템을 만들어 준다[6].

재구성(Restructuring)은 이름을 변경(Rename)하거나 형태(Type) 변환을 위주로 한 구조의 재구성을 하는 재구조화의 작업을 통해 소프트웨어 재공학 작업을 수행한다[3].

Refactoring과 재구성의 공통점은 작업 전후 기본 기능의 변동이 없고, 시스템의 이해를 돕는다는 것이다[5].

소프트웨어를 재사용하는 기법인 재공학은 재구성을 이용하는 것보다 Refactoring을 사용하는 것이 소프트웨어 성능향상을 위한 작업의 효율성 측면에서 매우 유리함을 알 수 있다. 또한, 기존의 구조적 개발 방법으로 만들어진 소프트웨어를 재사용하는 구조적 Refactoring은 작업의 규모나 비중에 비해 효율의 성과가 크지 못하지만, 객체지향 개념이 도입되어 객체지향 개발 기법과 도구들을 이용하여 개발되어진 소프트웨어들은 새로운 개념의 객체지향 Refactoring 기법을 사용하여 성능이 향상되는 효과를 기대할 수 있다[3].

이번 연구에서는 객체지향 Refactoring 프로세스를 제시하고, Refactoring단계에서 객체지향의 중요한 특징 중의 하나인 다형성과 상속성(Polymorphism & Inheritance)의 특성을 이용한 PI-Refactoring 다이어그램 모델을 이용하여 객체지향 Refactoring 프로세스 기법으로 각 단계별 수행과정을 보인다[7]. 그림1은 재사용 기법들을 정리하여 나타낸 계층도이며, PI-Refactoring은 객체지향 Refactoring의 수행 범위 안에 속한 새로운 프로세스 기법이다[8,9].

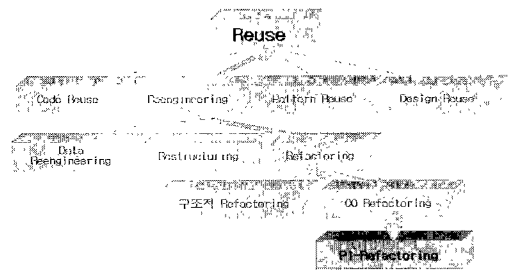


그림 1 재사용 기법과 PI-Refactoring

2.2 Serge Demeyer와 Stephane Ducassse의 Refactoring 프로세스

클래스 다이어그램(Diagram)을 중심으로 Refactoring을 수행하게 한 5단계 기법을 제시하고 있으며, 다음의 작업 단계를 갖는다[10].

- 1단계: Create Subclass
- 2단계: Move Attribute
- 3단계: Move Method
- 4단계: Split Method + Move Method
- 5단계: Clean-up

이 프로세스의 장점은 미리 서브 클래스를 생성한 후 요소들의 자리를 이동시키기 때문에 중복되거나 작업 중 지나칠 수 있는 요소들을 찾아내기가 쉽다. 즉, 요소들의 목록을 만들기 때문에 요소들에 대한 관리와 변경이 체계적이다.

단점은 1단계에서 미리 서브 클래스를 생성한 후 요소들의 이동을 시작하기 때문에 꼭 필요한 요소임에도 불구하고 생성한 서브클래스로 이동시키는 오류를 범하거나, 각각의 다른 기능을 수행하는 요소이지만 다형성을 지키지 못한 채, 하나의 같은 요소로 인식하여 중복

되게 덮어쓰는 경우 등이 생긴다.

2.3 Martin Fowler의 Refactoring 프로세스

클래스 다이어그램을 중심으로 Refactoring을 수행하게 한 4단계 기법을 제시하고 있으며, 작업 수행은 다음과 같다[11].

- 1단계: Extract Method
- 2단계: Move Method
- 3단계: Extract and Move Method 적용(1~2단계에서 미흡한 단계의 반복 적용)
- 4단계: Replace Temp with Query

이 프로세스의 장점은 요소들을 추출하여 반복적으로 이동하면서 적절한 위치를 찾아 변경하도록 하게 하기 때문에 간단한 자리 이동을 통해 시스템 성능향상을 위한 작업을 수행할 수 있다. 또한 쿼리(Query)를 이용한 임시저장소(Temp)를 사용하기 때문에 별도의 추가 공간이나 클래스 생성이 필요 없다.

단점은 작업 공간에 불필요한 반복적인 이동이 발생하기 때문에, 최적화된 성능을 발휘할 수 있는 요소들까지도 자리이동 되어지게 된다.

2.4 PI-Refactoring 다이어그램 모델링 기법

본 절에서는 식1과 같이 Refactoring 작업을 사용자가 좀 더 쉽게 이해하고, 용이한 작업을 할 수 있도록 도와주기 위해서 UML의 클래스 다이어그램과 순차(Sequence) 다이어그램을 재구성한 다이어그램을 PI-Refactoring이라고 정의한다[12,13].

PI-Refactoring Diagram
=Class Diagram+Sequence Diagram (1)

PI-Refactoring다이어그램은 클래스 다이어그램과 순차 다이어그램들을 이용한 Refactoring 반복작업으로 인한 중복 시간을 감소시키고, 다이어그램들의 UI(User Interface)를 통합하여 좀 더 쉽게 사용할 수 있도록 새롭게 정리한 Refactoring기법을 말한다. PI-Refactoring 다이어그램을 간단하게 그림으로 나타내면 그림 2와 같다[13].

3. Refactoring 프로세스 설계

본 장에서는 2.2절과 2.3절에서 연구한 두 가지 객체지향 Refactoring 방법들의 장점들을 정리하여, 일반적이고도 실증적인 소프트웨어 재사용과 유지보수를 하기 위한 객체지향 Refactoring 프로세스를 제시하고, 객체지향의 특성인 상속성과 다형성 등을 이용하여 작업을 수행하는 세부 기법과 지침을 보인다.

표 1 Refactoring 프로세스의 단계별 용어

(a) 역공학 단계 : [Rv]=Reverse Engineering
(b) 대상선정 단계 : [Td]=Target Definition
(c) Refactoring 단계 : [Re]=Refactoring
(d) 최적화 단계 : [Op]=Optimization

객체지향 Refactoring 프로세스에서 표현되는 단계 및 기법들을 위해 본 연구에서는 표1의 용어들을 정의

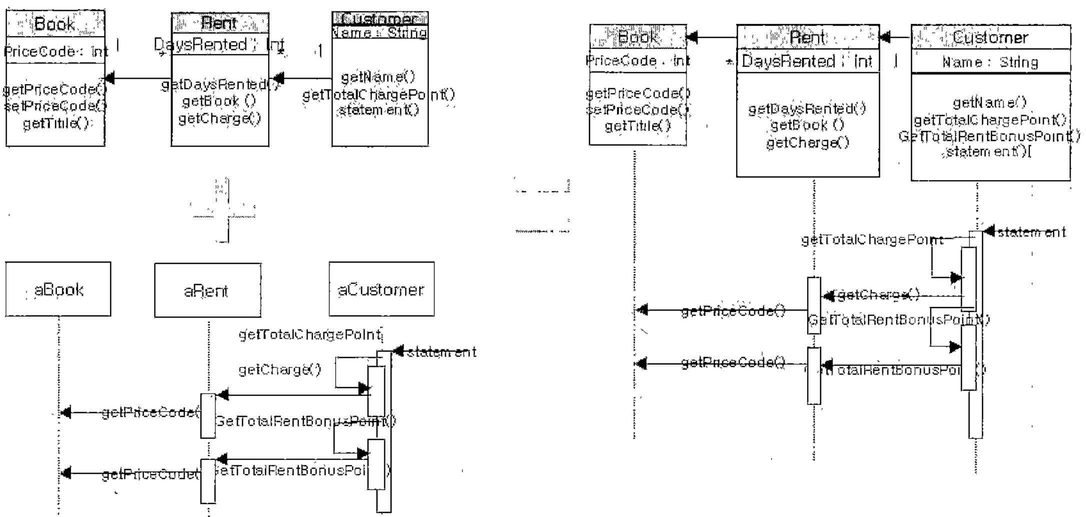


그림 2 PI-Refactoring다이어그램 모델

하고, 사용한다.

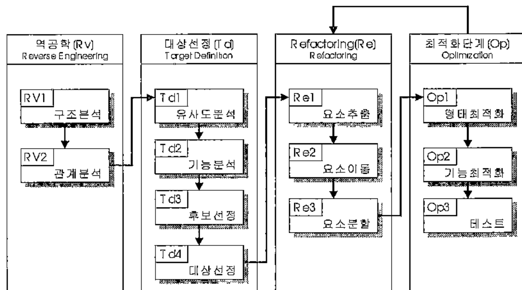


그림 3 Refactoring 상세 프로세스

그림3은 객체지향 Refactoring의 각 단계별 수행과정을 나타내는 상세 프로세스를 나타내고 있다.

3.1 역공학 단계 [Rv]

역공학 단계에서는 구조와 관계 등을 분석하여 Refactoring할 수 있는 근거 자료들을 분석해 내는 단계이다. 즉, 클래스 내의 변수 및 함수의 형태 와 이들의 연관 관계 등을 분석하고, 이 자료들을 문서화하여 정리하는 단계이다.

- [Rv1] 구조 분석 : 각 도메인 별 소속된 요소들을 알아내고 정리
- [Rv2] 관계 분석 : 각 요소들 간의 관계를 나타내는 클래스 다이어그램 생성

3.2 대상선정 단계 [Td]

역공학 단계에서 찾아진 클래스와 메소드(method)들의 분석 자료를 가지고 Refactoring 작업 범위를 선정하고, 이들에 대한 기능이나 형태들을 문서로 정리하여 문서화하는 단계이다.

- [Td1] 유사도 분석 : 역공학 도구인 McCabe를 이용하여 유사도 평가 요소와 기준을 제정, %로 표현한다. McCabe에서는 90~100% 유사도 측정치를 Very High Similarity라하고, 75~89%의 유사도 측정치를 High Similarity라 한다.
- [Td2] 기능 분석 : 실제로 수행되어지는 기능을 분석
- [Td3] 후보 선정 : 유사도 분석[Td1] 결과에서 유사도 측정치가 High Similarity인 75% 이상을 후보로 선정
- [Td4] 대상 선정 : 후보 목록에 있는 자료들 중에서 기능 분석[Td2]에서 분석된 유사한 기능을 수행하는 클래스와 메소드에 대해 Refactoring 작업을 수행할 대상으로 선정

3.3 Refactoring 단계 [Re]

시스템 성능 향상을 위해 대상 요소들을 변경하는 단계이다. 중복 요소 삭제, 메소드와 속성들의 이동, 독립 클래스로의 분할 등의 작업들이 이루어진다. 즉, 각각의 요소들을 다른 클래스로 옮기거나 중복된 요소들을 삭제하는 단계이며, 메소드와 속성들이 더 많이 사용되는 함수를 찾아 기능별로 이동시킨다. 함수를 이동한 후 이전 함수에서 호출(Call)하는 방법이다.

이 단계에서 PI-Refactoring 다이어그램을 사용하여 작업의 효율을 높이고, 기존 소프트웨어 시스템에 대한 이해를 향상시킬 수 있다.

- [Re1] 요소 추출 : Refactoring 작업을 수행할 대상의 요소들을 추출하여, 기능과 형태를 추상화하고, 단위별 추상화 수행 보고서를 작성
- [Re2] 요소 이동 : 중복 및 유사 요소들을 삭제하거나 다른 클래스로 이동, 기능별로 더 많이 호출되는 클래스로 이동
- [Re3] 요소 분할 : 기능별 독립 클래스로 분할, 슈퍼 클래스 생성

3.4 최적화 단계 [Op]

Refactoring되어진 클래스와 요소들을 최적화시키는 단계이다. Refactoring되어진 함수들의 기능이 정상적으로 작동하는지의 여부와 추가적으로 향상시키려 한 기능적 요소들이 성공적으로 작업이 되었는지를 테스트하는 단계이다. 또한 분할되어진 함수의 메소드와 속성들을 자연스럽게 함수별로 정리하고, 수정하게 된다.

- [Op1] 형태 최적화 : 통합/분할되어진 클래스의 메소드와 속성들을 자연스럽게(Clean up) 정리
- [Op2] 기능 최적화 : 기능이 정상적으로 작동하는지의 여부 검사
- [Op3] 테스트 : 추가적으로 향상시키려 한 기능적 요소들이 성공적으로 작동되는지를 테스트

4. 객체지향 기반의 Refactoring 프로세스 적용

4.1 적용환경

본 절에서는 D사에서 개발한 윈도우 환경 시스템 개발도구 인 *D2D*의 성능 향상을 위해 이번 연구에서 제시한 객체지향 Refactoring 프로세스를 적용하여 그 설계의 장단점을 검토하고 그 결과를 측정하였다. *D2D*는 C/S 환경에서 현재 사용되고 있는 시스템으로, 서버(Server)측은 C로, 클라이언트(Client)는 Visual C++로 개발되었으며, Refactoring의 대상으로 잡은 클라이언트 코드만 해도 300여 개의 클래스와 4000여 개의 메소드를 가지고 있는 방대한 시스템이다. 하지만, 다음과 같

은 문제점이 존재하고 있었다.

- 첫번째, 체계적인 방법론에 의해 개발되지 않아 방대한 규모에도 불구하고 설계 문서 및 클래스 다이어그램 또는 계층(Hierarchy)이 존재하지 않아, 재사용을 위해 기존 클래스를 사용하려 해도, 어떤 기능들이 존재하는지 파악하기가 어려운 상태였다.
- 두 번째, 컴포넌트 별로 여러 작업 그룹이 개별적으로 클래스를 작성하였기 때문에, 클래스간 유사한 기능이 여러 컴포넌트에 중복되어 존재하고 있었다.
- 그리고, 세 번째로 메소드의 이름이 동일하면서 매개변수(Parameter)가 유사하여도 전혀 다른 기능을 수행하는 것이 많았다.

이러한 문제점으로 인하여 기존의 *D2D*시스템을 새로운 환경변화에 따라 기능을 추가하거나 변경하기 위하여 일련의 유지보수를 진행하거나 기존의 클래스를 재사용하고자 할 때 어떠한 클래스를 이용해야 할 지 모르고, 또 한 클래스를 변경시켰을 때 미치는 영향을 파악하기 어려워 작업을 쉽게 진행할 수 없었고, 새로운 클래스를 개발하는 것이 오히려 개발기간 및 비용 측면에서 효율적인 상태에 있었다.

이러한 문제를 해결하기 위하여 본 사례연구에서는 중복된 기능이면서 여러 클래스에 분산되어 존재하는 메소드들을 한 클래스로 슈퍼 클래스화하며, 동일한 이름의 메소드는 동일한 기능을 하도록 변경하는데 그 작업의 초점을 두어, 전체 코드의 축소와 역공학을 통한 설계문서의 도출을 통해 유지보수성과 재사용성을 향상시키고자 하였다.

그림4는 객체지향 Refactoring 프로세스의 각 단계별 수행작업과 산출물을 나타낸다.

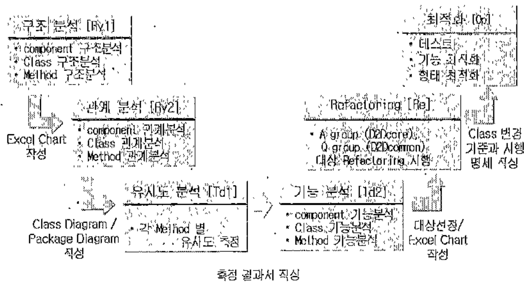


그림 4 각 단계별 수행작업과 산출물

4.2 역공학 단계 [Rv]

Refactoring의 대상을 식별하기 위하여 그 대상이 되는 소스의 전체적인 구조와 관계를 파악하는 단계이다.

4.2.1 [Rv1] 구조분석

Rational의 Rose2000 도구의 Reverse기능 을 통해 역공학 분석을 시행하고, 이를 통해 클래스의 구조적인 측면인 각 컴포넌트 별 클래스와 이에 포함된 메소드 이름, 매개변수(Parameter), 형태 등의 요소들을 분석하여, 이를 그림5에 보이고 있다.

D2D analysis Chart<Attributes>

D2D Refactoring Project 대신정보통신(주)

Component Name	Class Name	Methods			기 타
		Access Specifier	Name	Type	
AS2DGN	ADgnGridSct				derived from CDialog
	AEventDisplay	Public	m_stxEventData	CString	derived from CWnd
		Private	m_stxTempBuf	CString	
	APropextyView	Public	m_hinsPropextyInstance	HINSTANCE	derived from CWnd
			m_stxProperty	CString	
			m_stxPropName	CString	
			m_iCuxEvent	int	
			m_iCuxIndex	int	

그림 5 구조 분석 산출물

4.2.2 [Rv2] 관계분석

앞에서와 같이 Rational의 Rose2000에서 제공하는 Reverse 기능을 통해, 각 클래스간의 관계(Inheritance, Dependency)를 추출한다. 전체적인 클래스 간의 관계와 조합(Composition)과 상속이 집중되어 있는 클래스를 추출한다.

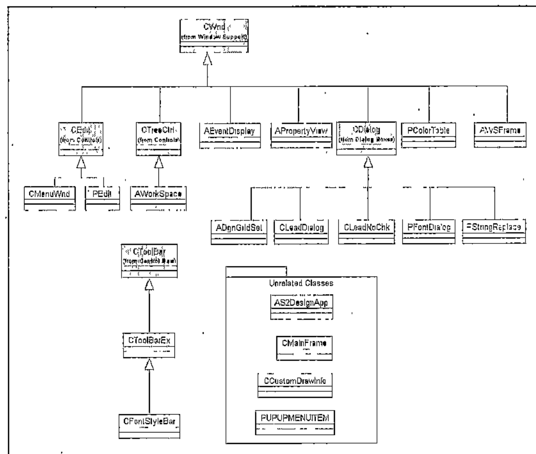


그림 6 D2Dcommon그룹의 관계분석(계층구조) 산출물

그림 6과 그림7은 두 그룹의 계층구조를 분석한 관계분석 산출물이며, Refactoring 작업 전의 계층구조를 보여준다.

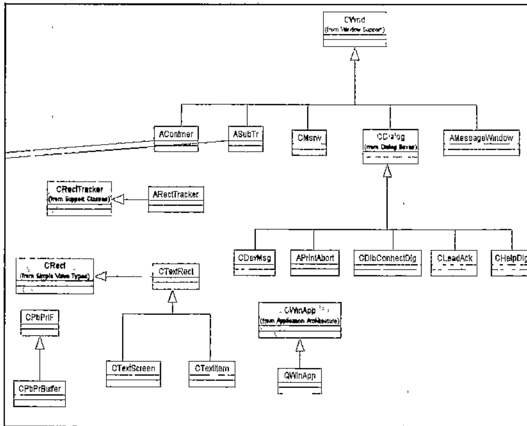


그림 7 D2Dcore그룹의 관계분석(계층구조) 산출물

(a) 단위별 원시코드 분류 (b) 기능 분석 보고서

그림 9 기능 분석 산출물

4.3 대상선정 단계 [Td]

Refactoring을 수행하기 위한 대상을 선정하는 단계로, 역공학 단계에서 분석된 자료들의 기능과 유사도를 분석하고, 이들 중에서 Refactoring 후보를 선정하여 실제 Refactoring작업을 수행하는 대상들을 결정한다.

4.3.1 [Td1] 유사도 분석

역공학 도구인 McCabe사의 McCabe 6.0도구를 사용하여, 전체 소스 중 형태가 유사한 함수를 추출한다. 이 과정에서 유사도가 McCabe 유사 복잡도 기준으로v(g) 100%~80%에 해당하는 것을 가지고 작업을 시행한다. 그림8과 같은 산출물에서 대부분의 유사한 메소드들이 D2Dcommon과 D2Dcore 컴포넌트에 집중되어 있음을 확인할 수 있다.

● Source Component : D2DCommon, Class : AAPPObj, Method : CreateTrWindow

Component	Class	Method	%	P/C	P/T	Comments	Method	Local	Return	Type	Function	+	-	Check
D2DCore	CMemoryData	ReadToken(char*)	75	X		X	X	X	X	X	X		O	
D2DCore	AListRow	GetPropertyTitle()	75	1	1	X	X	O	X	X	X		O	
D2DCore	CStream	GetDataPtr()	75	X	0	O	O	O	O	O	O		O	

※ P/C . Parameter Count, P/T · Parameter Type,

그림 8 유사도 분석 산출물

4.3.2 [Td2] 기능 분석

유사도 분석[Td1]단계에서 유사한 형태를 가진 메소드들의 기능을 분석, 비교하는 단계이다. 그림 9는 기능 분석을 하기 위해 재분류하여 문서화한 작업 범위 소스코드 산출물과 그 소스코드들이 실제로 수행하고 있는 기능들을 분석하여 문서화한 보고서를 나타낸다.

4.3.3 [Td3] Refactoring 후보 선정

유사도 분석[Td1] 결과에서 유사도 측정치가 High Similarity인 75% 이상인 것 중에서 기능 분석[Td2]결과 유사한 기능을 하는 메소드를 선택하여 Refactoring을 시행할 대상 후보로 선정한다.

4.3.4 [Td4] Refactoring 대상 선정

이번 사례 연구에서는 D사의 *D2D* 시스템 중 가장 핵심이면서 비주얼(Visual) 컴포넌트의 집합인 D2Dcommon 컴포넌트와 D2Dcore 컴포넌트를 그 대상으로 선택하였다. 두 컴포넌트는 각각 18개의 클래스를 가지고 있으며, D2Dcore 클래스의 모든 클래스는 D2Dcommon에 포함된 유사한 이름의 클래스와 1:1의 의존 관계가 존재하였다.

D2Dcommon 클래스는 MFC에서 제공하는 기본 컨트롤을 API와 MFC 클래스를 이용하여 추가 기능을 더해 자체적으로 생성한 클래스들이며, D2Dcore의 클래스는 화면 설계 도구 생성에 필요한 기능을 수행하도록 작성되어 있었다. 하지만, 두 클래스의 메소드들이 유사한 모양과 기능을 가진 것이 많아 이를 Refactoring의 대상으로 선정했다.

그림 10은 Refactoring 작업을 위한 후보 선정 목록과 수행 대상 선정을 나타낸 그림이다.

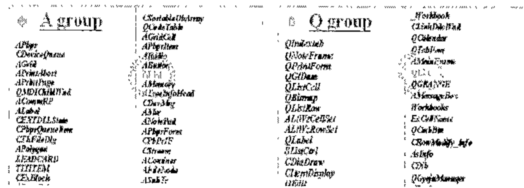


그림 10 Refactoring 후보 목록과 Refactoring 대상 선정

4.4 Refactoring 단계 [Re]

4.4.1 [Re1] 요소 추출

후보 클래스로 선정한, D2DCommon과 D2DCore의 36개 클래스와 1196개의 메소드 가운데, 유사한 모양을 가지면서, 기능적 유사도가 높은 것(Alist와 Qlist)을 선택하여 동일한 기능과 모양을 갖도록 추상화한다.

4.4.2 [Re2] 요소 이동

한 컴포넌트의 클래스들이 특정 컴포넌트의 클래스를 과다하게 호출하여 사용하는 경우, 호출을 받는 클래스 및 클래스 군을 호출하는 컴포넌트로 이동한다.

그림 11은 프로세스의 작업을 수행하는 과정을 나타낸 그림이다.

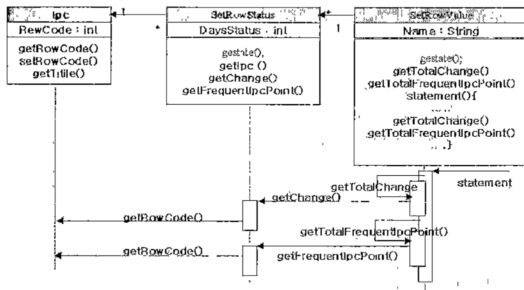


그림 11 Refactoring 작업 수행 요소 이동

4.4.3 [Re3] 요소 분할

한 컴포넌트 내에서, 동일한 기능 및 형태를 가진 메소드를 한 클래스로 이동하여 통합한다. 또 이러한 메소드가 많은 클래스끼리는 슈퍼 클래스로 도출하여 이를 상속받아 사용하도록 처리한다. 또한 모양만 동일하고, 내부 알고리즘(Algorithm)이 달라야 하는 경우 상위 클래스에서 추상함수로 정의하고 이를 상속받은 클래스에서 내부 알고리즘을 구현할 수 있도록 하여 차후 확장성을 가지도록 처리한다.

그림 12는 요소들을 통합하여 새로운 클래스로 분할하는 과정을 도식화하여 나타낸 것이다.

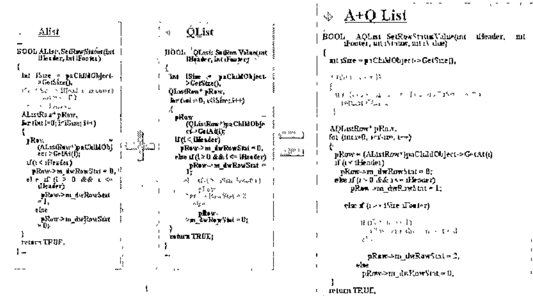


그림 12 요소 분할

표 2는 요소 이동 및 분할 단계[Re3]에서 진행하는 작업의 세부 지침을 표현하고 있다.

4.5 최적화 단계 [Op]

이 단계에서는 Refactoring Activity를 행한 Alist 와 Qlist를 중심으로 메소드와 변수들을 최적화시킨다. 본

표 2 Refactoring Activity

Refactoring Activity	Pre Condition	세부 작업 지침
컴포넌트 간 클래스 이동	• A 컴포넌트 그룹의 클래스들이 B 컴포넌트 그룹의 특정 클래스 또는 클래스 계층을 집중적으로 호출하여 사용하는 경우	• A 컴포넌트로 B 컴포넌트 그룹의 클래스 및 클래스 계층을 복사한다. • A 컴포넌트의 클래스들에서 호출관계를 수정한다.
슈퍼 클래스로 통합	• 동일한 컴포넌트 그룹 내에서 상속관계 또는 조합 관계에 있는 클래스 A, B가 있다면, 상속받은 B 클래스가 별다른 기능을 하지 않는 경우	• B 클래스의 구성요소와 메소드를 A 클래스로 이동하고 통합한다. • B 클래스를 삭제한다. • B 클래스와의 직접적인 호출관계에 있던 다른 클래스와의 호출관계를 A 클래스로 수정한다.
슈퍼 클래스 도출	• 동일한 컴포넌트 그룹 내에 있는 호출관계를 가지지 않은 A, B 클래스가 존재하고, 이 두 클래스 의 메소드가 다수 기능적으로 유사한 경우	• A 클래스와 B 클래스의 유사한 메소드의 공통부분을 슈퍼 클래스에 생성시킨다. • A 클래스와 B 클래스를 새로 생성한 슈퍼 클래스 C로부터 상속 받도록 관계를 수정한다. • 통합한 메소드에서 공통부분이 아닌 개별적으로 구현해주어야 하는 기능은 A 클래스와 B 클래스에서 상속하여 생성한다.
상위 클래스로 메소드 이동	• 동일한 컴포넌트 그룹 내에서 상속관계 또는 Aggrega- tion 관계에 있는 클래스 A, B가 있고, Sub 클래스 B의 메소드가 A 클래스로 옮기는 것이 A 클래스의 완전성을 향상시키는 경우	• 다른 컴포넌트 그룹에 있는 클래스의 특정 메소드를 사용하기 위해 빈번한 호출이 있는 경우 • 해당 메소드를 복사하여 옮긴다.
상위 클래스에 가상함수 생성	• 동일한 컴포넌트 그룹 내에 있는 여러 클래스 내에 동일한 Signature를 가진 메소드가 존재하지만, 모두 다른 Algorithm을 가진 경우	• 상위 클래스에 가상함수를 선언하고 이를 상속 받은 하위 클래스에서 개별적으로 Algorithm을 구현한다.
일반 클래스 생성	• 동일한 컴포넌트 그룹 내에 있는 여러 클래스들이 Type만 다르고, 메소드 및 구성요소가 거의 유사한 경우	• 상위에 일반 클래스를 생성하고, 상속관계를 맺어준다.

단계에서는 기능과 형태적인 측면에서 Refactoring의 수행 결과를 확인하며, 또한 이의 최적화가 진행된다. 본 사례연구에서는 기능과 형태에 대한 결과 확인에 초점을 맞추어 진행한다. 변경/분할되어진 함수의 메소드와 변수들을 클래스별로 정리하고 Refactoring 되어진 함수들의 기능이 정상적으로 작동하는지의 여부와 추가 기능 요소들이 성공적으로 작업되었는지를 테스트하는 과정으로 구성된다.

4.5.1 [Op1] 형태 최적화

통합/분할되어진 두 클래스 그룹의 메소드와 변수들을 정리한다(Clean up).

표 3은 두 요소들의 통합 작업 후 형태적인 면을 올바른 기능이 되도록 설정해 주는 작업이다.

표 3 Alist(a)와 Qlist(b)의 통합

(a) 기능 : 각 하위 Row Size을 확인 후 개체의 Header 또는 Footer의 정보를 0 또는 1로 설정 (b) 기능 : 각 하위 Row 개체의 Header 또는 Footer의 정보를 0 또는 1로 설정

또한 형태의 최적화를 위해 컴포넌트 간 클래스 이동, 가상함수와 제네릭 클래스 생성에 대한 적절성을 확인하고, 검증 하였다. 그 결과를 5.2 클래스의 완전성 증가에 나타내었다.

4.5.2 [Op2] 기능 최적화

Refactoring을 통해 변경된 클래스의 기능이 정상적으로 동작하는지를 검사한다. 표 4는 두 그룹의 통합으로 인한 공통 기능 수행 문제를 해결하기 위해 기능적인 정보를 설정해 주는 작업이다.

표 4 AQlist 기능 최적화

각 하위 Row Size을 확인 후 개체의 Header 또는 Footer의 정보를 0 또는 1이나 2로 설정

또한 기능의 최적화를 위해 동일한 컴포넌트 그룹 내의 클래스 간 수퍼 클래스로의 통합, 여러 컴포넌트 그룹간 수퍼 클래스 도출, 상위 클래스로의 메소드 이동을 시행하고, 이에 대한 적절성을 확인하고, 검증하였다. 그 결과를 5.2 클래스의 완전성 증가에 나타내었다.

4.5.3 [Op3] 테스트

추가적으로 향상시키려 한 기능적 요소들이 성공적으로 작동되는지를 테스트한다. 또한 전체 Refactoring 프로세스를 통해 수정된 컴포넌트 및 클래스의 성능 향상을 검증하기 위하여 각 컴포넌트 별로 실행 속도(객체 생성속도) 및 실행 모듈 크기 등의 성능 향상을 검증하였다. 그 결과를 5.3 실행 속도 증가에 나타내었다.

5. 성능평가

본 장에서는 논문에서 제시한 프로세스의 성능 평가를 위해 2장에서 제시한 기존 Refactoring 프로세스들과 비교하고, 실제 작업 결과 및 성능 향상 테스트의 사례를 기술한다.

5.1 기존 프로세스와의 비교

다음의 표5에 기존 Refactoring 프로세스 연구와의 비교를 보이고 있다. 기존의 연구가 객체지향의 일부 특성을 부각하여 기존 코드를 정제하는 기법에 지나지 않는 반면 본 논문에서는 Reengineering의 측면에서 각 단계와 세부 지침을 제시하여 사용자가 실무에서 쉽게 적용할 수 있도록 하였다. 또한 대상 시스템을 다양한 관점에서 Reverse할 수 있는 기법들을 제시하여 기존 연구에서 클래스들의 관계에만 집중하여 범할 수 있는 오류를 극복하도록 하였다.

표 5 기존 프로세스와의 비교

Process	본 논문에서 제시하는 프로세스	기존 프로세스	
		Demeyer & Ducassse	Fowler
핵심 Diagram	Class Diagram Sequence Diagram Component Diagram	Class Diagram	Class Diagram
주요 Refactoring 기법	Polymorphism Inheritance	Sub Classing	Method Moving
대상코드 선정 Level	Project	Class	Method
프로세스의 적용성	Easy	Difficult to find target Source	Difficult to Determine Iteration Phase
Reengineering Process의 완전성	전체 프로세스 제시	기법 수준	기법 수준
유지보수성	High	Low	Low
대상 시스템에 대한 View	Behavior Relation Architect	Relation	Relation

5.2 클래스의 완전성 증가

본 논문에서 제시한 프로세스를 통해 객체지향에서 제공하는 여러 장치들을 기존 소프트웨어 시스템의 정제를 위해 적용하여, 클래스에서 충분히 가져야 할 속성들을 갖도록 하였고, 불필요한 클래스 간의 관계를 삭제하여 정제된 객체지향 시스템으로의 성능 향상이 가능하였다. D2DCommon에 위치한 클래스 중 D2DCore에서 조합 방법으로 호출되던 3개의 클래스를 D2Dcore

로 복사(Replication)하였고, D2DCommon과 D2DCore 중 불필요하게 상속되어 사용되던 클래스를 상위 클래스로 통합하였다. 또한 동일한 컴포넌트 내에서 유사한 기능을 제공하는 메소드가 다수를 포함하는 클래스들을 슈퍼 클래스를 도출하여 이를 상속받도록 처리하였으며, 상속의 원칙상 상위 클래스로의 이동이 필요한 메소드도 처리를 하였다. 그리고, 같은 모양을 가지면서도 다른 알고리즘을 가져야 하는 메소드는 가상함수로 상위 클래스에 선언을 하고, 이를 하위 클래스에 구현을 하였으며, 클래스간 형태가 달라야 하면서 형태는 동일한 클래스를 일반 클래스로 생성하였다. 표 6은 Refactoring 작업을 수행한 내용을 보여준다.

표 6 Refactoring 수행 작업 표

작업 내용	D2Dcore	D2Dcommon
컴포넌트간 클래스 이동	D2Dcommmond에서 D2Dcore (3)	0
슈퍼 클래스로 통합	3	5
동일한 컴포넌트 내에서 유사한 메소드를 다수 포함하여 슈퍼 클래스로 도출	2	3
상위클래스로 메소드 이동	34	21
상위 클래스에 가상함수로 생성	21	12
일반 클래스 생성	2	2

5.3 실행 속도 증가

*D2D*는 실시간으로 1일 평균 수백만 건의 Data를 처리하는 GUI를 지원하는 도구로, 실행 속도가 매우 중요한 요소로서 소프트웨어의 실행 모듈의 크기가 작으면서 객체의 생성속도가 빨라야 할 필요성이 제기되고 있었다.

객체의 생성 속도를 평가하기 위해서, 본 테스트에서는 Intel Pentium III 700Mhz CPU와 256M RAM을 장착한 PC 환경에서, 수정한 Qedit 클래스와 Alist 클래스에 대한 각각 1000개의 생성속도로 이를 평가하였다. 표7은 Refactoring 작업 수행 전후의 크기변화와 속도변화에 대한 비교 값을 나타낸다.

표 7 Refactoring 작업의 수행 전후 비교

작업 내용	수정 전		수정 후	
	D2DCommon	D2Dcore	D2DCommon	D2Dcore
실행모듈 크기 (DLL Size)	913k	1529k	809k	813k
객체 생성 속도 (Qedit, Alist)	5.129ms	3.273ms	4.725ms	3.187ms

5.4 다단계 작업의 통합

기능적인 면에서의 Refactoring 작업을 수행할 때 사용하는 순차 다이어그램과 형태적인 면에서의 Refactoring 작업을 수행할 때 사용하는 클래스 다이어그램을 하나의 PI-Refactoring 다이어그램으로 통합하여 작업함으로써 사용자가 분석된 자료를 가지고 클래스 다이어그램과 순차 다이어그램을 중복하여 만들거나 찾는 번거로움을 줄일 수 있다는 것이다.

즉, 구조의 형태적인 측면에서의 Refactoring작업을 수행하는 UML 다이어그램인 클래스 다이어그램과 기능적인 측면에서의 Refactoring 작업을 수행하는 순차 다이어그램 각각의 작업을 PI-Refactoring 다이어그램 모델링 기법을 사용하여 통합하고 이들 중복 작업의 통합으로 작업 수행 속도의 향상을 얻을 수 있다.

5.5 사용자의 이해력 향상

PI-Refactoring다이어그램의 사용으로 각각 독립적으로 중복되어 반복작업을 하던 다이어그램들의 통합으로 UI 가 간결하게 정리되어 사용자의 작업이 매우 간단하고 쉽게 처리되므로 이해력을 향상시킬 수 있다.

단, 사용자의 이해도를 향상시키는 UI를 가진 PI-Refactoring 다이어그램도 한 단계의 과정 안에서 처리하는 작업이 제한이 되기 때문에 지금까지는 대규모 프로젝트에서는 유용하지 못한 단점을 나타내고 있으나 자동화 도구의 개발 연구가 병행되어 완성되어진다면 이러한 문제점은 해결될 것이다.

5.6 재사용성 및 확장성 증가

이번 연구에서 진행한 역공학 단계의 산출물을 통하여 기존 소스 코드의 전체 구조 및 관계, 기능을 파악할 수 있어, 차 후 재사용을 위해 어떠한 클래스 및 컴포넌트를 사용해야 하는지 판단할 수 있었고, 클래스의 완전성 증가로 차 후 확장이 용이하게 되었다.

역공학 단계의 분석 문서나 객체 단위를 저장소나 데이타베이스의 구축으로 재사용시에 유용하게 사용할 수 있게 된다. 기존 소스코드의 재사용 단위가 객체단위의 재사용으로 확장되었으며, 앞으로의 재사용과 유지보수에 있어서 최적화된 시스템은 재사용 요소들의 저장소를 추가로 보유하게 되는 효과를 볼 수 있다.

6. 결론 및 향후 연구과제

이번 연구에서는 객체지향 소프트웨어를 정제하고, 재사용성을 향상시키기 위해 객체지향 Refactoring의 핵심적인 프로세스를 정의하고, 단계별 기법과 모델을 정리한 PI-Refactoring모델링 기법을 적용하여 각 단계별 적용을 보였다. 이를 통하여 소프트웨어 유지보수와 재

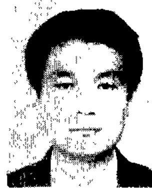
사용을 위해 좀더 정형적이며 경제적으로 Refactoring을 적용할 수 있게 되었다. D사의 *D2D* 시스템에 적용해본 결과 클래스의 완전성이 증가하였고, 실행속도가 증가하는 등의 성능향상의 효과를 확인할 수 있었다.

소프트웨어의 규모와 유지보수 비용이 날로 증가하고 있기 때문에, 새롭게 발생되는 소프트웨어 위기를 극복하는 방안으로 객체 또는 컴포넌트의 재사용의 중요성이 더욱 부각되고 있다. 그러므로 앞으로는 Refactoring 과정에서 찾아지는 요소들을 단위별, 기능별 데이타베이스화시키는 재사용 분석요소의 저장소 개발 기술과 데이타베이스 화 기술이 더욱 연구되어야 하며, 각각의 프로세스 단계별 상세과정이 자동화 도구로 개발되어, 프로토타입(Prototype) 모델을 사용하여 실생활에 적용할 수 있는 소프트웨어의 구현도 더욱 연구되어야 할 것이다.

특히, 재사용의 관심이 높아지고 있는 컴포넌트의 기술들을 적용하여, 기능 단위의 분할이나 이동 등을 좀더 쉽게 할 수 있고, 컴포넌트화 시키는 객체지향 컴포넌트 추출과 컴포넌트화 자동화 도구의 개발 분야의 연구가 더욱 진행되어야 할 것이다.

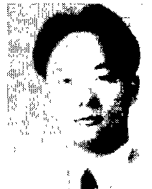
참 고 문 헌

[1] 권오천, 신규상, 역공학 및 재공학의 기술동향, 한국정보과학회, 소프트웨어공학회지, pp.6-21, 1999년 3월호.
[2] Ivar Jacobson, Martin Griss, Patrik Jonsson, Software Reuse, Addison-Wesley, 1997.
[3] 최순규, 이명재, 우치수, 3R (역공학, 재공학, 재사용), 한국정보과학회, 소프트웨어공학회지, pp.16-34, 1995년 3월호.
[4] Martin Fowler, UML Distilled, Addison-Wesley, 1999.
[5] Martin Fowler, Refactoring-Improving the Design of Existing Code, Addison Wesley Longman Inc., 1999.
[6] Bruce Eckel, Thinking in Patterns with JAVA, Revision 0.3, 2000.
[7] Roger S. Pressman, Software Engineering A Practitioner's Approach, 4th Edition, McGraw-Hill, 1997.
[8] Scott Tilley, A Reverse-Engineering Environment Framework', Carnegie Mellon Software Engineering Institute, April 1998.
[9] Nicolas Anquetil and Timothy Lethbridge, Extraction Concepts from File Names; a New File Clustering Criterion, IEEE, 1998.
[10] Serge Demeyer, Stephane Ducassse, Object-Oriented Reengineering, OOPSLA, 1999.
[11] Martin Fowler, Refactoring: Improving the Design of Existing Code, OOPSLA, 1999.
[12] 박진호, 김수동, 류성열, UML 다이어그램들 간의 일관성 검증방법, 한국정보과학회, 춘계학술발표논문집, 1998.
[13] 박진호, 이종호, 류성열, 소프트웨어 유지보수와 재사용을 위한 재공학 Refactoring 기법 연구, 한국정보과학회, 춘계학술발표논문집, 2000.



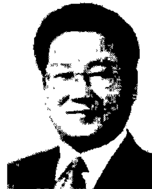
이 종 호

1985년 숭실대학교 전자계산학과 졸업(공학사). 1998년 숭실대학교 정보과학대학원 졸업(공학석사). 1998년 ~ 현재 숭실대학교 대학원 전자계산학과(박사과정). 1994년 정보처리기술사. 1997년 정보처리시스템감리인. 관심분야는 리엔지니어링, 분산 객체 컴퓨팅, 소프트웨어 재사용, 소프트웨어 리팩토링, 컴포넌트 시스템 정보처리시스템감리



박 진 호

2001년 숭실대학교 대학원 전자계산학과(공학석사). 2001년 ~ 현재 숭실대학교 대학원 전자계산학과(박사과정). 관심분야는 리엔지니어링, 소프트웨어 리팩토링, 소프트웨어 재사용, 소프트웨어 유지보수



류 성 열

1997년 아주대학교 컴퓨터학부(공학박사). 1997년 ~ 1998년 George Mason Univ. 교환교수. 1981년 현재 숭실대학교 컴퓨터학부 교수. 1998년 현재 숭실대학교 정보과학대학원 원장. 1998년 ~ 현재 숭실대학교 전자계산원 원장. 관심분야는 리엔지니어링, 분산 객체 컴퓨팅, 소프트웨어 재사용