□사례발표□

소프트웨어 유지보수와 리엔지니어링 기술 동향

최 은 만†

목 차

1. 노후 시스템

최근 기업 내에서 일어나는 여러 활동 및 조직들을 다시 생각해 보고 개선해 나가는 BPR(Business Process Reengineering)에 대한 관심이 고조되고 있다. 기업 조직의 규모가 거대해지면서 고객의 요구나 기술의 변화에 대하여 대처할 수 있는 의사결정과 처리과정이 효율적으로 신속하게 이루어지지 못하고 있다. 이는 업무의 흐름이 조직론, 노후 기술에 의존하여 설계되어 변화된 환경에 맞지 않는 시스템이 되어 버렸기 때문이다. 결과 중심으로 비지니스 프로세스를 재정비하고 기업을 재조직하여 체질을 개선하는 비지니스 리엔지니어링은 기업의 생산성 향상을 약속하고 있다.

동일한 개념으로 소프트웨어 시스템도 노후 시스템(Regacy System)으로 변하고 있다. 2~30 년전 어셈블리 언어나 3세대 언어로 쓰여진 온래된 시스템, 따라서 구조나 동작의 이해가 쉽지 않지만 기업 내에서 아직도 중요한 역할을 하는 시스템을 노후 시스템이라고 한다. 개발된 후 오랜 시간이 경과하여 사용된 언어나, 코딩 스타일, 사용자 인터페이스, 자료처리 방식 등이 이미 노후된 기술이며, 반복되는 유지보수에 의하여 소프트웨어의 구조가 망그러지고 문서화마저 잘되지 않은 시스템들이 많아지고 있다. 하지만 시스템을 파기하고 새로운 기술로 시스템을 구성하기에는 매우 많은 인력과 재원을 재투자하여야 한다. 예를 들어 미국방성의 1,700개 데이타 센터에 있는 20개의 급여 시스템은 연간 9천억불의 정보 기술 관련 비용이 드는 대규모 시스템이다[1]. 기능적으로 중복되어 있고 유사 정보가 다른 형태로 제공되는 비효율적인 면이 있지만 관련 부서에 중요한 정보를 제공해 주는 중요한 역할을 하기 때문에 전면 파기하고 새로운 시스템을 구축하는 것은 많은 자원이 낭비된다.

노후 시스템을 계속 고집하면 다음과 같은 난관에 빠지게 된다. 2~30 년전의 소프트웨어 설계 방식은 현재와 큰 차이가 난다. 소요 기억공간을 최소화하기 위하여 변수를 중복 사용하거나 광역변수 하나로 통합하는 경우가 있다. 또한 이해하기 쉬운 코드보다는 실행 효율을 중요하게 생각하였다. 이러한 설계 방식이 유지보수와 테스트를 어렵게 하고, 소프트웨어 구조를 망그러뜨리며 소프트웨어의 이해를 어렵게 만

† 정회원 : 동국대학교 컴퓨터공학과 교수

든다. 따라서 노후 시스템의 유지보수 활동의 대부분은 프로그램을 이해하는 것이 된다. 노후 시스템은 대부분 10만줄 이상의 대규모 소프트웨어이기 때문에 유지보수가 쉽지 않다.

노후 시스템은 기술적인 문제뿐만 아니라 관리적인 문제도 따른다. 보통의 프로그래머들은 노후 시스템의 유지보수보다는 새로운 시스템의 개발에 참여하기를 원한다. 인원의 부족뿐만 아니라 경우에 따라서는 노후 시스템에 사용된 기술에 능숙한 기술자가 없는 경우도 있다. 이렇게 기술적으로 취약함에도 불구하고 현재의 시스템은 사용 가치가 크다. 사용자들은 변화를 두려워하고 시스템을 완전히 새로운 기술로 재개발하여 교체하는 것도 많은 위험 요소가 따르기 때문에 노후 시스템을 고수하려고 한다. 한편으로는 노후 시스템의 유지보수는 점점 어려워지고 기능 향상 요구도 만족시키지 못한다면 소프트웨어 시스템으로서의 기능은 상실하고 말 것이라는 반대 의견이 따르는 것이다.

유지보수 비용이 증가하고 BPR에 의한 소프트웨어 기능 향상이 요구되면서 노후 시스템에 대한 관심이 증가되고 있다. 또한 새로 소개되는 기술들, 예를 들면 객체지향 파라다임이나 분산처리 기술들은 소프트웨어 구조를 모듈화하여 유지보수를 용이하게 한다. 이러한 최신 기술을 소개받으면서 이러한 기술들을 동원하여 사용자에게 신뢰받고 있는 시스템을 더욱 개선해 보려는 요구가 노후 시스템의 리엔지니어링이다.

이 논문에서는 노후화해 가는 소프트웨어 시스템을 유지보수하는데 사용되고 있는 최신기술들의 동향을 소개하고 평가한 후 소프트웨어를 단순히 유지하는 것이 아니라 재정비하고 혁신적으로 발전시켜 나가는 리엔지니어링 개념과 그 기술에 대하여 소개한다.

2. 최신 유지보수 기술

유지보수에 대한 문제와 원인에 대한 여러 차례의 조사[2][3][4]에 의하면 유지보수 작업의 관리 부재, 조직 환경의 미비, 개인의 기술 부족, 대상 시스템이 안고 있는 문제들이 중요한 원인으로 인식되고 있다. 최근의 조사[5]에 의하면 제일 공감하고 있는 유지보수 작업 도중에 일어나는 문제점은 작업 우선순위가 자주 바뀌므로(Change priorities) 현재 진행되고 있는 유지보수 작업이 새롭게 중요하게 인식된 수정 요청에 의하여 중단된다. 새로운 수정 요청에 대한 처리를 위하여 문제의 파악부터 프로그램의 이해, 수정되어야 할 부분을 찾아내 수정한 후 다시 원래의 일로 되돌아 가 중단된 작업을 계속하려 할 때 연결되지 않는 경우가 많다. 이 문제는 형상관리(Configuration Management) 활동이 없거나 있더라도 잘 운용되지 못하기 때문이다.

다음은 테스트 방법이 적절하지 못하다는 문제이다. 유지보수 작업에서의 테스트는 개발 단계의 테스트와 매우 다르다. 개발 단계의 테스트는 프로그램의 전반적인 기능이나 프로그램의 모든 실행가능 경로를 점검해 보는 것이나 유지보수 단계의 테스트는 변경된 일부의 기능이나 경로를 테스트한다. 또한 유지보수 단계의 테스트는 매우 신속하게 이루어져야 한다. 따라서 변경된 부분을 중심으로 테스트하면서 나머지 부분이 변경에 대한 파급효과가 없는지 검사하는 리그래션 테스트가 필요하다.

결국 유지보수 단계에 가장 중요한 기술은 변경에 의한 파급효과 분석(Change analysis)과 리그래션 테스트이다. 이외에도 성능 측정의 어려움, 문서화의 미비 또는 부재, 비지니스 환경의 빠른 변화, 자료의 방대함 등이 있으나 지면 관계로 여기서는 자세히 설명하지 않는다. 소프트웨어가 방대해지며 복잡해질수록 소프트웨어 이해가 유지보수 작업의 성패에 큰 영향을 미친다. 따라서 유지보수 단계에서 중요한 기술 동향인 소프트웨어의 이해를 위한 역공학 기술을 설명한다. 또한 최근 ISO 9000 시리즈가 발표되면서 소프트웨어 품질에 대한 관심이 높아지고 있는데 ISO 9000이 유지보수 활동에 미치는 영향에 대하여 다룬다.

2.1 파급효과 분석

소프트웨어를 수정한 후 다른 부분에 영향을 주지 않는지 다음과 같은 세 가지 측면에서 살펴볼 필요가 있다.

- 수정이 바르게 되었는지 검증한다. 만일 소프트웨어의 변경이 여러 부분에 광범위하게 영향을 줄 가능성이 있다면 안전한지 여러 번 확인할 필요가 있다.
- 변경 후 다시 테스트할 필요가 있는 부분을 찾아낸다.
- 프로그램의 아주 중요한 기능을 담당하는 부분을 수정하였다면 그 파급효과가 매우 크므로 자세히 검증한다

소프트웨어의 분량이 커진다면 이상과 같은 작업은 자동화되어야 한다. 의존도 분석(Dependency analysis)이 소프트웨어 유지보수와 디버깅에 필요한 중요한 기술이다. 특히 의존도 분석 기술은 소프트웨어의 수정이 다른 부분에 어떤 영향을 미치는가를 잘 나타낸다. 모듈 내부의 의존도를 분석하는 방법은 코드 분할(Slicing) 방법이 많이 이용된다. 원래 코드 분할은 관심의 대상이 되는 변수 값을 변화시키는 부분들을 프로그램에서 분할 추려 내는 것을 말한다. 개념이 발전되면서 실행 경로에 의한 분할 방법도 제안되고 있다. 유지 보수 프로그래머가 많이 사용하는 의존도 분석은 호출 그래프에 나타난 모듈간의 관계이다. 유지보수 프로그래머들은 프로그램의 자세한 부분을 이해하는 것보다 모듈 단위의 동작, 즉 시스템을 기능별로 파악하려 한다. 따라서 모듈 사이의 관계 파악, 기능 중심 테스트에 적용될 수 있는 도구가 필요하다.

의존도 분석을 이용한 파급효과 분석은 여러 가지 접근 방법이 있다. 이를 대상, 모델, 분석 방법, 분할 방법, 의존 관계 저장 등의 기준으로 분류하면 다음과 표 1과 같다.

이제까지 연구 개발되거나 사용되는 의존도 분석 기술들을 이러한 기준에 의하여 살펴보면 다음의 표 2와 같다.

〈표 1〉 의존도 분석 방법

분 류 기 준	기 준 설 명	사 례
대상	의존 관계의 대상	원시코드의 객체, 문서, 논리 흐름도
의존 관계 분석 모델	의존 관계를 분석할 때 사용하는 모델	자료흐름도, 논리흐름도, 문자열, 객체 의존도
의존도 분석 방법	파급효과를 추적하는 방법(Tracing approach)	분할(Decomposition), 패턴매칭, 휴리스틱 탐색
저장 방법	의존 관계 및 그 대상을 저장하는 방법	RDBMS, 화일 시스템, 객체지향 데이타베이스
기능	저장된 대상과 관계를 보여주고 변경할 수 있는 기능	Load, Modify, Browse

〈표 2〉 파급 효과 분석 도구

분 석 도 구	대 상	모 델	분 석 방 법	저 장 방 법	기 능
Program slicer[6]	변수, 문장 등 프로그램 안의 객체	자료 흐름도 (Define-use graph)	Decomposition slice, program slice	화일 시스템	Load, Modify, Browse
Cross reference[7]	문자열	패턴 매칭	패턴 매칭	화일 시스템	〃
ALICA[8]	DOD-STD-2176	〃	휴리스틱 탐색	RDBMS	〃
SODOS[9]	문서 안의 객체	하이퍼텍스트	Traceability relationship	RDBMS, OODBMS	〃
Control flow analyser[10]	프로그램 객체	논리흐름도, 호출 그래프	논리흐름 의존	화일 시스템	〃

2.2 리그레션 테스트

리그레션 테스트는 유지보수 과정에서 수정되지 않은 부분이 예상하지 못한 영향을 받는지(이를 리그레션 오류라 함) 테스트하는 것이다. 즉 수정되지 않은 원래 소프트웨어의 기능이 잘 작동되는지 검증하는 과정이다. 리그레션 오류를 검출하려면 변경되지 않은 부분의 기능에 대하여 개발 당시 테스트 케이스들이 보관되어야 하는데 이를 리그레션 테스트 슈트(regression test suite)라 한다. 유지보수를 위하여 변경된 부분을 테스트할 수 있는 새로운 리그레션 테스트 슈트를 만들어주는 리그레션 도구가 필요하다.

리그레션 테스트 문제도 적용하는 대상과 목표가 다르다. 프로시듀어 안의 코드가 변경된 경우 단위 리그레션 테스트가 적용되며 새로운 프로시듀어가 추가되어 기존 프로시듀어와의 인터페이스를 점검하는 통합 리그레션 테스트가 적용된다. 소프트웨어 수정 후 기능 중심으로 리스레션 테스트를 할 수도 있다. 또한 변경 후 하드웨어를 포함하여 전체 시스템의 성능을 재점검하는 시스템 테스트도 있다.

리그레션 테스트에 적용되는 기술을 분류하면 〈표 3〉과 같다.

리그레션 오류를 찾아내기 위한 분석에는 주로 의존도 분석기술을 사용한다. 논리흐름에 의존되어 있거나 자료를 선언하고 사용하는(define-use) 관계에 있거나 전역 변수를 사용하는 프로시듀어들의 관계에 있다면 이들은 리그레션 테스트의 대상이 된다. 또한 화일이나 통신 라인을 공동 이용한다면 서로 영향을 줄 만하므로 리그레션 테스트되어야 한다.

최근의 기술은 변경된 부분만을 따로 컴파일하고 저장하여 변경되지 않은 부분과 합하는(incremental parsing) 방법을 이용하여 리그레션 테스트도 원래의 테스트 케이스가 변경되어야 할 부분만을 파악하는 방법(incremental regression test)이 연구되고 있다.

2.3 프로그램 이해를 위한 역공학

소프트웨어를 유지보수하거나 점증적으로 발전시켜 나갈 때 가장 중요하고 우선적인 과제는 프로그램의 이해이다. 노후 시스템의 경우 프로그램의 이해에는 많은 어려움이 따른다. 유지보수 단계에 많은 비용이 소요되는 것도 하위 수준의 방대한 원시코드에서 상위 수준의 소프트웨어 구조를 파악하는데 많은 시간이 걸리기 때문이다. 특히 노후 시스템은 소프트웨어 구조를 이해하는데 도움이 되는 개발 문서가 없는 경우가 많고 수정이 많이 가해져서 프로그램 구조나 기능의 파악이 어려운 경우가 많다.

하드웨어 역공학 개념을 소프트웨어 시스템에 도입하여 원시 코드로부터 설계 정보와 추상화된 시스템 모델을 역으로 추출해 내는 기술을 소프트웨어 역공학(Software Reverse Engineering)이라 한다. 소프트웨어 역공학 기술을 이용하여 소프트웨어의 구조에 관한 설계정보를 추출하면 대규모 소프트웨어의 이해에 매우 유용하다. 즉 대규모 소프트웨어를 이루는 모듈들을 파악하고 이들 사이의 관계 및 인터페이스를 자동적으로 파악해 내는 방법이다.

소프트웨어를 이해하는 작업은 여러 가지 수준이 있을 수 있다. 원시코드에 담겨진 의미를 이해하는 프로그래밍 언어 수준의 이해, 시스템을 이루는 모듈

〈표 3〉 리그레션 테스트 기술

분 류	이 용 기 술	적용 분야	연 구
unit regression test	dependency analysis	단위 테스트	[6][11][12]
input partition	프로그램 명세에 따라 input domain을 나눔	단위 테스트	[13]
path regression test	변경된 경로를 찾아내 테스트(실행 슬라이스)	단위 테스트	[14]
incremental data flow test	data flow를 기반으로 한 테스트에 incremental data flow analysis를 가미	단위 테스트 및 통합 테스트	[15]

〈표 4〉 프로그램 이해를 지원하는 프로그램 분석 방법

분 석 방 법	Abstraction	인식 규모	인식 빈도	연 구
Functional abstraction	predicate	소규모 블럭	많음	[16]
Knowledge-based approach	program plan	중규모 블럭	가변적	[17]
Graph-parsing	program plan	대규모 블럭	적음	[18]

단위의 이해, 요구명세서에 나타난 수준의 이해, 응용 분야의 업무나 전문지식 수준의 이해 등이다. 인간이 소프트웨어를 이해하는 과정을 자세히 연구하는 논문[16]에 의하면 소프트웨어를 이해하는 과정에 이러한 여러 가지 수준의 지식이 복합적으로 작용한다고 한다. 즉 운영체제의 원시코드를 이해하려 한다면 프로그램에 사용된 변수의 이름에서 힌트를 얻어 블럭의 의미를 해석하고 블럭의 기능을 파악한다. 이미 응용 분야에 대하여 충분히 이해하고 있다면 프로세스 관리, 기억장치 관리 등 프로그램의 전체적인 구조를 쉽게 파악할 수 있다. 결국 여러 수준의 추상화된 표현들이 제공되어야 소프트웨어를 효과적으로 이해할 수 있다.

프로그램을 분석하여 원시코드 수준의 이해를 돕는 방법이 연구되고 있다. 이 분야의 기술은 〈표 4〉와 같이 크게 세 가지 방법으로 나뉜다. 원시코드 수준의 이해를 돕는 방법은 프로그램 안에 있는 의미있는 단위(program prime unit)들을 파악하여 프로그래머에게 알려주자는 것이다. 즉 프로그램에서 sorting 부분이 있다면 그래프 파싱이나 패턴 매칭을 이용하여 이를 자동적으로 파악한다. 또한 프로그램 블럭에 나타난 의미를 Predicate으로 추상화하면 그 의미를 쉽게 알 수도 있다.

2.4 ISO 9000과 유지보수

제품을 생산하거나 서비스를 제공하는 조직이 좋은 품질의 제품을 생산하고 양질의 서비스를 제공하기 위하여 갖추어야 할 품질에 대한 규정이 1987년 ISO에서 제정되면서 많은 산업부분에 영향을 미치고 있다. 소프트웨어 산업 분야에도 다음과 같은 세 가지 규격으로 추진되고 있다.

- 소프트웨어를 개발하고 공급 및 유지보수하는 기관을 위한 특별 지침(ISO 9000-3 : 1991)
- 소프트웨어 품질 검사 기준(ISO 9126 : 1991)
- 소프트웨어 프로세스 평가 기준을 개발하고 이를 이용하여 기관을 테스트할 수 있도록 하는 SPICE 프로젝트

최근 ISO 9000 품질 인증을 받기 위하여 국내 여러 소프트웨어 개발 기관들이 노력하고 있다. 유지보수 활동에 대한 ISO 9000의 기준에서 중요한 것은 변경 관리의 과정을 명확히 규정한 유지보수 과정이 확립되어 있어야 한다는 사실이다. 즉 소프트웨어 유지보수를 위한 계획과 이를 시행할 팀이 구성되어 있어야 하며 유지보수 활동에 대한 모든 기록이 자세히 보관되어야 한다. 또한 유지보수에 관한 기록을 모아서 분석한 후 다음 계획에 반영하여야 한다. 이러한 과정에 대한 규정은 ISO 9000-3의 6.1 Configuration Management에 잘 기술되어 있다.

두 번째로 중요한 사항은 소프트웨어를 개발하는 동안에 품질과 관련된 데이타를 수집하는 일이다. 즉 개발하는 동안 모은 품질에 관한 기록은 특수한 응용분야 또는 개발 환경에서 일어날 수 있는 오류 유형에 대한 정보를 제공한다.

ISO 9000 품질 규정에서 중점을 두고 있는 또하나의 사실은 설계 오류에 의하여 다시 작업하는 경우를 줄이기 위하여 검사(inspection) 과정이 있는가 하는 것이다. 연구에 의하면 설계 및 코드 검사를 실시하면 오류의 90% 이상을 시스템 테스트 이전에 발견할 수 있다고 한다. 이렇게 되면 오류를 수정하는 유지보수(corrective maintenance) 형태가 줄어든다. 따라서 개발 이후의 작업이 소프트웨어의 발전(evol-

ution)에 중심을 두게 되어 시스템이 더 유연한 구조로 개선되어 갈 수 있다.

하지만 ISO 9000 시리즈에서 아쉬운 점은 품질 개선을 위한 프로세스는 강조하였지만 궁극적으로 관심이 되는 품질을 재는 방법(metric)에 대해서는 언급하고 있지 않다. 또한 개발과정에 기록되어야 할 최소한의 품질 데이타의 내용도 제시되지 않고 있다. 이는 ISO 9000 시리즈가 단지 소프트웨어를 개발하는 과정에 할 일이 정확히 기술되어 있고(Say what you do), 정해진 대로하고(Do what you say), 한대로 기록하고(Record what you did), 수행한 것과 기록을 반영하여야 한다는(Review your record and act) 철학에 의하여 작성되었기 때문이다.

마지막으로 ISO 9000에서 필요한 것은 유지보수 작업의 형태를 하나로 고정하지 않고 여러 가지 파라다임(예를 들면 quick-fix, evolution 등)을 포괄하는 내용이 되어야 한다.

3. 리엔지니어링 기술

소프트웨어 리엔지니어링은 소프트웨어 시스템을 정확히 이해하고 변경시키는 여러 가지 작업과 관련된다. 리엔지니어링을 위하여 먼저 현재의 시스템을 이루는 각 부분을 인식하고 이들 사이의 관계를 파악하는 일이 중요하다. 어떤 블럭이 어떤 기능을 하며, 자료구조는 무슨 정보를 저장하며, 함수가 계산하여 전달하는 값의 의미는 무엇인가 등이다. 이러한 작업은 한마디로 시스템에 대한 고수준의 요약 정보(high-level description)를 만들어 내는 일이다. 시스템이 어떻게 구성되어 있으며, 어떻게 작동되는가를 발견하여 기술하는 작업이다. 다음에는 새로운 기술을 이용하여 이해한 시스템을 다시 작성하는 과정이 필요하다. 예를 들면 관계형 데이타베이스 기술이나 새로운 통신 프로토콜, 분산처리 기술, 객체지향 기술 또는 새로운 사용자 인터페이스 기술들을 적용하는 것이다.

소프트웨어 리엔지니어링의 필요성은 다음과 같다. 앞서 언급한 바와 같이 처음에 모든 것을 완성한 시스템이라도 시간이 경과하고 기술이 발전하면서 결함이 발견된다. 더구나 하드웨어 기술의 발전으로 미래의 요구를 정확히 반영한 시스템은 되지 못한다. 따라서 천천히 계속적으로 변경하고 개선시키는 방향으로 시스템을 구현하여야 한다. 결국 앞으로의 소프트웨어 개발은 새로운 시스템을 이해하고 만들어 나가는 일 뿐만 아니라 과거 시스템을 이해하고 이를 리엔지니어링하는 기술과 도구가 공급되어야 한다.

리엔지니어링이 필요한 이유는 대부분 소프트웨어를 유연하게 하여 유지보수 비용을 줄여 보려는 것이 대부분이며 변경 요구를 수용하려는 것, 그리고 새로운 최신 기술을 적용해 보려는 것이다. 또한 비지니스 프로세스를 이해하고 개선하려는 이유로 소프트웨어 시스템을 리엔지니어링하는 경우도 있다.

리엔지니어링 기술의 접근 방법은 다음 세 가지로 생각할 수 있다. 먼저 단순한 프로그램 제어의 재구조화이다. 예를 들면 오래된 COBOL로 쓰여진 시스템의 경우 대규모 시스템이 큰 프로그램 하나로 작성되어 있다면 프로그램 수정 및 유지보수에 어려움이 많다. 프로그램의 제어 구조를 살펴보면 perform 문장을 사용하여 모듈화될 수 있는 요소들이 있다. 따라서 이러한 부분들을 따로 떼 내어 독립된 모듈로 분리시킨 후 다른 화일에 관리하면 유지보수를 용이하게 할 수 있다. 두 번째 방법은 새로운 시스템으로 구성하기 위하여 노후 시스템의 설계 사양을 복원하는 것이다. 대규모 소프트웨어 시스템을 이해하기 위하여 시스템 구조와 각 모듈의 관계를 파악하는 것이 필수적이다. 마지막으로 노후 시스템을 파악하여 변경하기 어려운 경우, 시스템을 동결시키고 이를 새로운 시스템의 부분 요소로 캡슐화한다. 즉 노후 시스템을 그대로 두고 그 위에 새로운 층을 형성하여 사용자 인터페이스 등을 개선할 수 있다.

리엔지니어링 기술의 특성은 논리적 모델링과 슬라이싱이다. 원시코드에 파묻힌 데이타 모델, 프로세스 모델, 논리 모델을 추출해 내는 기술이다. 데이타 모델은 고객, 주문, 재품 등과 같은 자료나 개체와 관

련된 모델이며, 프로세스 모델은 구매요구, 선적 등 자료에 가해지는 동작이나 기능이며, 논리 모델은 신용허가, 견적준비 등 비지니스 업무와 관련된 모델이다. 이러한 모델 추출 기능과 함께 원시코드에서 필요한 부분을 추적하여 찾아내는 슬라이싱(slicing) 기술이 절대 필요하다.

리엔지니어링 과정에서 가장 중요한 단계는 비용 절감 효과에 대한 예측이다. 과연 새로 개발하는 것보다 기존의 시스템을 리엔지니어링하는 방법이 비용 절약되는지, 어느 정도의 소프트웨어 품질이 향상되는 것인지, 유지보수 과정이 얼마나 향상되는지, 회계 장부로 얼마의 비용이 실제 절감되는지 예측하여야 한다. 따라서 소프트웨어 품질을 측정할 수 있는 원시코드 검증 장치와 메트릭 자료를 보관할 수 있어야 한다.

리엔지니어링의 다음 과정은 노후 시스템을 동결시키고 원시코드나 관련 문서에서 자료의 정의와 의미, 블럭의 기능들을 찾아낸다. 이를 정리하여 문서화한 후 자료나 프로그램 제어에 대하여 개선할 점이 있으면 최선의 방법을 찾아낸다. 새로운 원시코드를 작성한 후에는 단위 테스트로 프로그램을 검증하고, 데이타 리엔지니어링이 있었다면 변환된 데이타를 이용하여 통합 시험한다.

3.1 데이타베이스 리엔지니어링

오래된 데이타베이스에 대한 문제는 다음 세 가지로 구분된다. 첫째 자료의 정의에 관한 문제들, 예를 들면 자료의 이름이 명확하지 않거나, 필드 길이가 맞지 않는 경우, 레코드 레이아웃이 부적절한 문제, 상수가 hard-code된 경우 등이다. 대규모 자료의 경우 자료를 구성하는 각 필드에 대한 설명이 자세히 자료사전에 기록되는데 정확한 정보를 담고 있어야 한다.

두 번째 문제는 자료값 자체에 대한 문제이다. 디폴트 값이 일치하지 않거나 자료값이 존재하지 않거나(missing data), 또는 자료값이 절단된 경우, 자료값의 단위가 일치하지 않는 경우 문제가 된다.

데이타베이스 리엔지니어링이 불가피한 가장 큰 요인은 데이타베이스 파라다임의 전환이다. 화일 저장 방식에서 데이타베이스를 도입하는 경우, 계층적 또는 네트워크형 모형의 데이타베이스에서 관계형 데이타베이스로 전환하려는 경우이다. 경험에 의하면 이 작업은 데이타베이스 구조와 밀접히 관련되어 매우 어려운 작업이다[19]. 즉 계층적 구조에서는 child 레코드와 parent 레코드가 묶여 있어 관계형 데이타베이스에서 필드의 값으로 레코드의 관계를 표현하기가 쉽지 않다.

관계형 데이타베이스로 변환하는 방법은 상향식(bottom-up)으로 이루어진다. 먼저 현재 시스템에서 정확한 자료 정의를 얻어낸다. 다음에는 현재 화일 관련 원시코드의 구조나 화일 사이의 관계를 알아내고, 레코드나 화일의 레이아웃을 작성한 후 논리적으로 그루핑한다. 다음 작업은 정규화된 데이타 모델을 얻어내기 위하여 논리 데이타 모델링 기법을 적용한다. 즉 Cobol 레코드 정의로부터 E-R 모델을 만들어 내고 이를 데이타 모델로 발전시킨다. Silverrun과 같은 도구는 이 과정을 자동화한 것이다.

서로 다르게 구현된 데이타베이스 시스템간의 리엔지니어링도 필요하다. 예를 들면 특정 회사의 관계형 DBMS를 사용하다가 다른 관계형 DBMS를 이용하여 데이타베이스를 구축하려 할 경우이다. 이러한 형태의 데이타 리엔지니어링에서는 메타-데이타 모델의 표현이 중요하다. 즉 구현 방법에 관계없이 데이타들 사이의 관계를 잘 표현하는 방법이 필요하다. Premerlani는 여러 가지 방법 중에 객체지향 방법에 의한 리엔지니어링 방법을 제안하고 있다[20]. 먼저 관계 데이타베이스의 각 테이블을 객체로 보고 테이블의 각 열을 객체의 속성으로 본다. 다음은 데이타의 패턴을 보고 후보키(candidate key)를 알아낸다. 객체간의 일반화(generalization) 관계를 찾아내기 위하여 먼저 외부키(foreign-key) 그룹들을 알아내고 수직으로 분해된 클래스, 즉 같은 의미를 갖는 클래스들을 결합한다. 마지막으로 일반화 관계와 결합 관계를 파악하여 객체지향 모델을 만든다.

3.2 사용자 인터페이스 리엔지니어링

대부분의 노후 시스템은 문자 중심의 사용자 인터페이스 형태를 취하고 있다. 사용이 더 간편하도록 인터페이스를 리엔지니어링한다면 시스템의 수명이 더 연장되고 널리 사용될 수 있을 것이다. 사용자 인터페이스를 리엔지니어링하려면 우선 현재의 인터페이스가 어떻게 구현되어 동작되는지 알아야 하며 새로 만들어진 인터페이스가 시스템의 나머지 부분과 잘 융합되기 위한 조건 등을 파악하여야 한다. 무엇보다도 문자형 사용자 인터페이스에서 그래픽 사용자 인터페이스로 리엔지니어링하는 과제가 제일 큰 관심거리이다[21].

사용자 인터페이스 리엔지니어링에서 가장 중요한 정보는 노후 시스템의 사용자 인터페이스에 대한 구조(structural) 및 동작(behavioral) 명세이다. 노후 시스템의 원시코드를 보고 사용자 인터페이스가 어떻게 구성되었으며 동작하는지 파악하여 구조는 인터페이스 명세 언어로 표현하고 동작은 그래프 형태로 표현한다. 이런 모델을 이용하여 노후 시스템의 인터페이스 구조와 흐름에 문제가 없는지 분석하고 이를 바탕으로 새 시스템을 설계, 구현한다.

사용자 인터페이스를 리엔지니어링하는 과정은 먼저 노후 시스템의 인터페이스 코드를 역공학하여 Abstract Syntax Tree를 만들고 여기서 인터페이스 부분을 추출한다. 인터페이스 부분이 독립된 모듈로 만들어졌다면 쉽게 분리되지만 그렇지 않다면 슬라이싱 기법으로 해당되는 각 부분을 잘라 내야 한다. 잘라 낸 원시코드와 논리흐름을 잘 살펴보면 사용자 인터페이스 화면을 구성하는 객체, 화면 및 화면의 흐름을 알아낼 수 있다. 이를 객체 지향으로 표현하는 방법이 AUIDL(Abstract User Interfece Descrio-ption Language)이다.

모델로 표현한 사용자 인터페이스는 이제 쉽게 변형될 수 있다. 예를 들면 기능 키에 의한 작동을 마우스 버튼에 의하여 작동되도록 재구성할 수 있고, 대화 화면의 흐름을 분석하여 더 효율적으로 개선할 수 있다. 마지막으로 새로운 인터페이스를 원래의 시스템과 통합하는 작업은 경우에 따라 그 난이도가 다르다. 노후 시스템의 대부분을 대형 기종 위주의 중앙 집중 형태로 구성되어 있어 이와 통신할 수 있는 에뮬레이션 기능을 추가하고 클라이언트의 사용자 인터페이스를 변경하면 가능하다. 그러나 대형 중앙 시스템의 원시코드를 변경하여야 하는 경우 시스템 원래의 기능을 보호하여야 하며 사용자 인터페이스를 위하여 추가하는 부분이 변수 이름 중복, 매개변수 값의 지정으로 인하여 방해받지 않도록 배려하여야 한다.

사용자 인터페이스 리엔지니어링의 주요 작업은 특정 언어로 쓰여진 현재 시스템의 인터페이스를 추상화된 일반적인 표현으로 바꾸는 것이다. 인터페이스를 이루는 각 요소들의 관계, 즉 인터페이스 구조를 표현하여야 하며 인터페이스를 이용하여 사용자와의 대화가 이루어질 때 상호 동작의 흐름도 표현되어야 한다. 인터페이스 구조는 화면에 표현되는 요소들의 포함(containment)관계와 각 요소들의 값과 속성이 화일의 무엇과 연결(importation)되어 있는지를 의미한다. 인터페이스를 분석하기 위하여는 동적인 면, 즉 이벤트에 대하여 어떻게 반응하는지를 잘 이해할 수 있도록 나타내야 한다. 일반적으로 인터페이스의 상태와 상태의 변화를 나타내는 상태천이도(Transition graph)로 그려본 후 그래프를 반복적으로 Traverse하여 경로의 집합을 구한다. 구해진 경로에서 정상적으로 마치지 않고 중간에 없어진 인터페이스 동작에 관한 부분을 가려낸다. 이런 부분을 집중적으로 분석하고 문제가 있으면 해결하는 것이 인터페이스 개선의 주된 작업이다.

3.3 분산 시스템으로의 리엔지니어링

여러 가지 이유로 메인프레임에서 분산 클라이언트 서버 시스템으로 전환하려는 추세가 있다. 값비싼 하드웨어의 구입 및 유지보수 비용을 줄이려는 목적뿐만 아니라 사용자에게 더욱 친근하고 유연한 인터페이스를 제공하기 위하여 분산 시스템으로 전환한다. 분산 시스템을 도입하기 위하여 오랫동안 많은

투자를 하여 개발한 시스템을 버리고 새로 개발하는 것은 큰 낭비가 아닐 수 없다. 따라서 노후 시스템을 리엔지니어링하여 분산 시스템으로 전환하는 기술이 연구되고 있다[22].

분산 시스템으로 전환하는 과정, 즉 큰 응용 프로그램을 적당히 분리하여 분산 재배치하는 작업은 다음과 같은 이유로 쉽지 않다. 먼저 프로그램이 메인프레임 데이타베이스에 많이 의존되어 있다. 특히 자료처리가 주기능인 비지니스 응용 프로그램은 데이타베이스 관리 시스템의 기능과 형식에 크게 의존하는 부분이 많다. 따라서 분산 시스템으로 구성될 경우 워크스테이션이 호스트의 데이타베이스를 접근하기 위하여 에뮬레이션 기능이 필요하다. 즉 IMS나 IDMS 명령어를 SQL로 바꾸어 주어야 한다.

또 한가지 장벽은 비분산 시스템이 대부분 메인프레임의 온라인 트랜젝션 프로그램이라는 사실이다. 호스트의 모니터 프로그램이 터미널에서의 반응에 따라 서브 프로그램을 호출하게 되어 있는데 트랜젝션 모니터 기능을 네트워크 관리 소프트웨어가 담당할 수 있도록 하여야 한다. 응용 프로그램에서 message passing 부분은 클라이언트 컴퓨터에 body 부분은 서버에 남도록 배치하여야 한다. 메인 프레임에서 수행되는 응용 프로그램이 매우 큰 경우 워크스테이션에서 컴파일도 안되는 문제가 있다. UNIX나 AS/400 운영체제에서 12,000줄 이상의 단일 프로그램을 올리는 것은 불가능하다.

이런 장벽이 해결된다면 결국 분산 시스템으로의 전환은 모듈로 쪼개는 과정이라 할 수 있다. 즉 복잡한 프로그램을 독립적으로 컴파일할 수 있는 모듈로 나누는 작업인데 여기에 대한 구체적이고 실제적인 연구 결과는 미약하다. 여기에는 그래프 이론과 추상 데이타 기법이 응용되고 있다. 분산 시스템으로 구성하기 위하여 프로그램을 도들화하는 방법은 〈표 5〉와 같이 세 가지 방식이 있다.

3.4 객체지향 시스템으로의 리엔지니어링

3세대 프로그래밍 언어로 쓰여진 소프트웨어를 4세대 언어로 전환하는 리엔지니어링은 이제 어느 정도 정착된 것 같다. 이제는 제4세대 및 제3세대 언어의 프로그램을 객체지향으로 전환할 필요가 있다. 객체지향 프로그램으로의 리엔지니어링 기술은 원래의 프로그램에서 객체가 될 만한 것을 뽑아 객체선언을 알맞게 하고 프로그램 슬라이싱을 이용하여 절차중심의 원시코드에서 Method를 추출하는 것이 핵심이다.

절차중심의 노후 시스템을 객체지향 시스템으로 변환하는 과정은 Jacobson의 연구[23]에 잘 기술되어 있다. 또한 노후 시스템을 객체지향 개념으로 잘 포장하는 기술은 [24]에서 다루고 있으며 자료 흐름 모델에서 객체 지향 설계로 전환하는 방법은 [25]에 기술되어 있다. 절차 중심 프로그램으로 이루어진 시스템을 객체지향 시스템으로 자동 변환하는 과정은 [26]에 잘 설명되어 있다. 여기서는 절차중심 COBOL 프로그램을 객체지향 시스템으로 변화하는 기술을 간단히 다룬다.

메인프레임의 COBOL 원시코드 분석에 의하여 객

〈표 5〉 분산 시스템으로의 전환 방법

전환 방법	프로그램 View	설 명	장 단 점
Procedural approach	Directed graph	프로그램 그래프에서 Procedural cluster를 발견하여 쪼개고 call subroutine 문장을 삽입	전환이 빠르고 정확, 반면에 성능에 취약
Functional approach	Functional hierarchy	기능 구조도에서 functional boundary를 정하고 나눔	Domain 지식 필요, 사용자에 의한 전환에 적합
Data-type approach	Cooperative processing objects	여러 가지 데이타 객체(화일, 리포트, 내부 테이블, 레코드, View 등)를 파악하고 이를 기준으로 원시코드를 분할	변경에 유연함, 전환이 쉬움, 유지보수 용이

체가 될 수 있는 7가지는 다음과 같다.

- 사용자 인터페이스 객체(Panel에서 추출)
- 정보 객체(데이타베이스 정의 명세에서 추출)
- 화일 객체(JCL에서 추출)
- 레코드 객체(프로그램에서 추출)
- View 객체(프로그램에서 추출)
- Work 객체(프로그램 work area에서 추출)
- Link 객체(프로그램 파라미터에서 추출)

다음 과정은 각 객체에 대한 오퍼레이션을 찾아 추출하는 것이다. 앞뒤 참조표(cross reference table)를 조사하면 객체를 접근하거나 변경시키는 프로그램 세그멘트를 찾아낼 수 있다. 프로그램 세그멘트를 구성하는 작은 단위를 찾아내려면 제어 흐름을 분석하여 Branch 사이의 레이블이 붙여진 블럭을 찾으면 된다. 이것이 객체의 Method가 된다. 다음은 오퍼레이션의 타입을 결정하여야 한다. 오퍼레이션의 대상(operand)이 객체의 속성이면 그 객체의 Method로 등록하고 오퍼레이션 타입(Insert, Select, Update, Delete)에 따라 그루핑한다.

다음 과정은 객체들을 연결시키는 작업이다. 오퍼레이션이 사용하는 객체로 선언되지 않은 변수, 즉 외부 변수(foreign variables)들을 찾아내어 알맞은 객체의 Method로 호출하는 문장을 만들어 낸다. 마지막으로 오퍼레이션 순서를 바로 정하는 작업을 한다. 이 과정은 원래 프로그램의 제어 구조를 새로운 오퍼레이션을 이용하여 그대로 복원하면 된다.

4. 결 론

소프트웨어의 의존도가 커 가고 기술 발전 속도가 빠르게 되면서 노후 시스템에 대한 개선이 더욱 중요한 과제가 될 것이다. 지난 7월에 캐나다에서 열렸던 역공학 컨퍼런스에서는 앞으로 소프트웨어 역공학 기술자들이 많이 필요할 것이며 이들의 일과 기술은 마치 소프트웨어 고고학자라 불릴 수 있는 일이라고 하였다. 오래 묵은, 잊혀진 소프트웨어를 꺼내 먼지를 털고 필요 없는 것은 버리고 의미 있는 부분은 살려서 새로운 모델로 재창조하는 과정이다. 앞으로는 노후 시스템을 유지보수하는 방법, 리엔지니어링하는 기술 더 나아가 재사용할 수 있는 기술과 도구들이 활발히 연구되고 적용될 것이다. 기업이나 공공기관 및 가계에서 소프트웨어와 노후 시스템은 하나의 자산으로 인식하기 시작하였기 때문이다.

한편으로는 미래의 노후 시스템이 될 소프트웨어를 현재 개발되고 있다. 객체지향, 클라이언트/서버, 분산 시스템도 미래에는 암호와 같은 노후 시스템이 될 것이다. 그때 "이것을 개발할 때 무슨 생각을 하였을까?" 이러한 물음의 해답을 찾을 찾을 때 필요한 기술이 노후 시스템에 대한 유지보수 및 리엔지니어링 기술이다.

참 고 문 헌

1. P. Aiken, A. Muntz, R. Richards, "DoD Legacy Systems", Comm. of ACM, Vol. 37, No. 5, pp. 26-41, May 1994.
2. B. Lientz, E. Swanson, *Software Maintenance Management*, Addison-Wesley, 1980.
3. N. Chapin, "Software Maintenance : A Different View", Proc. of the National Computer Conference, AFIPS 54, pp. 507-513, 1985.
4. N. Shnidewind, "The State of Software Maintenance", IEEE Trans. on Software Engineering, Vol. 13, No. 3, pp. 303-310, 1987.
5. S. Dekleva, "Delphi Study of Software Maintenance Problems", Proc. of Conference on Software Maintenance, pp. 10-17, 1992.
6. K. Galagher, J Lyle, "Using Program Slicing in Software Maintenance", IEEE Trans. on Software Engineering, Vol. 17, No. 8, pp. 751-761, 1991.
7. "Automated Life Cycle Impact Analysis System", RADC-TR-86-197, Rome Air Deveploment

Center, 1986.

8. R. Arnold, S. Bohner, "Impact Analysis-Towards A Framework for Comparison", Conference on Software Maintenance, pp. 292-301, 1993.
9. E. Horowitz, R. Williamson, "SODOS : A Software Document Support Environment-Its Definition", IEEE Trans. on Software Engineering, Vol. 12, No. 8, pp. 849-859, 1986.
10. A. Podgurski, L. Clarke, "A Formal Model of Program Dependences and Its Implication for Software Testing, Debugging, and Maintenance", IEEE Transactions on Software Engineering, Vol. 16, No. 9, pp. 965-979, 1990.
11. S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependency Graphs", ACM Symposium on Principles of Programming Language, 1993.
12. D. Binkley, "Using Semantic Differencing to Reduce the Cost of Regression Testing", Conference on Software Maintenance, pp. 41-50, 1992.
13. D. Richardson, L. Clarke, "Partition Analysis : a Method of Combination Testing and Verification", IEEE Transaction on Software Engineering, Vol. 11, No. 12, pp. 1477-1490, 1985.
14. P. Benedusi, A. Cimitile, U. Carlini, "Post-maintenance Testing Based on the Path Change Analysis", Conference on Software Maintenance, pp. 352-361, 1988.
15. M. Harrold, M. Soffa, "An Incremental Approach to Unit Testing During Maintenance", Conference on Software Maintenance, pp. 362-367, 1988.
16. P. Hausler, et. al, "Using Function Abstraction to Understand Program Behavior", IEEE Software, 1990.
17. M. Harandi, J. Ning, "PAT : A Knowledge-Based Program Analysis Tool", Conference on Software Maintenance, pp. 312-318, 1988.
18. C. Rich, L. Wills, "Recognizing a Program's Design : A Graph-Parsing Approach", IEEE Software, 1990.
19. B. Francett, "From IMS or Non-IBM, The Move is on to DB2", Software Magazine, pp. 50-61, September, 1989.
20. W. Premerlani, M. Blaha, "An Approach for Reverse Engineering of Relational Databases", Communications of the ACM, Vol. 37, No. 5, pp. 42-49, 1994.
21. E. Merlo, A. Thiboutot, "Inference of Graphical AUIDL Specifications for the Reverse Engineering of User Interfaces", Conference on Software Maintenance, pp. 80-88, 1994.
22. T. McDonald. "Converting Legacy FORTRAN Applications to Distributed Applications", International DCE Workshop, 1993.
23. I. Jacobson, "Re-engineering of Old Systems to an Object-Oriented Architecture", OOPSLA, pp. 340-350, 1991.
24. W. Dietrich, et. al, "Saving a Legacy with Objects", OOPSLA, pp. 77-88, 1989.
25. B. Alabiso, "Transformation of Data Flow Analysis Models to Object Oriented Design", OOPSLA, pp. 335-353, 1988.
26. P. Newcomb, "Reengineering Procedural into Object-Oriented Systems", 2nd Working Conference on Reverse Engineering, pp. 237-249, 1995.
27. M. Hanna, "Legacy Apps : To the Scrap Heap or a New Platform?", Software Magazine, pp. 39-44, 1994.

최 은 만

1982년 동국대학교 전자계산학과(학사)
1985년 한국과학기술원 전산학과(석사)
1993년 일리노이 공대 전산학과(박사)
1985~1988년 한국표준연구소 연구원
1988~1989년 한국데이타통신(주) 주임연구원
1993~현재 동국대학교 컴퓨터공학과 조교수
관심분야 : 소프트웨어 유지보수, 역공학, 재사용, 객체지향 소프트웨어 공학

E-MAIL ID 양식

수신 : 한국정보처리학회 FAX : (02)593-2896

성 명	(한글) (영문)	전화번호	( ) -
주 소		팩스번호	( ) -
근 무 처	(직위)		
E-MAIL ID			

* 반드시 팩스 이용