기간계 객체지향 시스템의 유지보수성에 관한 현장연구

임 좌 상* · 정 승 렬**

A Field Study on the Maintainability of Mission Critical Object-Oriented Systems

Joa Sang Lim* · Seung Ryul Jeong**

Abstract

Empirical evidence on the maintainability of object-oriented systems is far from conclusive, partly due to lack of representativeness of the subjects and systems used in the study. The present research empirically examined this issue with the systems that are mission-critical, currently operational and maintained by professionals. It was found that the OO group appeared to consume less time while maintaining more amount of software artifacts than the NOO counterpart. This economical utilization of time appeared evident regardless of software development life cycle. This was due to the usefulness of UML for impact analysis which contributed to effective comprehension and communication. Insufficient design specifications led to ambiguity and costly defects in transferring design solutions to development. Also, the encapsulation of OO seemed to reduce mental loads at maintenance tasks and improved code reuse. However, the number of files to manage increased and thus, dependency management is required for the OO systems.

Keywords : Object Oriented Development, Maintainability, UML

논문접수일 : 2004년 6월 15일 논문게재확정일 : 2004년 8월 18일
※ 2004년 상명대 교내연구비에 의해 수행되었음.
* 상명대학교 미디어학부 교수
** 국민대학교 비즈니스IT전문대학원 교수

1. 서 론

소프트웨어는 생명주기를 걸쳐 진화해 가는데, 그 수명은 1990년대 9.4년으로 조사된 바 있으며, 이는 1980년대의 4.75년에 비해 2배 정도 증가한 것이다[28]. 이러한 소프트웨어의 생명주기 동안 유지보수는 요구사항의 변화를 수용하기 위해 피할 수 없는 활동이다[24]. 물론 유지보수를 어떻게 정의할 것인지에 관해서는 이견이 있다. ISO 9126에서는 시간을 기준으로 시스템 구현 이후의 모든 활동을 유지보수로 정의한다[18]. 이는 생명주기 과정 전반의 모든 유지보수 활동을 개발로 분류하는 단점이 있다. 이러한 단점을 보완하기 위해 어떤 활동이든지 정보시스템을 수정, 변경하는 것을 유지보수라고 정의하기도 한다[17]. 유지보수는 세 가지로 구분하는데[16], 우선 하드웨어 또는 소프트웨어에 내재한 결함의 수정 및 조치를 의미하는 오류유지보수(Corrective Maintenance)가 있다. 둘째는 적응유지보수(Adaptive Maintenance)로서 판매세율의 변경과 같이 현재 운영하고 있는 시스템 환경의 변경에 따른 기능개선과 관련한 활동을 말한다. 마지막으로 완전유지보수(Perfective Maintenance)는 현행시스템의 성능개선과 같은 확장을 말한다.

유지보수의 중요성은 오래 전부터 인식되어 왔다. 1970년대 후반 UCLA대학의 Lientz와 Swanson 교수는 설문조사를 통하여 많은 기업이 개발보다 유지보수에 훨씬 더 많은 인력을 투입하는 것을 보고하였다[24]. 이러한 현상은 그 후 2, 30년이 지난 현재 그 간 수많은 신기술이 등장하였음에도 불구하고 개선되지 않았다. 비용 면에서 보면 유지보수는 개발과 비교해서 그 비용이 2/3에서 무려 4/5까지 달한다는 주장도 있다[2]. 또한 오류유지보수의 경우, 결함이 적시에 발견되어 조치되지 못하면 그 비용이 급증하게 된다. Kan et al.[23]은 요구사항 단계에서 결함을 발견하지 못하고 구현으로 전이될 경우 그 비용이 무려 52배로 증가하게 된다고 지적하였다. 이러한 유지보수의 문제점을 해결하기 위한 방안으로 이론과 실무에서 객체지향 기술이 많은 주목을 받고 있다[19]. 하지만 객체지향 기술이 제공할 것이라는 향상된 유지보수성은 아직 충분히 검증되지 못한 상황이며[14], 오히려 학습의 어려움 또는 충분치 못한 성능 등으로 인해 객체지향 기술에 대한 회의적인 시각이 존재하기도 한다. 이러한 상황에서 실제 객체지향 기술이 투자 가치가 있을 만큼 유지보수성이 확보되는지에 대해 살펴보는 것은 매우 시의적절하다고 하겠다. 물론 이러한 이슈를 다루는 일련의 연구가 이미 발표되기도 했으나 그 연구결과들은 일관적이지 않은 것으로 나타난다[20]. 이들 연구들은 대부분이 잘 통제된 실험실 환경에서 저자들이 개발한 작은 시스템을 이용하여 실험을 수행하였다. 따라서 현장에서 실제로 사용하는 규모와 복잡성을 가진 시스템과 관련하여서는 충분한 통찰력을 제공하기 힘든 면이 있다는 단점을 가지고 있다. 기업의 핵심업무를 지원하는 기간계 시스템(mission critical systems)은 그 역할이나 규모 등을 감안하면 앞선 연구의 결과가 그대로 적용하는 것은 매우 위험할 수 있다. 특히 최근 관련기술의 발전으로 은행과 같은 기간계업무에도 객체지향의 도입이 심각하게 검토되고 있다는 점을 감안하면, 현 시점에서 객체지향기술이 기간계 업무에도 유지보수성을 향상할 수 있는 대안이 될 것인지를 연구하는 것은 매우 유의미한 일이다. 물론 기간계 시스템을 대상으로 유지보수성을 실험할 수 있는 환경을 마련한다는 것은 매우 어렵다. 이와 같은 까닭에 많은 연구가 손쉬운 실험환경에서 수행되었고, 현장실험을 통한 연구는 상대적으로 그 어려움으로 인해 많이 시도되지 않았는

지 의문이 들 수 있다. 지금까지 나타난 연구 결과들은 위의 논의를 바탕으로 볼 때 객체지향 시스템의 유지보수 관련 효과가(실제 현업의 상황과는 다른 시스템인) 이론적으로 아주 잘 설계된 작은 시스템에서만 나타날 수 있음을 보여준 것은 아닌가 하는 조심스런 질문을 던지게 된다. 따라서 본 연구는 이러한 이슈들을 다루기 위해 실험실이 아닌 일평균 25만 건 정도의 트랜잭션이 발생하는 실제 운용중인 기간계시스템을 대상으로 객체지향 기술과 유지보수성과의 연관성을 실증적으로 살펴보았다.

2. 문헌 연구

앞에서 논의되었듯이 유지보수는 정보시스템의 결함수정, 기능향상, 기능확장과 같이 시스템의 변경과 관련한 활동이라고 정의되었다. 한편 유지보수성(maintainability)은 유지보수와는 다른 개념으로, 유지보수의 용이성을 의미한다[16]. 따라서 유지보수성이 높은 정보시스템이란 그렇지 않은 시스템에 비해 결함수정, 기능향상, 기능확장과 같은 유지보수가 더욱 용이하게 될 것이다. 유지보수성은 품질표준 ISO 9126의 소프트웨어 품질을 결정하는 요소로서[18], (1) 분석용이성(Analyzability), (2) 테스트(Testability), (3) 안정성(Stability), (4) 변경성(Changeability) 등으로 이해할 수 있다. 즉, 유지보수가 용이한 시스템은 이해하기 쉽고, 변경이 일어나더라도 그 영향이 국지화 되도록 안정적이어야 하며, 쉽게 변경하고 테스트할 수 있어야 한다는 것인데, 이러한 시스템은 생명주기의 한 과정인 구현뿐만 아니라 모든 단계에 걸쳐 효과적인 산출물의 작성이 전제되어야 한다.

객체지향 시스템이 내포하고 있는 유지보수성의 우월성을 실증적으로 검증한 연구는 많지 않은 편이다[1]. 실제 객체지향 시스템의 유지보수성과 관련한 많은 실증연구들은 주로 유지보수성을 향상시키는 요인에 관한 연구이거나 유지보수성 모형을 검증하는 방법에 관한 연구들이었다. 좋은 예로서 Binkley and Schach[4]의 연구를 들 수 있다. 이들은 벤더빌트 의과대학에서 C++로 개발된 시스템(82,000라인)에 대하여 유지보수와 객체지향 메트릭스의 연관성을 조사하였다. 유지보수는 각 클래스에 대한 추가, 삭제, 수정과 같은 변경 횟수로 정의하였다. 연구에 포함한 메트릭스는 (1) 클래스간 메트릭스(CDM, Fan-in/ Fan-out, CBO 등), (2) 클래스내 메트릭스(RFC, WMC 등), (3) 상속관련 메트릭스(CHNL, NOC, NCIM, NOD)로 구분하였다. 그 결과 클래스간, 클래스내 메트릭스는 유지보수성과 관련성이 유의적인 반면, 상속관련 메트릭스는 유의적이지 못한 것으로 나타났다. 또한 객체지향에서 상속이 유지보수성에 미치는 영향은 더욱 비관적으로 나타난다. Daly et al.[10]은 상속계층을 0, 3, 5로 구분해서 유지보수성을 비교했는데, 3수준이 가장 효과적이었다. Harrison et al.[15]은 상속이 없었을 경우에 오히려 유지보수성이 좋아진다고 보고하였다. Binkley and Schach[4]는 상속과 결함은 무관하다고 하였다. 또한 Chaumun et al.[8]은 변경영향모델(Change Impact Model)을 제시하고 C++로 구현된 통신시스템을 대상으로 변경용이성과 WMC의 유의적인 관련성을 찾아내었다. 이는 앞서 Binkley and Schach[4]와 일치하는 결과이다. 물론 비객체지향에서도 유지보수를 측정하는 모델이 연구된 바 있다. 회귀분석모델을 사용해 복잡성(cyclomatic complexity, Halstead Volume V), 크기(lines of code), 문서정도(lines of comments)로 구성한 모델을 개발해서 성공적으로 적용한 바 있다[26].

객체지향은 효과적으로 사용할 경우 유지보수성에 긍정적으로 작용한다는 연구가 있다. 최

근 Deligiannis와 그의 동료들은 일련의 연구를 통해[11, 12] 객체지향의 설계원칙이 유지보수성에 미치는 영향을 연구했다. 여기서 설계원칙은 집중과 분산의 문제로서, 분산이 유지보수에 유리할 것이라고 추정하였다. 즉 클래스에 기능을 집중하는 것은 응집성을 낮추고, 결합성을 높인다는 점에서 나쁜 영향을 미치는 것이다. 그 결과 분산설계가 유지보수 산출물의 완전성, 정확성, 일관성을 향상시킨다는 측면에서 유지보수성에 긍정적인 효과를 미쳤다. 이 뿐만 아니라 분산설계가 다른 설계원칙의 적용을 용이하게 한다고 보고하였다(예 : 상속). 이러한 연구결과는 Briand et al.[6]의 연구를 뒷받침하고 있다. Briand et al.[6]은 설계원칙을 준수하는 것이 유지보수성에 긍정적인 효과가 있는지를 실증적으로 조사하였는데, 그 결과 Coad와 Yourdon [9]이 제시한 결합성, 응집성, 일관성(용어, 책임), 일반화와 같은 설계원칙을 준수하는 것이 유지보수성을 향상시키는 것으로 나타났다. 그러나 객체지향이 구조적방법에 비해 유지보수성을 향상시키는 것은 아니었으며, 객체지향을 적용할 경우 설계원칙을 준수하는 것이 더욱 어렵다고 하였다.

설계문서의 효용성에 대한 의문은 Lindvall과 Runesson[25]에 의해 제기되었다. 즉 변경에 대해 설계문서가 유지보수에 필요한 자세한 내용을 보여주지 못해 그 유용성이 적다고 주장하였다. 이들은 Ericsson사에서 사용하는 셀룰러 통신량 관리 소프트웨어를 대상으로 R6에서 R10으로 버전업이 될 경우, 변화에 대한 클래스의 변화를 살펴보았다. R6에서는 총 158개의 클래스가 개발되었는데, 그 가운데 1개가 삭제되었고, 7개가 추가되면서 R10은 164개를 갖게 되었고 상당히 많은 수(157개)가 재사용되는 결과를 보여 주었다. 이 가운데 63개의 클래스가 변경되었는데 모델에서는 불과 28개만이 그 변화를 보여주고 있어 문제점으로 지적되고 있다.

앞서 언급한 바와 같이 객체지향의 사용이 반드시 유지보수성의 향상에 긍정적인 역할을 하는 것은 아니었다. 그러나 좋은 설계, 즉 결합성 및 응집성과 같은 원칙을 준수할 경우에는 객체지향이건 구조적 방법이건 유지보수성이 향상된다는 점을 인지해야 한다. 그러나 이러한 연구결과는 전문인력이 아닌 학생들을 대상으로 얻어진 것이며 실험용 작은 시스템으로 연구가 수행되었으므로 적용에 한계가 있다. 본 논문에서는 전문인력을 대상으로 실제 운용중인 시스템에 대하여 객체지향과 비객체지향의 유지보수성을 비교하고자 한다. 본 논문의 피험자는 해당 업무를 수년간 지속해 온 전문인력이고, 시스템은 그들에 의해 구현되었기 때문에 소프트웨어의 품질이 선행연구의 그것과 비교해 매우 우수하리라고 믿어진다. 따라서 본 논문의 대상 시스템은 이른바 품질이 좋은 현행 시스템으로 객체지향과 비객체지향을 비교한다는 의미가 있다. 명제는 앞서의 문헌 연구에서 살펴본 내용을 기반으로 다음과 같이 설정한다.

명제 1 : 객체지향시스템과 비객체지향시스템의 설계이전 단계(요구사항, 분석 및 설계 단계)에서의 유지보수성에는 차이가 없다.

명제 2 : 객체지향시스템과 비객체지향시스템의 설계이후 단계(구현, 테스트 및 배치 단계)에서의 유지보수성에는 차이가 없다.

3. 연구방법

3.1 실험 설계

실험은 현장연구 방식으로, 국내 카드사에서 현재 운용 중인 시스템을 대상으로 수행되었다. 시

스템은 전국적으로 약 2000만 사용자, 약 4000여 개의 가맹점에서 일평균 약 250만 건 발생하는 트랜잭션을 처리하고 있다. 현장연구는 연구하려는 변인을 효과적으로 통제하기 어렵지만, 현장에서 실제로 발생할 수 있는 변인의 인과관계를 관찰할 수 있다는 장점이 있다. 이러한 필드 연구방식은 C로 구현된 현지시스템을 C++로 재개발하는 과정에서 일정기간 두 시스템을 병행적으로 운용하면서 가능하게 되었다. 따라서 본 연구에서는 앞에서 설정한 명제의 검증을 위해 객체지향(OO)과 비객체지향(NON-OO)의 2개의 실험집단을 구성한다.

3.2 대상업무

본 연구에서 실험은 시나리오 방식을 이용하여 수행되었다[3]. 매년 현업에서 발생하는 유지보수 요구사항(CR : Change Request)은 기획부서에 접수되어 반영여부를 결정하게 된다. CR은 2002년 1376건, 2003년 488건이 발생하였는데, 2003년 건수가 적게 발생한 것은 신용위기에 따라 회사가 공격적인 경영을 가제하였기 때문이었다. 기획부서는 CR을 크기에 따라 1~9등급으로 구분하여 필요공수를 결정하는데 등급이 높을수록 투입공수가 많아진다. 상위 1~2등급은 정부정책의 변화 또는 경영 외적인 환경의 변화로서 수개월이 소요되며, 중간규모인 3~5등급은 1개월 정도가 소요되는 CR이고, 하위 6~7등급은 화면이 수개인 프로그램 1본 정도의 규모를 나타낸다. 마지막으로 8~9등급은 매우 단순한 것으로 수 일 내에 완료될 수 있는 CR이다. <표 1>은 등급별 건수를 나타내고 있는데 8등급이 전체의 50%를 점하고 있다. 본 논문에서는 중간 규모의 4등급 시나리오인 '통합한도'를 선택하였다. 통합한도는 신용위기를 해결하기 위한 방안의 하나로 현재 상품별(예를 들어, 신용판매, 현금서비스, 할부금융 등)로 개별 설정된 한도를 통합해서, 회원별로 단일 한도를 부여하는 것이다. 이렇게 함으로 개인이 상품별로 사용한 한도금액을 공유, 통합관리하게 되어 개인별 신용저하에 대비하는 효과를 거둘 수 있게 된다. 결국, 본 연구에서는 동일한 '통합한도' 시나리오(신용판매와 현금서비스)에 대해 OO와 NON-OO 두 실험그룹이 유지보수 활동을 수행하고 그 차이를 분석하였다.

<표 1> 년도별 유지보수 건수 분포

등급	1	2	3	4	5	6	7	8	9	합계
2002	4	18	40	147	175	81	108	803	-	1376
2003	1	2	4	19	21	71	94	276	-	488

3.3 참여자

본 실험에는 총 12명의 전산실 실무인력이 참여하였다. NON-OO 그룹에는 현행시스템의 운영자 가운데 선발된 6명이 참여했는데, 이들은 과거 수년간 지속적으로 유지보수 업무를 수행하면서 한도관련 업무를 잘 숙지하고 있었다. 반면 OO 그룹에는 C++ 개발인력 6명이 참여하였는데, 약 3개월 동안 한도업무를 분석, 설계하면서 현행시스템의 담당자와 유사한 수준으로 해당 업무를 숙지하고 있었다.

3.4 실험절차

우선 한도업무 관련 현행시스템의 운영담당자와 협의하여 우선순위가 높은 CR에서 적합한 시나리오를 선정하도록 하였다. 선정된 시나리오의 업무담당자가 실험에 참여하게 되었고, 그들에게 필요한 작업을 설명하였다. 각 참여자는 작업시간, 작업내용, 작업 산출물을 매일 기록하고 보고하도록 하였다. 회사의 관리자는 참여자에게 작업의 중요성을 설명하고 과대, 과소하게

기록하지 말도록 하였다. 참여자는 작업이 실험의 일부라는 것을 알지 못하였으며, 일상적인 유지보수 활동으로서 작업내역을 보고하는 것으로 인식하였다. 단지, 이렇게 작업결과를 보고하는 것은 모든 유지보수에 대하여 적용되는 것으로 활동시간과 작업산출물을 개발주기에 따라 구분하여 보고하는 규정에 따랐다.

3.5 종속변수

유지보수성을 평가하기 위하여 시나리오를 시스템에 반영하기 위한 (1) 유지보수 공수와 (2) 유지보수 정도를 측정하였다[7]. 이들은 요구사항, 분석, 설계, 구현, 테스트, 배치 단계 등의 모든 개발주기에 걸쳐 평가되었다. <표 2>에서와 같이 (1) 유지보수 공수는 시간 기준으로, (2) 유지보수 정도는 문서의 경우 A4 용지를 기준으로, 프로그램은 LOC(Lines of Codes)를 기준으로 측정하였다. LOC는 측정기준에 차이가 있을 수 있지만[21], 공백과 주석문을 제외하고 계산하였다. 테스트는 테스트케이스 수를 측정하여, 테스트 양을 비교할 수 있게 하였다. 작업의 완료는 운용에 필요한 모든 기능이 결함 없이 배치된 것을 기준으로 하였다.

〈표 2〉 유지보수성 측정변수

구 분	변 수	변수의 측정
요 구 사 항	산출물 변경 정도	페이지
	요구사항 공수	명수 x 투입시간
분 석 설 계	산출물 변경 정도	페이지
	분석설계 공수	명수 × 투입시간
구 현	LOC 변경 정도	변경된 소스코드 줄 수
	변경 파일 수	소스코드가 변경된 파일의 수
	구현공수	명수×투입시간
테스트	테스트 파일 수	변경으로 인한 테스트 파일 수
	테스트케이스 수	완전한 테스트를 위한 테스트케이스 수
	테스트 프로그램	테스트 케이스 실행을 위한 프로그램의 줄 수
	테스트 공수	명수×투입시간
배 치	배치 파일 수	변경으로 인한 재배치 파일 비율
	테스트 공수	명수×투입시간

4. 결 과

앞서 설명한 바와 같이 명제의 검증에 필요한 종속변수는 (1) 유지보수 공수와 (2) 유지보수 정도였다. 먼저 유지보수 공수는 참여자의 작업내용과 기록을 취합하여 활용하였다. 양측에 서로 다른 항목을 포함함으로써 발생할 수 있는 차이를 최소화하기 위해 작업기록에 대해 참여자와 1 : 1 면접과 더불어 동일한 작업에 대한 양측 참여자와의 면접을 수차례 실시하였다. 우선 취합된 기록을 1차 분석하고, 담당자와 인터뷰 실시해서 과대계상, 과소계상, 항목불일치를 조정하였다. 조정된 작업결과를 재분석 한 후, 측정된 값에 대해 상담시간, 중간 요구사항 변경, 인수테스트와 같은 항목에 대한 차이를 보정하였다. 그 후 작업을 수행한 모든 참여자와 관리자를 소집하여 코드분석과 워크스루를 통해 측정에 오류가 있었는지 협의하였고, 그 결과 요구사항 변경에 따른 재작업 공수에 미세한 차이가 있어 이를 다시 보정하였다. 두 번째로 산출물 변경 정도는 유지보수로 인해 변경된 문서와 소스코드를 취합하여 측정하였다. LOC는 동일한 기준을 적용하여 측정하였다.

명제의 검증에 앞서 양측에서 작성한 산출물과 그 절차를 우선 이해하는 것이 필요한데, <표 3>에서 보듯이 OO와 NON-OO에서 작성한 산출물은 서로 상이했다. 이러한 차이는 적용한 방법론과 프로그램언어가 서로 차이가 있었기 때문이었다. 객체지향은 C++ 언어를 사용해서 UML

과 반복적 방법론을 적용하였고, 반면, 비객체지향은 C 언어를 사용하면서 전통적인 방식으로 산출물을 작성하였다.

〈표 3〉 단계별 유지보수 산출물 목록

	NON-OO	OO
요구사항	없 음	행동다이어그램, 이벤트목록, 유즈케이스명세
분석설계	개발내용분석문서	클래스다이어그램, 오퍼레이션명세, 테스트케이스, 파일목록
구 현	프로그램	프로그램
테스트	테스트 결과서 (단위 및 기본 테스트케이스)	테스트 프로그램 (통합), 테스트 결과서
배 치	배치목록, 실행파일	배치목록, 실행파일

서로 다른 방법론을 적용하면서 OO와 NON-OO의 유지보수 절차는 많은 차이가 있었다. <표 4>에서 보듯이, 객체지향 시스템에서 유지보수는 표준화된 UML에 의해 각 단계가 수행된 반면 비객체지향 시스템에서는 문서보다는 직접적인 '의사소통'에 많이 의존하였다. 요구사항 단계에서 객체지향 시스템은 기존 요구사항 문서를 수정한 반면, 비객체지향 시스템은 별도의 문서를 작성하지 않고 분석 및 설계단계를 수행하면서 산출물을 작성하였다. 유지보수가 필요한 영향분석 위해서 비객체지향 시스템에서는 기존 프로그램 파일을 열어서 이해하는 것이 필수적인 반면, 객체지향 시스템에서는 클래스와 오퍼레이션을 검토하면서 수행되었다. 이와 같이 산출물을 표준화하고 작성함으로 인해 구현단계에서도 많은 차이를 볼 수 있었다. 객체지향 시스템에서는 분석 및 설계단계에서 구현단계로 이행할 때 그 내용이 산출물에 담겨서 개발자가 이를 참조할 수 있었던 반면, 비객체지향 시스템에서는 분석설계자의 설명이 절대적인 역할을 하였다. 이러한 '의사소통'에 의존하는 방식은 필수적으로 설명되지 않거나, 잘못 설명된 분석설계 내용이 개발까지 전이되어 수정하려면 비싼 비용이 지불되었던 경향이 있었다. 테스트단계에서 비객체지향 시스템은 현업사용자에 의한 인수테스트가 실시되었고 이로 인한 요구사항의 변경이 발생하였다(앞서 설명한 바와 같이, 이로 인한 차이는 분석 전 보정되었음).

〈표 4〉 단계별 유지보수 절차

	비객체지향시스템 유지보수 절차	객체지향시스템 유지보수 절차
요 구 사 항	• 현업에서 CSR 접수 • IT부서에서 분석, 등급결정 • IT부서에서 CSR 배정 현업 인터뷰 (회의, 전화)	• (1)~(4) 동일 • (5) 행동다이어그램 수정 • (6) 유즈케이스명세 수정
분 석 설 계	• 현행프로그램 분석 • 수정항목 찾기 • 요구사항 정리하면서 산출물 작성 • 개발자에게 업무설명	• 수정항목 찾기 • 클래스다이어그램 수정 • 오퍼레이션 명세 수정 • 테스트케이스 작성 • 파일목록 수정 • 엔티티관계도 수정
구 현	• 프로그램 이해 • 프로그램 유지보수 (구현) • 디버깅 • 분석설계자가 검토 (개발 지원) • 프로그램 수정	• 설계문서 이해(오퍼레이션 명세) • 프로그램 유지보수 (구현) • 디버깅 • 단위 테스트 • 프로그램 수정
테스트	• 테스트 케이스 작성 • 프로그램 테스트 • 단위테스트 • 인수테스트	• 테스트 프로그램 작성 • 단위테스트(model, controller)
배 치	• Compile • Link	• Compile • Link

명제 1 : 설계 이전 단계에서의 유지보수성

객체지향 시스템과 비객체지향 시스템 간에 한도변경 시나리오에 대한 설계 이전 단계에서

의 유지보수성에 차이가 없다는 첫 번째 명제를 검증하기 위해 먼저 투입시간을 조사하였다. 표 5에서 보듯이 NON-OO의 유지보수 공수는 OO에 비해 약 2배 많이 소요되었다(1260 vs. 570). 요구사항 단계에서는 거의 차이가 없었는데(240 vs. 260), 이는 요구사항을 문서를 통해 같은 현업사용자에게 동일하게 서로 공유하였기 때문이다. 하지만 분석 및 설계단계에서는(1020 vs. 310) 약 3배의 차이가 있었다. 따라서 이러한 결과는 NON-OO 시스템이 OO 시스템에 비해 더 많은 유지보수 공수를 필요로 한다는 것을 보여준다.

〈표 5〉 설계 이전단계의 유지보수성

단계	측 정 치	NOO	OO
요구사항 단계	유지보수 공수	240	260
	유지보수 정도	없음	행동다이어그램 2장 사용사례 명세 2장
설계 및 분석 단계	유지보수 공수	1020	310
	유지보수 정도	설계 사양서 2장	클래스 다이어그램 1장 오퍼레이션 명세 5장 파일 목록 1장

한편 <표 5>는 설계 이전단계의 유지보수 정도, 즉 산출물 변경정도를 보여주기도 한다. 이 표에 따르면 요구사항, 분석 및 설계단계 산출물은 객체지향 시스템에서 비객체지향 시스템에 비해 더 많이 작성되었다. 이는 OO 그룹이 NON-OO 그룹에 비해 더 많은 산출물을 유지보수 해야 하는 사실을 감안하면 그리 놀라운 것도 아니다. 특히, 흥미롭게도 객체지향시스템에서 소요된 시간은 오히려 적었는데, 이는 비객체지향 시스템에서 '수정항목'을 인식하는 것과 인식된 내용 및 분석설계내용을 개발자에게 설명하는데 더 많은 시간이 소요되었기 때문이었다. 정리해 보면 OO 시스템은 NON-OO 시스템에 비해 훨씬 적은 공수가 필요하지만 산출물 변경에 있어서는 약간 더 많은 노력이 들었다. 하지만 전체적으로는 OO 시스템에 든 노력이 더 적으므로 첫 번째 명제는 기각된다고 판단할 수 있다.

명제 2 : 유지보수정도

두 번째 명제는 두 집단 간에 설계 이후 단계에서의 유지보수성에 차이가 있는지를 검증하는 것이었다. <표 6>은 구현단계에서의 투입 공수가 660대 215로 OO 그룹이 훨씬 적다는 것을 보여준다. 또한 테스트 단계에서도 약 6배 정도의 많은 차이가 있었는데(1103 vs. 182) 두 팀간의 개발방법론의 차이로 인해 NON-OO에서 시스템의 테스트에 많은 시간이 소요되었다. 즉, OO에서는 반복적인 방식을 적용함으로 인해 테스트케이스가 구현 전에 도출되었고, 반면 NON-OO에서는 폭포수모형을 적용하여 구현이 종료되고 나서 테스트케이스를 도출하였기 때문이었다. 이와 같은 차이는 다음 장에서 변경된 산출물과 더불어 자세히 비교하겠다. 한편 배치단계에서도 예외 없이 OO 시스템이 NON-OO 시스템에 비해 더 작은 공수가 투입되었다(30 대 17). 이러한 결과는 설계 이전단계에서의 투입공수를 비교한 결과와 다르지 않았으며 결국 OO 시스템의 유지보수 공수는 NON-OO 시스템의 그것에 비해 훨씬 적다는 것을 알 수 있다.

한편, 유지보수 정도를 살펴보면 구현단계에서 작성한 LOC는 객체지향 시스템에서 비객체지향 시스템에 비해 많았다(277 대 182). 그러나 업무를 구분해서 보면, 객체지향 시스템에서 신판 업무에서 작성한 LOC는 268줄로서 비객체지향 시스템에서 작성한 51줄에 비해 많았지만, 론패스 업무에서는 반대였다. 즉, 비객체지향 시스템에서는 151줄을 작성하였고, 객체지향시스템에서는 9줄에 불과했다. 이는 프로그램 재사용으로 인한 차이로서, 객체지향 시스템에서는 신판에서 작성한 대부분의 로직을 론패스에서 수정

없이 사용하였다. 반면, 비객체지향 시스템에서는 로직이 중복해서 프로그램 되었다. 테스트 단계의 차이는 객체지향 시스템에서는 테스트 프로그램(CUnit, LOC = 283)을 작성하였다는 점이 큰 차이였는데, 이로 인해 산출물 작성 부담이 많았다. 또한 배치단계에서도 객체지향 시스템에서는 프로그램이 클래스로 나누어지게 되어서 컴파일과 배치한 파일 수가 많았다. 비록 OO 시스템은 프로그램과 파일에 좀 더 많은 유지보수 활동을 하게 되지만 투입공수를 함께 고려해 볼 때 전체적인 유지보수에 대한 투입노력은 OO 시스템이 NON-OO 시스템에 비해 적다는 것을 알 수 있고, 따라서 명제 2는 기각된다고 판단한다.

〈표 6〉 설계 이후단계의 유지보수성

단 계	측 정 치	NOO	OO
구 현	유지보수 공수	66(	215
	LOC 변경	182(51, 151)	277(268, 9)
	파일 변경	5(2, 3)	11(6, 5)
테스트	유지보수 공수	1103	182
	테스트 케이스	15(8, 7)	39
	테스트 파일 수	1	4
	LOC 테스트 스크립트	NA	283(278, 5)
배 치	유지보수 공수	30	17
	컴파일 파일 수	2	6(5, 1)
	배치 파일 수	2	6(5, 1)

주) 괄호안 숫자는 각각 신판업무와 론패스

5. 결 론

급격하게 변화하는 경영환경 가운데 경쟁우위를 확보하기 위한 대안으로 객체지향은 주목을 받고 있다. 그러나 객체지향이 주장[5]과는 달리 생산성, 재사용성, 유지보수성을 향상시키는지에 관해서는 회의적인 시각도 적지 않다[13]. 이러한 배경으로 객체지향의 특징으로 제시되고 있는 상속, 다형성, 은닉성이 프로그램을 쉽게 이해하고, 쉽게 수정하는데 오히려 걸림돌이 될 수 있다는 흥미로운 결과가 발표되기도 하였다[14]. 따라서 객체지향을 무분별하게 받아들이기 보다는 비객체지향적 방식에 비해 투자에 상응하는 효과를 거둘 수 있는지 의문을 제기할 필요가 있다. 이러한 의문과 관련하여 객체지향을 비객체지향과 직접적으로 비교하여 유지보수성을 검증한 연구는 많지 않았다[6, 14]. 이들 연구는 공통적으로 객체지향이 비객체지향에 비해 언제나 유지보수성을 향상시키지 못한다는 부정적인 결과를 보여주고 있다. 부정적인 관점을 뒷받침하는 이유로 설계의 중요성을 꼽을 수 있는데, 객체지향은 잘 설계하기 어려우며, 잘 설계될 경우에 한해 유지보수성이 향상된다는 연구를 주목할 필요가 있다[11, 12].

본 논문에서는 객체지향을 적용할 경우 유지보수성이 향상할 것이라는 주장[9]을 실제로 카드시스템의 '통합한도' 변경 시나리오를 통해 검증하였다. 그 결과 객체지향 시스템은 비객체지향 시스템에 비해 산출물의 유지보수는 많았지만, 유지보수에 투입된 공수는 오히려 적은 것으로 나타났다(약 1/3). 특히 객체지향 시스템의 경우 산출물의 재사용이 상당히 높아(신용판매의 산출물을 론패스에서 재사용) 유지보수성이 향상된 것은 매우 흥미로웠다. 이런 차이는 객체지향의 '제품' 측면의 우월성(예 : 인터페이스를 적용하여 비즈니스로직의 은닉)과 UML 및 반복적 개발방법론을 적용함으로 인한 '절차'측면의 우월성에서 설명할 수 있다[22]. 우선 설계측면에서 보면, 객체지향시스템의 개발에서는 (1) 설계를 업무로직까지 매우 자세하게 문서화하였고, (2) 설계문서는 표준에 따라 작성되었고, (3) 작성된 설계문서는 개발에 적절하게 활용함으로 인해 (A) 개발자의 개인적 편차를 줄일 수 있었다. 예를 들어, 비객체지향에서는 동일한 SQL

문이 개발자에 따라 서로 다르게 작성되는 경향이 있었다(모든 속성 vs. 일부 속성). 또한 반드시 돌려주어야 하는 메시지와 로그의 기록을 누락하기도 하였다. 객체지향에서는 이러한 개발자의 편차가 아키텍처 중심의 설계(폐쇄형아키텍처 원칙에 따라 엔티티레이어를 두고, SQL의 중복을 허용하지 않고 재사용), 프레임워크의 적용으로 인해 상당히 감소하였다. 또한 (B) 객체지향에서는 문서에 의해 부정확한 의사소통이 방지되었으나, 비객체지향에서는 분석설계자가 개발자에게 일일이 설명, 지원이 필요하였다. 이러한 부정확한 의사소통은 해야 될 설명을 누락하거나 부정확하게 함으로 (C) 개발자의 임의의 코딩으로 인해 결함이 발생하였고, 또한 이러한 결함이 테스트에서야 발견되어 '비싼' 변경을 수반하였다(<표 7> 참조). 둘째로 객체지향 시스템에서 인터페이스를 사용함에 따라 구현은닉이 유지되었다. 따라서 (A) 인터페이스를 추가하고 구현하면서 구현은닉, (B) 전역변수의 사용이 감소함에 따라 수정이 용이했다. 비객체지향에서는 업무별로 전역변수를 선언하여 사용하였고, 누구나 수정하는 것이 허락되어 예기치 못한 결과를 초래할 위험이 있었다. 반면에 객체지향에서는 구조적으로 전역변수의 사용이 어려워, 특정 클래스에 변수를 선언하고 참조하게 하였다. 세 번째로, 컴포넌트 방식을 적용함으로 (A) 기능의 통합, (B) 재사용 증가, (C) 기능분리가 가능했다. 네 번째로 아키텍처의 사용으로 인해 (A) 의존성 관리에 따라 변경의 최소화, (B) 전사적인 수정의 용이, (C) 개발자의 차이가 감소했다. 마지막으로 테스트에 자동화를 적용해서 (A) 분석설계자의 역할이 감사 기능으로 확장되었고, (B) 변경에 따른 회귀테스트 (Regression test), (C) 테스트 결과의 관리가 용이했다.

<표 7> 단계별 유지보수 차이

단 계	항 목	비 객 체 지 향	객 체 지 향
분석설계	의사소통 (분석설계팀과 개발팀)	높은 언어의사소통 (Code Inspection, 개발자 설명)	설계문서 의사소통 (업무검토, 기술검토)
	설계 문서	매우 적음	자세함
	개발오류 수정	개 발	설 계
	문서 작성	사내 표준	UML
개 발	개발자 편차	많음(SQL문, 메시지, 로그)	적음(표준화)
	전역변수	많음(별도 업무별 전역변수파일)	국지화
	재사용	적음(프로그램 복사)	많음(클래스 단위)
	API	전역변수	인터페이스
	기능분리	어려움(전역변수 의존성)	용이(컴포넌트 의존성 관리)
	전사적 수정	어려움	용이(FW의 변경)
	의존성관리	없 음	있 음
테스트	테스트 케이스	개발하면서 추가. 인수 테스트	분석설계자(미리 정의 (감사기능))

객체지향 시스템의 사용에 따른 단점도 지적할 수 있다. 본 논문에서 지적한 바와 같이 객체지향을 사용하게 되면서 (1) 산출물 작성이 증가(분석설계, 테스트케이스), (2) 프로그램이 더 많이 분할되어 관리노력이 증가(구현, 테스트 파일, 배치 파일), (3) 분석설계와 개발의 분리에

따른 개발자의 직무만족도, 업무지식 축적기회가 감소, (4) 공통 컴포넌트의 재사용으로 인해 패키지 간 의존성을 심화시키는 경향이 있었다. 향후에는 유지보수성을 향상하기 위한 방안으로 업무규칙의 분리, 패키지의 의존성 관리, 유지보수 영향분석 강화, 모델과 프로그램의 동기화를 위한 모형에 대한 연구가 필요하다. 또한 ISO 9126의 품질요소로서 성능과 유지보수의 상충에 대한 연구를 위한 리팩토링과 객체지향에서도 유지보수모델의 자동화에 대한 연구가 필요하다.

본 논문은 그 결과가 사례에 국한될 수 있다는 점에서 일반화에 신중을 기해야 한다. 여기서 실험한 유지보수는 중규모인 CR 4등급 시나리오의 결과이며, 적은 규모인 7, 8 등급이 전체의 약 70%를 차지 한다는 점을 감안하면 그 차이는 적어질 것으로 예상할 수 있다. 또한 지금까지의 시스템 개발 관행을 보면 전통적인 시스템 개발과정이 이론적으로 제시되는 것과 같이 엄격하게 진행되지 않는 측면들이 있어 각 종 산출물이나 관련 태스크들의 완전성이 떨어지는 경우가 많다. 이러한 경우, 실제 실험연구 결과에서 객체지향 기술의 유지보수성이 우월하게 나타나더라도 이를 객체지향 기법이나 방법론의 우월성 보다는 전통적인 시스템의 개발방법론이 열악해서 나타나는 현상으로 간주할 수도 있다. 하지만 비록 이러한 시각이 일부 타당성을 가지더라도 전통적인 방법론의 열악함이 실제 흔히 나타나는 현실이며 이러한 현실을 바탕으로 객체지향 기법의 유지보수성을 비교 분석하는 것도 실증연구로서 유의미한 작업이란 점이 강조되어야 하겠다. 향후 연구에서는 이러한 부분을 보다 심도 있게 살펴보는 것이 필요하다. 마지막으로 본 실험에 있어 객체지향의 전환에 따른 현행시스템 운영자의 저항이 적지 않았다는 점을 감안하면 객체지향의 도입에 따른 변화관리가 필요하겠다. 유지보수는 단순히 산출물 또는 프로그램 측면에서만 추론할 수 있는 문제가 아니라, 요구사항 종류, 유지보수 절차, 조직과 같은 다양한 요인에 의해 영향을 받는 까닭에 체계적이고 학제적인 연구가 필요하다고 하겠다.

참 고 문 헌

[1] Basili, V.R. and Lanubile, F., "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, 1999, pp. 456-473.

[2] Bass, L., Clements, P. and Kazman, R., "Software Architecture in Practice", Addison Wesley, 1998.

[3] Bengtsson, P. and Bosch, J., "An Experiment on Creating Profiles for Software Change", *Annals of Software Engineering*, Vol. 9, 2000.

[4] Binkley, A.B. and Schach, S.R., "Inheritance Based Metrics for Predicting Maintenance Effort : An Empirical Study", TR-97-05, 1997.

[5] Booch, G., "Object-Oriented Developments", *IEEE Trans. Software Engineering*, Feb. 1986, pp. 211-221.

[6] Briand, L.C., Bunse, C., Daly, J.W., Differding, C., "An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents", *Empirical Software Engineering*, 2, 1997, pp. 291-312.

[7] Briand, L.C., Bunse, C. and Daly, J., "A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs", *IEEE Transactions on Software Engineering*, Vol. 27, No. 6, 2001, pp. 513-529.

[8] Chaumun, M.A. Kabaili, H., Keller, R.K. and

Lustman, F., "A Change Impact Model for Changeability Assessment in Object Oriented Software Systems", *Science of Computer Programming*, 45, 2002, pp. 155-174.

[9] Coad, P. and Yourdon, E., *Object-Oriented Design*, Prentice-Hall, 1991.

[10] Daly, J., Brooks. A., Miller, J., Roper, M. and Wood, M., "Evaluating Inheritance Depth on the Maintainability of Object Oriented Software", *Empirical Software Engineering*, Vol. 1, No. 2, 1996, pp. 109-132.

[11] Deligiannis, I., Shepperd, M., Roumeliotis, M. and Stamelos, I., "An Empirical Investigation of an Object Oriented Design Heuristic for Maintainability", *The Journal of Systems and Software*, 65, 2003a, pp. 127-139.

[12] Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M. and Shepperd, M., "A Controlled Experiment Investigation of an Object-Oriented Design Heuristic for Maintainability", *The Journal of Systems and Software*, To Appear, 2003b.

[13] Glass, R.L., "The Realities of Software Technology Payoffs", *Communications of the ACM*, Vol. 42, No. 2, 1999, pp. 74-79.

[14] Hatton, L., "Does OO Sync with How We Think?", *IEEE Software*, May/June 1998, pp. 46-54.

[15] Harrison, R., Counsell, S. and Nithi, R., "Experimental Assessment of the Effect of Inheritance on the Maintainability of Object Oriented Systems", *The Journal of Systems and Software*, 52, 2000, pp. 173-179.

[16] IEEE, *IEEE Software Engineering Standards Collection*, IEEE, 1993.

[17] ISO, IEC, "ISO/IEC 12207 : 1995, Information Technology - Software LifeCycle Processes", Geneva : ISO/IEC, 1995.

[18] ISO 9126, Software Product Quality Characteristics, http://www.cse.dcu.ie/essiscope.

[19] Johnson, R.A., "The Up's and Down's of Object-Oriented Systems Development", *Communications of ACM*, Vol. 43, No. 10, 2000, pp. 69-73.

[20] Johnson, R.A., "Object-Oriented Analysis and Design-What Does the Research Say", *Journal of Computer Information Systems*, Spring 2002, pp. 12-15.

[21] Jones, C., *Programming Productivity*, New York : McGraw Hill, 1986.

[22] Kan, S.H., *Metrics and Models in Software Quality Engineering*, 2nd Ed., Addison Wesley, 2003.

[23] Kan, S.H., Dull, S.D., Amundson, D.N., Lindner, R.J. and Hedger, R.J., "AS/ 400 Software Quality Management", *IBM Systems Journal*, Vol. 33, No. 1, 1994, pp. 62-88.

[24] Lientz, B.P. and Swanson, E.B., "Discovering Issues in Software Maintenance", Data Manage, 16, 1978, pp. 15-18.

[25] Lindvall, M. and Runesson, M., "The Visibility of Maintenance in Object Models : An Empirical Study", *The Proceedings of the International Conference on Software Maintenance*, 1998, pp. 54-62.

[26] Oman, P., Hagemeister, J. and Ash, D., A Definition and Taxonomy for Software Maintainability", report SETI, Report 91 08 TR, University of Idaho, 1991.

[27] SEI Software Technology Review, Maintainability Index Technique for Measuring Program Maintainability, URL : http://www.sei.cmu.edu, 2002.

[28] Zvegintzov, N., "Software should live longer", *IEEE Software*, Vol. 15, No. 4, 1998, pp. 19-21.

저자소개

임 좌 상

호주 뉴사우스웨일즈 대학에서 정보시스템 박사를 취득하고 현재 상명대학교 미디어학부 교수로 재직중이다. 그는 객체지향/컴포넌트 개발, 소프트웨어 품질공학 및 감성공학 등의 주제와 관련하여 컨설팅 및 개발 프로젝트를 수행하였으며 국내외 유명 학술지에 다수의 논문을 발표하였다.

정 승 렬

미국 사우스 케롤라이나 대학에서 경영정보학 박사를 취득하고 현재 국민대학교 비즈니스IT전문대학원 교수로 재직중이다. 그는 ERP 시스템을 포함하여 다수의 공공분야 및 금융권 대형 시스템 개발 프로젝트에 참여하였으며 유명 국내외 저널에 시스템 개발 및 구현, 정보시스템 감리, ERP, process management 등의 주제와 관련하여 많은 논문을 발표하였다.

◈ 이 논문은 2004년 6월 15일 접수하여 1차 수정을 거쳐 2004년 8월 18일 게재확정되었습니다.