□研究報告□

韓國 軍事運營分析 學會誌
第15卷 第1號, 1989. 6. 30.

소프트웨어 재사용과 설계에 관한 고찰
(Reusable Software Design Guidelines)

윤 창 섭*

요 약

소프트웨어 위기(software crisis)라는 용어가 소프트웨어 공학분야의 연구보고서에서 자주 언급되고 있는 이유는, 오늘날의 소프트웨어 개발환경에서 소프트웨어의 품질과 생산성의 문제해결이 매우 어렵기 때문이다. 소프트웨어 개발과 관련되는 프로그램의 설계, 프로그램의 명세서, 개발방법론 및 기술과 도구들 중에서 기존의 개발사업에서 이미 사용하였던 요소들을 새로운 개발사업에서 효율적으로 재사용할 수 만 있다면, 품질과 생산성에 관한 위험 부담을 최소화할 수가 있다.

본 연구는 지금까지 연구발표된 연구보고서를 중심으로 소프트웨어 재사용의 대상과 잇점, 재사용에 따른 장애요인들을 소개하였고, 재사용을 목적으로 소프트웨어를 설계할 때에 고려할 몇가지 지침들을 고찰하였다.

I. 서 언

일반적으로 관리자는 위기상황에서 미래예측의 불확실성 때문에 적절한 의사결정을 하기가 어렵다. 소프트웨어 관리자의 경우도 개발에 대한 위험부담이 따르기 때문에 의사결정이 매우 어려워서 오늘의 소프트웨어 개발환경을 "소프트웨어 위기"로 묘사하고 있다. 소프트웨어 개발단계에서 위기상황으로 인식되는 주요 원인은,

- 개발기간의 지연,
- 개발비용의 초과,

* 국방대학원

· 사용자 요구사항의 불만족,

· 소프트웨어 신뢰도와 가용도의 저조,

등으로 알려져 있다.

컴퓨터 기저 시스템(computer based system)에서 소프트웨어는 점점 복잡화되고 있는 반면에, 하드웨어는 비용면에서 점점 저렴화되고 있는 추세에 있다. 이러한 추세 때문에 전산비용(cost of computing)중에서 소프트웨어 비용이 지배적인 고려 요소가 되고 있다. [1] 소프트웨어의 개발과 개발완료 이후의 운용정비에 소요되는 비용증대의 원인은 소프트웨어 규모가 대형화, 복잡화 되면서 개발에 투입되는 고급인력의 인건비 증대와 생산성 향상의 저조 [2], 운용정비 대상 소프트웨어의 수적증가와, 오류수정과 조정 및 성능개선을 위한 노동 집약적인 처리과정 때문이다. [3]

소프트웨어 재사용(software reuse)은 소프트웨어의 품질과 생산성을 개선하고, 결과적으로 소프트웨어 개발기간을 단축하며, 개발과 운용정비 비용을 개선하는데에 기여한다. [4]

본 연구는 지금까지 연구 발표된 연구보고서를 중심으로 소프트웨어 재사용의 대상과 잇점, 재사용에 따른 장애요인들을 소개하고, 재사용을 목적으로 소프트웨어를 설계할 때에 고려할 몇가지 지침들을 고찰하여 제시하고자 한다.

Ⅱ. 재사용성(Reusability)의 개념

1. 정 의

재사용성이란 소프트웨어 구성단위(software component)가 갖고 있는 능력으로써 그 구성단위가 최초에 개발되었던 응용분야(application)가 아닌 새로운 응용분야에서 반복적으로 사용되는 능력을 말한다. 효율적으로 재사용되기 위하여 새로운 응용분야의 요구사항에 적합하도록 소프트웨어 구성단위에 수정작업이 이루어질 수도 있다. [5]

재사용성은 소프트웨어 구성단위의 수준에서(component level)만 재사용되는 정도로 표현되고 있고 응용분야의 수준(application level)에서는 고려되지 않는다. 재사용은 응용분야가 시행하여야할 전체 기능을 구성하는 부분적인 기능에 해당하는 subprogram 또는 subsystem을 어떻게 재사용하도록 하는가에 관심을 갖고 있다. 재사용의 형태는 calling component 또는 called component를 반복 사용하는 경우이다.

재사용성과 이식성(portability)의 개념이 가끔 상호 동일한 의미로 혼용되고 있으나 이식성은 특정 응용분야 전체 소프트웨어가 갖추고 있는 능력으로써 최초의 개발 목적환경이 아닌 새로운 환경에서 반복, 재사용되는 능력을 뜻하며, 새로운 운영환경에 적합하도록 수정 작업이 이루어질 수도 있다. [5]

요약한다면 전체 소프트웨어(entire application)가 새로운 환경으로 이식된다고 하고 그 소프트웨어의 구성단위가 새로운 응용분야의 소프트웨어 구성단위로 재사용된다고 하여야 한다.

두 개념간의 큰 차이점은 재사용성은 설계에 관한 문제(design consideration)이고 이식성은 구현에 따른 문제(implementation consideration)로 구분할 수 있다. 재사용성을 극대화하기 위하여 반드시 이식성을 고려하여야 한다.

재사용을 할 것인가 혹은 재사용을 하지 않을 것인가의 판단기준은 수정에 소요되는 비용과 재사용으로 절감되는 비용간의 선택에서 결정된다. 그림 1은 이들의 관계를 제시하고 있다.

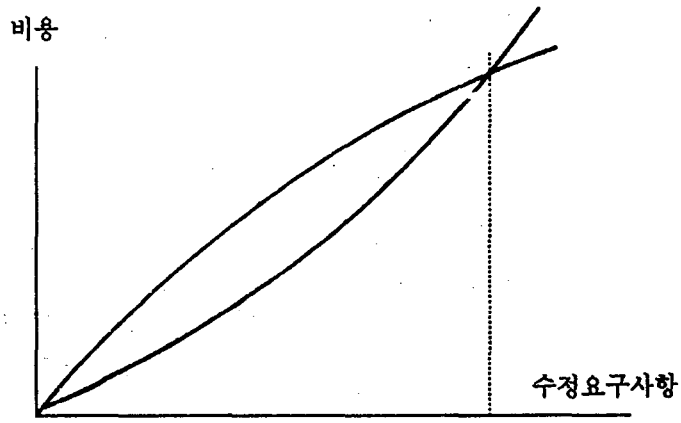


그림1 수정요구사항과 재사용의 비용관계

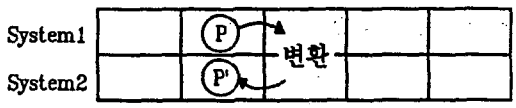
2. 소프트웨어 재사용의 고려대상

일반적으로 소프트웨어의 재사용을 기존의 운용중인 source code모듈의 반복적인 사용이라는 협의의 의미로만 인식하고 있으나 새로운 소프트웨어를 개발할 때에 기존의 다른 사업에서 사용되었던 모든 정보를 재사용하는 광의의 의미로 인식하는 것이 바람직하다.

재사용을 위한 설계와 코딩으로 얼마의 비용을 절감할 수 있는가를 Raytheon's Missile System Division's Information Processing System Organization의 연구 보고서[4]에서 살펴보면 설계, 코딩, 시험, 문서화 작업노력의 60%를 절감할 수 있었고, 그 절감된 시간과 노력만큼을 개발대상 시스템의 특수한 부분에 할당할 수 있다는 경험적 결과가 제시되고 있다. 또한 프로그래머가 learning curve에 의하여 3번 이상만 동일한 논리구조(logic structure)를 반복 사용하게하면 50%

의 생산성을 향상한다고 보고 되고 있다. 또한 프로그래머가 작성하지 않았던 프로그램이라도 프로그램을 수정(modifying)할 때에 재사용의 설계지침에 의한 일관성 있는 프로그램 형식(consistent programming style)을 이해하고 있기 때문에 유지, 보수 단계에서도 효율적이라고 하였다. 그러나 소프트웨어 수명주기 단계중 어느 한 단계에서 생산성 향상이 이루어졌다하더라도 수명주기 전체의 생산성 향상에는 크게 기여하지 못한다. (2)

소프트웨어는 수명주기 각 단계를 거쳐서 개발되고 유지, 보수된다. 수명주기는 계획단계, 개발단계, 유지·보수단계로 이루어지고 개발단계는 설계, 코딩, 시험단계로 세분화 된다. 소프트웨어는 각 단계별로 여러 형태의 생산물(product, deliverable)로 변환하면서 개발되게 된다.



System 수명주기의 단계

위의 그림에서 system 1에 있는 소프트웨어 생산물 P는 system 2에서 재사용이 가능한 것으로 식별되어서, 적절한 수정 작업을 거쳐서 하나의 생산물 P'가 되었음을 나타내고 있다. 소프트웨어를 수명주기의 각 단계별 생산물로 생각하면 반드시 모듈의 source code만이 재사용으로 고려되어야 할 대상이라고 생각할 필요는 없다.

소프트웨어 재사용에 고려되는 대상은 아래와 같다. (6)

- 소프트웨어 개발 방법론(methodology).
- 요구사항과 설계에 관한 명세서(specification).
- Source code와 모듈(module).
- 소프트웨어 도구 및 개발 환경(tools and support environment).
- 분석 자료(analysis data).
- 시험 자료(test information).
- 운용 정비에 관한 자료(maintenance information base).

위에서 제시한 바와 같이 소프트웨어의 무엇을 재사용할 것인가를 결정하기 위하여 수명주기 전체단계에서 생산되는 모든 생산품과 그 생산물이 만들어지는 처리과정도 그 대상으로 하여 고려되어야 한다. 예를들면 일반적으로 많이 사용되는 기능(function)이 필요할 때에 그 기능에 해당하는 source code를 직접 사용할 수 없다하더라도 그 기능에 대한 사용자 요구사항 명세서나 또는 설계 명세서를 재사용할 수도 있다.

3. 재사용의 잇점과 장애요소

소프트웨어를 재사용함으로 얻어지는 잇점

은 소프트웨어의 복잡도, 사업의 규모, 새로운 개발분야와 기존 응용분야의 차이점에 따라 다르겠지만 일반적으로 그 잇점은 아래와 같다. [6]

- 개발 및 운용정비 비용의 절감
- 생산성 향상과 개발기간의 단축
- 소프트웨어 신뢰도의 개선
- 자원의 효율적 활용

소프트웨어 수명주기의 단계별 생산물을 재사용하게 되면 그 자체를 직접 개발하는 것 보다 경제적인 면에서 비용절감의 잇점이 있고, [4, 7] 이미 기존의 응용 분야에서 사용되는 과정에서 신뢰도가 개선된 생산물을 재사용하게 되므로 신뢰도와 품질개선의 효과도 기대할 수 있다. 구성품을 재사용하게되면 생산성의 향상으로 개발기간을 단축하고 재사용 만큼의 절감된 노력을 기타 부분의 개발노력에 투입할 수 있는 기회를 갖게 되어서 자원의 효율적인 면에도 기여할 수 있다.

재사용으로 고려되는 대상이 복잡한 소프트웨어일수록 재사용의 노력과 비용도 또한 많이 소요되게 된다. 복잡한 시스템은 이해하기가 어려울 뿐만 아니라 개발중인 소프트웨어에만 특수한 사용자의 요구사항에도 적합하도록 수정작업이 이루어져야 하므로 수정에 따른 디버깅도 실시되여야 한다. 그러나 재사용으로 시스템을 구축하게 되면 장차 재사용의 기회가 점진적으로 증대하게되고 재사용된 빈도 만큼의 효과가 기대된다. 연구결과[2]에 의하면 재사용을 통하여 40%의 생산성 향상이 있었으나 비용면에서는 재사용을 위한 소프트웨어를 개발하는데에 소요되는 최초의 비용이 20-25% 추가 소요되었다는 점이다. 따라서 손익분기점은 재사용의 빈도와 적용의 의지에 따라 결정된다고 보아야 한다.

소프트웨어 재사용은 경제적, 기술적인 관점에서 바람직한 방향이면서 동시에 기존의 전통적인 개발원칙의 수정과 변화를 초래하게 한다. 왜냐하면 소프트웨어의 재사용이 소프트웨어를 장차 재사용할 수 있도록 생산하는 시도로 될 수가 있고, 또는 단순히 기존의 소프트웨어를 반복 사용하는 측면으로도 추구될 수 있기 때문이다. 하나의 개발조직에서 재사용 개념을 적용하고 이를 개발에 실행하는 데에는 초기에 많은 어려움이 예상된다. 재사용에 따른 장애요소를 열거하면 아래와 같다. [6].

- 소프트웨어 개발관리자와 개발담당자들의 저항
- 재사용 기술적용 등기의 결여
- 재사용에 부적당한 기존 소프트웨어의 설계
- 표준화의 부재
- 사회적, 법적 장애

소프트웨어 개발관계자들은 다른 조직에서 개발한 생산물을 재사용한다는 것을 부정한 것으로 생각하고 있고('not invented here') 재사용의 결과로 예상되는 예산의 삭감과 인력의 감축등을 우려하여 재사용기술을 적용하는데에 소극적이다. 또한 재사용을 목적으로 설계하지 않았던 소프트웨어를 사용할 때에 통상 성공적이지 못했다는 과거의 경험도 부정적 요인으로 작용한다.

재사용이 확대 적용되기 위해서는 반드시 표준화문제가 해결되어야 한다. 모듈의 기능을 명세화 하고 모듈의 사용시에 필요한 접속관계(interface)의 명세화를 표준화하여 이를 기반으로 cataloguing, searching index, component interface mechanism 등에 관한 조직내, 외적 준비와 절차가 마련되어야만 재사용의 기회를 증진할 수 있다.[8] 그러나 오늘날의 현실은 그렇지 못하고 단지 조직내의 문제로만 해결하려고 하고 있다. 소프트웨어 보호, 지적소유권에 관한 사회적, 법적 조치도 원활한 재사용을 권장하기 위하여 마련되어야 하나 아직도 미비한 상태에 있다.

소프트웨어의 재사용은 소프트웨어 위기"에 대비할 수 있는 잠재적 대비책이라 할 수 있다. 재사용의 목표는 생산성의 향상과 품질 향상으로 개발비용과 운용, 정비비용을 절감하고 신뢰할 수 있는 고품질의 소프트웨어 구성품을 생산하는데 있다. 효율적인 재사용의 시행은 장차 예상되는 이득을 위한 기반을 구축하기 위해서 초기단계에서 부가적인 투자를 필요로 한다. 재사용에 있어서 사업의 관리자, 전문요원들의 참여가 무엇보다도 중요한 과제이다.

Ⅲ. 재사용성을 위한 설계 지침

재사용성(reusability)은 소프트웨어를 설계할 때에 고려되어야할 가장 중요하고 최우선적인 과제이다. 갖고자하는 소프트웨어가 재사용성을 염두에 두지 않고 설계되었다면 비록 소프트웨어의 코딩과 문서화가 잘 되어 있다 하더라도 재사용성을 기대할 수는 없다. 설계시에 재사용성을 고려하지 않고 작성된 소프트웨어는 구성품과 구성품간의 상호 접속관계가 재사용되기 어렵기 때문이다.

지금까지 알려진 기본적인 최적의 설계원칙을 충실히 준수하였다 하더라도 재사용성의 특성을 갖춘 소프트웨어를 개발할 수는 없다. 설계원칙의 적용은 단지 재사용성의 목표만을 지원하는 것이 아니고 여러 특성을 감안한 제원칙 들이기 때문이다.

재사용성에 관한 연구 보고서를 수집하여 1984년 9월에 "Special Issue of the IEEE Transactions on Software Engineering"이 발간 되었다.[9] 재사용성에 관한 소프트웨어 공학적인 연구가 제한적이면서 최근의 관심사항임을 알 수 있다.

본 장에서는 소프트웨어의 재사용을 위한 설계지침을 고찰하고자 한다. 설명에 앞서 먼저 일반적인 설계지침을 제시한다면

· 모델의 사용(use of model)

· 계층구조의 설계(layered architecture)

· 접속관계 고려(interface consideration)

· 효율성의 선택(efficiency trade-offs)등

이 있다. [5]

설계지침들의 목적은 소프트웨어 설계자가 설계에 관한 결심을 하고자할 때에 이들을 고려함으로써 재사용 가능한 소프트웨어를 설계할 수 있도록 하는데 있다.

1. 모델의 사용

일반적으로 모델(model)은 공통된 관점을 정의하는데에 사용된다. 소프트웨어 개발시 분석가들은 분석모델을 사용하여 소프트웨어의 요구사항을 분석하고 정의할 수도 있고 이러한 요구사항들을 기존의 능력과 비교하여 얼마 만큼이나 일치하는가를 판단한다. [10]

소프트웨어의 재사용을 고려할 때는 통상 다음의 두가지 문제들을 해결해야 한다. 그 중 하나는 기존의 소프트웨어를 재사용하기 위하여 새로운 시스템을 어떻게 설계하는가이고 또 다른 문제는 개발하고자 하는 소프트웨어가 지금까지는 구현되지 않았지만 그러나 장차 예상되는 여러 형태의 새로운 시스템에서 재사용할 수 있는 소프트웨어를 어떻게 설계할 것인가 이다. 이러한 문제들이 모델의 사용으로 해결 가능하다. 왜냐하면 모델은 시스템의 기능을 표현하고 있고[10] 그 기능에 해당하는 모듈의 재사용을 모델에서부터 판단할 수가 있기 때문이다. [11]

재사용 가능한 소프트웨어를 구현하기 위한 모델을 형성하는 과정은 실세계의 문제를 인식하고 그 문제와 관련되는 발생 가능한 문제들을 축출하여 이를 일반화하고, 일반화된 기능 중심의 모델을 형성하는 것이다. [12] 개발하고자 하는 시스템에 대한 사용자의 요구사항은 그 개발 대상이 다루어야할 문제들만을 구체적으로 정의하고 있개 때문에 이러한 사용자의 요구사항만을 만족하는 소프트웨어를 개발하기 전에, 반드시 먼저 범용으로 사용할 수 있도록 사용자의 직접적인 요구사항을 일반적인 요구사항으로 전개하면서(generalized) 또한 원래의 요구사항을 만족할 수 있는 소프트웨어(customized)가 될 수 있도록 설계하여야 한다.

이러한 접근방법을 사용하게 되면 단기적인 사용자의 요구사항을 만족하는 소프트웨어를 설계할 수 있고 또한 유사한 소프트웨어의 개발시에 재사용 가능한 시스템을 설계할 수가 있다. 모델은 사용자의 직접적인 요구사항뿐만 아니라 장기적인 필요성까지 수렴하는 모델이 되며 그 모델을 이용하여 소

프트웨어 기능과 기능간에, 소프트웨어와 외부 환경과의 접속관계를 규정하고 설계할 수가 있다.

2. 계층구조의 설계

계층구조란 시스템을 순서화된 계층으로 분할(partitioned)한 구조를 의미하며 분할된 계층은 그 계층에 적합한 기능만을 실행한다. 또한 각각의 계층간에는 실행하고자 하는 기능을 지시하는 계층과 실행을 하는 계층으로 상호관계를 갖고 있고 계층간에 상호 규정된 접속관계를 유지하고 있다.

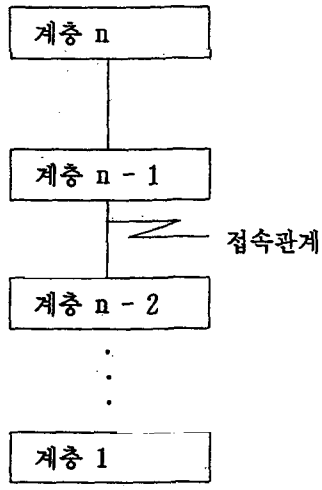


그림2 계층구조의 설계

계층구조의 설계 개념은 그림2에서 제시한 바와 같이 소프트웨어가 수행할 전체의 기능을 계층으로 설계하고 계층간의 접속관계를 설정하는 것이다. [5]

소프트웨어를 계층구조로 설계하게 되면 개별계층이 독립적인 분할 기능을 수행함으로 재사용하려고할 때에는 이를 대치하여 재사용의 목적에 맞도록 조정할 수가 있다. 또한 대치할 때나 또는 변경 및 수정을 할때에 파급효과를 국지화할 수 있으므로 수정노력을 최소화하게 하고 계층간의 접속관계를 표준화 한다면 개별계층의 재사용 기회를 증진하는 효과도 있다.

3. 접속관계의 고려

소프트웨어 구성단위의 접속관계는 그 구성단위가 외부 환경과 관련되는 역할을 규정하고 있고 접속관계를 이용하여 구성단위를 명확히 설명할 수가 있다. 그 구성단위를 재사용하고자 하는 설계자에게 접속관계에 관한 정의 자료는 재사용 가능성 여부를 판단하는 의사결정에 유익한 지침이 된다.

소프트웨어 설계자는 재사용성을 제고하기 위하여 접속관계를 명확히 인식하고, 이를 설계하여야 하고, 완벽한 문서화 과정을 통하여 정의하고 필요시 참조할 수 있도록 문서로 남겨야 한다. Ada 환경하에서 소프트웨어를 개발할 때에 top-down 또는 bottom-up 개발 절차를 이용하는데, 이들 절차 모두가 system에서 이미 정의된 모듈(package 또는 subprogram)들을 이용하여 system building-up 하게 된다. [12, 13] 이때에 sys-

tem에서 정의된 모듈에 대한 적절한 문서화 작업과 문서가 준비되어 있지 않았다면 설계자는 무엇을 재사용할 것인가를 모르게 된다. 따라서 Ada 환경하에서의 개발은 적절한 문서의 준비가 무엇보다 중요하다고 제안하였다. [14]

접속관계를 설계함에 있어 설계자는 모듈이 기능면에서 고도의 응집도(high cohesion)를 갖도록 하고 모듈간의 접속은 낮은 결합도(low coupling)를 유지하도록 설계하여야 한다. 낮은 결합도를 갖기 위하여 인수에 의한 접속(parameterized)을 하도록 하여야 하고 인수에 의한 접속을 통하여 모듈 상호간의 상호 의존성을 최소화하고 모듈의 독립성을 유지할 수가 있다. 모델의 사용에 관한 설계지침에서 언급한 바와 같이 직접적인 요구사항과 장차 예상되는 요구사항을 만족하는 구성품을 설계하기 위하여 일반화 과정을 거치게되고 일반화 과정에서 인수는 과도하게 많이 정의될 수도 있다.

접속관계의 설계지침은 융통성과 일반성(flexibility and generality)을 갖도록 설계하는 것이다. 과도한 접속은 오히려 재사용 가능한 경우를 제한할 수도 있다. [5] 일반적으로 알려진 바로는 적은 수의 단순한 접속관계를 갖고 있는 구성단위가 재사용의 가능성이 더욱 높다는 것이다. [15]

적속관계는 의도적인 사용범위내에서 인수화 되면서(parameterized) 또한 목적에 알맞게 조정(customized)될 수 있도록 설계되어야 재사용이 가능하다. 예를들면 n차 다항식을 parameterized하여 설계하고 이를 4차 다항식으로 customized한다면 재사용성을 보장할 수 있다.

또한 접속관계는 모듈의 활용 범위를 제한하여 모든 경우의 기능을 하나의 모듈로 처리하지 않도록 설계하여야 한다. 예를들면, 삼각함수의 전부를 처리하는 모듈의 접속관계로 설계하였다면 너무나 복잡한 접속일뿐만 아니라 그 모듈 자체 코드의 복잡도도 대단할 것이다. 그러나 실제 사용시에는 부분적인것만 사용하게 됨으로 사용되지 않는 함수는 overhead로 남게된다. 따라서 재사용의 활용범위를 제한하고 그 범위내에서 접속관계를 설계함이 바람직하다.

4. 효율성의 선택

재사용을 위한 소프트웨어 설계는 과도한 일반화 과정의 작업으로 인하여 그 성능면에서 저조한 것으로 생각될 수가 있고, 또한 특별한 사용자의 관점에서 본다면 불 필요한 능력이 추가된 상태의 소프트웨어 구성품으로 느껴질 수도 있다.

소프트웨어 구성품을 재사용하려는 사용자의 관점에서 다음의 몇가지 사항은 효율성면에서 부정적인 요소가 되고 그 부정적인

요소에 대한 개선책이 필요하게 된다. [5] 먼저 부정적인 요소를 제시하면 다음과 같다.

- 융통성과 일반화를 위한 과도한 인수화
- 발생빈도가 희박한 경우의 처리를 위한 코드의 집합
- 모듈 코드의 변경없이 소프트웨어 구성품을 재사용하기 위한 부프로그램 호출 구조의 추가
- 여러 기능을 하나로 결합한 모듈

효율성을 개선하기 위한 방법을 Ada의 환경에서 제시하면 다음과 같다.

- 일반화 처리(generic)
- Inline Code
- Constant Folding과 Dead Code Elimination
- Subunit화

융통성과 일반화를 위하여 과도하게 인수화가 된 소프트웨어의 구성단위는 Ada의 generic에 의하여 Ada의 compiler로 하여금 불필요한 인수들을 상수(constant)로 처리케 하고 parameter passing의 overhead를 제거할 수도 있다. 발생빈도가 희소한 경우의 코드 집합은 generic에 의하여 경우를 결정하는 condition이 상수로만 제시되면(generic in parameter) Ada의 compile time에서 condition이 평가되고 평가 결과에 따라 그 경우에 해당하는 처리 기능의 코드는 제거될 수도 있다. constant folding과 dead code elimination은 앞에서 언급한 상수 처리와 condition의 평가에 따라서 불필요한 code를 제거하는 과정이며 효율성을 개선하는 방안이 된다.

Ada의 inline(pragma inline)기능은 Ada의 compiler로 하여금 call mechanism을 최적화 할때에 사용된다. 과도한 인수화의 경우에 인수가 상수일 때에는 상수값으로 call mechanism을 단순화 하게 한다. 또한 subprogram call의 과도한 호출 사용은 run time overhead가 되는데, subprogram의 모듈화를 설계와 코딩에서 유지하면서, run time새의 overhead를 최소화하기 위하여 parameter passing과 linkage 부분을 최적화하여 실행 코드로 작성되도록 한다. [2]

프로그래밍 언어의 관점에서 compiler와 linker는 프로그램을 구성하는 모든 참조 구성단위를 결합하는 작업을 하고 또한 불필요한 구성단위를 제외하는 작업도 한다. Ada 환경에서 subprogram을 보다 적은 규모의 subunit으로 설계하고, 작성한다면 linker로 하여금 프로그램의 실행에 필요한 subunit만을 선택하여 효율성을 제고할 수 있다.

요약하면, 소프트웨어의 구성단위를 재사용할 때에 효율성 면에서 부정적인 설계로 만들어진 구성단위가 있을 수 있으나 Ada의 환경에서 이를 개선하는 방법도 있다. 과도

한 인수화는 generic으로, 발생이 희박한 경우에 해당하는 코드의 집합은 constant folding과 dead code elimination으로, 부프로그램 호출 구조의 추가는 inline으로, 여러 기능을 수행하는 부프로그램의 비효율성은 sub unit화 처리로 개선이 가능함을 제시하였다.

Ⅳ. 결 언

소프트웨어의 재사용은 "소프트웨어의 위기"에 대비하는 잠재적인 대비책이라할 수 있다. 재사용성의 목표는 생산성의 향상과 품질의 개선에 있다. 소프트웨어의 재사용은 경제적, 기술적인 측면에서 바람직한 시도이면서, 동시에 기존의 전통적인 개발원칙의 대폭적인 수정과 적용을 요구하고 있다.

설계시에 재사용성을 고려하지 않고 작성된 소프트웨어 구성품은 재사용성을 보장받을 수가 없다. 알려진 최적의 설계원칙을 충실히 적용하였다 하더라도 재사용의 원칙을 고려치 않았다면 그 구성품을 현재 또는 미래에 다른 시스템에서 재사용 하리라고 기대할 수 없다.

본 고에서는 소프트웨어 재사용을 위한 설계지침으로

- 모델을 사용하여 현재와 미래에 사용자가 요구할 기능을 정의하고,
- 이들을 계층구조로 설계하며,
- 소프트웨어 구성품간의 접속관계를 명확히 명세화하고 이를 문서화하는 절차를 준비하도록 하고,
- 재사용의 설계에서 예상되는 몇가지 비효율적인 점들을 Ada의 compiler와 linker의 기능으로 처리 가능한 방안을 고찰하였다.

효율적인 재사용을 위하여 초기에 개발환경의 변화에 따른 부가적인 투자가 필요하며 재사용의 빈도에 의하여 생산성과 품질의 향상을 기대할 수 있다. 사업의 관리자와 전문가들의 적극적인 참여와 수용이 무엇보다도 중요하다.

앞으로 설계지침의 세부적인 처리에 관한 절차와 설계지침을 지원하는 방법론의 연구가 필요할 것이다.

참 고 문 헌

〔1〕 B. W. Boehm, "Software and its Impact : A Quantitative Assessment", Datamation, May. 1973, p. 13.

〔2〕 Ellis Horowits and John B. Munson, Senior member, IEEE, "An Expansive View of Reusable Software", IEEE Trans. on Software Engineering, Vol. SE-10, No. 5, September, 1984.

〔3〕 B. P. Lientz and E. B. Swanson, "Problems in Applications Software Maintenance", Communication of ACM, Vol. 24, No. 11, Nov., 1981.

〔4〕 Robert G. Lanergan and Charles A. Grasso, "Software Engineering with Reusable Design and Code", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, Sept., 1984, pp. 498-501.

〔5〕 Christine Ausnit, Christine Braun, Sterling Eanes, John Goodenough, and Richard Simpson, "Ada Reusability Guideline", ESD-TR-85-142, SoftTech, Inc. April, 1985.

〔6〕 William Wong, "A Management Overview of Software Reuse", NBS Special Publication 500-142, U. S. Department of Commerce, September, 1986.

〔7〕 Simcox, L. M. and Ball, R. G. "Reusable Software", Memorandum 3884, Royal Signals and Radar Establishment, 1985, pp. 4-1~4-14.

〔8〕 Patrick A. V Hall, "Software Components and Reuse - Getting More out of Your Code", Information and Software Technology, Vol. 29, No. 1, January/February 1987, pp. 38-43.

〔9〕 Special Issue on Software Reusability, "IEEE Transactions on Software Engineering", Vol. SE-10, No. 5. September, 1984.

〔10〕 Meilir Page-Jones, "The Practical Guide to Structured Systems Design", 2nd Ed., Prentice-Hall International, Inc. 1988. pp. 1-14.

〔11〕 Bruno Alabiso, "Transformation of Data Flow Analysis Models to Object Oriented Design", OOPSLA '88 Proceedings, pp. 335-353, September 25-30, 1988.

(12) Grady Booch, "Software Engineering with Ada", 2nd Ed. The Benjamin/Cummings Publishing Company, Inc. 1986. pp. 169-172.

(13) Jean D. Ichbiah, "On the Design of Ada", Information Processing 83, Edited by R. E. A., 1983. pp. 1-10.

(14) Geoffrey A. Pasco, "Elements of Object Oriented Programming", Tutorial: Object Oriented Computing, Vol. 1, Concepts, 1987. pp. 15-20.

(15) David N. Card, Victor E. Church, and William W. Agresti, "An Empirical Study of Software Design Practices", IEEE Transaction on Software Engineering, Vol. SE-12, No. 2, Feb., 1986.