# 프로그램 관리 관점에 기반을 둔 소프트웨어 아키텍처 생애주기 모델 : 확장된 나선형 모델

고석하*

# A Software Architecture Life Cycle Model Based on the Program Management Perspective : The Expanded Spiral Model

Seokha Koh*

## Abstract

The expanded spiral model in this paper consists of five processes of architecture design, architectural construction, architectural maintenance, operation, and architectural management. The former four processes are executed alternatively, while the latter architectural management process is executed continuously interacting with the other processes during the whole life cycle of the system. The expanded spiral model provides a conceptual framework to sort discussions of architectural degeneration into those of product-oriented processes and those of management processes, making it possible to incorporate the models and body of knowledge about project and program management, especially those of Project Management Institute, into discussions of architectural degeneration.

A good architecture decomposes the software-intensive system into components mutually interacting in a well-formed structure. The architecture design process and the architectural construction process together create the object system with well-designed architecture. The architectural maintenance process prevents the implemented architecture deviate from the designed architecture. The architectural management process monitors the changes of requirements including architecturally significant requirements, supports the other processes to be executed reflecting various perspectives of stake-holders, and creates and documents the reasons of architectural decisions, which is considered as a key element of the architecture.

Keywords : Software Architecture, Software Architecture Degeneration, Spiral Model, Program Management, Project Management Institute

논문접수일 : 2013년 03월 28일 논문게재확정일 : 2013년 06월 15일

※ 이 논문은 2011년도 충북대학교 학술연구지원사업의 연구비 지원에 의하여 연구되었음.

* 충북대학교 경영정보학과 교수, e-mail : shkoh@cbnu.ac.kr

## 1. 서 론

애자일 방법론에서는 가시적인 사용자 가치를 최대한 빨리 그리고 자주 인도하는 것을 목표로 삼으며, 이러한 과정에서 종종 아키텍처에 대한 고려가 결여된다[Brown et al., 2010]. 즉, 애자일 방법론은 아키텍처[1] 프로세스를 최소화하고, 개발 도중에 발현하는 필요(emergent needs)에 따라서 아키텍처가 '설계되도록' 하여야 한다고 주장한다[Abrahamsson et al., 2010; Brwon et al., 2010]. 반면에, "BDUF(Big Design Up-Front)" 소프트웨어 개발 초기에 현재만이 아니라 미래에 발현할 상호의존성까지 최대한 충족시킬 수 있는 아키텍처를 설계하여야 한다고 주장한다[Abrahamsson et al., 2010].

상호의존성은 도메인의 안정성이나 기술적 성숙도를 불문하고, 또한 시스템이 개발 초기에 있느냐 또는 이미 수년간 운영 중이냐를 불문하고 존재한다[Brown et al., 2010]. 즉, 모든 소프트웨어는 '일정한 구조하에서 서로 상호작용하는 컴포넌트들'로 구성되며, 이러한 의미에서 모든 소프트웨어는 아키텍처를 갖는다[Rozanski and Woods, 2012]. BDUF와 애자일(agile) 방법론을 구분하는 것은 서로 상호작용하는 컴포넌트들에 의도적으로 '정돈된 형태의 구조'를 생성하기 위한 절차, 즉 아키텍처적 절차(architectural process)의 유무이다[Brown et al., 2010].

BDUF에서 아키텍처를 설계하는 전형적인 방법은 프로젝트 초기(예를 들어서, UP(Unified Process) 또는 Open UP에서는 두 번째 단계인 일래보래이션 페이즈(elaboration phase)에 ASR(architecturally important requirements)을 식별하고; 아키텍처 패턴과 같은 일군의 지식을 선택하여 통합하고; 그것들을 사용하여 시스템 전체의 기능성을 하위 시스템들로 ASR을 충족시키는 방법으로 분할하고; 그 하위 시스템들을 동일한 방법으로, 아키텍트의 판단에 더 이상의 분할이 필요 없다고 판단되는 수준까지, 재귀적으로 분할하는 것이다[Bass et al., 2003; Bosch, 2000; Eclipse, 2012; Mattsson et al., 2009]. 모델-뷰-컨트롤러(Model-View-Controller), 칠판(Blackboard), 계층(Layers) 등과 같은 일반적인 아키텍처 패턴들에 의하면, 이것은 전형적으로 시스템이 다양한 종류의 클래스들(모델, 뷰, 그리고 컨트롤러와 같은)로 구성되는 하위시스템으로 분할되는 것을 의미한다[Buschmann, 1996; Mattsson et al., 2009].

미래에 발현할 아키텍처적 상호의존성(architectural interdependency)은, BDUF이나 애자일이냐를 불문하고, 예측에 의존할 수밖에는 없다. 일반적으로, 소프트웨어의 생애주기 전체에 발생할 상호의존성에 대한 예측은 매우 불확실하다. 따라서 애자일 방법을 선호하는 사람들은 이러한 상호의존성을 예측하고 대비하기 위해서 다량의 문서를 작성하고 시간을 소모할 가치가 없다고 주장한다.

그러나 애자일 방법론은 종종 상호의존성을 충분히 고려하지 못하게 하는 문제점을 발생시킨다[Brown et al., 2010]. 결과적으로 애자일 방법론에서는 종종 개발 후기로 갈수록 점증적으로 코드의 구조가 나빠지고 오류가 증가하며 시간과 자원이 많이 들고, 심지어는 모든 것을 폐기하고 처음부터 다시 재개발해야 하는 경우가 발생하기도 한다[Brown et al., 2010]. 실증조사들에 의하면, 많은 애자일 팀들이 실제로는 개발 초기에 아키텍처 프로세스를 상당한 정도로 수행한다[Blair and Watt, 2010]. 이것은, 소프트웨어 생애주기 전체를 걸쳐서, 견실한 아키

1) 이후로는 특별한 언급이 없는 한, '아키텍처'는 '소프트웨어' 또는 '소프트웨어 집약적 시스템'의 아키텍처를 의미한다. 또한 '소프트웨어'와 '시스템'을 모두 "소프트웨어 집약적 시스템"을 지칭하는 동일한 의미로, 혼용하여 사용하겠다.

텍처가 매우 중요하다는 것을 반증한다.

아키텍처 퇴화(architecture degeneration)는 '소프트웨어가 개발된 이후의 후기 생애주기 변경들을 통하여 구현된 아키텍처가 설계된 아키텍처와 달라지는 것'을 말한다[de Silva and Balasubramaniam, 2012; Hochstein and Lindvall, 2005]. 아키텍처가 퇴화되면 개선비용이 누적적으로 증가하며, 더 이상 시스템을 개선할 수 없어서 조기에 폐기해야만 할 필요성도 발생하게 된다[Banker et al., 1993; Hochstein and Lindvall, 2005]. 소프트웨어가 더 이상 사용자 요구사항을 잘 충족시키지 않게 되는 것은 일반적으로 아키텍처 퇴화로 간주되지 않는다.

아키텍처 퇴화에 관한 논의들은 건실한 아키텍처의 장기적 효과를 논증한다. 그러나, Abrahamsson et al.[2010]의 표현을 따르면, 아키텍처의 비용은 눈에 보이나 그 가치는 파악하기 힘들다. 그러나 아키텍처적 프로세스와 그 결과물인 아키텍처는 분명하게 구별되어야 한다. 그러한 의미에서, Abrahamsson et al.[2010]의 표현은 '아키텍처의 효과는 파악하기 어려우나, 아키텍처적 프로세스의 비용은 눈에 보인다'로 바뀌어야 한다. 본 논문에서는 잘 설계된 아키텍처 그 자체의 비용은 그 효과에 비해서 거의 무시할 수 있다고 전제한다.

본 논문에서는, 이러한 전제하에서, 소프트웨어 생애주기 전체에 걸친 아키텍처적 프로세스의 비용을 최소화할 수 있는 소프트웨어 아키텍처 생애주기 모델을 제시한다. 구체적으로는, Weinreich and Buchgeher[2011]의 소프트웨어 아키텍처 생애주기 모델에 PMI(Project Management Institute)의 프로그램 개념을 결합하고 개선함으로써 전체 생애주기 관점에서 개선된 아키텍처적 프로세스 모델을 제시한다.

PMI는 프로젝트의 작업 또는 프로세스를 관리 프로세스와 제품 지향적 프로세스로 분류한다. 본 논문에서는 기존의 아키텍처 퇴화에 관한 논의들을 관리 프로세스와 제품 프로세스에 해당하는 것으로 분리하고, 관리 프로세스에 해당하는 것들을 PMI의 프로젝트 및 프로그램 관리 모델의 개념들과 결합함으로써 PMI의 프로그램 및 프로젝트 관리 지식체계를 효과적으로 아키텍처 퇴화 문제에 적용할 수 있게 하는 것을 목적으로 한다.

## 2. 상호의존성과 프로그램 관리

PMI[2008b]는 아키텍처를 한 프로그램 내의 프로젝트들 간의 상호의존성을 정의하는 것으로서 다음과 같은 요소로 구성된다고 정의한다.

- 프로그램 또는 시스템의 컴포넌트들의 구조에 대한 정의.
- 컴포넌트들 간의 상호관계에 대한 확인.
- 그 상호관계와 프로그램 또는 시스템의 진화를 관장할 일군의 규칙들.

여기에서 컴포넌트는 프로그램의 구성 프로젝트 또는 그 프로젝트의 산출물을 의미한다.

PMI의 이러한 일반적인 아키텍처에 대한 정의는 소프트웨어 아키텍처에 대한 전형적인 정의에도 부합한다. 고석하[2012]에 의하면, 소프트웨어 아키텍처의 정의의 일반적인 핵심적 키워드는 컴포넌트(component), 상호작용(interaction), 그리고 정돈된 형태를 갖는 구조(well-formedness of structure)이며, 이에 대한 구체적인 의사결정들과 관련된, 다음과 같은 항목들이다.

- 이유(reason) : 고려되었던 대안들에 대한 분석과 의사결정들에 대한 구체적인 이유가 명시적으로 제시되고 기록되어야 한다.
  ◦ 아키텍처적 지식(architiectural knowledge) :

개발 조직은 정돈된 형태를 생성하기 위한, 아키텍처에 대한 기존에 잘 확립된 지식, 전략, 원칙, 지침, 규칙, 방법, 프로토콜, 패턴, 그리고 베스트 프랙티스 등으로 구성되는 지식 베이스를 유지해야 하며, 그 중에서 어떤 것이 해당 아키텍처의 개발에 적용되었는가가 명시적으로 기록되어야 한다.

- ◦ ASR(architecturally significant requirements) : 해당 프로젝트에 고유한, 아키텍처 개발에 적용된 구체적인 제약들이 명시적으로 기록되어야 한다. 지식은 실제의 프로젝트에서는 제약 없이 무조건적으로 그리고 무제한적으로 적용될 수는 없으며, 모든 의사결정이 궁극적으로는 해당 프로젝트의 요구사항 또는 이해관계자들의 필요에 기반을 두어야 한다.
- ◦ 논리(rationale) : 선택된 지식들이 구체적인 제약 하에서 어떻게 적용되었는지, 그리고 해결안이 왜 다른 대안들에 우선해서 선택되었는지에 대한 구체적인 설명이 명시적으로 제시되어야 한다.

• 다양한 관점(various perspectives) : 아키텍처에 반영되어야 할 이해관계자들의 필요는 일반적으로 서로 상충하며, 따라서 아키텍처에는 다양한 이해관계들의 관점이 반영되어야 한다. 이러한 다양한 관점들은, 특히, 생애주기 관점에서 정리되어야 한다.

소프트웨어는 전형적으로 프로젝트를 통해서 개발된다. 프로젝트는 '유일한 제품, 서비스 또는 결과를 만들어 내기 위해 한시적으로 수행되는 점진적인 노력'으로 정의할 수 있으며, 반복적으로 수행되는 운영과 구별된다[PMI, 2008a]. 프로그램은 '개별적으로 관리해서는 달성하기 힘든 이익 획득 및 비용 절감을 위해 통합하여 관리되는 일련의 프로젝트들'로 정의할 수 있다[PMI, 2008b]. 한 프로그램 내의 프로젝트들은 인도되는 공통의 결과나 집단적인 능력을 통하여 연계된다. 한편, 포트폴리오는 해당 작업의 효과적인 관리를 촉진하여 전략적 비즈니스 목표들을 달성하기 위해서 함께 묶은 컴포넌트들(예 : 프로젝트들, 프로그램들, 포트폴리오들, 그리고 유지보수 및 연관된 지속적 운영들)의 모음으로 정의할 수 있으며, 프로젝트들 간의 연관이 단지 공유되는 고객, 판매자, 기술, 또는 자원만이라면, 하나의 프로그램보다는 프로젝트들의 포트폴리오로 관리되어야 한다[PMI, 2008b]. 한 포트폴리오의 프로젝트나 프로그램들은 반드시 상호의존적이거나 직접적으로 연관되어 있을 필요가 없다.

소프트웨어가 프로젝트에 의해서 개발된 후에 또 다른 프로젝트에 의해서 보수된다면 이 두 프로젝트는 공통의 산출물을 통해서 서로 연계되며, 따라서 하나의 프로그램으로 묶어서 관리할 수 있다. 소프트웨어 아키텍처 퇴화(architecture degeneration)와 관련된 논의들은 이러한 관점을 잘 반영한다.

## 3. 아키텍처 퇴화

모든 소프트웨어는 노후화된다. 소프트웨어 노후화(software aging)는 '시간 경과에 따른 소프트웨어 가치의 저하'로 정의할 수 있으며, 그 원인은 크게 '시간의 경과에도 불구하고 고치지 않는 것'과 '잘 못 고치는 것'의 두 가지로 나눌 수 있다[Parnas, 1994]. '고치지 않는 것'으로 인한 소프트웨어 노후화를 방지하기 위해서, 즉 기술 발전 또는 업무환경의 변화로 인한 새로운 요구사항에 맞도록 최소한 한 번의 개발 주기가 완료되고 시스템의 운영되기 시작한 후에 소프트웨어를 변경하는 것을 후기 생애주기 변경(late-lifecycle changes)이라고 한다[Williams and

Carver, 2010].

소프트웨어 진화(software evolution)는 후기 생애주기 변경으로 인한 소프트웨어의 변화(경우에 따라서는 아키텍처의 심대한 변경을 수반한) 과정으로 정의할 수 있다[Ozkaya et al., 2010]. 소프트웨어 진화는 오랫동안 연구되어온 주제이며, 다음과 같은 항목들을 포함하는 소프트웨어 진화 법칙(Laws of Software Evolution)이 개발되어 있다[Capiluppi et al., 2007; Lehman, 1980, 1996a, 1996b; Lehman and Belady, 1985; Lehman and Ramil, 2000; Sadou et al., 2005; Williams and Carver, 2010].

- 지속적인 변경 : 소프트웨어는 현재의 능력과 환경적 요구사항의 불일치를 해소하기 위한 유지보수와 개발로 인하여 끊임없이 변한다.
- 증가하는 복잡도 : 이러한 변경은 컴포넌트 간의 상호작용과 의존도를 증가시킨다.
- 지속적인 성장 : 사용자의 만족도를 유지하기 위해서 기능이 전체 생애주기를 통해서 지속적으로 증가한다.
- 낮아지는 품질 : 시스템의 지각된 품질이 낮아지며, 유지보수 비용도 또한 점점 증가한다.

이렇게 변경이 통제되지 않아서 시스템의 품질이 저하하고, 급기야는 더 이상 유지보수가 불가능해지기까지 하는 현상을 아키텍처 퇴화(architectural degeneration/erosion/drift/decay) [de Silva and Balasubramaniam, 2012, Hochstein, and Lindvall, 2005, Lindvall et al., 2002, Perry and Wolf, 1992], 또는 코드 부식(code decay) [Eick et al., 2001], 설계 부식(design decay) [Izurieta and Bieman, 2007], 설계 침식(design erosion) [van Gurp and Bosch, 2002], 소프트웨어 노후화(software aging) [Parnas, 1994], 소프트웨어 침식(software erosion) [Dalgarno, 2009] 등으로 부른다. 그러나 이런 다양한 명칭에도 불구하고 모든 유형의 소프트웨어 노후화는 아키텍처와 직간접적으로 관련이 있으며, 관련된 논의들을 아키텍처 퇴화의 관점에서 통합할 수 있다[de Silva and Balasubramaniam, 2012].

de Silva and Balasubramaniam[2012]는 아키텍처 퇴화를 진화 과정에서의 공학적 품질의[2] 전반적인 악화로 간주한다. 소프트웨어의 공학적 품질의 가치를 결정하는 가장 중요한 요인은 아키텍처적 상호의존성(architectural dependency)이며, 변화하는 환경과 요구사항에 맞추어서 경제적이고 신속하게 소프트웨어를 유지보수하기 위해서는 높은 공학적 품질의 아키텍처를 개발하고 퇴화를 예방하는 것이 매우 중요하다[de Silva and Balasubramaniam, 2012]. 아키텍처의 품질은 신중한 개선에 의해서 향상될 수 있다[de Silva and Balasubramaniam, 2012].

퇴화된 아키텍처를 개선하기 위해서 현재 주로 사용되는 방법으로는 다음과 같은 것들이 있다[de Silva and Balasubramaniam, 2012].

- 최소화(minimise) : 시스템의 개발과 유지보수 활동 동안에 아키텍처 준수를 보장하는 프로세스를 채택하거나, 구현과 병행하여 아키텍처 명세의 진화를 관리할 방법을 사용하거나, 아키텍처 모델을 구현으로 전환하는 방법이나 도구를 이용함으로써 아키텍처 퇴화가 최소한으로 발생하도록 한다.
- 방지(prevent) : 소스코드 내에 아키텍처 모델을 삽입하고 실행시간에 아키텍처 준수를 모니터할 수 있는 메커니즘을 이용하거나, 실행상태에서 구현이나 실행상태에 변경이 이루어진 후에 시스템이 아키텍처에 맞추어 스

2) 아키텍처의 품질과 공학적 품질에 관해서는 부록 A를 참조할 것.

스로 재구성할 수 있는 기술을 사용함으로써 아키텍처가 퇴화하는 것을 방지한다.

- 수리(repair) : 일단 아키텍처가 퇴화되면 소스 코드와 기타 인조물들로부터 구현된 아키텍처를 추출하거나, 아키텍처 문서가 없을 때 시스템의 발현적 속성들로부터 설계된 아키텍처를 밝혀내는 기술을 사용하거나, 설계된 아키텍처와 구현된 아키텍처간의 격차를 줄이는 데 도움이 되는 방법들을 사용하여 수리함으로서 아키텍처를 복구(restore)할 수 있다.

ASR을 잘 관리하는 것은 아키텍처 퇴화가 없는 소프트웨어 진화를 위한 핵심 요소이다 [Miksovic and Zimmermann 2011; Mirakhorli 2011; Ozkaya et al., 2010]. Ozkaya et al.[2010]은 아키텍처 퇴화를 방지 위해서는 다음과 같은 프로세스를 통해 진화 계획을 작성하고, 그에 따라서 진화를 관리해야 한다고 주장한다.

단계 1 : 진화 요구사항, 즉 아키텍처 진화의 지침을 제공할 ASR를 밝히고 정리한다.
단계 2 : 필요한 과업들과, 그것들 간의 상호의존성을 확인하고, 과업의 수행 순서를 정한다.
단계 3 : 진화 계획에 대한 대안들을 작성한다.
단계 4 : 대안들 중에서, 비용-효익 분석을 통해서, 최적의 계획을 선택한다.

그들은 이러한 프로세스가 다음과 같은 특징을 지닌 폐쇄적 진화(closed evolution)의 경우에 특히 효과적이라고 주장한다.

- 기술의 갱신 또는 변경을 요구하는 비즈니스 필요 그리고/또는 제약에 의해서 진화가 촉발된다.
- 진화의 목표 시스템에 대한 주요 비즈니스 구동 요소들과 요구사항들이 알려져 있다.
- 작성할 목표 시스템의 아키텍처가 알려져 있거나, 또는 구축 중에 있다.

모든 아키텍처는 '일정한 구조 하에서 서로 상호작용하는 컴포넌트들'로 구성된다[Rozanski and Woods, 2012]. 즉, 퇴화된 아키텍처에서도 컴포넌트들과 상호작용은 있으나, 아키텍처 퇴화의 정의상, 정돈된 형태의 구조는 파괴되어 있다. 아키텍처가 퇴화되는 과정에서는 일반적으로 변화된 제약과 관점들이 적절히 다루어지지 않으며(그래서 아키텍처가 퇴화한다), 따라서 더 이상 기존의 지식과 논리도 또한 더 이상 유효하지 않게 된다. 즉, 퇴화된 실제 아키텍처는 더 이상 이유와 다양한 관점을 포함하지 않는다(<표 1> 참조).

애자일 방법론과 같이, 잘 정제된 아키텍처 설계 과정이 없는 프로세스에 의해서 생성된 아키텍처에는 잘 정돈된 구조가 존재할 가능성과

〈표 1〉 아키텍처에 포함된 구성요소의 차이 : 잘 통제된 아키텍처 프로세스에 의해서 설계된 아키텍처, 발현적으로 생성된 아키텍처, 그리고 퇴화된 실제 아키텍처

| 아키텍처의 구성요소 | 잘 설계된 아키텍처 | 발현적으로 생성된 아키텍처 | 퇴화된 구현된 아키텍처 |
|---|---|---|---|
| 컴포넌트 | 포함 | 포함 | 포함 |
| 상호작용 | 포함 | 포함 | 포함 |
| 정돈된 형태 | 포함 | 아마도 | 불포함 |
| 이유 : 기록된 | 포함 | 불포함 | 불포함 |
| 전체 생애주기적 관점 | 포함 | 아마도 | 불포함 |

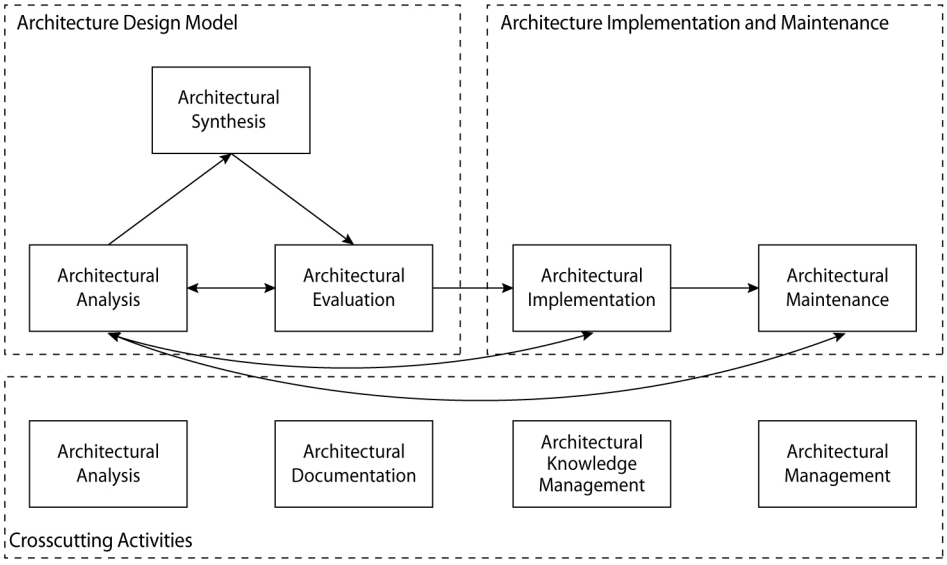
아키텍트(architect) 및 시스템 엔지니어와 같은 전기 생애주기 이해관계자들 외에 병행 운영자 및 관리자와 같은 후기 생애주기의 이해관계자들의 관점이 반영될 가능성이 모두 필연적으로 낮아진다. 또한, 애자일 방법론은 아키텍처 활동에 포함된 방대한 문서의 생성을 제거하는 것을 그 목적중의 하나로 삼으므로, 이유에 대한 명시적인 기록이 결여될 가능성이 높아진다.

## 4. 소프트웨어 아키텍처 생애주기 모델

Hofmeister et al.[2007]은 아키텍처적 분석, 아키텍처적 종합, 그리고 아키텍처적 평가의 세 가지 주요 활동으로 구성된 아키텍처 설계 모델을 발표하였다. Tang et al.[2009]은 Hofmeister et al.[2007]의 아키텍처 설계 모델에 아키텍처 구현과 유지보수가 추가된, 크게 두 단계로 구성된 아키텍처 생애주기 모델을 발표하였다. Weinreich and Buchgeher[2012]은 Tang et al.[2009]의 모델에 Christensen et al.[2009] 모델의 일부 활동들이 추가된 소프트웨어 아키텍처 생애주기 모델을 개발하였다(<그림 1> 참조). 각 활동들에 대한 Weinreich and Buchgeher[2012]의 정의를 소개하면 다음과 같다.

- 아키텍처 설계 모델(architecture design model) : 다음 단계로 투입될 확인된 아키텍처(validated architecture)를 산출한다.
  - 아키텍처적 분석(architectural analysis) : 해결되어야 할 문제, 즉 ASR을 식별한다.
  - 아키텍처적 종합(architectural synthesis) : 문제에 대한 잠정적인 해결안을 개발한다.
  - 아키텍처적 평가(architectural evaluation) : 개발된 해결안을 ASR에 대비하여 검증한다.

- 아키텍처 구현 및 유지보수(architecture implementation and maintenance) : 소프트웨어 생애주기 전체에 걸쳐서 실현된 아키텍처가 설계된 아키텍처에 부합하도록 아키텍처와 관련된 활동들을 지원한다.




자료원 : Weinreich and Buchgeher[2012].

〈그림 1〉 교차 활동이 있는 소프트웨어 아키텍처 생애주기 모델

◦ 아키텍처적 구현(architectural implementation) : 주어진 아키텍처를 기반으로 소프트웨어를 구현하는 것을 지원한다.
◦ 아키텍처적 유지보수(architectural maintenance) : 아키텍처 문서에 근거하여 변화하는 요구사항과 시스템 진화의 관점에서 아키텍처적 변경의 영향을 평가하는 것을 지원한다.

• 교차활동(crosscutting activities) : 소프트웨어 생애주기 전체를 통해서 상호작용하는 방식으로 수행된다.
◦ 아키텍처 분석(architecture analysis) : 이 활동의 목표에는 상충하는 ASR의 탐지, 아키텍처 설계 의사결정의 영향에 대한 분석, 아키텍처 모델들 간의 불일치 및 아키텍처와 구현간의 불일치의 탐지, 그리고 아키텍처 해결안들의 정의된 요구사항들에 대비한 평가 등이 포함된다.
◦ 아키텍처 문서화(architecture documentation) : 아키텍처를 명시적으로 표현하고 문서화한다. 아키텍처는 정형적 또는 비정형적으로 표현될 수 있다. 아키텍처 문서는 전형적으로 다양한 이해관계자의 관심을 반영하기 위해 아키텍처에 대한 다양한 관점을 제공한다.
◦ 아키텍처 지식 관리(architecture knowledge management) : 특히 아키텍처 설계와 구현 및 유지보수 단계들에서의 아키텍처 평가와 영향 분석을 지원하기 위해서, 설계 의사결정과 논리와 같은 아키텍처적 지식을 포착하고 관리하며, 이러한 정보들을 아키텍처적 구조와 연결한다.
◦ 아키텍처 관리(architecture management) : 아키텍처와 아키텍처 인조물들을 생성하고 관리한다. 여기에는 아키텍처의 기술적 표현, 그것의 리포지토리(repositories)와 도구 내에서의 관리, 아키텍처에 대한 서술과 아키텍처와 관련된 정보들의 배치하고 공유하기가 포함된다.

Weinreich and Buchgeher[2012] 모델의 장점은 기존의 아키텍처 생애주기 모델들을 통합하며, 관련된 다양한 활동들을 포괄한다는 것이다. 그러나 바로 그러한 점이 또한 이 모델의 가장 큰 한계이기도 하다. 즉, 이 모델에서는 Tang et al.[2009] 모델에 있던 아키텍처 설계 모델의 활동들과 Christensen et al.[2009] 모델에 있던 아키텍처 분석이 잘 구별되고 차별화가 되지 않는다. 이러한 문제점은, 아키텍처 설계 모델에 포함되어 있는 아키텍처적 분석이 Tang et al.[2009] 모델에 해당하는 모든 활동들과 상호작용한다는 것, 즉 이 활동이 실질적으로는 교차활동이라는 것으로도 재확인되기도 한다. 뿐만 아니라, 아키텍처 관리는, 적어도 Weinreich and Buchgeher[2012] 자신들의 표현상으로는, 다른 활동들 거의 모두, 특히 아키텍처 설계, 구현 및 유지보수의 활동들을 포함한다. 아키텍처 생애주기 활동들은 모두 결국은 아키텍처와 아키텍처와 관련된 인조물들을 생성하고 관리하는 활동이다.

또한, Weinreich and Buchgeher[2012] 모델에서는, 아키텍처적 유지보수는 아키텍처적 구현 다음에 아키텍처의 재설계 없이 수행될 수도 있지만, 아키텍처의 재설계를 수반할 수도 있다. 그러나 아키텍처의 재설계를 포함한다면, 실질적으로 아키텍처적 유지보수는 아키텍처적 구현과 동일하게 된다. 이러한 문제는 근본적으로 각 활동간의 순서가, 아키텍처적 분석을 중심으로, 너무 복잡하게 설정되어 있다는 것에서 기인한다. 즉, Weinreich and Buchgeher[2012] 모델에서는 아키텍처적 분석을 중심으로 그 전후

에 거의 모든 활동들이 임의로 수행될 수 있으며, 이러한 점에서 특정한 생애주기 모델이라고 할 수 없다.

## 5. 토 론

PMI 프로젝트 및 프로그램 관리 모델의 특징은 범위 및 아키텍처와 관련된 요구사항 활동들을 관리 활동으로 간주한다는 것이다[PMI 2008a, 2008b]. 반면에, 소프트웨어 생애주기 모델과 방법론들은 전형적으로 요구사항 활동들을 제품 프로세스로 간주한다[Boehm, 1988, 1996, 1998; DeGrace and Stahl, 1990; McConnell, 1996; Larmen, 2005; Royce, 1970; Stapleton, 1997]. 그러나, Young[2001]에 의하면, 요구사항과 관련된 활동들은 '요구사항을 도출하기'를 포함해서 17개 범주로 세분할 수 있다. 한편, 아키텍처는 소프트웨어 프로젝트의 성패를 가름 지을 수 있는 주요 위험 요소들과 연관되어 있는 가장 핵심적인 요소 중의 하나이다[Boehm, 1988, 1996, 1998]. 따라서, '요구사항의 우선순위를 정하기'를 포함하는, 범위 및 아키텍처와 관련된 요구사항과 관련된 활동은 관리 프로세스로 분류하고, 도출된 요구사항(derived requirements)를 포함하는 기타 요구사항과 관련된 활동들은 제품 프로세스로 분류할 수 있다. 이러한 관점은 PMI의 프로젝트 및 프로그램 관리의 전통하에서 축적되어온 위험 관리를 비롯한 풍부한 지식과 경험을 소프트웨어 아키텍처 퇴화에 관한 논의에 자연스럽게 통합할 수 있게 해준다.

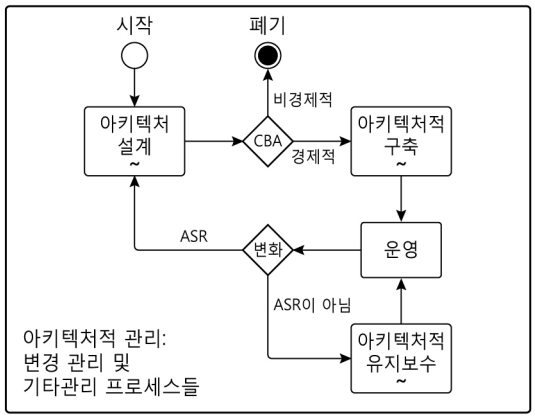
한편, PMI 프로그램 관리 모델은 주로 컴포넌트 프로젝트가 병렬적으로 동시에 수행되는 경우를 상정하고 있다. 즉, PMI 프로그램 관리 모델은 아키텍처 관리의 주안점을 전체 프로그램 범위를 병렬적으로 수행되는 컴포넌트 프로젝트에 할당하고, 컴포넌트 프로젝트간의 상호작용을 관리하는 데에 두고 있다. 따라서 PMI 프로그램 관리 모델은 전체 범위를 하나의 프로젝트에 할당하고 시간이 경과함에 따른 범위 및 기술 및 기타 환경 변화와 관련된 아키텍처 퇴화와 관련된 논의에 그대로 적용하기에는 한계가 있다. 이러한 한계 내에서 Weinreich and Buchgeher[2012] 모델의 각 활동들을 PMI 모델의 각 요소에 대응시키면 <표 2>와 같이 된다.

일군의 소프트웨어를 개발하고, 그 소프트웨어들을 폐기할 때까지 변경하는 전체 과정을 하나의 프로그램으로 간주하여 관리할 수 있다. 이런 경우, 소프트웨어의 후기 생애주기에서의 변경은 수반되는 아키텍처의 변경의 정도를 기준으로 다음과 같이 크게 두 가지로 구분할 수 있다.

〈표 2〉 Weinreich and Buchgeher[2012] 모델과 PMI 모델간의 대응관계

| Weinreich and Buchgeher | PMI |
|---|---|
| 아키텍처적 분석 | 범위 및 요구사항과 관련된 프로세스들 |
| 아키텍처적 종합 | 아키텍처를 개발하기 프로세스 |
| 아키텍처적 평가 | 범위 및 요구사항과 관련된 프로세스들 |
| 아키텍처적 구현 | 소프트웨어 개발/유지보수 프로젝트 |
| 아키텍처적 유지보수 | 아키텍처를 관리하기 프로세스 |
| 아키텍처 분석 | 범위 및 요구사항과 관련된 프로세스들 |
| 아키텍처 문서화 | 의사소통 관리 지식 영역 |
| 아키텍처 지식 관리 | 조직 프로세스 자산 |
| 아키텍처 관리 | 변경 관리, 의사소통 지식 영역 |

- 아키텍처적 유지보수(architectural maintenance) : 기존의 아키텍처를 유지하면서, 예를 들어서 아키텍처 퇴화 최소화 및 방지 기법들을 사용하여, 기능을 개선한다.
- 아키텍처적 재구축(architectural reconstruction) : 아키텍처를 재설계하고, 새로운 아키텍처에 맞추어서 시스템 전체를 변경한다. 퇴화된 아키텍처를 복구하는 즉, 실제 아키텍처가 설계된 아키텍처와 다를 때, 실제 아키텍처를 설계된 아키텍처에 맞추어서 시스템을 수리하는 경우를 포함한다.




〈그림 2〉 확장된 나선형 모델

여기에서는, Weinreich and Buchgeher[2012] 모델과는 달리, 아키텍처적 유지보수는 아키텍처의 재설계를 수반하지 않으며, 아키텍처의 재설계를 수반하는 변경은 아키텍처적 재구축으로 분류된다. 아키텍처적 재구축은 독립적인 프로젝트로 관리하고, 아키텍처적 유지보수는 독립적인 프로젝트로 관리하지 않는 것을 원칙으로 한다. 그러나, 실무적으로는, 비교적 규모가 큰 유지보수 작업은 프로젝트로 관리하고, 별도의 프로젝트로 관리할 필요가 없는 정도의 경미한 아키텍처의 변경을 수반하는 변경은 아키텍처적 유지보수로 분류하고 독립적인 프로젝트로 수행하지 않을 수도 있다.

<표 3>과 <그림 2>는, 이러한 구분에 기반을 둔, 시스템이 개발되어 폐기되기까지 아키텍처의 퇴화를 효과적으로 통제하기 위한, 규범적(normative)인 소프트웨어 생애주기 모델의 개요를 보여준다. <그림 2>에서 아키텍처적 구축은 전기 생애주기에서의 구축과 후기 생애주기에서의 재구축을 포괄한다. PMI에 의하면, 모든 프로젝트와 프로그램 활동들은 관리 프로세스와 제품 프로세스로 분류할 수 있다. 유지보수 또는 (재)구축이 완료되면, 시스템은 운영상태로 들어간다. 운영상태에서, 관리 활동은 프로그램 수준에서 수행되며, 요구사항의 변화를 모니터링하고, 요구사항의 변화가 탐지되면 그것이 ASR과 관련 있는가를 확인하는 것을 포함한다. 만약, ASR이 변하지 않았다면 아키텍처적 유지보수를 수행한다.

〈표 3〉 확장된 나선형 모델의 각 프로세스의 주요 활동

| | 관리 프로세스 | 제품 프로세스 |
|---|---|---|
| 운영 | • 요구사항의 변화를 모니터링 한다. | • 현재의 시스템을 이용한다. |
| 아키텍처적 유지보수 | • 요구사항(의 변화)을 분석하고, ASR의 변화를 식별한다. | • 요구사항을 정제한다.<br>• 상세 설계, 코딩 등의 작업을 수행한다. |
| 아키텍처적 구축 | • ASR의 변화를 분석하고 정제한다.<br>• 변경된 ASR에 맞추어 아키텍처 대안들을 생성하고 평가한다.<br>• 아키텍처의 프로토타입을 작성하고 주요 위험을 해소한다.<br>• 아키텍처 프로토타입에 맞는 생애주기 모델과 개발 방법론을 선택한다. | • 아키텍처를 정제한다.<br>• 요구사항을 정제한다.<br>• 상세 설계, 코딩 등의 작업을 수행한다. |

이때, 경제적인 이유 등으로 유지보수를 일정 기간 보류할 수도 있으나, 그림에서는 이러한 것은 표현하지 않았다. 유지보수를 위한 제품 프로세스에는 요구사항을 정제하기 위한 추가적인 요구사항 활동과 상세 설계 및 코딩 등이 포함된다.

만약, ASR이 변했다면 프로그램 관리팀은 변한 ASR을 정제하고, 정제된 ASR에 맞추어 새로운 아키텍처의 프로토타입을 작성한 후, 재구축에 대한 비용/효익 분석(CBA)을 수행한다. 만약, 비용/효익 분석에서 재구축을 하는 것이 경제적이지 않다고 판정이 나면, 재구축을 보류하거나 적당한 시점에 시스템을 폐기한다. 재구축을 하는 것이 바람직하다고 판정이 나면, 재구축 프로젝트를 착수하고 관리의 책임을 프로그램과 프로젝트 수준으로 적절히 분할한다. 재구축 프로젝트의 제품 프로세스에는 선택된 아키텍처 프로토타입을 정제하는 것과 구현을 위한 기타 작업들이 포함된다.

아키텍처의 퇴화가 적절히 통제되지 못해서 아키텍처가 이미 퇴화되어 있는 경우에는, 비용/효익 분석을 통해서 적절한 시점에 아키텍처를 설계하고 아키텍처적 재구축 프로젝트를 수행함으로써 본 생애주기 모델로 진입할 수 있다.

애자일 방법론을 이용한 개발 프로세스에서는 '설계된' 아키텍처는 물론 실제 아키텍처에 대한 문서도 없을 수도 있다. 이런 경우에, 실현된 시스템으로부터 실제 아키텍처에 대한 문서를 작성하는 것은 프로그램 수준의 관리 활동에 포함된다. 산출물에 대한 문서를 관리하고 정보를 유통하는 것은 관리부문의 기본적인 책임이다. 만약, 파악된 실제 아키텍처가 현재의 ASR을 충족하지 못한다면 아키텍처적 재구축을 수행할 수 있다.

아키텍처 설계는 Weinreich and Buchgeher [2012] 모델의 아키텍처적 분석, 종합, 평가 그리고 아키텍처 분석을 포괄한다. 아키텍처 설계에서는 아키텍처적 제약과 아키텍처적 지식을 다양한 관점에서 논리적으로 연결하고 아키텍처 대안들을 작성하고 평가하며, 최종안을 선택하기 위한 프로토타이핑을 수행한다. 아키텍처 대안들은 전체 시스템을 컴포넌트와 컴포넌트들 간의 상호작용을 잘 정돈된 형태로 분리하고 표현하여야 한다.

아키텍처 설계에서 어떠한 활동들이 어떠한 순서에 따라서 수행될 것인가는 그때그때의 상황에 따른다. 즉, OMG(Object Management Group) [2010] BPMN의 용어를 이용하면, 본 모델에서 아키텍처 설계는 애드혹(ad-hoc) 프로세스로[3] 정의된다.

아키텍처적 구축에서는 아키텍처의 프로토타입을 정제하고 상세 설계를 하며, 최종적으로 소프트웨어 시스템을 구현한다. 아키텍처적 구축과 아키텍처적 유지보수도 애드혹 프로세스이며 작업의 구체적인 규모와 성격에 맞추어서 수행한다. 특히, 아키텍처적 재구축에서 구체적인 상황에 맞추어 적용할 방법론과 프로젝트 생애주기 모델을 선택하는 것은 관리부문의 책임이다. 개발자들은 정의된 아키텍처를 이해하고 준수하기 위해서 관리부문의 지원을 받는다.

아키텍처 설계 이후의 제품 프로세스에 대해서 미리 제약을 가하지 않는다는 점에서, 본 모델은 Boehm[1988, 1996; Boehm et al., 1998]의 나선형 모델[Spiral Model]과 동일하다. 실제로, 본 모델은 Boehm의 나선형 모델의 아키텍처 퇴화의 관점에서의, 전기 생애주기에만 국한된 것이 아니라, 소프트웨어 전체 생애주기에 걸친 확장이라고 볼 수 있다. 각 아키텍처적 구축 프로젝트에서 아키텍처 설계까지의 과정을 Boehm의 나선형 모델에 따르는 것은, 본 모델에서 적극적으로 권장된다.

관리 프로세스들도 애드혹 프로세스이다. 요구사항의 변화는 변경 관리의 일환으로 지속적

3) <그림 2>에서 기호 '~'는 해당 프로세스가 애드혹 프로세스, 즉 포함되는 활동들이 임의적인 순서로, 그때그때의 상황에 맞추어서 수행된다는 것을 나타낸다.

으로 모니터링 된다. 이러한 요구사항의 변화는 기술적 환경의 변화로부터 기인할 수도 있지만, 최근에는 비즈니스 환경 및 프로세스의 변화로부터 기인하는 경우가 많다[이현우 등, 2009]. 특히 비즈니스의 변화는, 기술부문이 적시에 모니터하기 어려우며, 관리부문이 특히 전적인 책임을 지고 모니터하여야 한다. 요구사항의 변화가 감지되면, 그 내용에 따라서 필요한 활동들을 수행한다. 이러한 관리 활동들을 주요 내용을 아키텍처를 중심으로 정리하면 다음과 같다.

- 아키텍처적 요구사항 및 제약 분석(architecturally significant requirement and constraint analysis) : ASRs을 비롯하여 아키텍처에 영향을 미치는 제약들을 식별하고, 이러한 제약들 간의 상충관계를 비롯한 관계들과 완전성에 대해서 분석하고, 현재의 설계된 아키텍처 또는 실제 아키텍처가 이들 제약들에 부합하는지를 평가한다.
- 아키텍처적 지식 관리(architectural knowledge management) : 해당 아키텍처 설계에, 최종적으로 적용된 지식을 포함하여, 이용될 가능성이 있는 모든 지식을 조직 프로세스 자산을 중심으로 탐색하고 식별한다.
- 아키텍처적 활동 관리(architectural activity management) : ASR의 변화를 포함하는 각종 변경을 관리하고, 아키텍처 문서를 작성하고 형상을 관리(configuration management)하며, 정보 및 의사소통을 관리하고, 기타 관리 및 지원 활동을 수행한다.

<그림 2>에서 관리 프로세스가 다른 프로세스들을 감싸도록 표현한 것은 관리 프로세스가 환경의 변화에 대한 완충작용을 하여 다른 프로세스들이 안정적이고 효율적으로 수행될 수 있도록 하여야 한다는 것을 강조하기 위해서이다. 그러나 현실적으로는 전체 생애주기에 걸친 관리 프로세스를 지속적이고 안정적으로 책임 질 조직이 없는 경우가 대부분이라는 것이 문제이다. 즉, 일반적으로는 시스템을 이용하는 조직과 구축이나 유지보수를 하는 조직이 다르며, 뿐만 아니라 각각의 구축과 유지보수에 대한 주된 책임을 지는 조직도 다를 수 있다. 따라서 전체 생애주기를 통해서 가장 큰 이해관계가 있는 조직의 관점에서 어떻게 이러한 전체 프로세스를 일관성 있게 정렬할 것인가에 대한 집중적인 연구가 필요하다. <표 4>는 확장된 나선형 모델과 기존 연구들 간의 차이점을 요약하여 보여준다.

〈표 4〉 확장된 나선형 모델의 기존 연구들과의 차이점

| 기존 연구 | 기존 연구의 한계 | 확장된 나선형 모델의 특징 |
|---|---|---|
| 소프트웨어 아키텍처 생애주기 모델 : Weinreich and Buchgeher | • 구성 활동들이 뚜렷이 차별화되지 않는다.<br>• 활동들 간의 순서가 너무 임의적이다.<br>• 본질적으로 전체 프로세스가 애드혹 프로세스에 가깝다.<br>• 각 활동들의 주관 조직에 대한 고려를 모델 안에 통합하기 어렵다.<br>• 아키텍처에 관한 논의들을 전체 프로젝트 또는 프로그램에 대한 논의와 결합하기 어렵다. | • 구성 활동들이 뚜렷이 차별화된다.<br>• 활동들 간의 순서가 명확히 정의되어 있다.<br>• 각 활동들의 주관 조직에 대한 고려를 모델 안에 통합하기 어렵다.<br>• 아키텍처에 관한 논의들을 전체 프로젝트 또는 프로그램에 대한 논의와 결합하기 쉽다. |
| 나선형 모델 : Boehm | • 개발 프로젝트에 국한된다.<br>• 관리 프로세스와의 연계에 대한 고려가 결여되어 있다. | • 전체 생애주기로 확장한다.<br>• 관리 프로세스와 쉽게 연계될 수 있다. |
| 프로젝트 및 프로그램 관리 모델 : PMI | • 일반적이다. | • 소프트웨어에 특화되어 있다. |

## 6. 결 론

아키텍처는, 고석하[2012]에 의하면, 시스템을 서로 상호작용하는 컴포넌트들로 분할하며, 이러한 컴포넌트들을 잘 정돈된 형태로 조직화하여야 하고, 관련된 의사결정들과 그 이유에 대한 다양한 관점에서의 서술을 포함해야 한다. 의사결정의 이유는 크게 아키텍처적 지식, 아키텍처적 제약, 그리고 추론 논리를 포함한다. 많은 논문에서, 그리고 본 논문에서도, 아키텍처적으로 중요한 요구사항(ASR)을 아키텍처적 제약과 동일한 의미로 사용한다.

본 논문의 생애주기 모델은 아키텍처 설계, 아키텍처적 구축, 아키텍처적 유지보수, 운영, 그리고 아키텍처적 관리의 다섯 개의 프로세스로 구성되어 있다. 이 중에서 앞의 네 프로세스는 일정한 논리적 순서에 따라서 수행된다. 관리 프로세스는 시스템의 전체 생애주기를 통해서 수행되며 나머지 프로세스들을 지원한다. 관리 프로세스도 함께 수행되는 기타 프로세스에 따라서 그 주된 내용이 변한다. 그러나 이 다섯 프로세스 그 자체는 모두, 포함된 활동들이 따라야하는 고정된 순서가 미리 정해지지 않은 애드혹 프로세스이다.

아키텍처 설계 프로세스는 아키텍처의 기본적인 구성 요소들인 컴포넌트, 상호작용, 그리고 잘 정돈된 형태를 결정하며. 아키텍처적 관리 프로세스는 이유와 다양한 관점을 생성하고 기록함으로써 아키텍처를 완성한다. 즉, 아키텍처적 관리 프로세스는 ASR를 식별하고 아키텍처 설계에 대한 지식 베이스를 제공하며, 아키텍처와 관련된 의사결정에 다양한 이해관계자들의 관점이 반영될 수 있도록 이해관계자 관리와 의사소통 관리를 수행하며, 이러한 과정이 질서정연하게 수행될 수 있도록 변경 관리를 수행한다. 아키텍처 설계와 아키텍처적 관리 프로세스는 전체 관리 프로세스의 일부이며, 서로 매우 밀접하게 상호작용한다. 저자는 이 두 프로세스에서 어떤 활동을 어떠한 순서에 의해서 수행할 것인가에 대해서는 구체적인 제약을 설정하지는 않지만, Boehm[1988, 1996; Boehm et al., 1998]의 나선형 모델에 따라서 수행할 것을 강력히 추천한다. 본 모델의 이름인 “확장된 나선형 모델”은 이러한 관점을 반영한다.

본 논문은 소프트웨어 개발과 유지보수에 관한 논의에 PMI의 프로젝트와 프로그램 관리에 대한 지식 체계를 도입함으로써, 바퀴를 재설계하는 낭비를 제거하였다. 그러나 일반적으로 하나의 조직이 관리의 책임을 지는, PMI의 기본 프로그램 관리 맥락과는 달리 확장된 나선형 모델에서는 장기간에 걸쳐 다수의 조직이 관리의 책임을 분담해야 한다. 따라서 확장된 나선형 모델에서는 이해관계가 서로 상충할 수도 있는 조직들, 예를 들어서 개발 조직들과 소유권을 갖고 있는 이용 조직간에 어떻게 관리 활동을 통합하고 조정할 것인가에 대한 연구가 필요하다. 그러기 위해서는, 기본적으로 포함되는 컴포넌트 프로젝트들이 병행적으로 수행되는 경우를 전제로 한 PMI 프로그램 관리 모델을 컴포넌트 프로젝트들이 순차적으로 수행되는 경우도 포함하도록 확장하는 것이 필요하다.

아마도 가장 중요한 관점은 어떻게 소유권을 갖는 이용 조직이 전체 생애주기 비용을 최소화할 수 있도록 자신의 관리 프로세스에 개발 및 유지보수 조직의 관리 프로세스를 통합시킬 것인가가 될 것으로 판단된다. 그러기 위해서는 감리(auditing)의 역할과 기능에 대한 재정의가 필요할 것으로 판단된다. 구체적으로는, 감리의 각 항목이 누구의 어떤(DP를 들어서, 단기적 또는 장기적) 이익을 보호하는가를 실증적으로 확인하는 것이 중요하다. 또한 전체적인 개념적 틀 안에서 각각의 조직이 자신의 프로세스를 어

떻게 최적화시킬 것인가에 대한 실증적 연구가 필요하다.

## 참 고 문 헌

[1] 고석하, "소프트웨어 아키텍처의 구성요소에 대한 포괄적 모델", *Journal of Information Technology Applications and Management*, Vol. 19, No. 2, June 2012, pp. 135-147.

[2] 이현우, 박찬석, 고석하, "객체지향 개발 프로세스에서 비즈니스 프로세스 모델과 소프트웨어 아키텍처의 관계 연구를 위한 접근 방법의 제언", *Entrue Journal of Information Technology*, Vol. 8, No. 2, 2009, pp. 19-29.

[3] Abrahamsson, P., Babar, M. A., and Kruchen, P., "Agility and Architecture : Can They Coexist?", *IEEE Software*, March/April 2010, pp. 16-22.

[4] Ambler, S., "Agile Architecture : Strategies for Scaling Agile Development", Agile Modeling, 2008, http://www.agilemodeling.com/essays/agileArchitecture.htm(참조일 : 2011. 10. 29).

[5] Banker, R. D., Datar, S. M., Kermer, C. F., and Zweig, D., "Software Complexity and Maintenance Costs", *Communications of the ACM*, Vol. 36, No. 11, 1993, pp. 81-94.

[6] Bass, L., Clements, P., and Kazman, R., Software Architecture in Practice (2nd ed.), Addison-Wesley, 2003.

[7] Blair, S. and Watt, R., "Responsibility-Driven Architecture", *IEEE Software*, March/April, 2010, pp. 26-32.

[8] Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., and Madachy, R., "Using the WinWin Spiral Model : A Case Study", *IEEE Computer*, Vol. 31, No. 7, July, 1998, pp. 33-44.

[9] Boehm, B. W., "A Spiral Model of Software Development and Enhancement", *Computer*, May 1988, pp. 73-82.

[10] Boehm, B. W., "Anchoring the Software Process", *IEEE Software*, July 1996, pp. 73-82.

[11] Bosch, J., Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000.

[12] Brown, N., Nord, R., and Ozkaya, I., "Enabling Agility through Architecture", CrossTalk, Nov/Dec 2010, pp. 12-17.

[13] Buschmann, F., Pattern-Oriented Software Architecture : A system of Patterns, John Wiley and Sons, 1996.

[14] Capiluppi, A., Fernandez-Ramil, J., Higman, J., Sharp, H. C., and Smith, N., "An Empirical Study of the Evolution of an Agile-Developed Software Systems", in : 29th International Conference on Software, 2007, *Minneapolis*, MN, pp. 511-518.

[15] Christensen, H. B., Hansen, K. M., and Schougaard, K. R., "An Empirical Study of Software Architects' Concerns", in : Asia-Pacific Software Engineering Conference, *IEEE Computer Society*, 2009, pp. 111-118.

[16] de Silva, L. and Balasubramaniam, D., "Controlling Software Architecture Erosion : A Survey", *Journal of Systems and Systems*, Vol. 85, 2012, pp. 132-151.

[17] de Grace, P. and Stahl, L. H., Wicked Problems, Righteous Solutions : A Catalog of Modern Engineering Paradigms, Englewood Cliffs, NJ : Yourdon Press, 1990.

[18] Eclipse, Opne UP ver. 7, 2012. 5. 30, http://epf.eclipse.org/wikis/openup/index.htm : 참조일 2012. 7. 16.

[19] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data", *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, 2001, pp. 1-12.

[20] Hochstein, L. and Lindvall, M., "Combating Architectural Degeneration : A Survey", *Information and Software Technology*, Vol. 47, No. 10, 2005, pp. 643-656.

[21] Hofmeister, C., Kruchten, P., Nord, R., Obbink, H., Ran, A., and America, P., "A General Model of Software Architecture Design Derived from Five Industrial Approaches", *Journal of Systems and Software*, Vol. 80, pp. 106-126.

[22] Larmen, C., Applying UML and Patterns : An Introduction to Object Oriented Analysis and Design and Iterative Development (3rd ed.), Upper Saddle River, NJ : Prentice Hall PTR, 2005.

[23] Lehman, M. M., "Feedback, Evolution and Software Technology", *in : Proceedings of the 10th International Process Support of Software Product Lines Software Process Workshop*, 1996b, pp. 101-103.

[24] Lehman, M. M., "Laws of Software Evolution Revisited", *in Proceedings of the 5th European Workshop on Software Process Technology*, Springer, 1996a, pp. 108-124.

[25] Lehman, M. M., "On Understanding Law, Evolution, and Conversation in the Large-Program Life Cycle", *Journal of systems and Software*, Vol. 1, No. 3, 1980, pp. 213-231.

[26] Lehman, M. M. and Ramil, J. M., "Towards a Theory of Software Evolution-and its Practical Impact", *in : Proceedings of the International Symposium on Principles of Software Evolution*, 2000, pp. 2-11.

[27] Lehman, M. M. and Belady, L., Software Evolution-Process of Software Change, Academic Press : London, 1985.

[28] Lindval, M., Tesoriero, R., and Costa, P., "Avoiding Architectural Degeneration : An Evaluation Process for Software Architecture", in : Proceedings of the 8th International Symposium on Software Metrics, *IEEE*, 2002, pp. 77-86.

[29] Mattson, A., Lundell, B., Lings, B., and Fitzgerald, B., "Linking Model-Driven Development and Software Architecture", *IEEE Trans. on Software Engineers*, Vol. 35, No. 1, January/February, 2009, pp. 83-93.

[30] McConnell S., Rapid Development : Timing Wild Software Schedules, Redmond, Washington : Microsoft Press, 1996.

[31] Miksovic C. and Zimmermann, O., "Architecturally Significant Requirements, Refernce Architecture, and Metamodel for Knowledge Management in Information Technology Services", (DOI 10.1109/WICSA.2011.43) Ninth Working IEEE/IFIP Conference of Software Architecture, 2011.

[32] Mirakhorli, M., "Tracing Architecturally Significant Requirements : A Decision-Centric Approach", (ACM 978-1-4503-0445-0/11/05) *ICSE*, Vol. 11, May pp. 21-28, 2011, Waikiki, Honolulu, HI, USA, 2011.

[33] OMG, Business Process Model and Notation(BPMN), ver.2.0, OMG, 2010.

[34] Ozkaya, I., Dias-Pace, A., Gurfinkel, A., and Chai, S., "Using Architecurally Significant Requirements for Guiding System Evolution", *14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 127-136.

[35] Parnas, D. L., "Software Aging", *in : Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy*, 1994, pp. 279-287.

[36] Perry, D. E. and Wolf, A. L., "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, Vol. 17, No. 4, 1992, pp. 40-52.

[37] PMI(Project Management Institute), A Guide to the Project Management Body of Knowledge (4th ed.), PMI, 2008a.

[38] PMI, The Standard for Program Management, 2nd ed., PMI, 2008b.

[39] Port, D. and Ligou, H., "Strategic Architectural Flexibility", in Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 2003, pp. 389-396.

[40] Rowe, D., Leaney, J. R., and Lowe, D. B., "Defining Systems Evolvability-A Taxonomy Change", In Proc. of ECBS'98, 1998.

[41] Royce, W. W., "Managing the Development of Large Software Systems", in Proceedings, *IEEE WESCON*, Aug. 1970, pp. 1-9.

[42] Rozanski, N. and Woods, E., Software Systems Architecture(2nd ed.), Pearson Education, Inc., 2012.

[43] Sadou, N., Tamzalit, M., and Oussalah, A., "A Unified Approach for Software Architecture Evolution at Different Abstract Levels", in : Proceedings of the 2005 8th International Workshop on Principles of Software Evolution, 2005, pp. 65-68.

[44] Stapleton, DSDM : Dynamic Systems Development Method, Harlow, England : Addison-Wesley, 1997.

[45] Tang, A., Avgeriou, P., Jansen, A., Capilla, R., and Ali Babar, M., "A Comparative Study of Architecture Knoweldge Management Tools", *Journal of Systems and Software*, 2009.

[46] Unphon, H. and Dittrich, Y., "Software Architecture Awareness in Long-term Software Product Evolution", *Journal of Systems and Software*, Vol. 83, 2010, pp. 2211-2226.

[47] van Gurp, J., J. Bosch, "Design Erosion: Problems and Causes", *Journal of System Software*, Vol. 61, No. 2, 2002.

[48] Weinreich, R. and Buchgeher, G., "Towards Supporting the Software Architecture Life Cycle", *The Journal of Systems and Software*, Vol. 85, 2012, pp. 546-561.

[49] Williams, B. and Carver, J. C., "Characterizing Software Architecture Changes : A Systemic Review", *Information and Software Technology*, Vol. 52, 2010, pp. 31-51.

[50] Young, R. R., Effective Requirements Practices, Boston : Addison Wesley, 2001.

## <부록 A> 소프트웨어 아키텍처의 품질

아키텍처의 품질의 측정을 위해서는 유연성, 진화성, 확장성, 또는 유지보수성 등의 다양한 척도가 사용된다[Abrahamsson et al., 2010; Unphon and Dittrich, 2010]. 유연성(flexibility)은 시스템이 계획되지 않은 수정을 수용할 수 있는 정도로 정의할 수 있다[Port and Ligou, 2003], 유연성은, 후기 변경이 시스템을 원래의 설계에서 멀어지게 함으로써, 감소한다[Williams and Carver, 2010]. 진화성(evolvability)은 '시스템의 전 생애주기를 통해서 요구사항의 변화를 아키텍처 무결성(architectural integrity)을 유지하면서 가능한 최소한의 비용으로 수용할 수 있는 능력과 관계된 속성'으로 정의할 수 있다[Rowe et al., 2006]. 이 두 정의에서도 확인할 수 있듯이, 아키텍처의 품질 척도들의 대부분은 아키텍처 퇴화의 개념과 연관되어 있다.

한편, 공학적 품질(engineering quality)은 다음의 것들을 포괄하는 개념으로 정의된다[de Silva and Balasubramaniam, 2012].

- 아키텍처 무결성(architectural integrity) : 완전성(ompleteness), 올바름(correctness), 일관성(consistency) 등,
- 품질 속성 요구사항들에의 적합성(conformance),
- 건전한 소프트웨어 공학 관행(예를 들어, 모듈화(modularity) 및 낮은 결합도(low coupling) 등)의 채택,
- 아키텍처 논리와 해당하는 설계 결정의 준수.

공학적 품질과 성능은 반드시 일치하는 않는다(de Silva and Balasubramaniam 2012). 즉, 공학적 품질이 낮은 시스템도 잘 기능할 수 있으며, 반대로 공학적 품질이 높은 시스템이 기대대로 기능하지 않을 수도 있다. 그러나 공학적 품질이 낮은 소프트웨어는 고치기 힘들다(Perry and Wolf 1992). 소프트웨어의 후기 생애주기에서 요구사항의 변경이 일어났을 때, 공학적 품질이 높을수록 이러한 변경을 수용하기 쉽다.

소프트웨어의 공학적 품질의 가치를 결정하는 가장 중요한 요인은 아키텍처적 상호의존성(architectural dependency)이다. 상호의존성이 클수록 높은 공학 품질의 아키텍처의 가치가 높아진다[de Silva and Balasubramaniam, 2012]. 따라서 평가기간이 길수록 품질 높은 아키텍처의 가치에 대한 평가치가 높아진다. 즉, 평가기간이 길수록 전체 생애주기에서 발생할 상호의존성 중에서 평가에 반영되는 부분이 커지며, 따라서 품질 높은 아키텍처의 가치에 대한 평가가 정확해짐과 동시에, 그 평가치가 높아진다. 결과적으로, 개별적인 개발 프로젝트 관점에서 평가할 때보다 소프트웨어 생애주기 전체에 걸친 프로그램의 관점에서 평가할 때에 개발 초기에 높은 품질의 아키텍처를 구축하려는 BDUF의 타당성이 더 높아진다.

## <부록 B> PMI의 프로젝트 및 프로그램 관리 모델의 개요

PMI의 프로젝트 및 프로그램 관리 모델의 개요는 다음과 같다[PMI 2008a, 2008b].

- 활동들을 서로 밀접하게 상호작용하는 프로세스로 분류한다.
- 프로세스를 다음과 같이 크게 두 가지로 분류한다.

◦ 제품 프로세스(product-oriented process) : 프로젝트/프로그램의 목표 인도물을 명기하고 생성하며, 응용 영역에 종속되는 프로세스.
◦ 관리 프로세스(management process) : 모든 프로젝트/프로그램에 공통적이며 인도물에 독립적인 프로세스.

• 관리 프로세스를 프로세스 그룹(초기화, 계획, 실행, 모니터링 및 통제, 종료)과 관리 지식 영역의 두 가지 기준에 의해서 세분한다.
◦ 프로젝트 : 42개의 프로세스가 있으며, 관리 지식 영역을 통합, 범위, 시간, 비용, 품질, 인적자원, 의사소통, 조달, 위험 관리의 9개 영역으로 분류한다.
◦ 프로그램 : 47개의 프로세스가 있으며, 관리 지식 영역을 통합, 범위, 시간, 의사소통, 조달, 위험 관리, 재무, 이해관계자, 거버넌스 관리의 9개 영역으로 분류한다.
◦ 각 지식 영역별로 관리 계획을 작성하며, 각 세부 관리 계획들을 프로젝트/프로그램 관리 계획으로 통합한다.
◦ '변경 관리'는 전체 관리 프로세스를 통해서 수행되며, 프로그램 수준에서는 '프로그램 변경을 모니터하고 통제하기,' 프로젝트 수준에서는 '통합된 변경 통제를 수행하기' 프로세스에서 통합한다.

• '이해관계자를 파악하기' 프로세스를 포함하는, 요구사항과 관련된 일부 프로세스들을 관리 프로세스로 분류한다.
◦ 프로젝트 : 요구사항 수집하기(collect project requirements), 범위를 정의하기, 범위를 검증하기, 범위를 통제하기.
◦ 프로그램 : 요구사항 개발하기(develop program requirements), 범위를 계획하기, 범위를 모니터링 및 통제하기.

• 프로그램 수준에서는 속해있는 프로젝트들 간의 상호의존성을 관리하기 위하여 '아키텍처를 개발하기'와 '아키텍처를 관리하기' 프로세스를 수행하고, 이에 대한 계획을 '프로그램 범위를 계획하기' 프로세스에서 수립한다.
◦ 아키텍처를 개발하기 프로세스 : 프로그램 컴포넌트의 구조를 정의하고 그것들 간의 상호관계를 확인한다.
◦ 아키텍처를 관리하기 프로세스 : 프로그램 아키텍처가 현재의 요구사항에 부합하도록 모든 프로그램 컴포넌트들 간의 상호관계를 관리하기.

• 종료 프로세스 그룹에서는 수집된 선례 및 교훈(lesson learned) 정보를 교훈 지식 기반으로 전송하고 조직 프로세스 자산(organizational process assets)을 개선한다. 조직 프로세스 자산은 크게 프로세스와 절차, 그리고 기업 지식 기반으로 나눈다.
◦ 교훈 지식 기반(lesson learned knowledge base) : 선행 프로젝트/프로그램에서 수집된 이슈와 위험, 향후에 적용할 수 있는 기법 등을 포함한다.
◦ 프로세스와 절차(processes and procedures) : 작업을 수행하기 위한 각종의 프로세스와, 절차, 그리고 관련된 지침, 기준, 템플릿 및 서식 등을 포함한다.
◦ 기업 지식 기반(corporate knowledge base) : 교훈 지식 기반, 선례 정보 외에 조직 내의 모든 공식적인 표준과 정책 등을 포함한다.

## 저자소개

**고 석 하**

현재 충북대학교 경영정보학과 교수로 재직 중이다. 주요 관심분야는 Software Quality Management, Business Process Modeling, Project Management, Software Engineering, IS Education and Curriculum 등이다.

◈ 이 논문은 2013년 03월 28일 접수하여 2013년 06월 15일 게재확정되었습니다.