소프트웨어 아키텍처 설계의 근본 원리들

(Fundamental Principles for Software Architecture Design)

강성원*

(Sungwon Kang)

요 약 이 논문에서는 소프트웨어의 개념과 소프트웨어 개발의 중요성을 알아본 뒤, 효과적인 소프트웨어 개발을 위한 필수적인 기술로서 열 두 개의 근본적인 소프트웨어 아키텍처 설계원리를 제시한다. 아키텍처 설계단계들을 문제의 파악, 아키텍처 모델링 방법의 결정, 아키텍처 설계의 수행 그리고 설계된 아키텍처의 평가의 네 개의 단계 군으로 나누고, 이 단계군 안에서 그리고 단계군 간에 활용되는 아키텍처 설계의 원리들을 식별하고 그 역할을 설명한다.

키워드 소프트웨어 아키텍처, 소프트웨어 아키텍처 설계, 소프트웨어 아키텍처 설계원리, 소프트웨어 아키텍처 설계절차

Abstract This paper first examines the notion of software and the importance of software development and then presents twelve fundamental principles for software architecture design as the key enabling technology for effective software development. This paper divides design steps into four groups, i.e. analyzing the problem, deciding architecture modeling methods, architecture design process and architecture evaluation. Then it identifies the principles within and across the various steps of software architecture design and explains their roles.

Key words Software architecture, software architecture design, software architecture design principles, software architecture design procedure

1. 컴퓨터 소프트웨어

1.1 소프트웨어와 컴퓨터 소프트웨어

소프트웨어라는 용어는 컴퓨터 하드웨어에 대비되는 표현으로서, 컴퓨터를 위한 프로그램을 지칭하는 용어로 사용된다. 컴퓨터 소프트웨어는 처음에는 컴퓨터를 동작하게 하는 일련의 컴퓨터 명령을 의미하였다. 그러나 프로그래밍 언어, 프로그램 라이브러리, 프로그램 프레임워크, 미들웨어 등의 발전으로 컴퓨터와의 세부적인 연결을 직접 표현하지 않고도 사용자가 필요한 연산, 데이터 처리, 활동 등을 표현할 수 있게 되면서, 컴퓨터를 제어하는 용도보다는 인간생활의 니즈(needs)를 표현하는 용도로 사용되는 방향으로 발전되어 가고 있다.

따라서 컴퓨터를 명령하기 위한 표현으로서의 소프트웨어를 넘어서, 소프트웨어가 무엇이며, 무엇을 위한 것인지를 생각해 볼 필요성이 자연스럽게 대두 되었다. 그것은 컴퓨터 소프트웨어의 역할이 본질적으로 무엇인가를 생각하며, 현대적인 컴퓨터가 존재하기 이전의 세계로 그 역할을 투영해봄으로써 더 잘 생각해 볼 수 있을 것이다.

1.2 역사 속의 소프트웨어 개념

인류는 오랫동안 천을 짜는 기계인 방직기를 이용하여 다양한 무늬를 갖는 천을 만들어 왔다.

* 종신회원 : 한국과학기술원 전산학과 부교수
sungwon.kang@kaist.ac.kr
논문접수 : 2010년 11월 20일
심사완료 : 2010년 12월 23일

이 때 다양한 무늬를 만들기 위하여 먼저 카드에 방직기의 동작을 제어할 수 있는 구멍을 무늬에 맞게 뚫는데, 이 때 방직기는 하나의 하드웨어이고 카드의 구멍의 배열은 하나의 소프트웨어로 방직기를 위한 프로그램인 것이다.[1] Ada Augusta (1815 – 1852)는 Charles Babbage (1791 – 1871)의 해석엔진(Analytical Engine)[2]을 위한 프로그램을 설계하였는데, 이 프로그램을 최초의 컴퓨터 소프트웨어로 간주하기도 한다. 또한 초기의 컴퓨터에는 전기-기계식 컴퓨터[3]도 있었으며 프로그래머가 스위치 보드 하드웨어에 직접 프로그래밍을 하기도 하였다.

이와 같이 현대적 컴퓨터가 존재하기 이전에도 소프트웨어는 존재하였고 사람들은 프로그래밍 활동을 하였으며, 현대적 컴퓨터가 등장한 후에 새로운 국면을 맞게 되지만, 소프트웨어와 프로그래밍은 이런 긴 역사를 가지고 있다고 볼 수 있다.

1.3 서비스, 소프트웨어 서비스, 컴퓨터 소프트웨어 서비스

컴퓨터 소프트웨어는 하나 하나의 단계가 엄밀히 수행될 수 있는 알고리즘으로서의 특징을 통하여 우리가 필요로 하는 계산, 정보의 전달, 물품의 구매, 전통적인 제반 분야의 서비스 등의 니즈를 달성하는 서비스 매체로서의 특징을 갖는다. 거꾸로, 컴퓨터를 사용하지 않더라고, 그러한 니즈를 달성하게 해주는 엄밀한 절차가 존재한다면 그것을 소프트웨어라고 부를 수 있을 것이다. 과거에는 오늘날 컴퓨터 소프트웨어가 하는 많은 서비스를 컴퓨터 소프트웨어 없이 인간의 두뇌를 통하여 달성하였던 것이다.

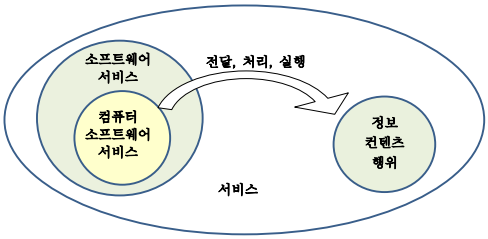

그림 1. 서비스, 소프트웨어 서비스, 컴퓨터 소프트웨어 서비스의 개념

그림 1은 전통적인 서비스의 개념과 소프트웨어 서비스 그리고 컴퓨터 소프트웨어 서비스의 개념 간의 관계를 보여 준다. 전통적인 산업의 분류상 1차, 2차 산업이 각각 원료의 수집과 원료의 가공을 통한 유형적 물건의 조달 및 제조활동이었던 비하여, 무형적인 대상으로 사용자의 니즈를 충족시켜주는 행위가 3차 산업인 서비스 산업으로 분류되었다. 소프트웨어 서비스는 정보와 컨텐츠를 제작, 전달, 가공하는 활동을 포함한다. 또 서비스는 우리가 소프트웨어 서비스라고 부르는 것과 아울러 정보와 컨텐츠를 포함한다. 정보와 컨텐츠는 디지털화 되면 컴퓨터 소프트웨어 서비스가 가능한 형태가 되어, 전달, 처리 등의 컴퓨터 소프트웨어 서비스를 통하여 가공되어 부가가치를 창출하는 새로운 것이 될 수도 있다.[4]

1.4 컴퓨터 소프트웨어의 역할의 발전

소프트웨어라는 용어가 컴퓨터 소프트웨어라는 용어를 통하여 일상화 된 지는 오래되지 않지만, 소프트웨어가 인간 사회에 등장한 것은 매우 오래 전이었다. 그러나 컴퓨터의 등장으로 인하여 인간이 소프트웨어를 이용하는 방법은 극적으로 넓어졌고, 이제는 컴퓨터 소프트웨어는 산업과 생활을 위하여 업무, 공장, 오락, 교육 등 광범위한 분야에서 필수적인 기능을 수행하게 되었고 나아가 과학기술의

1) Joseph Jacquard(1752-1834)는 1801년 이렇게 동작하는 자동 방직기를 설계하였다.
2) Charles Babbage는 결국 해석엔진을 완성하지 못하였다.
3) 1940년대 Konrad Zuse (1910-1995)는 최초의 전기-기계식 컴퓨터 Z3를 만들었다.
4) 이와 같은 통찰은 다시, 어떤 전통적인 산업분야나 생활분야가 컴퓨터 소프트웨어의 도움을 받아 발전될 수 있는가의 방향을 시사할 것이다.

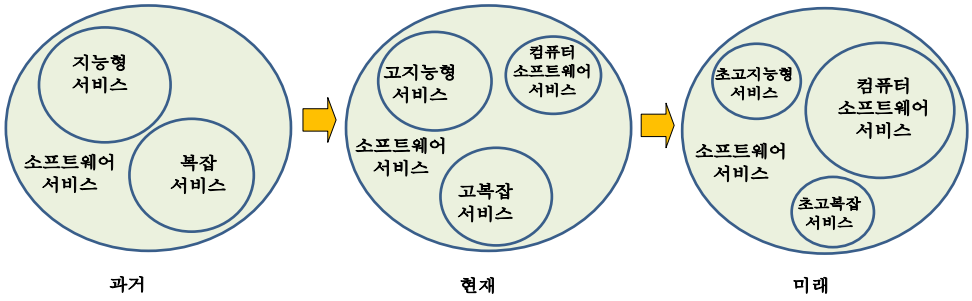

그림 2. 컴퓨터 소프트웨어의 역할의 발전

발전을 이끄는 도구로서도 눈부신 역할을 하게 되었다. 그림 2은 이런 컴퓨터 소프트웨어 역할의 발전 과정을 보여준다.

소프트웨어 서비스 중에서 단순하고 간단한 것들은 컴퓨터의 등장과 함께 바로 현재에 컴퓨터 소프트웨어 서비스로 바뀌어, 높은 지능을 요구하거나 복잡한 서비스가 아닌 것은 현재에는 컴퓨터 소프트웨어 서비스로 제공되게 되었다. 가까운 미래에는 매우 복잡하거나, 매우 높은 지능을 요구하는 서비스들을 제외하고 컴퓨터 소프트웨어 서비스에 의하여 제공될 것이면, 이 들 중에 많은 것들도 궁극적으로 컴퓨터 소프트웨어 서비스로 제공되게 될 것이다.

1.5 컴퓨터 소프트웨어 기술의 중요성

컴퓨터는 컴퓨터 소프트웨어를 이용하기 위한 장치이고 컴퓨터 소프트웨어를 이용 가능하게 만들어 주는 도구이다. 과거에 석기시대에는 돌을 잘 다루는 부족이, 철기시대에는 철을 잘 다룰 줄 아는 국가가, 산업혁명의 시대에는 기계를 잘 만들어 다룰 줄 아는 민족이 세계를 지배했던 것처럼 컴퓨터 시대에 컴퓨터 소프트웨어를 잘 만들어 다루는 사람들이 세상을 지배하게 될 것이라고 보아도 과언이 아닐 것이다.[5] 이와 같은 추측을 뒷받침하는 증거로, 현대의 전쟁이 로봇병사와 무인비행기를 이용하여 수행되고 있고, 현대적 기업이 비즈니스를 컴퓨터 소프트웨어로 수행하여 비용을 절감하고 고객의 취향을 실시간에 파악함으로써 기업이 경쟁력을 갖게 되는 등의 예에서 볼 수 있다. 그리고 컴퓨터소프트웨어를 잘 만드는 방법에 대한 연구도 당연히 컴퓨터의 역사만큼이나 길었던 것이다.

Mark Weiser는 수 많은 컴퓨터가 우리 생활 속에 밀집되어 퍼져있는(pervasive) 새로운 문명 세계에 대한 비전을 제시하였다 [9]. 이 비전은 이미 현실로 되어 가고 있다. 이러한 현상은 이 많은 컴퓨터들이 원하는 대로 기능하도록 하기 위한 많은 다양한 컴퓨터 소프트웨어가 필요하다는 것을 의미하고, 나아가 이들 컴퓨터를 이용한 다양한 (소프트웨어) 서비스들도 만들어져야 한다는 것을 의미한다.

1.6 컴퓨터 소프트웨어를 어떻게 잘 다룰 수 있는가?

좋은 소프트웨어를 만들기 위하여, 우선 정확한 요구사항의 명세가 있어야 한다. 올바른 명세를 확보한 후에는 물론 그것을 잘 설계하고 잘 구현하여야 한다. 또한 시험을 철저하게 하여야 할 뿐 아니라, 이 전체를 위한 공정이 잘 계획되고 준수되어야 한다. 그러나 좋은 소프트웨어가 만들어지기 위해서는 무엇보다도 설계가 잘 만들어져야

5) C++프로그래밍 언어의 설계자인 Bjarne Stroustrup는 "Our civilization runs on software."는 말을 하여 이를 잘 요약해 주고 있다. (http://www.handbookofsoftwarearchitecture.com/index.jsp?page=main&part=Preface 에서 재인용)

한다. 예를 들어 시험이나 공정이 아무리 효과적이어도, 잘못된 구현이나 설계의 문제점을 지적해 줄 뿐이고, 올바른 구현과 설계를 만드는 작업은 다시 수행되어야 하기 때문이다. 또한 구현과 설계 가운데 구현이 설계에 의존하기 때문에 구현에 문제가 있을 때보다 설계에 문제가 있을 경우 이를 바로 잡는데 훨씬 더 많은 비용이 들어가게 된다.

어느 정도의 규모를 가진 소프트웨어의 경우 반드시 설계를 거쳐서 구현을 해야 한다는 것은 잘 알려진 사실이다. 만들어야 하는 소프트웨어의 규모가 커지고 복잡도가 높아짐에 따라, 이 두 가지를 정면으로 다룰 수 있는 방법이 필요하게 되었다. 이를 위하여 설계는 다시 아키텍처 설계와 상세 설계로 나뉘어져 아키텍처 설계단계에서 이 문제들의 해결을 시도하게 된다.

2. 소프트웨어 아키텍처의 개념과 중요성

1969 NATO Conference on Software Engineering Techniques 에서 Ian P. Sharp는 다음과 같이 말하였다:

"우리가 필요한 것은 소프트웨어공학 이외에도 더 있다. 그것에 관하여 우리가 조용히 얘기하여 왔지만, 공개적으로 논의하고, 관심을 가져야 할 것이다. 그것은 바로 소프트웨어 아키텍처이다. 아키텍처는 공학과 다르다. 이 말의 의미를 생각해 보기 위하여 예를 들어 OS/360 운영체제의 예를 들어 보자. OS/360은 각 부분별로는 매우 코딩이 잘 되어 있다. 세부적으로 들어가 보면 OS의 부분 부분은 우리가 좋은 프로그래밍 기법이라고 보는 기법들과 모든 개념들을 쓰고 있다. 그런데 OS/360가 형체 없는 프로그램의 덩어리(amorphous lump of program)가 된 이유는 아키텍트가 없었기 때문이다. 설계는 많은 그룹의 엔지니어들에게 위임되었고, 그들은 각자의 아키텍처를 반영하여야 하였다. 그리고 이 덩어리들을 최종적으로 통합하였을 때에는 전체가 부드럽고 아름다운 소프트웨어가 되지 못하였다." [7]

2.1 전통적 설계 접근 방법의 불충분성

Ian P. Sharp의 말은 소프트웨어 개발에 있어서 아키텍처가 어떤 역할을 수행해야 하는지를 잘 보여준다. 전통적으로 소프트웨어는 그림 3이 보여주는 것처럼 요구사항분석, 설계, 구현, 통합 및 시험이라는 단계를 거쳐 개발되었다. 그러나 전통적 소프트웨어 개발에서의 설계의 역할은 오늘날 우리가 아키텍처에 기대하는 역할에 크게 미치지 못하였다. 예를 들어, 객체지향설계 방법의 경우 시스템을 많은 객체들의 구성으로 보라고 하지만, 많은 객체들을 쉽게 이해하고 다룰 수 있도록 어떻게 더 큰 단위로 볼 수 있는가에 대하여는 말해 주고 있지 않다.

그림 3. 개발순기

아키텍처 설계의 목적이 요구사항과 구현을 "연결" 시키는 것인데, 전통적인 설계방법들도 요구사항과 구현의 간격을 메워 이들을 연결시키는 역할을 수행한다. 그러나 이 둘은 그 접근방식에서 다르다. 그림 4에서 네모는 산출물 혹은 모델을 나타내고 화살표는 자료의 흐름을 나타낸다. 그림 4(a)처럼 설계나 아키텍처설계 없이는 구현자는 즉흥적인 방법으로 구현을 하게 된다. 전통적인 설계방법들은 요구사항들을 구현으로 가져가는 길을 제시한다(그림 4(b)). 반면 아키텍처 설계는

요구사항으로부터 설계로 가는 기계적인 단계들을 명시하려고 하는 대신, 서로 다른 대안 아키텍처가 이 들에 기초한 선택과 같은 문제에 관심을 둔다 (그림 4(c)). 따라서 전통적인 설계와 아키텍처 설계는 상호보완적이다.

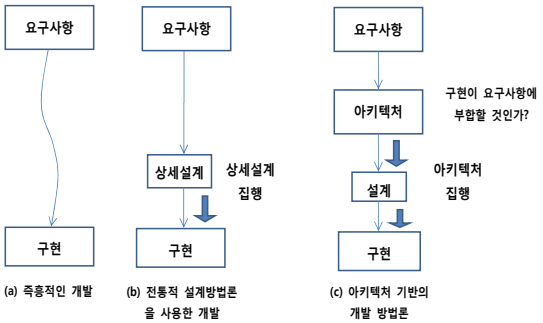

그림 4. 아키텍처 기반 개발의 장점

2.2 소프트웨어 아키텍처의 역할

따라서 M. Shaw, D. Garlan, P. Kruchten, Perry을 비롯한 소프트웨어 아키텍처 연구의 선구자들은 소프트웨어 개발의 중추적인 역할(pivotal role)을 수행하는 아키텍처의 필요성을 역설하였다.

중심축(pivot)이 한 방향으로부터 온 힘을 다 받아서, 다른 방향으로 전달하는 역할을 수행한다 (그림 5 참조).

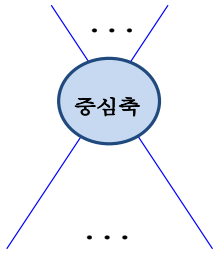

그림 5. 중심축의 개념

소프트웨어 개발에서 아키텍처도 바로 이러한 역할을 수행한다. 따라서 그림 6이 표현하고 있는 것처럼, 아키텍처 설계는 요구사항 분석 활동으로 얻어진 요구사항을 다 충족시킬 수 있는 시스템이 궁극적으로 만들어 질 수 있도록 하기 위한 설계 활동으로서 설계 및 구현을 위한 구조적(structural) 및 비구조적(nonstructural) 틀을 제공하는 것이다. 여기서 '틀'은 아키텍처 설계의 결과물로서 집행을 요구하는 결정 혹은 모델을 말한다. 따라서 구조적 틀이라 함은 시스템을 컴포넌트로 구성된 구조로 보아 시스템 개발문제를 용이하게 끌어가도록 결정된 컴포넌트의 구조모델을 말하고, 비구조적 틀은 이 구조모델 이외의 다른 아키텍처설계의 결정들을 말한다.

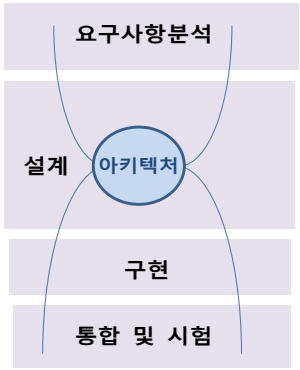

그림 6. 소프트웨어 개발의 중심축으로서의 아키텍처

3. 소프트웨어 아키텍처의 정의

이 논문의 제 1절에서는 소프트웨어 아키텍처의 기본적인 개념을 살펴보았다. 이 절에서는 소프트웨어 아키텍처에 대한 엄밀한 정의를 내려, 이어지는 소프트웨어 아키텍처 설계에 대한 논의 의 기반을 확립한다.

3.1 아키텍처 와 소프트웨어 아키텍처

1980년대까지 아키텍처라는 용어는 컴퓨터 아키텍처의 의미로 쓰였고, Fred Brooks, Butler Lampson, David Parnas, John Mills등을 통하여 시스템의 "소프트웨어 조직"의 의미로도 쓰이기 시작하여, 소프트웨어 아키텍처가 본격적으로 연구되기 시작하였다.

3.2 설계의 다양한 수준

건물, 교량, 도시를 건설할 때 건축가는 먼저 그 설계를 한다. 설계는 중요한 결정들을 하기 위한 아키텍처설계와 세부작업의 바탕이 되는 상세설계로 나눌 수 있다. 예를 들어 교량의 아키텍처설계에는 강물의 속도와 강의 폭, 단위시간에 교량을 건너는 차량의 평균수 및 최대수, 차의 최대 무게 등을 고려하여 다리의 공법, 구조, 폭 과 강도가 결정되어야 할 것이다. 교량의 상세 설계에서는 교량의 아키텍처 설계의 결정들이나 다른 상세설계 작업에 큰 파급효과가 없는 사항들이 결정된다.

이와 같이 소프트웨어 개발에서도 다양한 수준의 설계작업이 있고, 소프트웨어 시스템의 아키텍처 설계에 제약을 주는 상위설계로서의 시스템 아키텍처의 설계 가 있고 소프트웨어 아키텍처의 제약 안에서, 세부적인 설계작업을 수행하게 되는 상세설계가 있다. 그림 7은 이 세 개념들 사이의 관계를 보여준다.

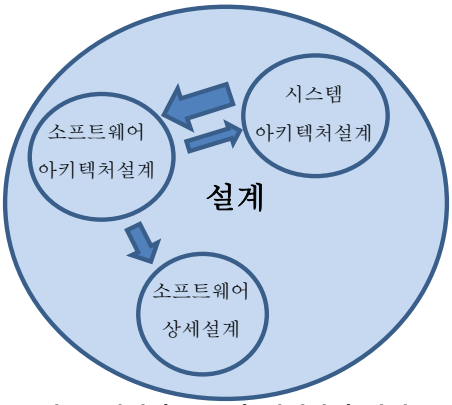

그림 7. 설계의 종류와 설계간의 관계

이들은 일반적인 설계활동의 하나로서 설계가 갖는 일반적인 특성을 공통적으로 갖는다. 그러나 또한 이들은 각기 다른 수준의 설계를 지칭하여, 시스템 아키텍처는 하드웨어와 소프트웨어로 구성된 시스템의 아키텍처를 지칭하며 소프트웨어 아키텍처의 결정을 제약하게 되고, 소프트웨어 아키텍처 설계는 소프트웨어 상세설계의 결정을 제약하게 된다. 그림 7에서 화살표의 굵기는 영향력의 크기를 나타낸다.

3.3 과거 연구자들의 소프트웨어 아키텍처 정의

과거 많은 소프트웨어 아키텍처의 연구자들이 소프트웨어 아키텍처에 대하여 다양한 정의를 내렸다. 그 중 다음과 같은 대표적인 정의들이 있다:

"소프트웨어 아키텍처는 소프트웨어 시스템들의 큰 규모의 구조와 실행에 관한 연구이다"[7].

소프트웨어 아키텍처란 "시스템의 근본적인 조직형태로써, 그것은 구성컴포넌트들과, 그들 서로와 환경에 대한 관계 그리고 그 설계와 진화를 관장하는 원칙들에 담겨있다" [4].[6]

소프트웨어 "아키텍처란 1)시스템의 구조를 나타내는 구조적 구성요소와 그들을 결합시키는 인터페이스, 2)그들의 협동을 통해 나타나는 구조적 구성요소와 행위적 구성요소들의 결합을 점진적으로 서브시스템에 맵핑을 하는 것, 3)구조를 이끌어 나가는 아키텍처 스타일의 선택"에 관한 중요한 결정들의 집합이다. [5][8].[7]

"프로그램 혹은 컴퓨팅 시스템의 소프트웨어 아키텍처란 소프트웨어 구성요소, 이들 구성요소의 가시적인 속성, 그리고 구성요소 사이에 관계로

6) Software architecture is "the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution" [6].

7) "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization – these elements and their interfaces, their collaborations, and their composition" [7,8].

구성된 시스템의 전체적인 구조 또는 구조들을 말한다"[1].[8]

3.4 소프트웨어 아키텍처의 정의

소프트웨어 아키텍처는 소프트웨어 시스템이 존재하기 이전에 그것에 관한 관한 많은 중요한 결정을 내릴 수 있게 하고 아키텍처 수준에서 소프트웨어 시스템의 분석을 가능하게 함으로써 계획된 소프트웨어 개발이 가능하도록 한다. 그림 8는 소프트웨어 아키텍처가 아키텍처 설계를 위하여 필요한 형태로 요구사항이 철저히 분석될 수 있도록 가이드 하는 한편, 아키텍처설계 단계에서 내려진 소프트웨어 구조를 포함한 제반 아키텍처적인 결정들이 상세설계, 구현, 통합 및 시험에까지 집행될 수 있도록 가이드 하는 구체적인 산출물임을 보여준다.

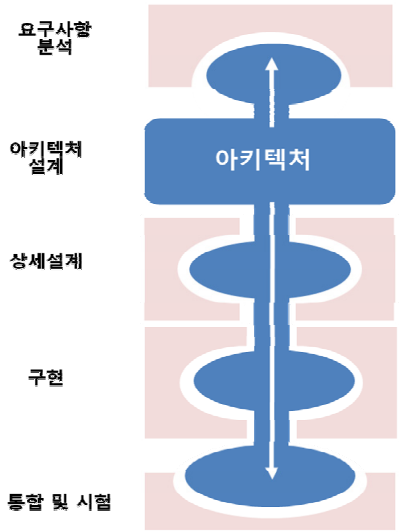

그림 8. 개발순기에 있어서 아키텍처의 역할

8) 이 정의는 논문 [4]에서 1994년에 SEI에서 논의되었다고 하는 "프로그램/시스템의 컴포넌트들이 갖는 구조, 그들의 관계 그리고 그들의 설계와 시간이 흐르면서 발생하는 진화를 지배하는 원칙과 가이드라인 (The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time)" 이라는 정의와 유사한 점이 있다.

소프트웨어 시스템은 개발 이후에도 많은 시간과 비용을 들여 진화하게 되는데, 아키텍처는 소프트웨어 진화과정에까지도 그 영향을 미친다. 따라서 우리는 다음과 같이 소프트웨어 아키텍처를 정의한다.

> **소프트웨어 아키텍처의 정의**
> 소프트웨어 아키텍처는 소프트웨어 시스템의 구조를 비롯한 시스템 개발에 중요한 영향을 미치는 결정들로, 소프트웨어 시스템 개발에서 특정 시스템에 대하여 요구되는 기능과 품질을 확보하고 또한 소프트웨어 시스템의 구축 및 지속적인 개선이 용이하도록 하는 역할을 한다.[9]

4. 소프트웨어 아키텍처의 설계

소프트웨어 아키텍처를 제 3절에서와 같이 정의한다면, 그러면 소프트웨어 아키텍처는 어떻게 만들 수 있을까, 즉 소프트웨어 아키텍처는 어떻게 설계하여야 하는가 하는 문제가 대두된다. 이 절에서는 소프트웨어 아키텍처 설계가 가져야 하는 본질적인 특징들을 살펴본다.[10]

4.1 분할정복으로서의 설계활동

분할 정복은 복잡한 문제를 풀기 위하여 원래의 문제를 작은 문제들로 나누고 먼저 작은 문제들에 대한 답을 얻은 뒤에 이 답들을 연결하여 전체 문제에 대한 답을 찾는 방법이다.

9) 이 정의는 논문 [4]에서 1994년에 SEI에서 논의되었다고 하는 "프로그램/시스템의 컴포넌트들이 갖는 구조, 그들의 관계 그리고 그들의 설계와 시간이 흐르면서 발생하는 진화를 지배하는 원칙과 가이드라인 (The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time)" 이라는 정의와 유사한 점이 있다.

10) 이러한 설계의 일반원리는 제 4절 그림 11 아키텍처 설계 절차에도 반영되어, 분할 정복원리는 컴포넌트와 커넥터의 이분법을 요구하고, 분석과 합성의 원리는 설계절차의 형태를 결정하고, 경험과 창의성의 결합원리는 설계에 적용되는 기법의 원리를 결정한다.

문제의 규모가 작고 복잡도가 낮을 경우에는 이런 접근 방법이 불필요할 수 있다. 그러나 문제의 규모가 클 경우 문제를 분할하여 작은 문제로 바꾸고 이들에 대한 해결을 통합하여 전체에 대한 해결하려 하는 것은 자연스럽고도 강력한 문제 해결 방법이다. 소프트웨어 아키텍처 설계는 본질적으로 규모와 복잡성을 다루는 접근방법이므로, 분할정복이 아키텍처 설계의 지배적인 해결 방식인 점은 아키텍처 설계를 일반적인 설계와 차별화해 주는 가장 큰 특징이다.

4.2 분석과 합성으로서의 설계활동

아키텍처 설계를 포함한 모든 설계는 창조적인 합성(synthesis)의 활동으로서[11] 좋은 결과를 가져오기 위하여는 분석(analysis)이라는 과정을 동반한다.[12] 그림 9는 통합과 분석이 서로 맞물리어 반복적으로 이어질 때, 좋은 설계결과가 만들어진다는 것을 나타낸다.[13] 아키텍처 설계도 하나의 설계 활동으로서 설계가 갖는 이러한 일반적인 특징을 갖는다.

그림 9. 합성과 분석의 반복으로 이루어진 과정으로서의 설계

4.3 경험과 창의성의 결합으로서의 설계활동

설계문제를 해결하기 위해서는 경험적인 수단과 창의적인 수단을 모두 동원하여야 한다. 앞에서 말한 분석과 합성은 일반적인 방법으로 원칙적으로 모든 문제에 대한 답을 줄 수 있는 틀이다. 그러나 합성과 분석에만 의존하는 것이 가장 효율적인 방법은 아니다. 경험적인 지식이 적용 가능할 때에는, 이를 잘 활용함으로써 많은 경우 짧은 시간에 더 높은 품질의 설계결과에 도달할 수 있다.

4.4 아키텍처 설계의 입력물과 출력물

설계가 분석과 합성의 반복적인 과정이라는 점에서 한 걸음 더 나아가서, 좀 더 상세히 설계의 과정을 들여다보기로 한다. 방법 혹은 과정은 그것이 어떤 것이든, 그것의 진행을 위하여 필요한 입력물이 있고,[14] 진행의 결과로서 만들어지는 출력물이 있다. 그렇다면 아키텍처 설계라는 활동의 입력물과 출력물은 무엇인가? 그림 10은 이를 도식적으로 보여준다.

소프트웨어 개발의 출발점은 요구사항이다. 아키텍처도 소프트웨어 개발을 체계적이고 목적에 맞게 달성할 수 있게 하는 역할을 수행하기 때문에 아키텍처 설계 역시 요구사항에서 출발하지만, 시스템 개발이 모든 요구사항을 그 달성대상으로 하는데 비하여, 아키텍처 설계는 아키텍처에 관련된 대표적인 요구사항들만이 주된 관심의 대상이 된다. 이러한 특별한 요구사항들을 아키텍처 드라이버라고 부른다. 한편 설계의 결과는 아키텍처를 문서화한 아키텍처 문서가 주된 출력물이고, 이에 대한 추가적인 관련사항을 정리한 아키텍처 가이드라인이 이차적인 출력물이다.

11) 또한 지적으로 "난이도가 높은 (challenging)" 활동이다.
12) 광범위한 의미의 설계는 이 둘을 포함하지만, 좁은 의미의 설계는 합성을 의미한다.
13) Kruchten은 논문 [10]에서 설계의 성격을 이와 같이 규정한다.
14) 입력물을 필요로 하지 않는 방법 혹은 과정은 그 다지 흥미롭지 못하다. 왜냐하면 그 결과가 항상 동일하거나, 달라질 경우에도 결과를 결정짓는데 그 방법 (또는 과정)의 이용자가 역할을 수행할 수 없기 때문이다. 시스템의 이용자는 유용한 방법 혹은 과정에서는 필요한 출력물이 나오도록 적절한 입력물을 제공하는 역할을 수행한다.

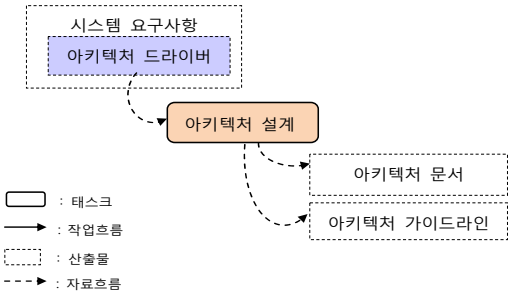

그림 10. 아키텍처 설계의 입력물과 출력물[15)]

5. 소프트웨어 아키텍처 설계의 근본 원리들

문제의 파악, 아키텍처 모델링 방법의 결정, 아키텍처 설계의 수행 그리고 설계된 아키텍처의 평가의 네 개의 단계 군으로 구성된 것으로 보고, 이들 단계군 안에서 그리고 단계군 간에 필요한 아키텍처 설계 원리들에 대하여 알아본다. 본 논문에서는 소프트웨어 아키텍처 설계에 필요한 근본적인 원리를 다음 12개로 본다:

1. 아키텍처 드라이버
2. 품질속성, 검증가능성, 품질속성시나리오
3. 문제분석
4. 컴포넌트와 커넥터
5. 아키텍처 스타일
6. 소프트웨어 아키텍처를 보는 관점체계
7. 설계의 일반원리
8. 아키텍처 설계 절차
9. 아키텍처 패턴
10. 품질속성 설계전술
11. 아키텍처의 분석
12. 아키텍처의 평가

15) 그림에서 "산출물(artifact)"이라 함은 소프트웨어 개발과정에 만들어지는 모든 결과물들을 총칭한다. "태스크(task)"는 한 명의 수행자가 수행하는 일을 지칭하며, 소프트웨어 설계 모델링의 표준적인 언어인 UML의 활동 다이어그램(activity diagram)은 더 이상 분해하지 않는 일의 단위를 태스크라 부르고 다른 태스크 혹은 활동으로 구성된 일의 단위를 활동이라고 부르고 있다. 그림의 다이어그램도 UML과 일관된 기호를 사용하고 있다.

이 12 개의 원리들이 아키텍처 설계 과정에 적용되는 단계를 그림 11은 보여준다. 그림 11은 또한 아키텍처 설계절차가 네 개의 단계군으로 구성되는 것으로 보여준다.

네 개의 단계군 가운데 첫 번째 단계군은 요구사항을 아키텍처 설계가 적용되기에 좋은 형태로 만들어 가는 원리에 대한 것이다. 두 번째 단계군은 아키텍처를 어떻게 표현할 것인가에 대한 원리로써 아키텍처를 보는 시각과 결과모델의 형태를 결정짓는다. 특히 컴포넌트와 커넥터의 이분법은, 아키텍처 스타일이나 아키텍처 관점과 같은 아키텍처의 다른 보다 구체적인 형태들의 공통적인 기반을 제시한다. 세 번째 단계군은 이 두 그룹의 원리를 적용한 결과를 바탕으로 하여 아키텍처 설계를 수행하는 원리들의 그룹이다. 이들은 일반적인 설계원리와 경험적인 지식을 이용하는 원리들의 두 가지로 구성된다. 이 세 번째 단계군은 전체 설계절차 중에서도 설계결과가 직접 결정되는 단계이므로 이 단계군을 좁은 의미의 아키텍처 설계라고 부르고, 이와 구별하기 위하여 전체설계절차를 넓은 의미의 아키텍처 설계라고 부르기로 한다. 네 번째 단계군에서는 아키텍처 평가가 수행되는데, 아키텍처 설계결과가 나오면 그에 대한 평가를 하여 아키텍처를 채택 또는 수정 결정을 하거나, 다른 대안을 찾는 결정을 내릴 수도 있게 된다.

이 절의 이하에서는 아키텍처 설계절차에 필요한 12개의 원리의 역할에 대하여 하나씩 설명한다.

5.1 아키텍처 드라이버

소프트웨어 시스템은 소프트웨어 시스템에 대한 요구사항으로부터 만들어진다. 소프트웨어 아키텍처는 소프트웨어 시스템의 아키텍처이기 때문에 아키텍처가 시스템에 대한 요구사항으로부터 설계된다는 것은 당연한 사실이다. 그러나 시스템에 대한 모든 요구사항이 아키텍처 설계를 위하여

필요하지는 않다. 시스템에 대한 요구사항들은 아키텍처에 영향을 주는 요구사항들과 그렇지 않은 요구사항들로 나뉘어 진다. 전자의 종류의 요구사항을 아키텍처 드라이버라고 부르는데, 따라서 아키텍처 설계에 더 잘 집중 할 수 있기 위하여는, 아키텍처 드라이버를 먼저 잘 가려낸 후 아키텍처 설계에 효과적으로 반영하여야 한다.

5.2 품질속성, 검증가능성, 품질속성 시나리오

아키텍처의 드라이버 중에서도 어떠한 결정요인들은 쉽게 아키텍처에 반영이 되는데 비하여, 어떤 결정요인들은 간단히 아키텍처에 반영될 수 없는 요인들이 있다. 예를 들어, '시스템이 계층적 구조를 가져야 한다'라는 제약사항이 있으면, 이 요구사항은 시스템의 아키텍처가 계층적 구조를 갖도록 함으

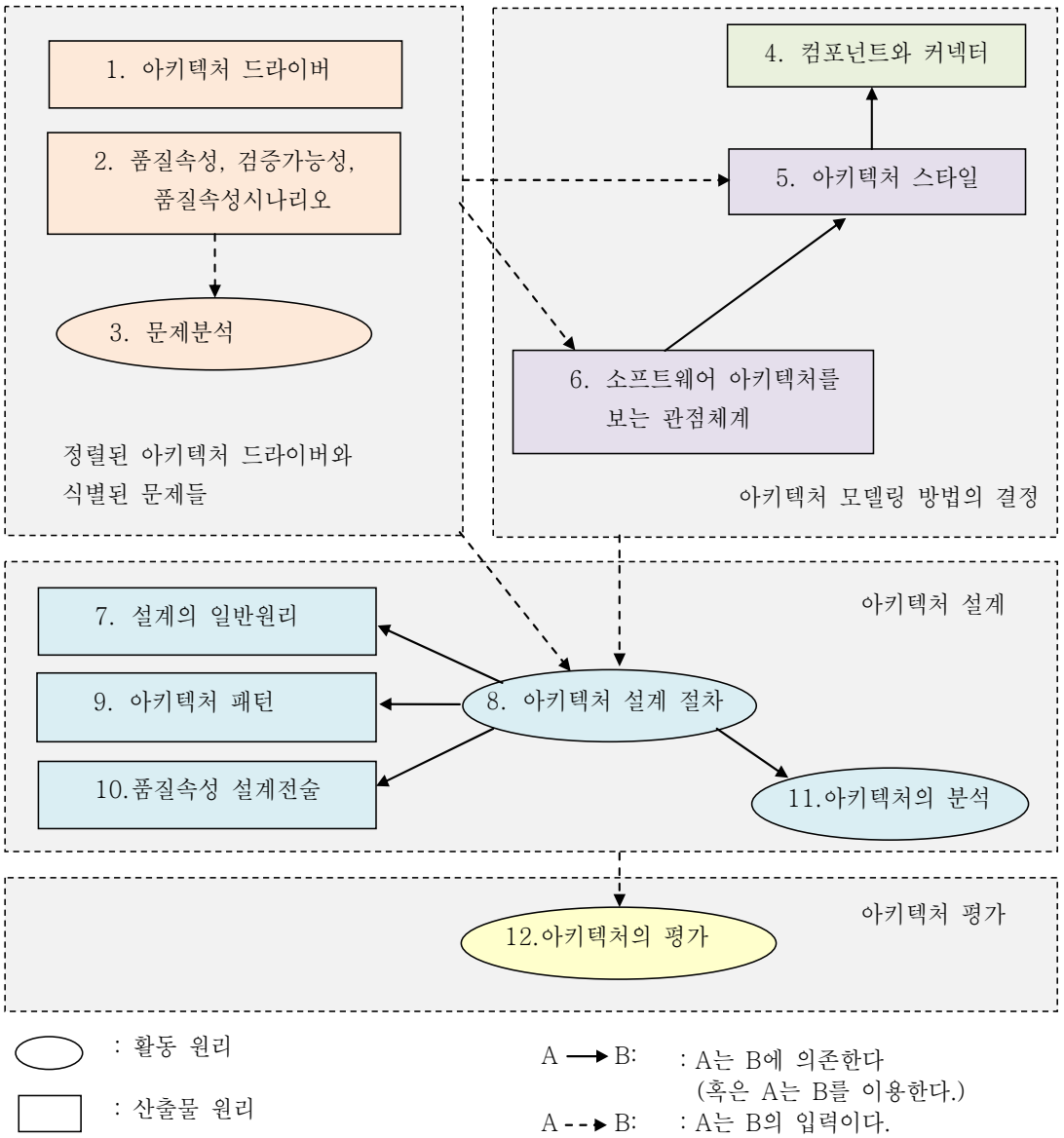

그림 11. 아키텍처 설계절차와 설계에 적용되는 원리[16)]

로써 충족된다. 그러나, 예를 들어, "높은 성능을 가져야 한다"라는 요구사항을 아키텍처 설계에 반영하기가 간단하지 않다. 이러한 요구사항은 일견 문제가 없어 보일 수 있으나, 잠시 생각해 보면 도대체 성능이 무엇을 말하는 것인지, 어느 정도의 성능을 충분히 높은 성능으로 볼 수 있는지 전혀 구체적이지 않다. 이러한 모호성은 당장 아키텍처 설계를 어떻게 해야 되는지에 대한 문제를 남길 뿐 아니라, 최선의 노력을 기울여 개발한 후에도 개발된 시스템에 대하여 고객이 높은 성능을 가진 것으로 판단할지 그렇지 않을 지를 전혀 알 수 없는 문제점을 갖고 있다. 따라서 아키텍처 드라이버들을 먼저 검증할 수 있는 형태로 먼저 바꾸어 놓은 것이 필요하다. 특히 품질속성[17]이라고 불리는 일군의 요구사항들은 아키텍처에 영향을 미치면서, 고객의 도움 없이는 개발자 단독으로 검증 가능한 형태로 바꾸기에 어려운 종류의 요구사항들이다.

검증 가능한 형태의 품질속성은 시나리오의 형태를 갖게 된다. 따라서, 아키텍처 설계에 들어가기에 앞서 품질속성을 검증 가능한 품질속성 시나리오 형태로 바꾸어 주어야 한다.

5.3 문제분석

좋은 아키텍처 설계 절차와 기법들을 우리가 잘 알고 있다고 하여도, 요구사항들이 제기하는 문제를 잘 이해하지 못한다면 올바른 아키텍처 설계가 나올 수 없다. 따라서 주어진 요구사항들이 어떤 문제의 해결을 요청하는 것인지 잘 이해하는 것이 필요하고, 가능한 한 알려진 문제의 형식으로 식별해 낼 수 있다면, 우리가 알고 있는 해법을 적용할 수 있게 됨으로써 문제 해결이 용이해 진다.

5.4 컴포넌트와 커넥터

소프트웨어 아키텍처를 사람들이 이해할 수 있는 형태로 나타내는 것을 소프트웨어 아키텍처 모델(model 혹은 representation)이라고 한다. 소프트웨어 아키텍처는 어떤 모습으로 모델링 되어야 하는 가? 소프트웨어는 궁극적으로 컴퓨팅 자원(data processing, data storage, data transfer)을 활용하여 이용자에게 필요한 서비스를 제공하도록 하는 '컴퓨팅자원의 제어 논리'로 불 수 있다. 그렇다면 컴퓨팅자원의 제어논리인 소프트웨어를 어떻게 나타내어야 하는가? 소프트웨어 아키텍처의 연구자들은 보통 컴포넌트와 커넥터로 표현해야 한다고 말한다. 여기서 컴포넌트는 처리(processing) 혹은 저장(storage) 혹은 전달(transfer)을 수행하는 소프트웨어 시스템의 구성 요소를 말하며, 커넥터는 컴포넌트를 연결하는 연결자(connection element)를 지칭한다.

커다란 컴포넌트의 경우 다시 그 자체가 컴포넌트와 커넥터로 구성될 수 있다. 커넥터는 컴퓨터 메모리의 어느 주소로 이동(jump)하라는 간단한 실행주소변경(transfer)에서부터 시작하여 응용 소프트웨어들을 연결하는 미들웨어에 이르기까지 매우 다양한 종류와 규모가 있다. 한편 미들웨어와 같은 커다란 커넥터는 그 자체가 더 작은 컴포넌트와 커넥터로 이루어진 하나의 컴포넌트이기도 하다. 이와 같이 컴포넌트와 커넥터의 양면성을 가진 가지 대상을 이 논문에서는 특히 커넥션 컴포넌트라고 부르기로 한다.

컴포넌트와 커넥터의 구별은 아키텍처의 표현 형태를 결정짓는 가장 근본적인 시각이며 이는 이어져서 설명되는 아키텍처 스타일과 아키텍처 관점의 적절한 결정에 앞서는 아키텍처 모델링의 기본적인 틀을 제공한다.

---

16) Brooks는 그의 최근 저서[2]의 제 8장 에서 설계의 접근방법을 합리적 방법과 경험적 방법으로 나누었다. 경험과 통찰에 의한 설계가 설계전문가들의 방법이고, Brooks는 설계자들은 "시각적이고 공간적 경향을 가진"오른쪽 뇌가 발달한 사람들이라고 말한다. 이 논문에서 제시하는 설계절차는 합리적 방법과 "시각적이고 공간적인" 방법이 서로 배타적이지 않음을 보여준다.

17) 대표적인 품질 속성으로 성능(performance), 보안성(security), 신뢰성(reliability), 확장성(extendibility), 사용자편의성(usability) 등이 있다.

5.5 아키텍처 스타일

아키텍처 스타일은 아키텍처의 유형을 말한다. 음악에도 재즈, 소울, 락 등의 다양한 “스타일”의 존재하는 것처럼, 아키텍처도 다양한 스타일이 있다는 시각이다. 그리고 적절한 아키텍처 스타일의 선택은 대상시스템의 개발을 효율적이고 효과적으로 만들 수 있다는 시각이다.

같은 스타일의 노래들이 서로 다르면서도 많은 공통점을 가진 것처럼, 같은 아키텍처 스타일을 갖는 시스템들 간에는 많은 공통점이 존재한다. 예를 들어, 파입(pipe)[18]과 필터(filter)라는 두 개의 서로 다른 성격의 컴포넌트로 구성된 다양한 시스템들이 존재할 수 있는데, 이러한 시스템들은 “파입-필터 스타일“을 가졌다고 말할 수 있다.

여기서 주목하여야 하는 것은, 아키텍처 스타일은 그 자체로서 주어진 요구사항을 충족시키는 해법을 제시하지는 않는다는 것이다. 단지 그러한 종류의 요구사항을 충족시키기 위하여는 이런 스타일의 아키텍처가 효과적일 것으로 예견하기 때문에, 그 스타일을 선택하게 되는 것이다. 구체적인 시스템에 도달하기 위하여는 스타일이 주는 장점들을 잘 활용할 필요가 있다. 그러나 시스템의 구체적인 아키텍처를 설계하기 위하여는, 아키텍처 스타일의 결정에 이어 설계의 일반원리나 아키텍처 패턴 혹은 품질속성 설계전술 등의 설계기법 원리들도 적절히 활용하여야 한다. 본 논문에서는 아키텍처 스타일이 직접적으로 설계결과를 주지 않고 간접적으로 효과적인 설계를 지원하기 때문에, 아키텍처 스타일을 설계 단계군에 속하는 설계 기법 원리의 하나로 보지 않고, 아키텍처 모델링 방법의 결정 단계군에 속하는 원리로 본다.

18) 파입은 구성상 필터와 필터 사이에 오기 때문에 이 둘을 연결하는 커넥션 컴포넌트다.

5.6 관점체계

그러면 이러한 입력물로부터 진행되는 설계활동의 궁극적 결과물은 어떻게 표현되어야 (혹은 나타내야) 하는가? 소프트웨어 아키텍처 연구를 통하여 사람들은 소프트웨어 아키텍처는 그 자체로서 우리가 볼 수 있는 혹은 ‘알 수 있는’ 대상이 아니라, 오직 우리가 특정한 ‘관점’으로 볼 때에만, 그 관점의 빛이 투영된 그림자로서 볼 수 있게 된다는 것을 발견하였다. 또한 하나의 관점만 존재하는 것이 아니라 여러 가지 관점이 존재하고 적절한 여러 개의 관점을 동원할 때 아키텍처를 더 잘 포착할 수 있다는 것을 발견하였다. 그렇다면 소프트웨어 아키텍처를 어떠한 관점들로 볼 수 있고, 어떠한 관점들의 집합이 개발 대상 시스템에 대한 적절한 관점들의 집합을 이루게 되는가 하는 문제가 대두된다.

여기서 한가지우리가 주목해야 하는 점은 아키텍처 스타일에 따다 적합한 관점체계가 서로 다를 수 있다는 것이다. 예를 들어, J2EE 기반의 응용, BPMS 기반의 SOA 응용, 내장소프트웨어, Web 응용 등의 아키텍처 스타일에 적절한 관점체계가 모두 동일하지는 않다.

5.7 설계의 일반원리

소프트웨어 아키텍처 설계도 다른 설계와 마찬가지로 적용되는 보편적인 원칙이 있다. 여기서 말하는 설계는, 아키텍처 드라이버부터 아키텍처 평가까지의 아키텍처 설계의 전체 과정을 말하는 것이 아니라, 정렬된 아키텍처 드라이버들로부터 설계선택들의 집합으로 옮아가는 단계를 말한다. 이 단계를 위하여 적용할 수 있는 많은 원리들이 존재하다. 그러나, 그 중에서도 많은 상황에 적용할 수 있고, 대부분의 문제의 해결에 단서가 되는 설계 원리는 무엇인가? 이 방법들은 우리가 의식하건 의식하지 않건 오래 세월을 두고 인간이 생활과

학문에서 생기는 문제들을 해결하는데 적용하여 왔고, 따라서 종종 너무나 당연한 것으로 받아들이기 쉬운 그러한 원리들이다. 이미 제 3절에서 이 들을 그림 11의 설계 절차의 큰 틀을 형성하는 원리로서 소개하였지만, 이 원리는 다시 좁은 의미의 설계를 수행하는 데에도 작용하여야 한다.

5.8 아키텍처 설계절차

아키텍처 설계 절차는, 아키텍처를 결정짓는 요구사항들로부터 먼저 어떤 아키텍처적인 설계의 선택들이 있는가를 판단함으로써 시작된다. 하나의 요구사항에 대하여 여러 가지 설계의 선택이 있을 수도 있고, 여러 개의 요구사항들에 대하여 여러 개의 설계선택들이 존재할 수 있다. 아키텍처 드라이버들을 충족시킬 수 있도록 어떻게 설계선택들을 잘 찾아가고 이들로부터 어떤 판단 하에 구체적인 선택으로 선택의 범위를 압축하여 가느냐 하는 것이 합성과 분석의 기술이다. 다양한 선택들을 찾는 것은 창조적인 과정이고, 그들 중에 장단점을 따져서 좋은 선택으로 좁혀가는 과정이 아키텍처 분석이다.

소프트웨어 아키텍처 설계는 일반적인 원리만이 아니라, 경험적 지식을 활용함으로써 설계의 효율을 높일 수 있다. 이미 검증된 아키텍처 설계를 새로이 발명할 필요는 없다. 경험적 지식은 흔히 많은 사람들에 의하여 오랜 기간 동안 확인된 좋은 해법이다. 그러한 처방(prescription)에는 크게 아키텍처 패턴(혹은 패턴)과 품질속성 공략전술 (혹은 품질속성전술)이 있다.

5.9 아키텍처 패턴

아키텍처 패턴은 반복적으로 발생하는 문제에 대한 미리 만들어진 솔루션으로, 아키텍처 스타일이 구체적인 요구사항의 해결을 제공해 주지 않는데 비하여 패턴은 시스템 전체 혹은 서브시스템에 대한 아키텍처 해법을 제시한다. 따라서, 흔히 패턴에 대한 체계적이고 깊은 이해는 훌륭한 아키텍트의 필수 조건으로 간주되기도 한다.[19]

패턴은 특정한 문제를 해결하는 준비된 처방이며, 품질 속성 설계 전술과 대조적으로 특정 품질속성에 한정되지 않는다. 따라서 아키텍처 설계자들은 설계의 해법을 패턴에서 먼저 찾고, 적절한 해답이 없는 경우 품질속성 설계전술을 사용하게 된다.

5.10 품질속성 설계전술

품질속성 설계 전술(tactic)은 단일 품질속성응답을 제어하는데 영향력 있는 설계결정이다. 시스템이 보통 여러 가지 품질 속성을 동시에 충족시킬 것을 요구하므로, 아키텍처 설계는 보통 여러 개의 요구사항의 충돌을 해결하여야 한다. 패턴은 문제에 대하여 일반적인 솔루션을 제공하여, 여러 가지 요구사항을 동시에 해결할 수 있는 패턴이 존재할 경우 이를 사용하면 되지만, 그렇지 않을 경우, 품질속성 설계 전술을 사용하여 문제를 해결하게 된다. 이 경우 여러 개의 힘의 충돌문제를 추가적으로 해결해야 한다.

5.11 분석

설계의 창조적 단계에 의하여 제시된 주어진 아키텍처가 적절한지 판단하기 위하여는 분석이 필요하다. 예를 들어, 이 아키텍처가 가져다 줄 처리량은 얼마인가와 같은 분석도 가능하다. 분석은 어떤 관점을 가지고 보느냐에 따라, 그에 따른 분석결과가 도출된다. 특별히 제시되는 분석관점을 제외하고 일반적으로 중요한 분석관점으로 위험요인과 민감점(sensitivity point), 상충점(tradeoff point) 등이 있다.

---

19) 물론 이 논문에서는 추가적으로 11가지의 원리를 더 알고 있어야 훌륭한 소프트웨어 아키텍트라고 보는 입장을 취하고 있다.

5.12 평가

아키텍처 분석이 아키텍처에 대하여 주어진 분석관점에서 분석결과를 도출하는 활동인데 반하여, 아키텍처 평가는 아키텍처 분석 또는 다른 방법을 통하여 얻어진 데이터를 종합하여 아키텍처에 대한 중요한 판단을 내리게 하는 활동이라는 점에서 서로 다르다.

문제의 해결방법에는 여러 가지가 존재할 수 있고, 각각의 해법마다 그 장단점이 있기 때문에 아키텍처 평가가 필요할 수 있다. 주어진 아키텍처에 대한 분석은 설계활동 안에서 더 나은 대안들을 찾아가는 과정이며, 평가는 완결된 설계를 대안과 비교함으로써 그 결과에 따라 설계라는 큰 과정을 다시 수행할 것을 요구할 수도 있는 활동이다.

6. 결론

본 논문에서는 효과적인 아키텍처 설계를 위하여 아키텍처 설계 절차를 네 개의 단계군으로 구성된 것으로 보고 아키텍처 설계에 필요한 12개의 근본적인 원리들을 식별하여 개략적으로 설명하였다. 각각의 원리에 대한 보다 상세한 설명과 본 논문에서 제시된 절차와 원리들이 적용되어 효과적으로 사용되는 사례는 다른 기회에 소개할 계획이다.

참고문헌

[1] Bass, L., Clements, P., Kazman, R., Software *Architecture in Practice, 2nd ed.,* Addison-Wesley, 2003.

[2] Frederick R. Brooks, Jr., *The Design of Design, Addison*-Wesley, 2010.

[3] Garlan, D., Perry, D., “Introduction to the Special Issue on Software Architecture,” IEEE Transactions on Software Eng., April 1995.

[4] Institute of Electrical and Electronics Engineers, *Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE Std 1471-2000)*, New York, NY: Institute of Electrical and Electronics Engineers, 2000.

[5] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[6] Kruchten, P., *The Rational Unified Process : AnIntroduction, 2nd Ed.*, Boston, MA: Addison-Wesley, 2001.

[7] Kruchten, P., Obbink, H., Stafford, J., “The Past, Present, and Future for Software Architecture,” IEEE Software, Volume 23, Issue 2, March 2006.

[8] Shaw, M., “Larger Scale Systems Require Higher-Level Abstractions,” Proc. Fifth Int’l Workshop on Software Specification and Design, pp 143 ~ 146, IEEE Computer Society, ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 3, May 1989.

[9] Mark Weiser, “The Computer for the Twenty-First Century,” Scientific American, September 1991

저 자 소 개

강 성 원

1982년 서울대학교 사회과학대학(정치학사)
1989년 Univ. of Iowa 전산학(전산학석사)
1992년 Univ. of Iowa 전산학(전산학박사)
1993년~2001년 한국통신 연구개발본부 선임연구원
1995년~1996년 미국 국립표준기술연구소 (NIST)
객원연구원
2001년~2005년 한국정보통신대학교 조교수
2003년~현재 미국 Carnegie-Mellon University
소프트웨어공학석사과정 겸임교수
2005년~2009년 한국정보통신대학교 부교수
2009년~현재 KAIST 부교수

〈관심분야〉 소프트웨어 아키텍처, 소프트웨어 프로덕트 라인, 엔터프라이즈 아키텍처, 소프트웨어 시험, 형식기법