소프트웨어 아키텍처 연구

Version: 1.0

Part 1/5

소프트웨어 아키텍처 개념에 대한 이해

작성 : 신현묵 (zetlos@naver.com, supims@gmail.com)

작성일 : 2007년 5월 20일

목 차

I 시작 배경

이 문서에는 소프트웨어 아키텍쳐에 대한 전반적인 소개자료와 기초개념에 대해서 설명하고 있다.

소프트웨어란 무엇인가에 대해서 다시한번 생각해보자.

여러가지 설명과 이해가 다 다르겠지만, 보편적인 부분만 설명한다면.

소프트웨어란 ' **사용자가 원하는 서비스를 제공하기 위하여 정보기술을 활용하여 정보서비스를 구축하는 것**'이라고 규정지을 수 있다. 이는, 정보통신 기술이 사용되는 수많은 분야중에 비즈니스나 공공서비스를 중점적으로 설명한다면 다음과 같은 사항들을 체크하게 한다.

기업의 변화와 사회의 흐름에 대한 구상이나 반영이 필요하다 볼 수 있다.

또한, 소프트웨어 개발의 변화가 과거의 1인 중심인 프로그래머가 혼자 모든 것을 다하던 시대에서 설계자와 코더가 나뉘어지고, 다시, 시대의 흐름이나 서비스중심의 구조적인 설계를 하는 아키텍트의 시대에 까지 시간이 많이 지나고 있다.

여기서는 소프트웨어 아키텍처의 중요성이 부각된 이유를 설명하고 소프트웨어 아키텍처를 도입함으로써 어떤 목표가 달성될 것인가에 대해서 알아본다.

소프트웨어 아키텍쳐란 다음의 두가지가 가장 중요하다.

하나.중요한 부분만 다루어야 한다.

둘. 아키텍트의 역할이 중요하다.

아키텍트는 소프트웨어를 개발할 때에 중요한 재료들을 잘 다루어야 한다. 그것은 비용, 적용기술, 조직의 역량, 기능, 목표, 시스템의 복잡도에 대해서 아키텍트의 중요한 판단을 통하여 시스템을 구상하고 추상화하는 방법이다.

이것이 바로 소프트웨어아키텍쳐이다.

그리고, EA에 대한 개념을 잠깐 집고 넘어가자.

어느날 사장님이 아침에 CEO 조찬모임에 나갔다.
다른 사장이 물어본다.
“어이 X사장.. 요즘 회사 어때? 어제 미국증시와 유가변동에 따른 국내 제조업의 변화동향에 대해서 어떻게 생각해?”

사장님 순간 당황한다... 그런 보고를 받은 적 없기 떄문에.. ㅡ.ㅡ;
(도대체 회사내의 정보전달체계가 마음에 안들어!)

... (그 다음날)
CEO는 CIO를 불러서 한소리 한다. 도대체 정보 보고가 어떻게 되었냐고!!
쪼인트 맞은 CIO.. 그런데, 이런 정보 시스템이 어떻게 전달되는가? CIO체크해보니, 자신의 부서에 수 많은 시스템들이 도입되어 있고 소프트웨어 아키텍쳐, 하드웨어 아키텍쳐등의 문서는 엄청많으나 중복되어있는 하드웨어와 소프트웨어가 무수히 많이 보인다.
(머리나쁜 CIO 순간 열받는다..

CIO열받는다.. 부장의 조인트를 까면서 한소리한다.. ‘큰 그림 그려와!’
부장은 과장, 과장은 대리.. (쿨럭~)
결국은 X대리 이것에 대한 해결책과 해결방안을 제시해주기 위해서 EA를 그린다.

EA는 회사의 청사진이라고 보면 된다. 우리 회사는 어떤 비즈니스가 존재하는가? 이 비즈니스를 위해서 어떤 데이터가 존재하는가? 또 이 데이터를 운용하기 위한 어플리케이션은 무엇인가? 이 어플리케이션을 지원하기 위한 정보기술은 또 무엇인가?

비즈니스와 기술은 어떤게 먼저이다 볼 수 없다. 닭과 달걀의 싸움이다. 결군, 유기적인 연관관계일뿐.

업무,비즈니스를 정의한 것은 Business Architecture, 데이터를 위한 Data Architecture, 어플리케이션을 위한 Application Architecture, 정보기술을 위한 Techinical Architecture, 소위 BA, DA, AA, RA... 다, 경영적인 마인드의 산출물들이다.

자! 그렇다면.. 현실은 어떠한가?

1 프로젝트의 현실

현재 일반적으로 SI이든 소규모개발사이던, 대형개발사이든 다음과 같은 문제들이 존재할 수 있다. 다만, 이러한 문제들이 도출되지 않는 조직에 계시다면 정말 행복한 사람이다.

이곳에서는 소규모 프로젝트라기 보다는 대규모 프로젝트인 경우에 한정을 지어서 생각해보도록 하겠다. 각각의 사례들은 그 규모에 따라 그 형태를 달리하는 경우가 많다.

프로젝트 A : EJB를 기반으로한 EAI 제품 개발, 개발기간 3년, 투입인원 팀 4개팀 26명

프로젝트 B : SyncML관련 모바일 솔루션 개발, 개발기간 6개월 투입인원 5명

프로젝트 C : 온라인 게임개발, 개발기간 2년 투입인원 12명 (디자이너 제외)

프로젝트 D : 공공기반 프로젝트, 개발기간 3년 투입인원 연인원 60명, 최대 130명

프로젝트의 성공여부는 어디에 달려있는것일까? 누가 성공여부를 결정하는가? 과연 프로젝트는 어떻게 진행되는 것이 맞는가?

과연 방법론의 사용여부는 어디까지이며? 산출물의 작성기준은 어디까지인가?

소프트웨어아키텍트는 이러한 것들을 커다란 개념으로 볼 수 있도록 한다.

예를 들어본 프로젝트들을 살펴보면 프로젝트 A와 B는 패키지 개발이고 실제 소프트웨어의 성향이나 목표에 대한 부분을 결정하는 것은 최고 경영자층과 CTO레벨에서 결정하는 내용이므로 오늘 설명할 내용에서 약간 벗어난다고 할 수 있다.

그리고, 프로젝트 C의 경우에는 일반 고객들을 상대로 하는 프로젝트이기 때문에 해당 프로젝트도 일단, 이 범위에서는 제외한다. 프로젝트 A,B,C에 대한 경험으로 별도의 각각 예제를 따로 준비중이고 아마도, 6회쯤에 해당 내용을 발표할 듯.

일단, 프로젝트 D를 기준으로 한다. 사용자들이 정해져 있고 보통 SI라고 불리우는 사업인 경우이므로 이에 대해서 기술한다.

프로젝트 D의 특징을 기술하면 다음과 같다.

하나.프레임워크나 솔루션이 투입되어 일정한 공수로 개발자들이 투입되어 개발한다.

둘. 기존에 정보시스템이 있거나 대체, 혹은 신규 개발하는 형태이다.

셋. 고객이나 사용자들이 일반사용자들에 가까우므로 IT관련 정보기술에 어둡다.

넷. 고급개발자들을 극소수로 밖에 사용할 수 없으며, 초급개발자들을 주로 사용하게 된다.

다섯.기간이나 비용등이 영업의 힘으로 결정되는 경우가 다반사이며, 생각보다 제약사항들이 많다.

여섯.고객의 요구조건을 어떻게 관리하고 품질의 영역이 불명확한 경우가 많다.

이러한 프로젝트들을 기준으로 세부적인 내용을 기술한다. 이 강의에서 이야기하는 '프로젝트'는 이러한 부분들을 중심으로 기술한다.

1.1 프로젝트의 일반적인 경향

첫째, 기술적인 결정을 내릴 때 비합리적인 결정을 내리는 경우가 많다.

대부분 기술적인 결정은 경험에 의해 이루어지는 경우가 많다. 그러나 경험적인 해결책이 현재 진행 중인 프로젝트에서 부적절한 경우 또한 대다수인 경우가 많다. 실제로 경험적인 결정만으로는 프로젝트에서 적용할 기술에 어떤 위험이 따르는지 판단하기 어렵다.

하지만, 보통 프로젝트의 경우 경험이 많은 개발자들을 중심으로 PM과 PL이 구성되어 지고 이들중에 PL들선에서 이러한 문제들을 해결하기 위한 미팅이나 회의가 많이 발생한다.

또한 기술적인 결정을 내릴 때 비용과 프로젝트 적합성을 따지지 않고 도입하는 경우가 많다. 이 경우 프로젝트의 기간을 줄이고 품질을 높이기 위해 도입된 기술이 오히려 프로젝트를 지연시키고 시스템의 품질을 떨어뜨린다.

기술적인 의사 결정이 프로젝트 중간에 흔들려서 프로젝트가 지연되는 경우가 종종 발생한다. 이것은 선택된 기술이 프로젝트에서 최적의 선택인지 고민하지 않았기 때문이다.

사용자의 요구사항에 기초하지 않은 의사 결정은 막연한 결과를 만들고 기술적인 의사 결정이 올바른 것인지 프로젝트 후에도 판단하기 어렵다.

의사 결정을 하기 위해 많은 시간을 소요하지만 합리적인 결정을 내리지 못할 때가 많다.

실제 영업적인 이유로 인하여 이와 같은 경우가 발생하는 경우도 다반사이다.

둘째, 새로운 프로젝트에서 과거의 경험을 답습하는 경우가 많다.

프로젝트는 전혀 새로운 환경 일때가 많고 프로젝트의 성격에 따라 최적의 선택을 해야 한다. 기술을 어떻게 선택할 것인지에 대한 정형화된 프로세스가 없으며 기술은 계속 변한다. 따라서 과거의 프로젝트 성공에 대한 경험으로 현재의 문제를 해결할 수 없다. 이점을 간과하는 경우가 많다.

따라서 많은 기술을 조합하여 현재 프로젝트를 위한 최적의 해결책을 찾아야 한다. 이 해결책은 사용자의 요구사항에서 논리적으로 도출된 것이어야 하며, 설명가능한 방식이어야 한다.

셋째, 과거의 프로젝트에서 배우지 못한다.

이것은 두번째 경우와 정반대의 경우이다. 과거의 프로젝트의 성공과 실패에서 원인 분석을 하지 못하면 성공한 이유도 실패한 이유도 알 수 없다.

프로젝트에서 대부분의 문제는 고객과 개발팀간의 커뮤니케이션 부재 때문에 발생한다. 또한 프로젝트의 범위 관리도 문제가 되며, 사용자의 요구사항을 어떻게 도출하고 관리할 것인가도 문제가 된다.

기술적으로는 시스템의 복잡성이 문제가 된다. 시스템의 복잡성은 개발자의 능력의 한계를 뛰어넘는 것으로 시스템을 복잡성을 줄이고 효율적으로 시스템을 구축하는 방안을 찾아야 한다. 기술적인 복잡성을 줄이고 합리적인 의사결정을 내리기 위해서는 기술적인 의사 결정을 초기에 내리고 적합한지 계속 테스트해야 한다.

또한 초기에 어떻게 프로젝트 조직을 구성할 것인가도 프로젝트 성공에 많은 영향을 끼친다. 역할 분담이 명확하지 않아서 책임 소재를 따지는데 많은 문제가 발생한다. 조직 구성도 프로젝트의 성격과 기술 Set에 따라 달라진다. 프로젝트 팀원들이 도메인을 잘 알고 사용하는 기술에 익숙한 경우와 프로젝트 팀원들이 도메인을 모르고 사용하는 기술도 모를 경우는 조직 구성이 달라져야 한다.

대부분 설계자가 요구사항을 정리하는 것 외에는 제 역할을 하지 못한다. 설계자가 어떤 일을 해야 하며 개발자에게는 무엇을 주어야 하는 지도 결정해야 한다.

1.2 국내 프로젝트의 경향

첫째, 신기술을 도입하여 무조건 쓰는 경우가 많다. J2EE가 프로젝트의 성격에 맞는지, CBD가 프로젝트의 성격에 맞는지 고민하여 프로젝트를 위한 최적의 선택을 한 경우가 많지 않으며 신기술에 대한 막연한 기대를 가지고 프로젝트를 시작한다.

둘째, 프로젝트의 복잡성이 증가되고 관리되지 않으면 어느 순간 통제할 수 상황에 빠진다. 이 경우 개발자

들이 자체적으로 communication을 통해 해결하는 경우가 많다.

셋째, 프로젝트 초기에 드러나지 않았던 문제들이 프로젝트 마무리 시점에서 드러나는 경우가 많다. 많은 오류가 프로젝트 마무리 시점에서 발생한다.

넷째, 설계자가 제 역할을 못하는 경우가 많음. 단순히 사용자의 요구사항을 정리하여 개발자에게 전달하는 역할만을 함.

다섯째, CASE 툴의 사용이 개발 생산성 향상에 큰 도움이 되지 않음. 개발자들이 시스템을 구축하기 위해 미리 준비되어야 할 많은 사항들을 개발자 스스로 만들어 가는 경우가 많다. 시스템 환경 셋팅, 테스트, 각종 툴 사용 기법, utility 공통 코드, 코딩 표준 등을 개발자 스스로 결정한다.

여섯째, 몇몇 능력 있는 개발자들의 도움을 받아 프로젝트가 진행된다. (결과적으로 프로젝트에서 누군가는 아키텍트의 역할을 함) 그러나 능력 있는 개발자가 자신의 능력을 발휘하는 시점은 프로젝트의 문제가 발생하는 시점에서 이루어지고 개발자의 능력만으로 해결할 수 없을 정도로 이미 문제가 심각해진 경우가 많다. (왜 프로젝트 초기 의사 결정에 이런 사람들을 참여시키지 않는가?) 프로젝트 팀원들이 프로젝트 관리자에게 의사결정을 요구하지만 기술적인 문제는 프로젝트 관리자가 의사결정을 내릴 수 없으며 몇몇 사람들의 의견을 구하거나 의사결정이 이루어지지 않은 채 프로젝트가 진행되는 경우도 있다.

일곱째, 요구사항에 대한 관리가 이루어지지 않는다. 요구사항이 시스템에 얼마 정도의 영향을 미칠 것인지, 아키텍처에는 얼마 정도의 영향을 미칠 것인지 판단하지 못함.

그렇다면 위와 같은 프로젝트의 경우에는 어떻할것인가?

프로젝트를 포기하는 것도 하나의 방법일 수 있다. 혹은, 제약사항을 머릿속에 두고 그와 같은 아키텍쳐를 그려낼 수 있는 방법을 도출하는 것도 방법이다.

어떤 프로젝트이던 최고의 방법은 없다. 최선의 방법만 존재할 뿐. 실제 프로젝트에서는 최고의 비용, 최고의 기간, 최고의 프로그래머와 일하는 것이 아니기 때문이다.

이러한 것들을 해결하기 위하여 무수히 많은 것들이 존재하지만, 이곳에서는 이러한 문제들의 구조를 파악하고 소프트웨어를 제작하기 위한 소프트웨어 아키텍쳐에 대해서 서술해보자.

그리고. 아주 보통 간과하고 넘어가는 경우가 많은데. 다음의 예를 하나 살펴보자.

사용자들이 요구하는 아주 비기능적인 요소들이 있다.

'동일 조회업무 화면을 여러 개 보이게 해주세요'

자.. 이러한 요구조건을 어떻게 해결해야 하는가? 생각보다 복잡하고 미묘한 문제들이 걸려있다. 어떤 부분인지 살펴보자.

일단, 어떤 사람이 개발한 업무화면이 'N'개 동작한다라는 것은 그 업무화면에서 여러 개 동작시키는 매커니즘을 활용하는 방법도 있을것이고, 해당 프레임워크를 제공할 수도 있을 것이고, 아니면, 웹과 마찬가지로 세션을 유지하지 않는 방법을 고려할 수도 있다.

사용자의 기능하나에 의하여 구조적인 모습이 전혀 엉뚱한 모습으로 변할 수 있다.

이러한 요구조건들이 품질요구사항으로 변하고 실제 소프트웨어의 아키텍쳐와 프레임워크, 솔루션의 도입, 개발자들의 가이드라인, 컴포넌트의 구성방식, 유지보수 방법등에 많은 영향을 주게된다.

이 부분을 간과하고 넘어간 경우를 살펴보자.

업무 분석을 하고 설계, 개발, 단위테스트, 통합테스트, 배포까지 무난하게 넘어갔다고 보자. 그런데, 실제 사용자가 요구한 것은 '전체 업무의 형태를 위와 같이 N개 동작하게 해달라'는 요구사항이었다고 나중에 밝혀졌다고 보자.

설계자는 요구조건에 맞는 프레임워크가 아니었다고 이야기 할 것이고, 개발자는 세션을 유지하는 C/S 프로그램으로 만들어 놨고, 공통 함수로 처리되는 방식으로 빠르게 개발할 것만 고려하고 플밍이 되었다고 보자.

과연 누구의 잘못인가?

이것이야 말로 아키텍트가 초기에 소프트웨어 아키텍쳐를 잘못설계한 케이스이다.

처음부터 이러한 상황을 고려하여 동작 가능한 구조로 만들어 주었다면, 문제가 발생하지 않았을 것이다.

개발자가 컴포넌트 개발도 할줄 모르는 공통함수개념과 화면별 단위 프로그램밖에 만들줄 모르는 개발자였다면, 해당 개발자들이 해당 요구조건을 모르고 코딩을 해도 실제 수행시에 아무 문제 없도록 시스템 메커니즘을 구현하였어야 했다.

해결방법은 아주 많은 방법들이 있다.

2 소프트웨어 아키텍처를 도입하면서 달성되는 목표

소프트웨어 아키텍처를 도입함으로써 다음과 같은 목표를 달성할 수 있도록 한다. 그 목표는 크게 다음의 세가지로 들 수 있다.

첫째, 견고하고 안정적이며 고품질의 시스템을 구축할 수 있다.

둘째, 시스템 구축 시 발생하는 문제들을 초기 단계에서 해결할 수 있다.

셋째, 아키텍트의 역할을 명확히 정의하여 20%의 고급 인력으로 80%의 프로젝트 인원을 리딩할 수 있다.

아키텍트는 비용과 목표에 대해서 가장 많은 고민을 하여야 한다. 최고의 개발자들로만 구성하여도 프로젝트는 실패할 수 있다. 튼튼하고 품질 좋고 나중에 문제 발생하지 않고, 비용이 적게든다면 최선이지 않은가?

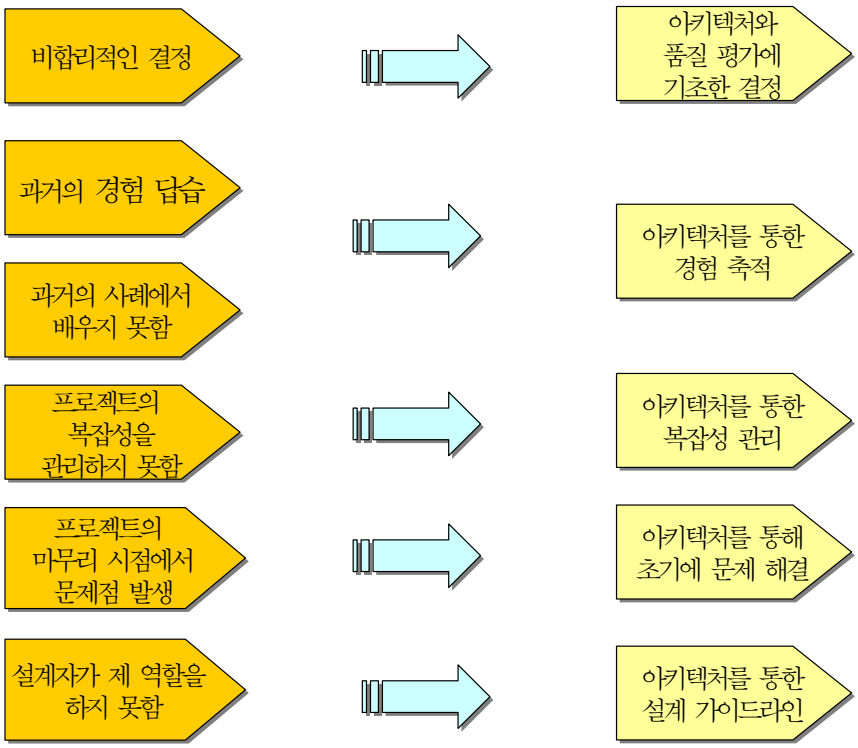


그림 I-1. 아키텍처 적용 시 장점

그림 I-1은 소프트웨어 아키텍처 도입 시 시스템 개발 프로젝트에서 어떤 도움을 받을 수 있는지를 보여준다.

첫째, 소프트웨어 아키텍처 도입 시 아키텍처의 품질을 평가하여 비합리적인 의사 결정이 합리적인 의사 결정으로 변경된다.

둘째, 과거의 경험을 답습하거나 과거의 사례에서 배우지 못했던 것에서 소프트웨어 아키텍처를 재사용함으로써 과거 프로젝트의 성공과 실패 사례를 배우는 것으로 변경된다.

셋째, 프로젝트의 복잡성을 아키텍처를 통해 관리하는 것이 가능해진다.

넷째, 프로젝트 마무리 시점에서 발생하는 문제들을 아키텍처를 도입하여 프로젝트 초기에 해결할 수 있다.

다섯째, 설계자가 단순히 요구사항을 개발자에게 전달하는 것을 아키텍처를 통해 설계자가 요구사항 분석뿐 아니라 기술적인 문제도 모델링 할 수 있게 하며 개발자는 더욱 효율적으로 시스템을 구현하는 것이 가능하다.

3 본 강좌의 목적

본 강좌의 목적은 다음과 같다.

첫째, 아키텍트가 프로젝트 시작 시점에서 투입되어 무엇을 하는지 시나리오를 작성할 수 있게 하며

둘째, 소프트웨어 아키텍처가 개발 프로젝트에 정착될 수 있도록 하고

셋째, 아키텍트의 역할에 대해서 정의를 하며

넷째, 아키텍트와 프로젝트의 구성 요소 사이의 관계에 대해서 이해를 한다.

다섯째, 소프트웨어 아키텍처가 포함된 개발 프로젝트의 프로세스는 무엇인가에 대해서 알아보고

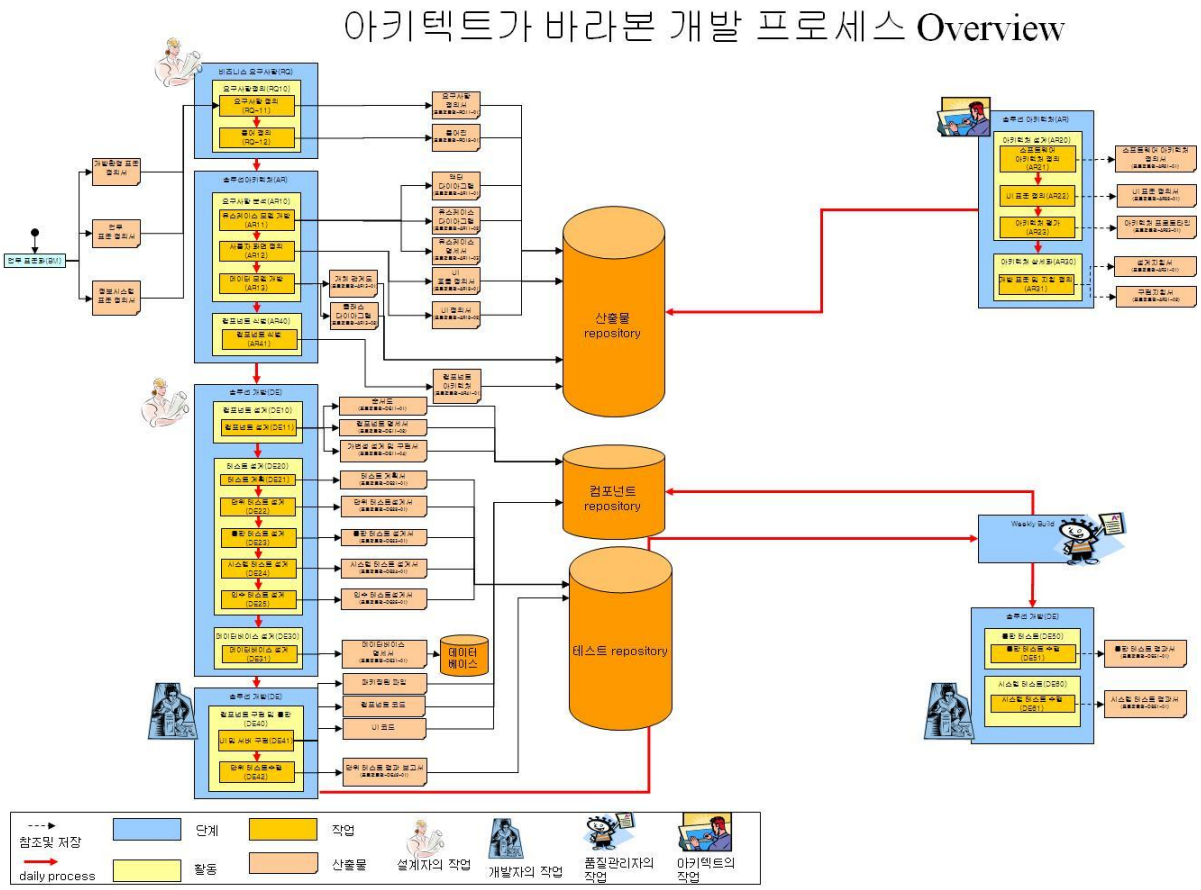
여섯째, 아키텍트가 프로젝트에서 결정해야 할 항목은 무엇인가에 대해서 고민하는 것이다.

II 소프트웨어 아키텍처 란?

본격적인 소프트웨어 아키텍쳐의 개념에 대해서 알아보도록 하자.

그렇다면, 소프트웨어 아키텍쳐란 무엇이며, 아키텍트란 누구이며, 그의 역할과, 소프트웨어 아키텍쳐를 통하여 개발 프로세스와 어떤 영향을 가지며, 소프트웨어 아키텍쳐와 다른 아키텍쳐가 어떤 관계를 가지는가에 대해서 알아봐야 한다.

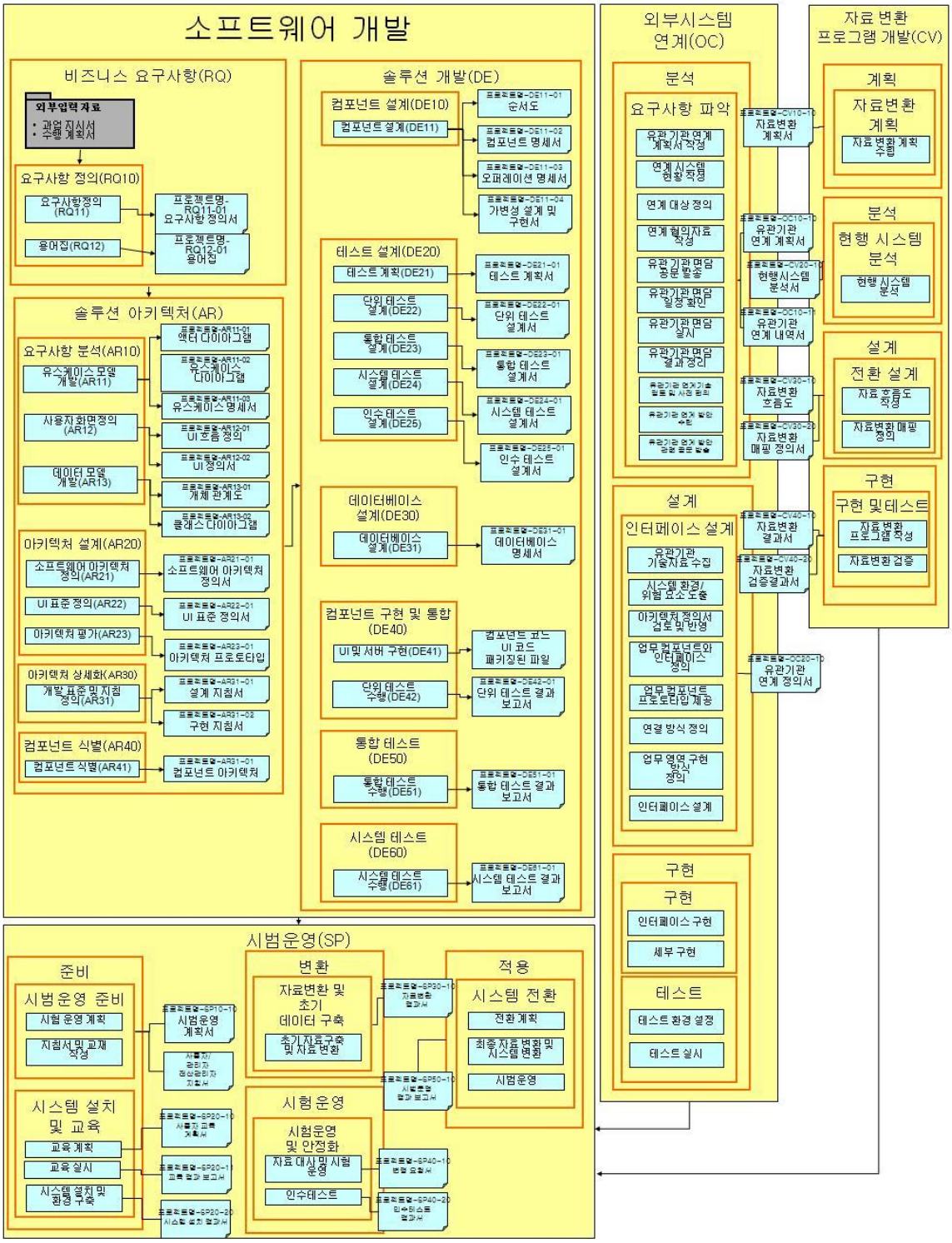
이 곳에서는 실제 CBD나 개발방법론에 대해서 서술하지는 않는다. 다음 강의 때에 CBD와 연관성이 있는 부분에 대해서 서술해볼 것이다.



이 그림은 실제 CBD로 개발되어지는 개발 프로세스에서 산출물과 개발흐름에 대해서 포괄적으로 서술되어진 그림이다.

이러한 CBD의 전체적인 산출물 구조는 다음과 같다.

CBD 산출물 흐름도



이 곳에서는 소프트웨어 아키텍쳐가 중요하므로 전반적인 프로세스에 대해서는 서술하지 않도록 한다. 다만, 참고적으로 살펴보기 바란다.

1 소프트웨어 아키텍처의 정의

가장 먼저 다른 사람들의 소프트웨어 아키텍쳐에 대한 정의에 대해서 알아보고 이 곳에서 사용할 소프트웨어 아키텍쳐의 정의에 대해서 알아보도록 한다.

아직도 소프트웨어 아키텍쳐에 대한 '정의'는 무수히 많은 곳에서 정의되고 정답자체가 아직은 모호한 분야라고 할 수 있으므로, 각자 머릿속에 그 '정의'를 선택하거나 정의할 수 있고, 그 부분들에 대해서 생각해 봐야 한다.

1.1 소프트웨어 아키텍처의 정의들

❑ 고전적인 정의

Rational Unified Process, 1999
Perry and Wolf, 1992
Garlan and Shaw, 1993
Bass, er al., 1994
Hayes-Roth, 1994
Garlan and Parry, 1995
Boehm, et al., 1995
Soni, Nord, and Hofmetster, 1995
Shaw, 1995

❑ ANSI/IEEE Std 1471-2000정의

Architecture is defined by the recommended practice as *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.*

This de.nition is intended to encompass a variety of uses of the term architecture by recognizing their underlying common elements. Principal among these is the need to understand and control those elements of system design that capture the system's utility, cost, and risk. In some cases, these elements are the physical components of the system and their relationships.

❑ Bass, Clements, Kazman의 정의

소프트웨어 아키텍처는 시스템의 구조, 시스템의 구조들에 대한 구조이다.

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
>
> The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects
>
> Bass, Len; Clements, Paul; & Kazman, Rick. *Software Architecture in Practice, Second Edition.* Boston, MA: Addison-Wesley, 2003
>
> "프로그램 또는 연산 시스템의 소프트웨어 아키텍처는 소프트웨어 구성요소와 구성요소의 외부로 보이는 속성 그리고 그 구성요소간의 관계로 이루어진 구조 또는 시스템의 구조이다. "
>
> " 그리고 아키텍처의 문서화와 관련된다. 상위단계의 디자인에 관한 초기 의사결정 사항과 프로젝트간의 패턴과 디자인의 재사용을 허용하기 위해 기록한 아키텍처의 문서화는 이해당사자간의 의사소통을 진행시킨다. "

❑ Dewayne E. Perry, Alexander L. Wolf의 정의

소프트웨어 아키텍처는 특별한 형태를 가진 아키텍처 구성 요소들의 집합이다. 아키텍처 구성 요소는 세가지로 구분된다. 프로세스 구성 요소, 데이터 구성 요소, 연결 구성 요소로 구분된다.

❑ Galarn and Perry의 정의

소프트웨어 아키텍처는 프로그램이나 시스템의 구성 요소들 사이의 구조, 관계, 설계와 시스템 업그레이드를 통제하는 지침과 원칙이다.

❑ Booch의 정의

소프트웨어 아키텍처는 소프트웨어의 구조에 대한 중요한 의사 결정의 집합이다. 이 의사 결정에는 소프트웨어 구성 요소에 대한 선택, 인터페이스에 대한 선택, 이 구성 요소들 사이의 상호작용, 더 큰 서브시스템을 구성하기 위한 구성 요소와 상호작용의 결합으로 이루어진다.

❑ Myron Ahn의 정의

소프트웨어 아키텍처는 모듈, 프로세스, 데이터, 이들의 구조, 구성 요소들 사이의 관계, 이 구성

요소와 관계들이 어떻게 확장되고 수정될 수 있는지, 사용하는 기술은 무엇 인지로 이루어져 있으며 소프트웨어 아키텍처를 통해 시스템의 유연성과 성능을 판단 할 수 있고 시스템을 어떻게 구현하고 수정할 수 있는지를 판단할 수 있다.

위의 같은 다양한 정의가 존재하므로 자신만의 정의를 만들거나 선택해보라, 실제적으로 이 개념들은 하고자하는 프로젝트에 따라 다르게 존재할 수 있다.

1.2 소프트웨어 아키텍처에 대한 정의

위의 정의들을 요약하여 소프트웨어 아키텍처를 다음과 같이 정의하여 보자..

소프트웨어 아키텍처는 시스템의 핵심 구성 요소와 구성 요소들 사이의 연결 관계로 이루지므로. 핵심 구성 요소는 시스템이 가지고 있는 모듈, 모듈 사이의 연결, 시스템의 변경, 진화하기 위한 기술적인 원칙, 모듈들 사이의 상호작용, 시스템이 동작하기 위한 기술을 모두 포함하는 것으로 정의한다.

즉 **소프트웨어 아키텍처는 구현할 시스템에 대한 top-down view이며 시스템에 대한 기술적인 명세서이며 공학적인 청사진이다.**

기본적인 개념은 MDA에서 취하고 있는 방향과 유사하다.

1.3 소프트웨어 아키텍처의 구성 요소

소프트웨어 아키텍처의 구성요소에 대해서 상세하게 살표보면 다음과 같은 구성 요소로 이루어짐을 알 수 있다.

- ❑ 시스템의 구성요소와 구성 요소 들 사이의 연결 관계
- ❑ 시스템의 설계와 진화를 통제하는 원칙과 가이드라인
- ❑ 시스템 구성 요소들의 collaboration (성과)
- ❑ 시스템이 어떻게 확장되고 수정될 것인가에 대한 결정
- ❑ 시스템의 구성 요소들이 가지고 있는 기술

다음 그림은 소프트웨어 아키텍처의 구성요소를 보여준다.

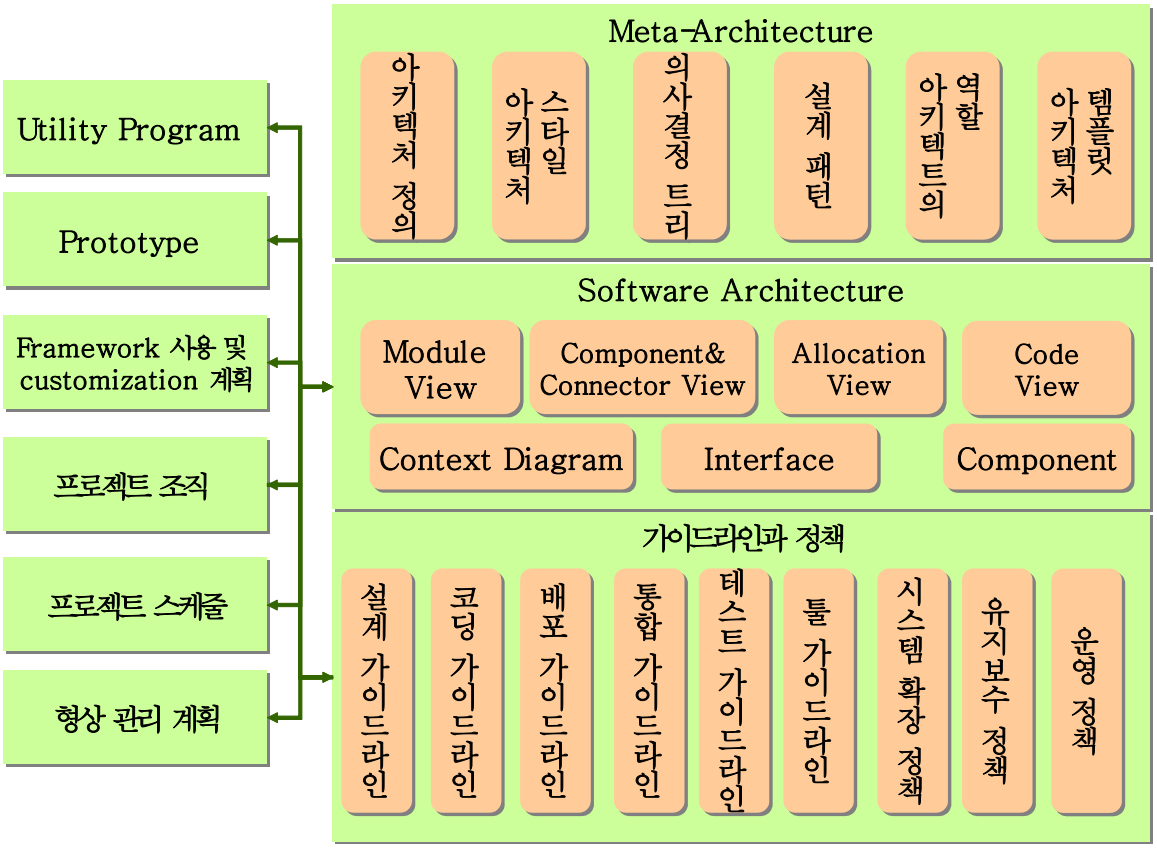


그림 II-1. 소프트웨어 아키텍처 산출물

그림 II-1를 보면 소프트웨어 아키텍처는 세가지로 구분된다.

Meta-Architecture는 아키텍처를 설계하기 위한 일반적인 지침이다. Meta-Architecture는 소프트웨어 아키텍처가 무엇인지 결정하고 아키텍처를 설계하기 위한 일반적인 의사 결정 트리와 아키텍트의 역할에 대한 정의, 아키텍처 문서에 대한 템플릿으로 구성된다.

소프트웨어 아키텍처는 네 가지 View인 module view, component&connector view, allocation view, code view로 구성된다. 또한 아키텍처 설계의 결과로 컨텍스트 다이어그램, 컴포넌트 명세, 인터페이스 명세가 도출되어야 한다.

View에 대해서는 사람마다 정의하는 방법이 다양하므로 이에 대한 논란은 나중에 하도록 하고, 먼저 그냥 서술해보자. 이 View는 사람들마다 서술하는 방법, 기술하는 방법, 다이어그램의 형태, 박스/라인/인터페이스/컴포넌트의 수준과 깊이/의미등이 다 다르다.

다만, View의 선정 기준과 기술할 때 주의점만 살펴보면된다.

건축물을 설계할때의 조감도 및 설계도, 세부 설계도, 투시도 등등의 형태와 동일하다고 보면 된다, 필요

수준까지만 각자 알아서 그리면 되는 것이다.

일반적인 view는 3가지이다. Allocation View, Component & Connector View, Module View라 이야기하지만 저는 Code view를 하나 더 구분해야 한다고 본다. 너무 상세한 설명은 나중에 거론하기로 하고 하여간 4개의 View가 있다고 보자.

이 부분은 적어도 1개 이상에 대해서 기술하고 몇 개가 적절한 가는 개발자들의 역량과 도메인에 대한 이해, 이전 프로젝트의 경험 및 주요 이슈 사항등의 요소에 의해서 좌우되어진다.

다만, 시스템의 큰 기능, 품질 속성 (가용성, 변경성, 보안성, 성능)의 판단 근거가 되어야 하며, 주요 모듈의 기능이 내/외부와 어떤 연관성을 가지는지 보여야 하고, 시스템의 규모와 범위를 파악할 수 있으면 된다.

또한, 위와 같은 아키텍쳐 기술 정의서를 구축할 때에 타당한 근거를 서술할 수 있으면 된다.

다만, 이 View를 UML 1.x영역에서는 표현하기 힘들었다. 너무 기술에 종속적인 패턴의 설계 혹은 컴포넌트의 배치 레벨에서의 작성이 주였기 때문이다. UML 2.0을 구성할때에 OMG에서 카네기 멜론 SEI의 아키텍쳐 정의를 상당부분 도입하여서 이 부분들에 대한 기술이 이젠 수월해진 편이다.

OMG는 재미있는 기관이다. 좌우당간...

가이드라인과 정책은 소프트웨어 아키텍처를 결정한 후 아키텍처를 기반으로 프로젝트를 진행하기 위한 가이드라인과 정책을 말한다. 아키텍트는 설계, 코딩, 배포, 통합, 테스트, 툴, 유지보수, 시스템 확장, 시스템 운영에 대한 가이드라인과 정책을 마련해야 한다. 정말 포괄적인 개발정책 부분을 모두 서술하고 결정한다는 것이다.

소프트웨어 아키텍처는 시스템 개발의 여러 구성요소와 관련을 갖는다. 프로젝트 조직, 프로젝트 스케줄, 형상 관리 계획은 소프트웨어 아키텍처에 근거하여 결정된다. 소프트웨어 아키텍처를 설계하는 데는 프로토타입, 프레임워크, 유틸리티 프로그램 등이 영향을 끼친다.

1.4 소프트웨어 아키텍처의 특징

소프트웨어 아키텍처의 특징은 다음과 같다.

첫째, 소프트웨어 아키텍처는 시스템에 대한 추상화이다. 따라서 지엽적인 정보는 포함하지 않는다. 큰 그림만을 가지고 있는다.

둘째, 소프트웨어 아키텍처는 한가지 다이어그램으로는 결정되지 않으며 여러 관점의 다이어그램으로 이

루어진다.

각각의 세부적인 것이 아니라, 건축물의 투시도와 같으며, 옷 디자이너의 스케치와 유사한 성격을 가진다.

1.5 소프트웨어 아키텍처의 역할

소프트웨어 아키텍처가 시스템 개발에서 담당하는 역할은 다음과 같다고 볼 수 있다.

첫째, 관련 당사자들 사이의 의사 소통의 수단이다.

둘째, 개발 프로젝트 초기 단계에서 의사결정 도구이다. 아키텍처는 품질 요소를 결정하며 프로젝트 조직에 영향을 끼친다.

셋째, 시스템의 전체 구조를 결정한다.

넷째, 개발 프로젝트의 조직을 결정하는데 참고할 수 있다.

다섯째, 시스템이 가져야 할 품질 요소를 결정한다. 따라서 아키텍처를 통해 시스템이 어떤 품질을 가질 것인지 예측할 수 있다.

여섯째, 아키텍처를 통해 시스템 개발자들에게 어떤 교육을 시킬 것이지 결정할 수 있다.

일곱째, 소프트웨어의 변경 사항을 어떻게 관리할 것인지를 알려준다.

상당한 역할이지 않은가? 이를 도식화 하면 다음과 같은 그림이 만들어진다.

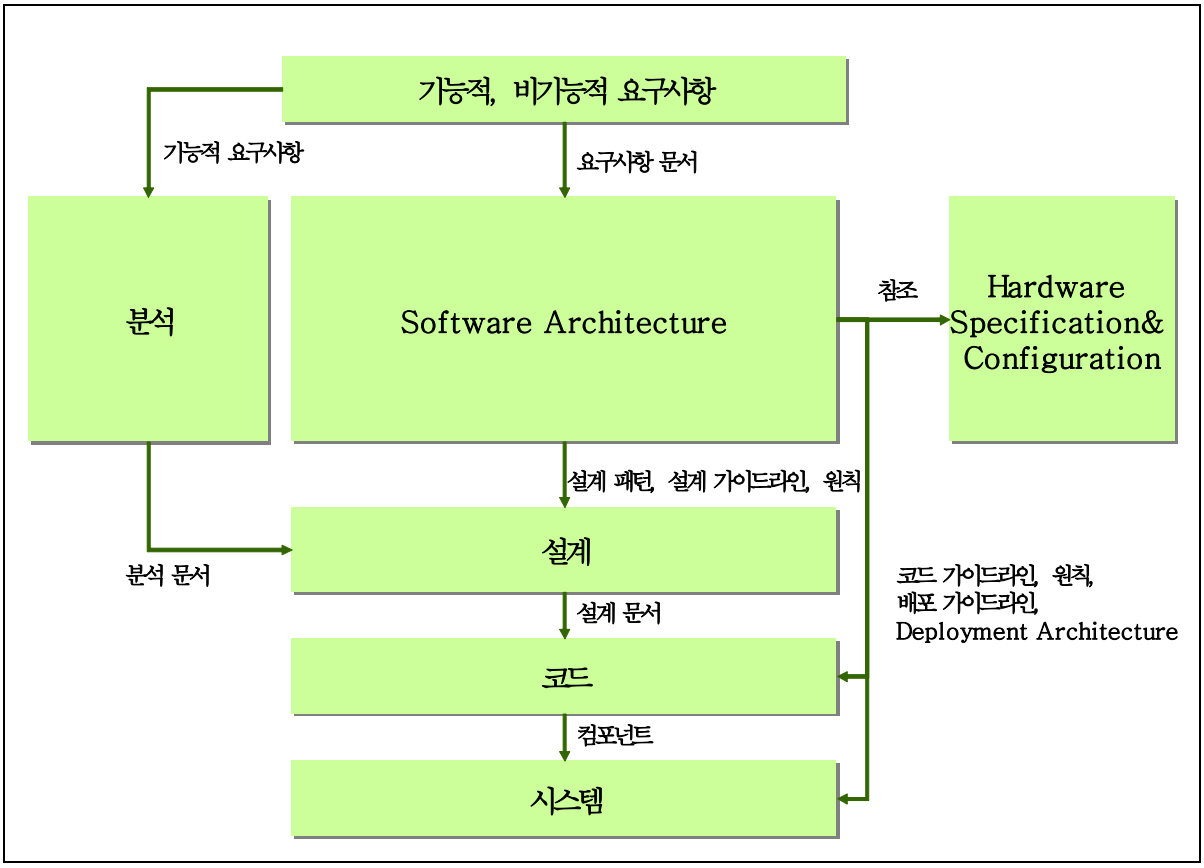


그림 II-2. 소프트웨어 아키텍처가 시스템 구축 시 하는 역할

그림 II-2는 소프트웨어 아키텍처가 시스템 구축 시 담당하는 역할을 보여준다.

기능적, 비기능적 요구사항은 소프트웨어 아키텍처를 설계하는데 입력 산출물로 작용한다. 소프트웨어 아키텍처가 결정되면 설계패턴과, 설계 가이드라인을 사용하여 소프트웨어를 설계할 수 있다. 또한 소프트웨어 아키텍처가 결정되면 코드 작성시 코드 표준, 코드 가이드라인을 제공할 수 있다. 또한 소프트웨어 아키텍처는 소프트웨어를 시스템에 배포 시에 가이드라인을 제공한다.

이를 반복에 의해서 구조적으로 안정적인 소프트웨어 아키텍쳐를 구성할 수 있다.

1.6 소프트웨어 아키텍처에 포함되지 않는 것

소프트웨어 아키텍처는 하드웨어, 네트워크, 물리적인 시스템에 대한 아키텍처를 포함하지 않는다. 따라서 소프트웨어 아키텍처 설계서는 시스템 안에 포함된 소프트웨어 만을 보여주며 시스템은 컨텍스트로서만 표현된다. 예를 들어 하드웨어 모델명, 하드웨어 구성, 라우터, 랜카드에 대한 정보는 소프트웨어 아키텍처에 포함되지 않는다.

이를 그린다는 것은 그 것에 종속된다는 의미이므로 '추상화'하려면 해당 정보가 표시되면 안된다는 것이다.

세부적인 구현 사항은 소프트웨어 아키텍처에 포함되지 않는다. 예를 들어 컴파일러 옵티마이제이션, DLL을 shared로 할 것인지, static으로 할 것인가는 소프트웨어 아키텍처에 포함되지 않는다.

이는 별도의 의사결정과정과 후반부에서 처리한다.

1.7 다른 아키텍처와의 관계

소프트웨어 아키텍처는 Enterprise Architecture에 포함된 business architecture, data architecture, Technical Architecture등을 참고하여 상세화 한다.

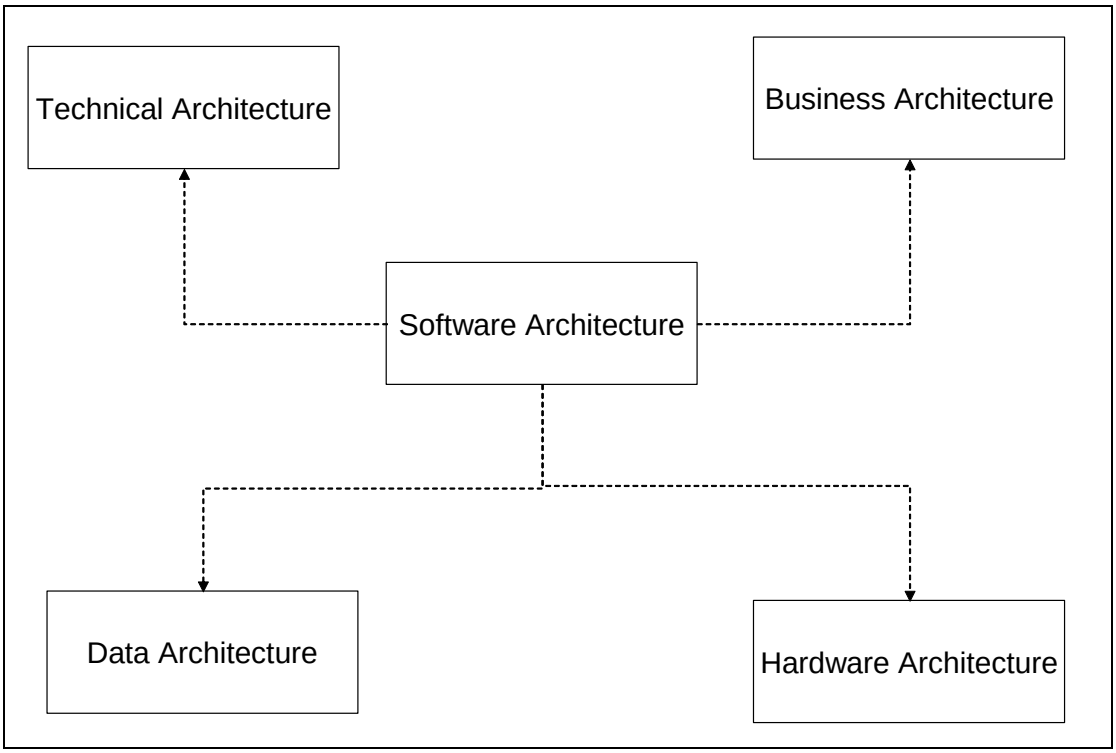


그림 II-3. 소프트웨어 아키텍처와 다른 아키텍처와의 관련성

다른 아키텍처는 소프트웨어 아키텍처가 작성되기 전에 완성되어 있으며 소프트웨어 아키텍처의 입력으로 받아들인다. 그러나 소프트웨어 아키텍처가 작성되면 다른 아키텍처의 변경이 불가피하며 소프트웨어 아키텍트는 다른 아키텍처도 변경해야 한다. 소프트웨어 아키텍처 문서에는 다른 아키텍처도 참고 자료로 포함해야 한다.

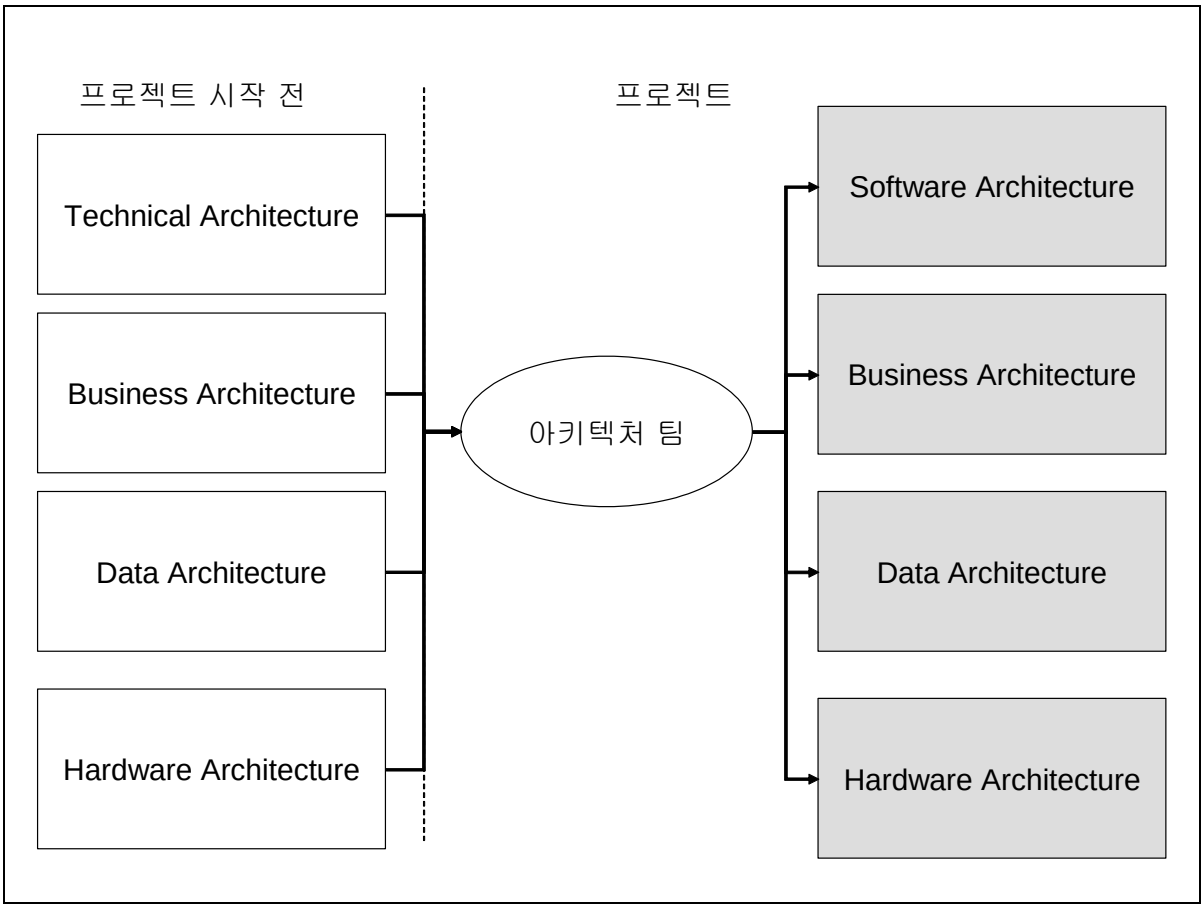


그림 II-4. 아키텍처 팀이 받아들이는 입력물과 생산하는 출력물

❑ IT Architecture

기업의 IT 자산을 사용하고 수정하고, 구축하고 획득하ㅡㄴ데 필요한 원칙, 가이드라인, 룰

❑ Business architecture

업무 프로세스, 조직, 전략

❑ Data Architecture

조직의 논리적, 물리적 데이터 구조

❑ Technical Architecture

어플리케이션이 배포되는 것을 지원하는 소프트웨어(미들웨어)에 대한 아키텍처

❑ reference architecture

특정 도메인에 대한 아키텍처를 지칭한다.

❑ enterprise architecture

소프트웨어 아키텍처, 데이터 아키텍처, 테크니컬 아키텍처, 비즈니스 아키텍처를 포함한다.

❑ Application architecture

Application architecture는 기업에서 사용되는 특정 시스템에 대한 소프트웨어 아키텍처이다.

❑ System architecture

구축할 시스템에 대한 아키텍처, software architecture는 system architecture의 subset이다. System architecture는 software와 hardware로 구성된다.

2 아키텍트의 역할 정의

아키텍트는 개발 프로젝트에서 초기에 기술적인 부분에 대하여 의사결정을 진행한다. 또한 프로젝트가 진행될 수 있도록 기술적인 이슈들을 해결한다. 아키텍트가 프로젝트의 각 phase마다 해결하는 문제는 다음과 같다.

표 II-1. 프로젝트의 각 phase마다 아키텍트가 수행하는 역할

단계	아키텍트가 수행하는 역할
● Inception 비즈니스 요구사항	● Architecture prototyping ● Make/buy trade-offs ● Primary scenario definition ● Archtecture evaluation ● CASE tool, 개발 툴 등 각종 툴 사용 방안 ● 설계 문서 템플릿 결정 ● 설계자와 개발자의 작업 규칙 결정
● Elaboration 솔루션 아키텍쳐	● Architecture baselining ● Primary scenario demonstration ● Make/buy trade-off baselining
● Construction 솔루션 개발	● Architecture maintenance ● Multiple-component issue resolution ● Performance tuning ● Quality improvements
● Transition 시범운영	● Architecture maintenance ● Multiple-component issue resolution ● Performance tuning ● Quality improvements

개발 프로젝트에서 아키텍트가 하는 역할은 다음과 같다.

아키텍트는 아키텍처를 만들고 컴포넌트와 컴포넌트 사이의 관계를 파악하고 인터페이스를 설계해야 한다. 프로젝트 관리자는 아키텍트가 아키텍처 문서를 생산하도록 관리해야 한다.

프로젝트에서는 아키텍처에 대한 일차 문서가 만들어지면 아키텍처 팀이 해체되고 각 subsystem에 대한 개발 리더로서 역할을 할 경우가 많다. 이 경우 시스템 전체를 보고 아키텍처를 upgrade하는 역할이 없어진다. 아키텍처가 일차 완성된 후에도 아키텍처는 자주 수정된다. 개발자들은 아키텍처 팀이 만든 문서를 받아들이지 않고 새로운 요구사항이 들어오면 아키텍처의 본래 목적에서 벗어난 방식으로 나름대로 개발하

려 한다. 따라서 아키텍처 팀은 프로젝트 끝까지 해체되지 않고 아키텍처에 대한 수정 및 업그레이드에 대한 책임을 져야 한다.

또한 아키텍트는 개발자들이 아키텍처를 이해하도록 도와야 하고 아키텍처 밑에 숨은 결정 사항을 설명해야 한다. 즉 아키텍트는 개발자들에게는 컨설턴트로서 리더로서의 역할을 해야 한다.

이를 정리하면 다음과 같이 정리될 수 있다.

1. 비즈니스 케이스 생성

   A. 비용과 기존 시스템과의 인터페이스 여부 설정

   B. 시스템의 한계 상황 설정

   C. 비즈니스 목표 설정

2. 요구사항 분석

   A. 아키텍쳐는 이해관계자들의 요구사항을 어떻게 반영할것이냐는 점이다.

   B. 기존 시스템을 어떻게 분석할 것인가?

   C. 프로토타입은 어느 수준까지? 어떻게?

   D. 품질요구사항에 따르는 아키텍쳐의 기준은 어떻게?

3. 아키텍쳐 구축

   A. 새로운 아키텍쳐를 만들것인가?

   B. 기존 아키텍쳐를 선택할 것인가?

2.1 아키텍트의 의무

1. 시스템의 목표를 찾아라
2. 고객이 기대하는 것이 무엇인가?
3. 시스템에 필요한 기술은 무엇인가?
4. 지금 쓸 수 있는 기술은 어떤것들이 있는가?
5. 현재 기술과 새로운 기술에 맞춰 고객의 기대와 비즈니스 목표를 고려한 전략을 세우고 유지하라.
6. 아키텍쳐의 일관성을 유지하라.
7. 설계와 관련된 기술 분야의 위험을 감지하고 대비하라.
8. 우선 순위를 결정해서 반복 개발 계획을 제안하라.

2.2 아키텍트가 가져야 할 역량

아키텍트가 가져야 할 역량을 기술 측면과 사업 전략 측면, 조직 측면에서 나누어 설명한다.

❑ 기술

아키텍트가 알아야 할 것	아키텍트는 무엇을 해야 하는가?	갖춰야 할 자질
● 도메인과 관련 기술에 대한 이해 ● 어떤 기술적인 이슈가 프로젝트 성공의 핵심인지 ● 개발 기술과 설계 기술	● 모델링 ● Tradeoff analysis ● Prototype/experiment/simulation ● 아키텍처 문서, 교육 자료, 프레젠테이션 준비 ● 기술적인 트렌드와 roadmap 분석	● 창조적 ● 실용적 ● 탐구적/분석적 ● 추상적인 단계에서 작업하는 것을 즐겨야 함 ● 모호함이 발견되었을 때 새로운 솔루션을 구하는 자세

❑ 사업 전략

아키텍트가 알아야 할 것	아키텍트는 무엇을 해야 하는가?	갖춰야 할 자질

● 조직의 사업 전략과 이유 ● 조직의 경쟁력 ● 조직의 사업 현황	● 사업 전략에 영향을 끼친다. ● 사업 전략을 기술적인 전략으로 옮긴다. ● 고객과 시장 트렌드를 이해한다. ● 아키텍처에 고객과 조직, 사업의 요구사항을 반영한다.	● 비젼 ● 기업 전체를 보는 관점

❑ 조직

아키텍트가 알아야 할 것	아키텍트는 무엇을 해야 하는가?	갖춰야 할 자질
● 조직에서 누가 핵심 역할을 수행하는가? ● 핵심 역할을 수행하는 사람이 사업적으로나 개인적으로 무엇을 원하는가?	● 핵심 역할을 수행하는 사람들과의 커뮤니케이션 ● 듣고 네트워크를 형성하고 영향을 끼친다. ● 비전을 세일즈한다. ● 아키텍처를 형성하는데 영향을 끼치는 요인들(혹은 사람들)을 수용한다.	● 다양한 관점에서 문제를 보고 다양한 관점으로 아키텍처를 선전할 수 있어야 한다. ● 때로 침묵을 지킬 수도 있어야 한다. ● 야심을 가지고 있어야 하며 다른 사람을 리드할 수 있어야 한다. ● 확신을 가지고 있고 자신의 입장을 명확하게 전달할 수 있어야 한다.

❑ 컨설팅

아키텍트가 알아야 할 것	아키텍트는 무엇을 해야 하는가?	갖춰야 할 자질
● 압축하여 전달하는 기술 ● 컨설팅 프레임워크	● 신뢰할 수 있는 어드바이져로서 관계 형성 ● 개발자들이 아키텍처를 통해 무엇을 원하는지 파악해야 함 ● 개발자들이 아키텍처의 가치를 이해하고 아키텍처를 어떻게 사용할지를 알도록 도와야 함. ● 주니어 아키텍트에 대한 멘터링	● 다른 사람의 성공을 돕는 능력 ● 호소력이 있어야 함 ● 작업 방식을 변경하도록 돕고 작업 절차에 정통해야 함 ● 좋은 멘터, 교사로서의 능력

❑ 지도력

아키텍트가 알아야 할 것	아키텍트는 무엇을 해야 하는가?	갖춰야 할 자질
● 자기자신의 역량과 능력	● 팀의 비전을 설정함 ● 의사결정 ● 팀을 조직함 ● 동기 부여	● 자기 자신이나 다른 사람이 아키텍트를 리더로서 볼 수 있도록 함 ● 카리스마가 있고 신뢰할 수 있어야 함 ● 참여적이고 헌신적이어야 함 ● 전체 사업과 개인 관점에서 결과를 볼 수 있어야 함

3 소프트웨어 아키텍처와 개발 프로세스의 관계

소프트웨어 아키텍처와 개발 프로세스와의 관계는 어떻게 정의되는것인지 살펴보자. 사실상 소프트웨어 키텍쳐를 중심으로 개발 프로세스를 구성하여야 하며 그것에 대한 관계는 다음과 같다.

3.1 소프트웨어 아키텍처와 다른 작업과의 관계

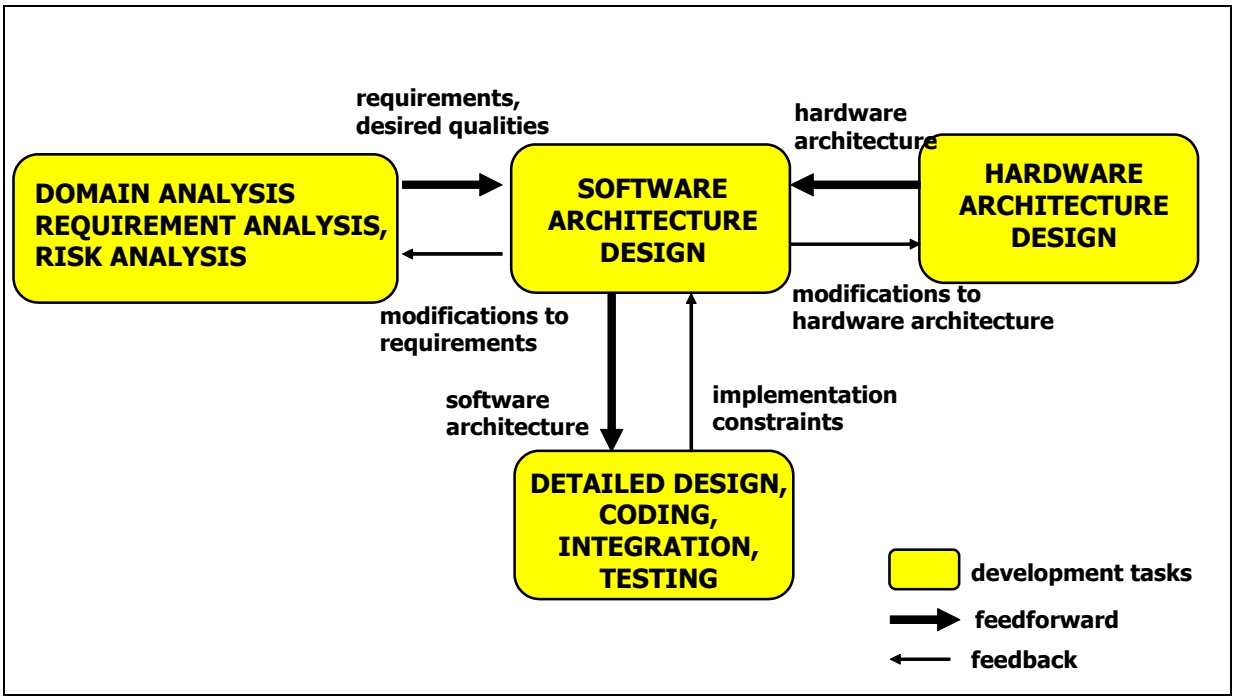


그림 II-5, 소프트웨어 아키텍처와 다른 작업과의 관계

그림 II-5은 소프트웨어 아키텍처와 개발 프로세스 상의 다른 작업과의 관계를 보여준다. 소프트웨어 아키텍처 설계는 요구사항 분석과 하드웨어 아키텍처를 참고하며 소프트웨어 아키텍처 설계가 완료되면 요구사항과 하드웨어 아키텍처도 영향을 받아서 수정되어야 한다. 요구사항의 경우는 소프트웨어 아키텍처가 정립되면 비기능적인 요구사항들이 구체화된다. 하드웨어 아키텍처는 소프트웨어를 동작하기에 적합한지를 판단하여 수정되어야 한다. 소프트웨어 아키텍처는 상세 설계의 입력물로 작용하며 구현에 대한 가이드라인을 제공한다.

3.2 소프트웨어 아키텍처와 개발 프로세스의 관계

소프트웨어 아키텍처가 하는 역할은 프로젝트의 각 phase마다 다르다.

❑ Inception phase (비즈니스 요구사항)

아키텍처 팀은 설계 팀과 함께 사용자의 요구사항을 듣고 비기능적인 요구사항을 추출한다. 아키텍처 팀을 구성하고 프로젝트 관리자에게 아키텍처 팀이 어떤 역할을 수행할 것인지 알려준다. 또한 아키텍처의 초안을 작성한다. 또한 어떤 제품을 살 것인지 개발할 것인지도 결정한다.

❑ Elaboration phase (솔루션 아키텍쳐)

요구사항을 받아서 아키텍처를 설계하고 구현한다. Elaboration은 전반적으로 아키텍트 팀이 리드하며 여기에는 핵심적인 Use Case에 대한 설계 및 구현이 포함된다. 따라서 설계자와 개발자를 포함시켜야 한다. 또한 아키텍트를 평가할 수 있도록 테스팅 인력도 포함시켜야 한다. 그리고 설계 문서 템플릿, 개발 문서 템플릿, 설계에서 개발로의 이행 방안, 툴 사용 방법등의 문서도 작성한다. 아키텍처는 Elaboration의 끝에서는 반드시 결정이 되야 한다. 아키텍처가 안정화 되었다고 판단하면 아키텍트 팀에 포함된 설계자, 개발자, 테스팅 인력을 재배치해야 한다.

❑ Construction (솔루션 개발)

이 phase에서는 시스템 구현에 초점을 맞추어 프로젝트 팀이 재 배치된다. 아키텍처 인력은 최소화되고 개발 중 이슈가 재기되었을 때 아키텍처를 재구성한다. 아키텍처 팀은 개발인력을 리드하고 시스템 전반의 구성을 교육시키고 컨설팅하는 역할을 담당한다.

❑ Transition (시험운영)

이 단계는 테스팅과 이행에 초점을 두고 있다. 따라서 테스팅과 이행을 위해 테스팅 조직, 지원 조직과 긴밀히 협조한다.

4 아키텍처와 개발 프로젝트의 다른 역할과의 관계, 개발 조직간의 관계

4.1 아키텍트와 다른 역할과의 관계

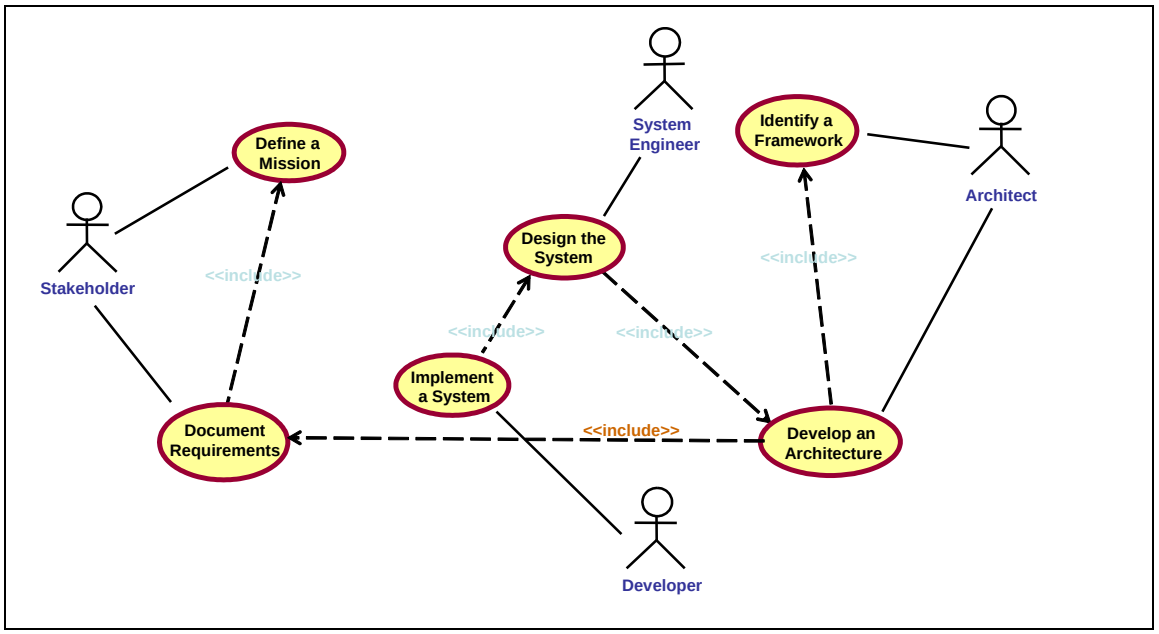


그림 II-6. 아키텍트와 다른 역할과의 관계

그림 II-6을 보면 관련 당사자는 요구사항을 정의하고 시스템의 개발 목적을 정의한다. 아키텍트는 아키텍처를 개발하면서 관련 당사자의 요구사항을 참고한다. 아키텍트는 아키텍처를 개발하면서 어떤 framework을 사용할 것인지도 결정한다. 시스템 엔지니어는 시스템을 설계하면서 아키텍처를 참고한다. 개발자는 시스템을 개발하면서 시스템에 대한 설계 문서를 참고한다.

4.2 프로젝트 조직 구성

아키텍처가 프로젝트의 핵심적인 역할을 수행하게 되면서 프로젝트 조직 구성도 변하게 된다. 프로젝트 조직은 한번 구성되면 프로젝트 마지막까지 변하지 않는 조직이 아니라 프로젝트의 각 phase가 끝날 때마다 재구성되는 동적인 조직이다. 아키텍처와 iteration을 고려한 프로젝트 조직의 특징은 별도의 아키텍처 팀이 존재하며 설계팀과 개발팀이 분리되어야 한다는 것이다.

아키텍처 팀, 설계 팀과 개발팀이 분리되어야 하는 이유

iteration에서는 한 iteration의 기간이 짧다. 또한 아키텍처가 완성되기 전까지는 개발이 시작될 수 없다. 따라서 아키텍처에 대한 작업이 진행되는 동안 설계자는 업무를 분석하고 개념 설계를 진행해야 한다. 아키

텍처가 완성되고 상세 설계에 대한 방침이 정해지면 개발이 시작될 수 있다. 개발팀은 개념 설계 문서와 상세 설계 방침을 바탕으로 개발을 진행한다. 어떤 경우에는 아키텍처 설계, 개념 설계, 개발이 동시에 진행될 것이다. 또한 개발팀이 개념 설계까지 담당할 수 없는 이유는 인터페이스에 대한 설계는 설계팀만이 결정할 수 있기 때문이다. 또한 설계팀은 업무에 대한 분석 및 설계만, 개발팀은 개발만 진행하는 것이 효율적이다. 또 한가지 방법은 설계팀은 업무 분석 및 개념적인 설계만 진행하고 개발팀은 아키텍처 팀의 지침을 따라 상세 설계를 진행한 후 개발해도 된다.

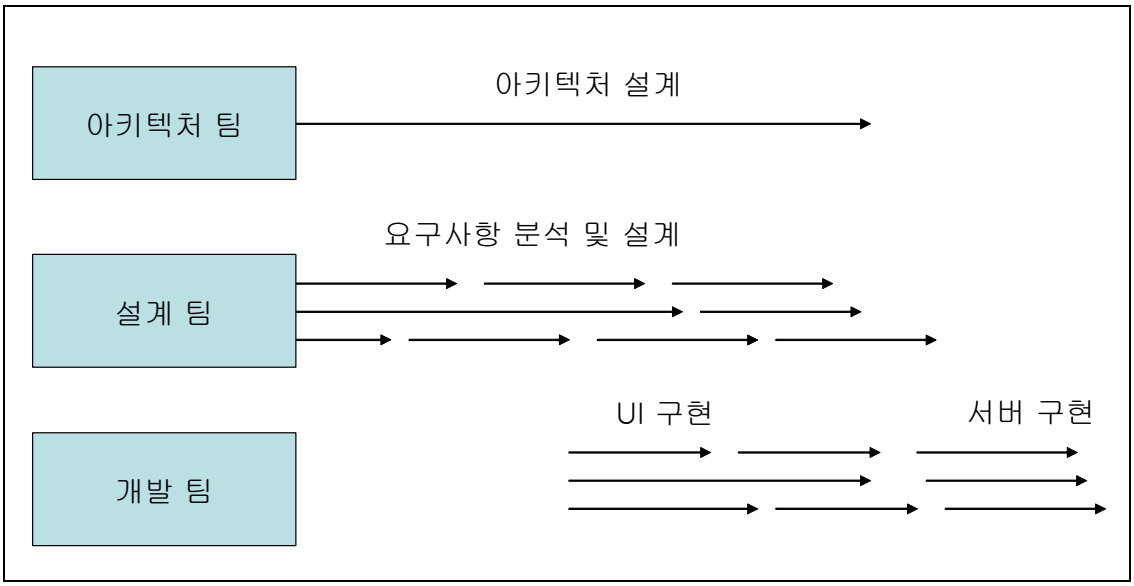


그림 II-7. 개발 조직의 작업 절차

그림 II-7를 보면 아키텍처 팀과 설계 팀, 개발팀의 작업이 동시에 진행된다. 그러나 동시 작업을 위해서는 다음 사항이 전제되어야 한다.

첫째, Inception 단계에서 각 팀이 어떤 문서를 주고 받을 것인지 어떻게 communication할 것인지가 결정되어야 한다.

둘째, 각 팀의 책임과 역할을 명확히 해야 한다.

셋째, 아키텍처 팀이 subsystem을 나누고 컴포넌트를 식별하기 전까지는 설계 팀은 요구사항을 분석할 수는 있지만 그 이상의 작업은 불가능하다.

넷째, 컴포넌트에 대한 초안은 아키텍처 팀에서 정의하지만 컴포넌트에 대한 정제, 상세 설계 및 인터페이스에 대한 결정은 설계 팀에서 한다. 대신 아키텍처 팀은 인터페이스를 잘못 설계한 것은 없는지 검증한다.

다섯째, 개발에 대한 검증은 설계 팀과 아키텍처팀에서 책임진다.

따라서 설계는 아키텍처 팀이 개발은 설계 팀과 아키텍처팀이 review한다.

4.3 각 조직이 하는 역할

각 팀이 해야 할 일들이 해야 하는 역할은 다음과 같다.

아키텍처 팀은 시스템을 초기 구현하고 설계, 개발에 대한 템플릿을 작성하고 설계와 개발의 이행 절차를 정의하고 툴 사용 절차를 정의한다. 설계 팀과 개발팀을 컨설팅하는 역할을 담당한다.

설계팀은 요구사항을 분석하고 화면을 작성하는데 중점을 둔다. 또한 인터페이스를 결정한다.

개발팀은 아키텍처의 가이드라인을 따라 설계팀에서 넘어온 문서를 바탕으로 상세 설계서를 작성하고 시스템을 구현한다.

4.4 개발 조직에 대한 구성 방안

개발 조직에 대해 다음과 같은 변형이 가능하다.

개발 조직이 개념 설계서만을 기초로 시스템을 개발하는 경우:

개발팀은 상세 설계서 없이 개념 설계서에 기초하여 아키텍처 문서의 가이드라인을 준수하여 시스템을 개발할 수 있다. 이 경우 개발 산출물이 아키텍처를 준수했는지는 아키텍처 팀에서 평가한다.

아키텍처 팀에서 상세 설계서를 작성하여 개발팀에게 넘겨주는 경우:

설계팀에서 개념 설계서를 작성한 후 아키텍처 팀에서는 상세 설계서를 작성하여 개발팀에게 넘겨준다.

MVC 패턴을 사용하는 경우:

MVC를 사용하는 경우 작업은 더 능률적이지만 아키텍트는 MVC의 일관성에 대한 검증을 더욱 철저히 해야 한다. 자신이 없다면 개발자를 MVC로 분리해서는 안 된다.

4.5 개발 조직과 아키텍처 팀 간의 관계

아키텍처 팀은 개발 팀, 설계 팀, 지원 팀, 테스팅 팀, 프로젝트 관리 팀과 관계를 갖는다. 또한, 아키텍쳐 팀은 TFT로 구성하는 것이 일반적이긴 하지만, 프로젝트의 규모에 따라서 상시조직으로 운영할 수도 있다. 다만, 상시운영이 될때에는 다음과 같은 운영 환경이어야 한다.

하나.의사결정을 해야할 고객 수준의 조직이 여러 개인 경우

둘. 아직 미결정된 하드웨어나 업무의 형태가 존재하는 경우

셋. 지속적인 성능향상이 필요한 경우

넷. 기술적인 요소에서 사례가 없거나 경험이 없는 경우 PM의 기술적인 참모조직이 필요한 경우

가장 중요한 것은 비용문제이기에 해당 문제에 대해서는 전반적인 협의가 필요하다.

아키텍처 팀과 개발 팀의 관계

개발 팀은 아키텍처 팀이 작성한 아키텍처 문서를 통해 시스템에 대한 전반적인 이해를 갖고 개발을 시작한다. 또한 개발 팀은 아키텍처 팀에서 작성한 코딩 표준안과 구현된 프로그램을 사용하여 개발한다.

아키텍처 팀과 설계 팀의 관계

아키텍처 팀이 초기에 설계한 문서와 아키텍처 상의 제약 사항을 염두에 두고 설계를 진행한다.

아키텍처 팀과 지원 팀의 관계

아키텍처로 인해 시스템 구성이 바뀌어야 하는 부분이 있으면 협의하여 수정한다.

아키텍처 팀과 테스팅 팀의 관계

아키텍처 팀의 지시를 받아 아키텍처가 사용자가 원하는 성능을 만족하는지 검사한다.

아키텍처 팀과 프로젝트 관리 팀의 관계

아키텍처가 결정된 후 프로젝트 관리자는 아키텍처를 반영하여 조직을 재구성한다.

5 소프트웨어 아키텍처 팀 운영 시 문제점에 대한 해결 방안

이 장에서는 소프트웨어 아키텍처 팀 운영 시 발생하는 다양한 문제점에 대한 해결 방안을 제시한다.

5.1 리더쉽에 대한 명확한 정의가 없는 경우

개발 조직에서는 조직원에게 명확한 권한과 책임을 부여하는 것이 매우 중요하다. 개발 팀이 지리적으로 두 개로 나누어져 있는 경우 각 팀마다 자체적으로 아키텍트의 역할을 하는 사람이 생기게 되며 이 경우 두 아키텍트 사이에 의견 충돌이 발생할 수 있다. 이 경우 프로젝트 관리자는 각 개인의 역할을 명확히 정의하고 지리적으로 나누어진 각 사이트에 테스팅 팀, 개발팀, 아키텍트 팀을 분배해야 한다. 즉 두 사람이 같은 역할을 소유하지 못하게 한다.

5.2 소프트웨어 아키텍트의 보고체계

소프트웨어 아키텍트는 프로젝트 팀장에게 직접 보고하는 경우가 많다. 그러나 너무 상세한 수준까지 보고하면 이 보고는 종종 실패하게 된다. 관리자가 기술적으로 불일치한 것을 검토할 수 있도록 보고서는 적당한 레벨을 유지해야 한다. 반대로 너무 추상적으로 보고하면 소프트웨어 아키텍트가 개발팀에서 소외 당하게 된다. 따라서 보고체계를 다음과 같이 설정하는 것이 좋다.

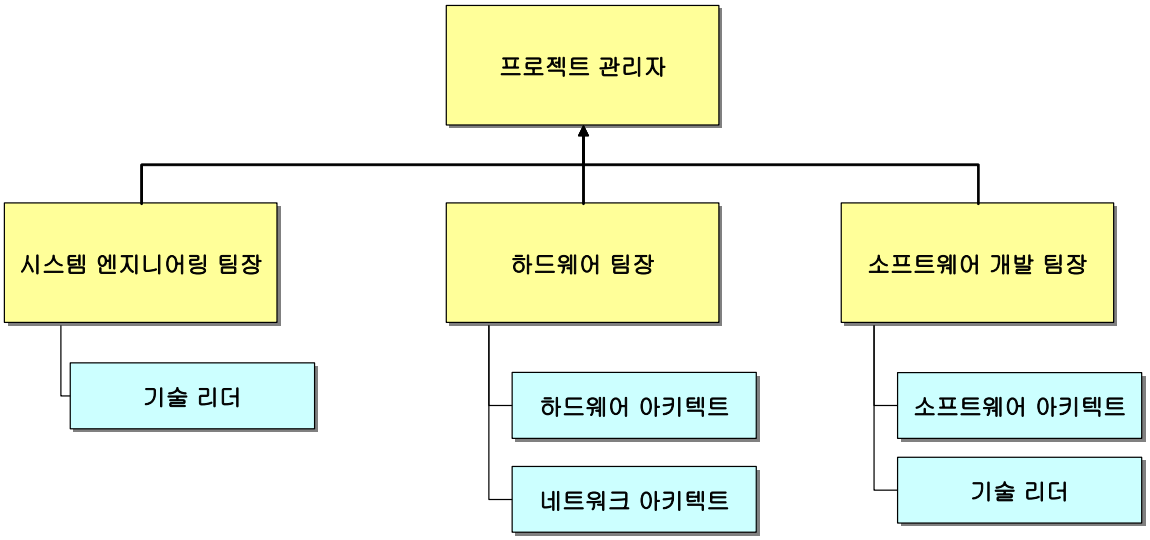


그림 II-8. 소프트웨어 개발 조직의 보고체계

그림을 보면 소프트웨어 아키텍트는 소프트웨어 개발 팀장에게 보고하고 하드웨어 아키텍트는 하드웨어 팀장에게 보고한다. 소프트웨어 개발팀장과 하드웨어 팀장은 결정한 사항에 충돌이 없는지 검사해야 한다.

5.3 소프트웨어 아키텍트와 개발팀의 지리적인 위치

지역적으로 분산된 개발 환경에서는 소프트웨어 아키텍트는 중요한 개발팀과 같은 곳에서 근무하거나 여러 개발팀을 계속 방문해야 한다. 지역적으로 분산되어 있을 경우 특정 사이트에서 기술적인 문제가 많이 발생하기 때문에 그 사이트의 기술자 중 한 사람을 아키텍트로 선정하여 여러 사이트를 방문하면서 기술적인 이슈를 해결하도록 해야 한다.

5.4 소프트웨어 아키텍트 팀의 규모

프로젝트 관리자가 종종 기술 리더가 아닌 사람을 아키텍트 팀에 넣는 경우가 있고 스스로 기술 리더가 아니면서 아키텍트 팀에 포함되기를 원하는 사람들이 있다. 이 경우 소프트웨어 아키텍트는 아키텍트 팀의 규모를 제어할 수 있어야 한다. 즉 소프트웨어 아키텍트는 팀 규모에 대한 가이드라인을 작성하여 프로젝트 관리자와 대화할 수 있어야 한다. 팀이 지나치게 크거나 기술 리더가 아닌 사람이 팀에 포함될 때 소프트웨어 아키텍트 팀의 효율성은 떨어진다.

팀에 불필요한 사람을 포함시키지 않도록 아키텍트 팀의 모임에는 다양한 사람들이 참가하는 것을 허용하고 결정은 아키텍트 팀에서 내려야 한다. 아키텍트는 팀 내 대화와 정보 공유, 의사 결정에 대한 규칙을 가지고 있어야 한다.

종종 아키텍트 팀에서 능력 있는 사람이 배제되는 경우가 있다. 따라서 초기에 3-4명 정도의 인원으로 아키텍트 팀을 구성하고 이슈가 발생하여 사람이 필요할 때 추가해야 한다. 팀에서 사람을 배제하는 것은 매우 어렵기 때문에 초기에 팀 구성을 잘 해야 한다.

5.5 소프트웨어 아키텍트가 다른 프로젝트로 이동하는 경우

종종 소프트웨어 아키텍트가 프로젝트가 끝나기 전에 역할을 다 했다고 생각하고 다른 프로젝트로 옮겨가는 경우가 있다. 그러나 프로젝트가 완료되기 전에 소프트웨어 아키텍트를 새로운 프로젝트로 옮기는 것은 피해야 한다. 첫째 이유는 작성한 소프트웨어 아키텍처가 효과적인지는 시스템을 운영하기 전에는 알 수 없으며 최종 사용자의 불만 사항을 알아야 하기 때문이다. 소프트웨어 아키텍처의 효과를 확인하지 않고 다른 프로젝트로 옮기게 되면 새로운 프로젝트에서도 결함이 있는 아키텍처를 사용하게 된다. 따라서 소프트웨어 아키텍트는 개발팀에 합류하여 소프트웨어 아키텍트의 역할을 계속 수행하거나 기술적인 리더로서 역할을 수행해야 한다.

6 소프트웨어 아키텍쳐의 참고사항

6.1 서비스 지향

아키텍트는 소프트웨어의 환경이 변화되고 있는 것을 감지해야 한다. 그것은 비즈니스의 신속성 (business velocity)를 증가시킨다.

'기업이 올바른 방향을 설정하는 한편, 그 운영 속도를 가속화 할 수 있는 능력'

그것을 빠르게 표현하는 것이다.

90년대의 소프트웨어의 화두는 '얼마나 빠르게 소프트웨어를 구축하는가?'라는 것이 최고의 목표였었다, 그러나 21세기의 화두는 '올바른 방향을 유지하는 동시에 신속하고 품질을 보장할 수 있는 소프트웨어를 만들것인가?'이다.

90년대의 소프트웨어는 '목표'만 정확하면 그 품질을 개량화 하기 수월한 소프트웨어들이 대다수였다. 일반 패키지들이 그러했고, 일반 업무들도 단위업무 중심이었기 때문이다.

이제 21세기 소프트웨어는 수많은 프레임워크와 수많은 업무들이 연관관계를 가지는 구조이기에 그 품질목표마져도 모호한 상태이거나 계량화 하기 힘들어진 상태이다. 그러하기 때문에 아키텍트는 Service중심의 소프트웨어를 구상하여야 한다.

이는 Service Oriented Architecture is an architectural style whose goal is to achieve loose coupling among interacting software agents 라고 정의할 수 있다.
(상호 작동하는 시스템 사이를 느슨하게 연결하려는 목적을 가진 아키텍쳐 이며 W3C의 Web Service Architecture Working Group에서 활동하고 있는 Hao He 박사의 정의이다.)

6.2 서비스 시스템의 목표

아키텍트는 소프트웨어 복잡도를 해결하면서 시스템의 품질 속성을 획득해야 한다. 이 서비스 시템은 물론 변경도 용이해야 한다.
유지보수성, 확장성, 재구조화성, 이식성을 가져야 한다는 점이기도 한다.

- 소프트웨어 복잡도 해결
- 시스템의 품질 속성 획득
  - 변경용이성
    - 유지보수성, 확장성, 재구조화성, 이식성
  - 상호운영성
  - 효율성
  - 안정성
    - 고장 감내, 강건성
  - 시험요잉성
  - 재사용성
- 문제를 해결하기 위한 기본 테크닉들
  - 추상화, 캡슐화, 정보은폐, 모듈화, 고려사항의 분리, 결합력과 응집력,
  - 충분-완전-원시성, 정책과 구현의 분리, 인터페이스와 구현의 분리, 한번의 참조, 분할 정복

그러나, 아키텍트들이 주의하여야할 점들이 있다.

가장 훌륭한 아키텍쳐는 목표시스템과 어울려야 한다는 것이다.

그렇다면. 반문할 수 있다. 어떤 아키텍쳐가 최선의 아키텍쳐냐고? 당근, 답변은 없다.

'그때 그때 달라요'

이런 예를 들어보자, 정말 엄청나게 짧은 기간에 대량의 데이터를 읽어서 실시간으로 레포트를 출력하는 프로그램을 만드는데 조직원들의 구성도 초보개발자들로 구성되어 있는데다가, PM도 성질을 부리고 고객도 성깔깨나 있는 곳이라면 어떻게 할것인가?
더군다나, 한 보고서 출력시 1초를 넘기면 안된다고 성질을 부리고 있다면?

물론, 아주 멋진 아키텍쳐를 제안해서 만들고 싶다.
AJAX로 화면을 구성하고 Spring Object로 비즈니스 컨테이너를 사용하고 하이버네이트로 OR맵핑을 한 아주 멋진 구성을 들이민다면?

아마도, 개발자들이 나자빠질것이다.

차라리.. 4GL도구나 레포팅 툴을 사용하십시요. 라고 제안을 하는 것이 올바른 아키텍트이다.

또하나의 주의점은 가이드라인이나 아키텍쳐라는 것 자체가 개발자들이 보기에는 무시무시하고 배우기 싫고. 귀찮은 부분이라는 점이다. 아마도, 개발을 진행하면서 무수히 많은 도전을 받을 것이기 때문에. 인내심과 이해관계자들을 설득하려는 뛰어난 구강신공을 가지는 것이 좋다.

아니면, 아예.. 고객을 설득해라!

심플하고 반복하지 않는 것!

6.3 간단한 아키텍쳐링 예제

가장 먼저 아키텍트가 할일은 '시스템'이란 무엇인가?를 찾고 식별해야한다.

첫째. 정치활동을 충실히 하라!

정치적인 변수들 (고객, 사업관리 조직 등등 - 심지어 고객의 사모님의 취향~)을 비밀문서에 잘 체크해두어야 한다.
간혹, 화면의 색만 가지고 시비거는 고객도 있다. (아예... 레이어를 분리해버리는 것도 방법이다.!)

정치활동을 통하여 아키텍트가 가진 컨셉을 모든 이해관계자들에게 설득하고 조율해야 한다. 당연히 엄청난 구강신공은 필수가 된다.

가장 뛰어난 아키텍트는 그 사람이 결정한 것이 아니라 그 결정 마져도 해당 관계된 사람들 사이의 회의에서 얻어진 결과물로 착각하게 만드는 것이 최선이기 때문이다.

둘째로 시스템의 구축 범위를 명확히 하라.

외부 시스템과 내부 시스템을 명쾌하게 구분해서 고객을 이해시켜야 한다. 또한, 솔루션이 할일과 외부 프로그램들이 할일에 대해서 명쾌하게 해야한다.
특히나 솔루션인 경우에 그 테스트 범위와 관련 일정등을 제대로 파악하지 못하면 프로젝트가 낭패를 당할 수도 있다.

결국, 이 내용들은 제안요청서(RFP), 비전기술서(안만들기도 하지만), 프로젝트 계획서나 수행계획서가 만들어질때에 그 '단어'의 선정과 설명을 명확하게 해야한다는 점이다.

세번째로 기능과 비기능을 명확히 하라.

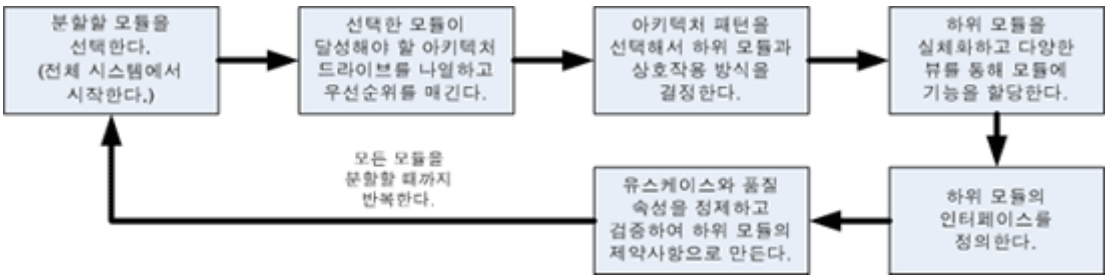
결국 품질부분이나 시스템의 구성형태에 대해서 전체적인 관련내용들에 대해서 고객만족을 위한 영역설정을 명확하게 하는 것이기 때문이다.

유스케이스로 표현되는 기능과 품질로 표현되는 비기능요소를 명확하게 구분해야 한다.

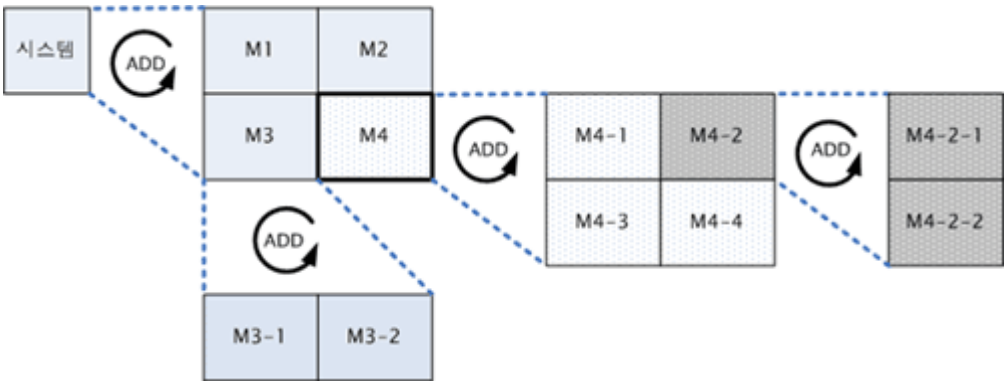
대표적으로 시스템이 높은 성능을 내어야 한다는 아주 모호한 부분을 실제 개량화하여 제시해야 하는 것도 아키텍트의 임무중의 하나이다. 결국, 품질과 공정(설계, 구현, 배치)는 떨어질 수 없는 끊임없는 반복의 결과이기 때문이다.

카네기-멜론 소프트웨어 엔지니어링 인스트튜트에서 제안한 ADD(Attribute-Driven Design, 속성 주도 설계)방법이든 아니면, 각자 고안한 방식이든. 큰 그림을 그리고, 시스템을 분할하여 나아가는 것이 필요하기 때문이다.
이를, '아키텍쳐 패턴'이라고 부르기도 한다.

간단하게 ADD 방법을 살펴보면 다음과 같다.
(상세한 것은 www.sei.cmu.edu/architecture/add_method.html 에 있다.)



간단한 방법에 대한 설명이다.



시스템을 분할해서 구성하는 방법을 간단하게 도식화 한것이다.

아주 좋은 이야기 하나 소개. [브라이언 푸트의 패턴이야기]

브라이언 푸트(Brian Foote)만큼 훌륭하게 패턴을 설명한 사람은 없는 것 같다.

게으르고 영약한 건축가가 있었다고 한다.

어느날 이 건축가가 대학 건물과 그 주변 통행로를 설계하게 되었는데 대학 건물만 설계하고 통행로는 방치한 채 게으름을 피웠다.

결국 통행로 없이 대학 건물만 지었지만 이 건축가는 여유만만이었다.

사람들은 통행로가 없었기 때문에 자기 좋을 대로 건물 주위를 돌아 다녔다.

겨울이 되었고 큰 눈이 내렸다.

이 건축가는 드디어 일을 시작했다. 사람들이 눈 위로 돌아다닌 발자국들을 사진으로 찍었다가 봄이 되자 찍어 두었던 사진들을 보고 통행로를 설계했다.

이렇게 완성된 통행로는 다니기에도 편했고 주변 건물과도 잘 어울렸다.

패턴이란 바로 이런 것이다.

패턴은 어떤 분야에서 계속 반복해서 나타나는 문제들을 해결해 온 전문가들의 경험을 모아서 정리한 것이다.

누가 뭐래도 대학 건물 주변의 동선(動線)을 제일 잘 알고 있는 전문가는 바로 매일 건물 주위를 돌아다니던 사람인 것이다.

게으르고 영약한 건축가는 누가 전문가인지 잘 알고 있었고 전문가들의 경험을 활용한 것이다.

이렇게 분할 하는 방법이 여러가지 view로 시스템을 바라보면서 아키텍쳐를 만들어 나아가는 것이기 때문이다. 따지고 보면, 건축물도 디자인을 먼저하고 세부적인 공정작업에 들어가는 것이기 때문이다. 가장 완성도가 높은 것은 고객에게 '완성'된 프로그램의 형태를 보여주고 그 형태를 만들기 위해 애쓰는 것이 아닐까?

결국 건축아키텍트는 수많은 투시도를 그리게 되고 설계도로 분화되면서 최후에 실제 프로그램을 구성하는 형태로 구성되어질 것이기 때문이다.

강사가 예전에 만들었던 소프트웨어에 이런 것이 있었다.
AutoCad에 애드인하는 소프트웨어 였는데. 건축설계사무소에서 Cad에 설계를 할때에 정의된 클립아트 형태 (계단, 지붕, 문 등등)를 사용해서 아파트나 주유소와 같은 건축물을 설계하면 관련 클립아트들이 컴포넌트 형태로 건축 일위대가라는 속성과 연결되어져서 실제 건축에서 산정되는 단가를 추출해주는 프로그램이었다.

건축가의 처음 '투시도'하나에서... 실제 '건축내역서'까지 파생되는 과정을 보면, 현재의 소프트웨어의 구상과 참으로 유사한듯 하다.

아마도, 완성된 프로그램을 빠르게 도식화해서 '고객'과 협의에 사용하게 하는 프로그램이 만들어진다면, 정말 괜찮을 듯 하다.

말이 조금 빗나갔지만, 아키텍쳐를 바라보는 View에 따라서 다 다르다.

업무관점에서 바라봤을때에..

구성	내용
애플리케이션 프레임워크 레이어	기술 기반 기능을 제공한다.
애플리케이션 프레임워크 레이어	업무 기반 기능을 제공한다.
업부 파티션 레이어	실제 업무 기능을 제공한다. 공통 업무 기능을 제공하는 코어 업무 파티션과 나머지 일반 업무 기능을 제공하는 단위 업무 파티션으로 나눌 수 있다.

구현관점에서 바라봤을때에..

구성	내용
프리젠테이션 레이어	뷰(View). 액터-시스템 인터페이스를 제공한다. 채널계 쪽으로 빠질 수도 있다.
애플리케이션 레이어	프리젠테이션 레이어와 비즈니스 레이어를 연결한다. 외부 시스템으로부터 요청을 받아들인다.

비즈니스 레이어	컨트롤(Control). 제공해야 하는 서비스를 실행한다.
커넥터레이어	다른 레이어를 리소스 레이어나 외부 시스템과 연결한다.
업부 파티션 레이어	모델(Model). 시스템이 다루는 정보를 관리한다.

시스템 관점에서 바라봤을때에.. 모든 시야에 따라서 그 레이어는 달라지게 된다.

네번째로 기술적으로 균일하게 만들 수 없다는 것을 알아야 한다.

다양한 기술과 다양한 개발자들과 일해야 하기 때문에, 그 기술 구조와 업무 구조에 따른 형태를 잘 구상해야 한다. 특히나 사업관리와 잘 이야기해야 할 것이 유지보수도 쉽고, 인력 수급도 쉽고, 개발 난이도도 낮은 수준의 시스템을 그리라는 것이다.
목표치를 조금은 낮추어 잡아야 프로젝트의 성공확률이 높아진다.

그리고, 개발언어, 툴에 너무 끌려다니면 안된다. 그래서, 구현을 생각하지 말고 생각해야 한다. 단순 CRUD성 프로그램을 만들기 위해서 복잡한 아키텍쳐를 구사할 필요가 없다.

가능하면 자동 코드 생성이 되는 툴을 사용하라.

다섯째로 일은 사람이 한다는 점.

이야기 많이 하고 술 같이 많이 마시고, 담배피고.. 결국 머리가 희게된다..ㅡ.ㅡ;
사람과의 관계가 가장 중요하게 된다는 점이다.

6.4 개념 설명

SOA와 CBD를 혼동하는 분을 위한 간단히 구분법을 설명드린다. 두 가지 모두 Reusable(재가용성)에서 중요한 개념이다.

하지만 중요하게 다른 점은 서비스가 서비스 소비자가 원하는 궁극의 결과를 얻기 위해 서비스 제공자가 처리해주는 한가지 일의 단위(W3C, Dr.Hao he)라고 보는 것이 SOA이고 컴포넌트는 다른 컴포넌트와 상호 작동하기 위해 만들어진 것으로 특정기능(functionality)또는 일련의 기능을 담고 있으며 명확히 정의된 인터페이스를 가지는 소프트웨어 객체(W3C WS-glossary) 라고 보는 것이다.

또한 가장 큰 차이점은 컴포넌트는 이미 구현된 것이기에 프로그래밍 언어에 중립적이지 않다는

점이며 CORBA의 대표적인 실패원인 이기도하다.

하나더 마이크로소프트의 비쥬얼스튜디오 닷넷에서 바라본 엔터프라이즈 템플릿

'닷넷 엔터프라이즈 아키텍쳐'

모델	관점
목표 과정 모델 [The Objectives and Goals Model]	비지니스 관점 [The Business Perspective]
수행절차와 조직 모델 [The Processes and Organization Model]	애플리케이션 관점 [The Application Perspective]
시스템과 데이터 모델 [The Systems and Data Model]	정보 관점 [The Information Perspective]
기술 모델 [The Technology Model]	기술 관점 [The Technology Perspective]

엔터프라이즈 템플릿을 통해서 TDL(Template Definition Language)를 구사.
엔터프라이즈 템플릿 빌딩 블록으로 닷넷 프레임웍과 CLR을 지원하는 언어로 작성되는 비쥬얼 닷넷 프로젝트.

6.5 소프트웨어 방법론에 대한 간략 소개

소프트웨어 방법론(software methodology)은 소프트웨어 개발방법에 대한 체계적인 접근방법이자 제어방식이다. 개발시의 'How'를 다루고 있다고 보면 간단하다.

그리고, CMM 프로세스는 그 방법론의 품질에 대해서 다루고 있다고 보면 된다.

일반적으로 유명한 RUP(Rational Unified Process)는 전통적인고 거대한 프로젝트를 구사할 경우에 그 성격이 맞으며 XP(eXtreme Programming)는 작고 가벼우며 신속하고 개발자 중심으로 흘러가는 경우에 적합하다.

중요한 것은 어느것도 정답은 아니다라는 것이다. 각각의 방법론은 일하는 조직과 일의 성격에

따라서 개별적으로 정하는 것이 옳다고 생각한다.

RUP(Rational Unified Process)에 대한 간략소개

그 목적은 '즉시'사용가능한 소프트웨어를 구현하는 IBM의 제품 오퍼링이며, 가장 중요한 것을 요구사항을 모으는 것이 중요하다고 역설하는 방법이다.
RUP는 시스템 디자인시에 UML(Unified Modeling Language)를 사용하고, 프로젝트 초기에 마지막 목표시스템에 대한 기능 및 기술적인 레이어를 포함한 완벽한 프로토타입을 구현해야 한다고 역설한다.
(문제는, 비기능 요소를 모조리 구현하려면 그 작업이 복잡하고 대규모적이 될 수 있기 때문이다.)

RUP의 기본단계는 다음의 4단계이다. 개념화(inception), 상세화(Elaboration), 구축(Construction), 전이(Transition)이다.

개념화는 프로젝트의 비전과 범위를 전개하고 초기 ROI를 만들며, 상세화단계에 요구사항을 수집하고 디자인 프로세스를 완료, 구축단계는 시간과의 싸움이라고 보며, 전통적으로 프로그램 개발을 하는 실제 단계가 이 구축단계라고 볼 수 있다.

XP(eXtreme Programming)에 대한 간략소개

이제는 7~8년 된 방법으로, 짧은 반복과 페어(pair)프로그래밍, 코딩 전 테스트의 개념으로 사용자 만족을 강조하는 방법으로 고안된 형식이다.

CRC(Class-Responsibility-Collaborator)카드 기술이나 사용자 스토리지를 정의하는 방법들로 요구사항을 빠르고 효과적인 방식으로 도입하고 이러한 요구사항들의 우선순위를 적절하게 분배하게 한다.

다만, 요구사항의 밸리데이션 수준까지는 다루지 않고 있다. 가장 간과하는 품질요소를 체크하지 못한다면 문제를 발생시키기도 한다.

7 1회강의 맺음말

소프트웨어 아키텍쳐에 대해서 기본적인 개념에 대해서 소개하였다고 생각한다.
마지막으로,
현재의 소프트웨어 환경에 대해서 몇가지 거론하고 마무리 하도록 하겠다.
소프트웨어의 환경은 이제야 가내수공업환경을 벗어나고 있다고 보인다. 이제는 좀더 전문적이고 체계적인 환경으로 분화되고 있다고 볼 수 있다. 사실상 아키텍트는 예전부터 존재해왔다. 다만, 실제 프로그래머가 아니라고 치부했을 뿐.

이제 아키텍트는 일종의 건축물의 설계자라고 볼 수 있다.

노가다 판에서 10년을 굴른 사람이라면, 어깨넘어로 배운 기술이 있어서 1층짜리 집은 지을 수 있다. 그것은 구조적인 계산 없이도 적당한 자재만 골라도 집을 지을 수 있고, 법적으로도 큰 문제가 되지는 않는다.

그러나, 노가다판 10년 굴렀다고 63빌딩을 지을 수는 없다.
높이 계산 하나만 하더라도 수많은 구조적인 계산과 복잡한 일정, 공정등을 알아야 하기 때문이다.

노가다를 뛴다고 건축계에서 일한다고 이야기할 수도 있다.
과연 그렇게 생각하는가?

건축을 예로 든다면, 프로그래머는 일당을 받는 노가다 인지도 모른다. 최소한, 전문 기술을 가진 건축기사나 전기기사의 역할은 해야하지 않을까?
특이한 기술 없이 반복적인 벽돌 쌓기를 하면서 남보다 벽돌 이쁘게 잘 쌓았다고 자랑하는 노가노가다 된 것은 아닌가?

건축에서도 예술적인 집을 지으러 다닐 수도 있고, 대규모적으로 평범한 아파트를 지으러 다닐 수도 있다. 따지고 보면 소프트웨어 계통도 유사한 경우가 많다.
따지고 보면, 가장 큰 이익을 창출하는 것도 아파트 공사장일 듯이 보이지만, 그것은 아니다, 미술관이나 예술적인 건물들, 난공사, 댐등의 공사도 있다.

더군다나, 그 도시를 설계하는 사람들도 있고, 예측하는 사람들도 있다.

이를 소프트웨어의 입장에서 생각한다면 어떠한 것들이 그려질까?

아키텍트는 이노베이터를 지향한다고 본다.

언젠가 한번쯤 '소프트웨어의 도시'를 설계하는 사람이 되어야 하지 않을까?

총 6회에 걸친 '소프트웨어 아키텍쳐'에 대해서
가장 '입문'분야인 1회과정을 마무리하고..

1 회 - ~~소프트웨어 아키텍쳐 개념에 대한 이해.~~
2 회 - 소프트웨어 아키텍쳐 설계를 하는 방법.
3 회 - 컴포넌트 설계를 하는 방법과 설명,
4 회 - 아키텍쳐에서 개발로의 이행하는 방법은?.
5 회 - 소프트웨어 아키텍쳐 산출물 작성법.
6 회 - 소프트웨어 아키텍쳐 설계(실전예제)에서 얻어보기.

의 내용을 좀더 충실하게 준비된 상황하에서 만나뵙고 싶습니다. 그럼.

개념과 정의
Developer - 계획적, 종합적 방식으로 개발하는 사람을 의미.
원래는 건축용어로써 도시주택에 관련된 개발사업자를 일컫는 말이었다. 디벨로퍼가 대규모적인 뉴타운의 건설과 도시재개발사업의 추진자로서 단순한 부동산업자와 구별되는 이유는, 각종 곤란한 사업에 도전하여 독자적인 이념하에 시대를 선견, 대규모 또는 계획적으로 도시조성을 목표로 지향하는데 있다는 것.

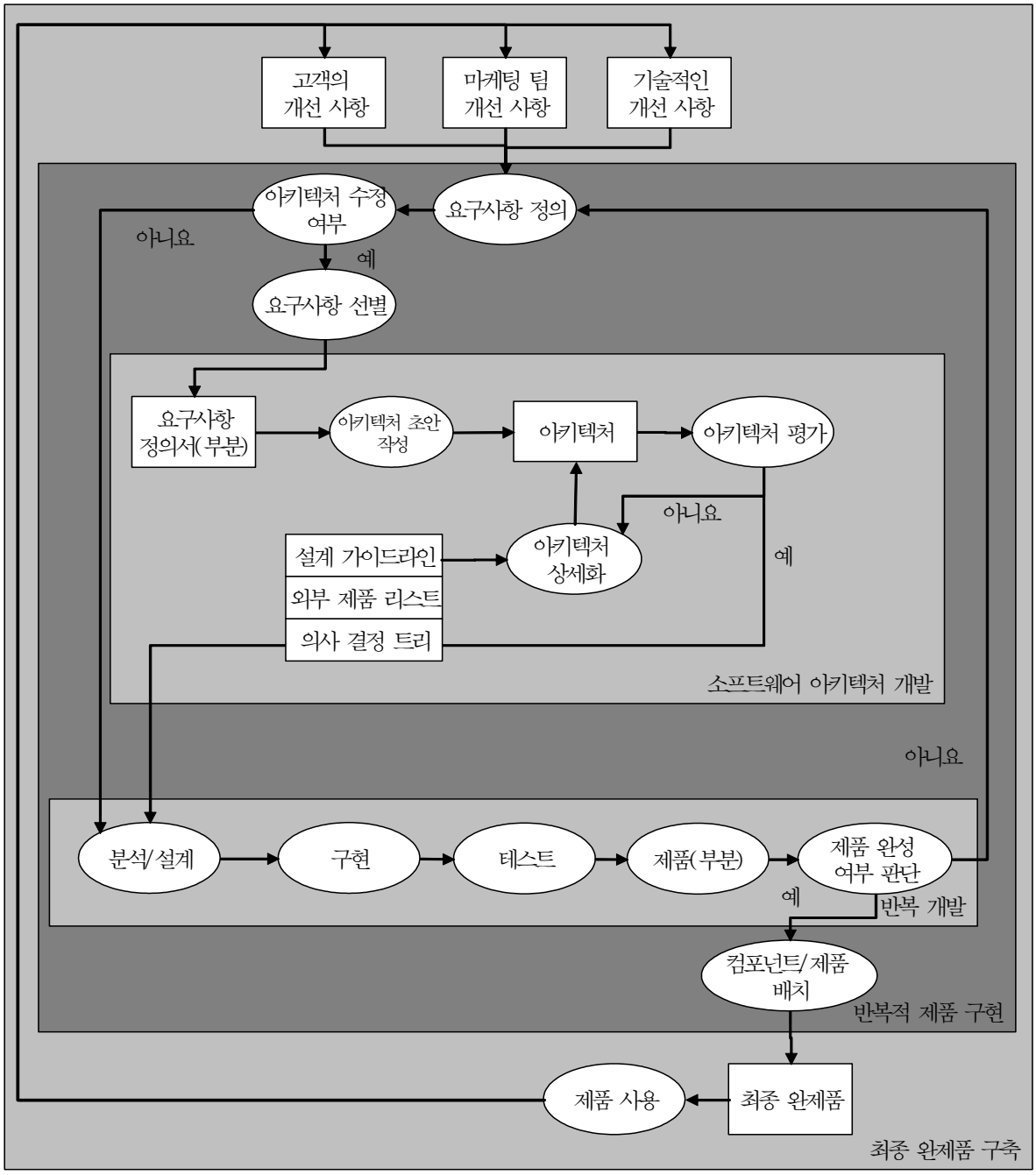
Programmer - 컴퓨터 프로그램의 논리나 알고리즘을 설계하고 프로그램을 작성하고 테스트하는 사람. 이제는 너무 방대한 직업이 되어버린듯.
이당시의 개념으로는 시스템 분석자(system analyst), 데이터베이스 관리자(DBA : database administrator)
프로그래머의 지식은 프로그래밍언어 + 오퍼레이팅 시스템의 명령 + 파일링 시스템의 운영법, 화면설정, 기타 작업도구 사용법을 숙지.

네이버에서 검색해보면.. 리누스 토발스, 미야모토 시게루, 제임스 고슬링, 클리부 배리 몰러등이 검색되네요. 이젠 프로그래머라는 단어는 광역의 단어가 되어버린듯 하므로 대명사가 아닌 소규모의 단어들을 다시 만들어야 할 듯.

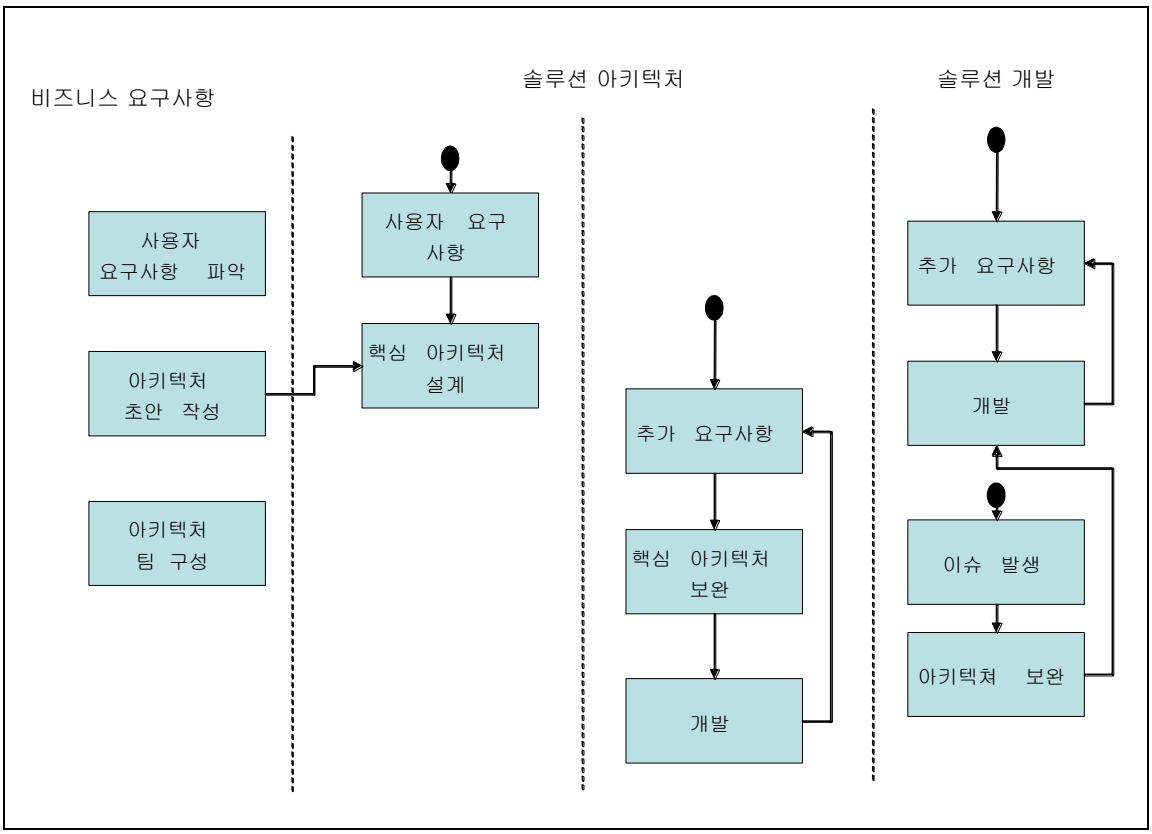
Ps~ 이 내용은 저 혼자만의 생각일 뿐입니다.

7.1 2차 강의에서 할 내용 맛배기~~

A 소프트웨어 아키텍쳐 설계 개요



B 반복을 통한 소프트웨어 아키텍쳐 설계 절차



C 아키텍쳐 관련 산출물의 개요

단계	활동	작업	입력산출물	출력산출물
비즈니스 요구사항	비즈니스 모델 개발	비즈니스 유즈케이스 모델개발		비즈니스 유즈케이스 모델 업무 규칙
		비즈니스 객체 모델 개발		비즈니스 객체모델
	요구사항 파악	요구사항 수집 및 기술	업무규칙	요구사항 정의서 요구사항 추적표
		현행 조직 및 시스템 정보수집		
		시스템 비전 개발	요구사항 정의서 현행운영 정보 분석서	시스템 비전 기술서
솔루션 아키텍처	요구사항 정의	유즈케이스 모델 개발	비즈니스 유즈케이스 모델 요구사항 정의서	유즈케이스 모델
		클래스 모델 개발	유즈케이스 모델	클래스 모델

		UI 프로토타입 수행	유스케이스 모델 클래스 모델	UI 프로토 타입
	요구사항 분석	유즈케이스 모델 분석	유즈케이스 모델 클래스모델	유즈케이스 모델
		클래스모델 분석	유즈케이스 모델 클래스모델	클래스모델 동적 모델
		UI 흐름 정의	유즈케이스 모델	UI 정의서
	아키텍처 설계	아키텍처 관련 요구사항 분석	요구사항 정의서 현행 운영 정보 분석서 업무 규칙 비즈니스 유즈케이스 모델 유즈케이스 모델 클래스 모델 동적 모델 제안서(시스템 구성도)	아키텍처 요구사항 분석서 품질 프로파일 요구사항 정의서 (업무규칙, 시스템 제약사항) 유즈케이스와 품질 관련표 하드웨어 아키텍처
		시스템 컨텍스트 정의	하드웨어 아키텍처	소프트웨어 아키텍처 정의서
		Component & Connector View 작성	아키텍처 요구사항 분석서 하드웨어 아키텍처 소프트웨어 아키텍처 정의서	소프트웨어 아키텍처 정의서 (추가)
		Module View 작성	아키텍처 요구사항 분석서 하드웨어 아키텍처 소프트웨어 아키텍처 정의서	소프트웨어 아키텍처 정의서 (추가)
		Allocation View 작성	아키텍처 요구사항 분석서 하드웨어 아키텍처 소프트웨어 아키텍처 정의서	소프트웨어 아키텍처 정의서 (추가)
		Code View 작성	아키텍처 요구사항 분석서 하드웨어 아키텍처	소프트웨어 아키텍처 정의서 (추가)

			소프트웨어 아키텍처 정의서	
	아키텍처 평가	아키텍처 평가 준비작업	소프트웨어 아키텍처 정의서 아키텍처 요구사항 분석서	테스트 시나리오 소프트웨어 아키텍처 평가 절차서 아키텍처 프로토 타입
		아키텍처 평가	테스트 시나리오 소프트웨어 아키텍처 평가 절차서 프로토타입 소프트웨어 아키텍처 정의서	소프트웨어 아키텍처 평가 결과서
	아키텍처 상세화	아키텍처 스타일 적용	아키텍처 설계 가이드라인	소프트웨어 아키텍처 정의서 (update)
		설계 패턴 작용	의사결정 트리 설계 가이드 라인	소프트웨어 아키텍처 정의서 (update)
		설계 지침 작성	소프트웨어 아키텍처 정의서 의사결정 트리 설계 지침	소프트웨어 아키텍처 정의서 (update) 설계지침(update)
		코드 지침 작성	소프트웨어 아키텍처 정의서	소프트웨어 아키텍처 정의서 코드 표준 코드 템플릿 코드 샘플
	컴포넌트 식별 및 명세	컴포넌트 식별 및 구조화	유즈케이스 모델 클래스 모델 동적 모델 소프트웨어 아키텍처 정의서	컴포넌트 명세서
		인터페이스 식별	유즈케이스 모델 클래스 모델	컴포넌트 명세서

			동적 모델 소프트웨어 아키텍처 정의서	
		컴포넌트 상호 작용 분석	유즈케이스 모델 클래스 모델 동적 모델 소프트웨어 아키텍처 정의서	컴포넌트 명세서
		컴포넌트 명세 작업	컴포넌트 명세서	컴포넌트 명세서
	컴포넌트 평가	컴포넌트 평가 기준	요구사항 정의서	컴포넌트 평가기준
	컴포넌트 조달	선정	유즈케이스 모델 클래스 모델 동적 모델 소프트웨어 아키텍처 정의서	
		컴포넌트 선정 및 구매	컴포넌트 평가 기준 컴포넌트 명세서	재사용 컴포넌트