



영국의 철학자 버틀란트 러셀경이 인식론에서 말했듯이 내 앞에 놓여있는 책상을 인식하지 못한다면 그것은 존재하지 않는 것과 마찬가지로 효과가 있다.

소프트웨어 아키텍처에 있어서 인식해야 할 많은 기법과 품질 요소들, 예를 들어 모듈화, 추상화, 응집성, 개념적 통일성, 구현용이성 등에 대한 인식이 없는 상태에서 구축된 아키텍처는 그러한 성질을 갖지 못할 확률이 높다.

The Way of Architect.

**송재하** raystorm@naver.com

1992년 학교 동아리에서 터보 파스칼을 배우면서 프로그래밍을 시작했다. 1995년에 'Gang of Four'의 디자인 패턴을 보면서 소프트웨어 엔지니어링에 관심을 갖게 됐다. 로코존 웹서비스 연구실에서 EJB 세션번 컨테이너와 C++ CORBA 커널 개발에 참여했고, 현재는 웹 서비스 브로커를 개발하고 있다.

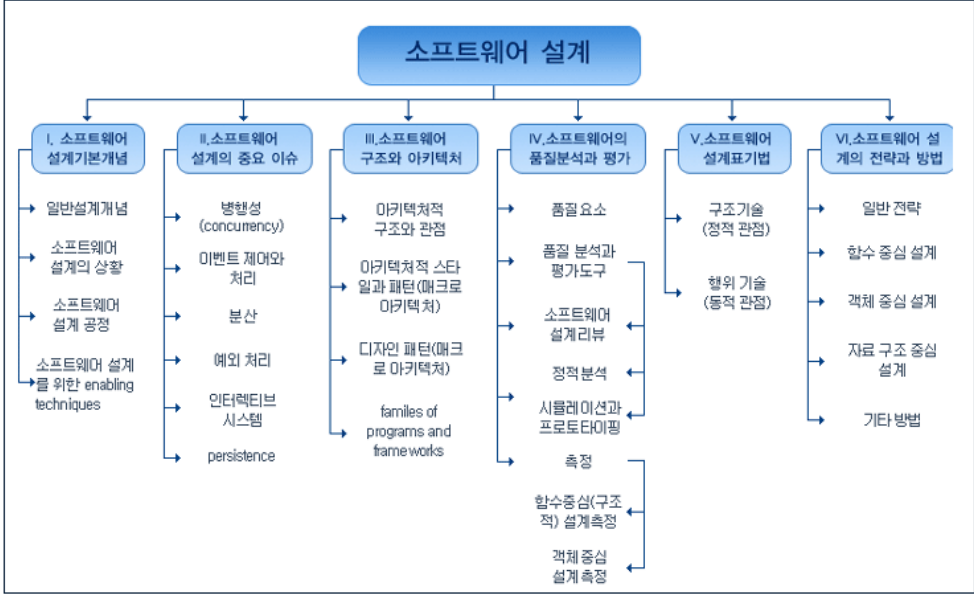
**박성운** redcloth@hitel.net

서강대학교 소프트웨어 공학 연구실에서 석사 과정에 재학 중이다. GoF의 『Design Patterns』, 『Refactoring』, 『POSA』 시리즈를 가장 사랑한다. 분산 환경의 프레임워크에 관심이 많고, 현재는 개미 사회의 행태를 모방한 에이전트 프레임워크에서 패턴 언어를 마이닝하는 연구를 하고 있다. Kent Beck, David L. Parnas와 같은 천재가 되는 게 목표다.

**필**

자들에게 올 한 해 동안 가장 심혈을 기울인 일이 무엇이라고 묻는다면 '개발자의 집단무의식' 연재라고 답할 것이다. 한 회 한 회 연재해 오면서 때론 모자란 능력에 탄식하고, 때론 독자들의 호의적인 반응에 뿌듯해 하면서 한 해가 다 지난 것 같다. 처음 시작할 때는 여러 가지 야심찬 계획으로 다양한 내용을 시도 있게 다루려 했지만 필자들의 모자란 능력으로는 쉬운 일이 아니었다. 이번을 마지막으로 연재를 마친다고 생각하니 애초에 계획했던 내용의 반도 다루지 못한 게 아닌가 싶다. 그런 의미에서 이번에는 먼저 본 연재에서 다루고자 했던 소프트웨어 아키텍처 영역 전반에 대해 소개하겠다. 이것은 소프트웨어 아키텍처 분야에 접근하기 위한 전체 지도에 해당한다. 그리고 그 안에서 우리가 지금까지 다루었던 부분과 그렇지 않은 부분이 되겠어 보겠다. 다음으로 소프트웨어 아키텍처를 구성하는 최고급 개발자인 소프트웨어 아키텍트가 갖춰야 하는 자질에 대해 소개하겠다. 우리는 독자들의 관심이 소프트웨어 아키텍처보다는 '좀 더 나은 개발자로서의 아키텍트', '개발자로서 존엄하게 늙어가기 위한 아키텍트'에 있음을 알고 있다. 그렇게 되기 위해 어떤 자질을 갖추기 위해 노력해야 하는지 알아 보겠다. 마지막으로 필자들의 학습 경험을 바탕으로 개발자들이 소프트웨어 아키텍트로 성장해 나가는 데 필요한 길잡이가 될 만한 서적과 학습 과정을 살펴보겠다. 물론 필자들은 아마추어 아키텍트 지망생들이고, 짧은 연문이지만, 소프트웨어 아키텍처 연재를 마무리하기에는 더 없이 좋은 소재인지라 부끄러움을 무릅쓰고 소개하겠다.

<그림1> 소프트웨어 설계의 지식 영역



**소프트웨어 설계**

소프트웨어 아키텍처는 설계의 일부분이면서 설계를 뛰어넘는 무언가가 있다. 정확히 표현하면 아키텍트-아키텍처를 수립하는 과정-에 필요한 지식은 일반적인 설계를 하는데 필요한 '기술적 지식+알파'의 관계에 있다. 그 알파가 무엇인지는 후반부의 '아키텍트가 갖출 일곱 가지 자질'에서 살펴보기로 하고 여기에서는 세부 설계와 아키텍처를 포함하는 개념으로서의 설계를 마스터하기 위해 필요한 지식 영역에는 어떤 것들이 있는지 살펴보자.

IEEE Computer Society의 전문가들은 여러 가지 기준을 토대로 소프트웨어 설계의 지식 영역을 (그림 1)과 같이 총 여섯 개의 대분류로 나누고 그들을 다시 24개의 소분류로 나누었다. 이 중에서 우리가 다룬 부분은 I의 Enabling Techniques, II의 분산, III의 아키텍처 스타일·디자인 패턴·프레임워크, IV의 품질 요소 등이다. 본 연재에서 주제 선정 기준은 본지 독자층에게 흥미로운 것, 아키텍처와 관련될 것, 누구나 아는 것이 아니라 무의식적으로만 느끼고 있는 것 등의 조건을 모두 통과한 내용이었다. 지금까지의 글들은 단행본이 아닌 기사 형태였기 때문에 이런 조건을 가장 강하게 만족하는 몇몇 특수한 주제만을 다루었다. 하지만 아키텍트가 되길 희망한다면 나머지 모든 주제에 대해서도 깊은 지식과 경험이 있어야 할 것이다. 다음 24개 소분류에 대해 설명이 필요한 몇몇 주제들에 대해 간략히 살펴보자.

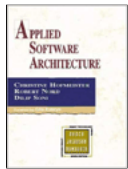
## 소프트웨어 설계 공정

소프트웨어 설계 공정은 크게 아키텍처 설계와 상세 설계의 두 단계로 이루어진다. 이 중 상세 설계 공정은 기존에 많은 책에서 설명하고 있고 대부분의 개발자들이 어느 정도 파악하는 부분이다. 하지만 아키텍처 설계 공정은 자료를 구하기가 쉽지 않다. 현재 공개되어 있는 자료를 살펴보면 (표 1)과 같다. 이 표에 있는 내용 말고도 더 많은 방법이 있지만 필자들이 직접 보고 가장 많이 도움이 되었다고 느낀 것들만 추려서, 좋다고 느낀 순서대로 정렬했다.

표1 아키텍처 프로세스

조직	이름	설명	구할 수 있는 곳
N/A	XP	XP는 엄밀히 말하면 아키텍처라고 명명한 공정은 없다. 하지만 아이러니하게도 아키텍처에 필요한 실질적인 방법을 가장 많이 담고 있다.	<a href="http://www.xprogramming.org">http://www.xprogramming.org</a> 그 외 인터넷 사이트, XP 관련서적
썬	썬톤	썬의 특색에 맞게 서비스 중심의 아키텍처를 제시하고 있다. J2EE 관련업무를 담당한다면 꼭 한번 읽어보라고 권하고 싶다.	<a href="http://www.suntone.org">http://www.suntone.org</a>
오픈그룹	TOGAF	일곱번의 버전업을 거친 검증된 프로세스로 가장 많은 자료가 공개되어 있다. TOGAF 하지만 RUP와 비슷하게 프로세스 프레임워크를 표방하기 때문에 이 내용을 소화하려면 엄청난 내공(경험과 지식)을 필요로 한다.	<a href="http://www.opengroup.org/togaf">http://www.opengroup.org/togaf</a>
래스널	RUP	아키텍처에 관련된 일반적인 내용이 '잡다하게 매우 많이' 있다.	<a href="http://www.rational.com">http://www.rational.com</a>
필립스	Gaudi	자료는 많지만 시스템 아키텍처이기 때문에 소프트웨어 아키텍처에겐 맞지 않을 수도 있다.	<a href="http://www.extra.research.phillips.com/natlab/sysarch">http://www.extra.research.phillips.com/natlab/sysarch</a>

## 아키텍처 구조와 관점



Applied Software Architecture

건축물의 구조를 표현하기 위해 여러 관점에서 도면을 그리듯이, 소프트웨어의 구조도 여러 관점에서 표현하게 된다. 이는 소프트웨어의 복잡도가 인간의 한계를 넘어서기 때문에 독립된 여러 관점으로 분해해서 다루기 위해서다.

널리 알려진 래쇼날의 4+1 관점-유스케이스, 논리, 프로세스, 컴포넌트, 배치의 다섯 가지 관점-은 소프트웨어 중심적인 시스템을 구축할 때 적절한 방법이다. 『Applied Software Architecture』에서 선보인 개념, 모듈, 코드, 실행의 네 가지 관점도 지멘스의 오랜 경험으로 검증된 모델이다. 하드웨어와 결합된 방법으로는 미국방 통신 아키텍처인 SCA(Software Communication Architecture)에서 사용한 소프트웨어 아키텍처, 배치, 애플리케이션, 네트워킹, 보안 관점 등이 있다.

독립된 몇 개의 관점으로 분리해서 설계할 때는 주의해야 할 사항이 몇 가지 있다.

- ◆ 개발하고자 하는 시스템에 적절한 관점을 선택해야 한다. 예를 들어 소프트웨어 중심적인 시스템이라면 4+1 관점이 적절한 것이다. 그 중에서도 위드 같은 독립형 소프트웨어라면 배치 관점은 제외할 수 있다.
- ◆ 각 관점의 내용물 간의 추적성을 유지해야 한다. A라는 유스케이스가 논리 관점의 어느 객체들에 의해 구현되고 그 객체들의 동기화가 어떻게 되는지 프로세스 관점에 등장할 때, 이들은 서로 추적성을 보장해야 한다.
- ◆ 독립적으로 생각할 수 있어야 한다. 각 관점을 서로 조합해야 전체를 이해하는 것이 가능하다면 관점을 도입한 의의가 없어진다. 이러한 관점의 선택과 관점 간의 연결 관계, 독립성 등은 급조할 수 있는 것이 아니다. 따라서 직접 만들어 쓰기보다는 지금까지 많이 사용되면서 검증된 방법들을 사용하는 것이 바람직하다.

## 품질 분석과 평가 도구

아무리 훌륭한 지도가 있어도 현재 나의 위치를 모르면 쓸모없다. 마찬가지로 현재 만들고 있는 아키텍처의 상태를 정확히 평가할 수 없으면 잠재된 위험을 인지하지 못해 많은 문제를 야기한다. 이러한 문제 해결을 위해 여러 가지 방법이 등장했다.

여러 사람이 같이 모여서 하는 리뷰나 체크리스트를 이용해 다시 한 번 살펴보는 확인(inspection)은 이미 널리 알려진 방법이다. 성능이나 안정성처럼 정량적으로 평가할 수 있는 품질 요소는 JInsight나 JMeter와 같은 도구를 이용해 구체적인 품질을 평가할 수 있다. 고객의 눈에 보이는 품질, 혹은 측정하지 않으면 프로젝트 성패에 큰 영향을 끼치는 품질들은 이런 도구들 혹은 수학적 검증 방법들이 존재한다.

하지만 이런 몇몇 눈에 띄는 품질 요소를 제외하면 대부분은 딱히 평가할 수 있는 방법이 없는 것이 현실이다. 변경용이성과 같은 일반적인 품질을 비롯해서 각 시스템마다의 특수한 품질들이 이에 해당한다. 이러한 품질을 평가하기 위한 방법으로 SAAM이나 ATAM과 같은 기법이 등장하고 있다(박스 기사 참조).

## 소프트웨어 설계 표기법

종대 표현력이 높은 언어를 가진 현대인은 단순한 표현만을 사용했던 원시인에 비해 상대적으로 더욱 효율적으로 깊은 사고를 할 수 있다. 마찬가지로 좋은 설계 언어는 설계자가 더 효과적인 사고를 할 수 있는 기반을 제공한다.

표기법은 구조 기술과 행위 기술의 두 가지 소분류로 나뉜다. 둘 다 현재는 UML이 표준이기 때문에 굳이 이 자리에서 다양한 방법을 소개할 필요는 없어 보인다. 하지만 대부분의 아키텍처 그룹은 현재의 UML이 아키텍처를 표현하기에 매우 많이 부족하다고 얘기하고 있고 실제로 그렇다. 현업에서는 부족하더라도 표준을 따라야 하겠지만 UML의 어떤 부분들이 부족하기에 여러 ADL(Architecture Description Language)이 등장하는지 조사해보면 아키텍처의 길로 한걸음 더 다가갈 수 있을 것이다.

ROOM(Real-Time Object Oriented Method)은 대중적으로 가장 성공한 ADL이다. ROOM은 현재 RUP에 Capsule이란 개념으로 포함되어 있고 MS 비지오 같은 도구에 확장팩을 설치해 모델링할 수도 있다. ROOM이 ADL로서 사용되는 이유는 아키텍처 기술에 필요한 컴포넌트, 커넥터, 포트, 프로토콜 등의 모델링 요소를 모두 제공하기 때문이다. ROOM은 이름에서 보듯 주로 실시간 시스템과 내장 시스템 등 제한된 분야에서 사용된다.

## 우리가 가고 싶은 길 : 아키텍처

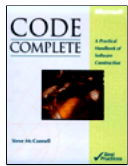
본 연재는 소프트웨어 아키텍처를 겨냥한 글이었지만 독자들의 관심은 아키텍처 자체가 아니라 좀더 성숙한 개발자로서의 아키텍트에 쏠려 있었을 거라 짐작한다. 그렇다. 독자들뿐 아니라 거의 대부분 개발자들의 관심은 스스로를 어떻게 갈고 닦으면 아키텍트의 반열에 오를 수 있을까 하는 데 있을 것이다. 이왕 개발자의 길에 들어섰으니 개발자의 도를 제대로 깨우쳐 아키텍트도 우뚝 서기를 바라는 것이 당연하겠지. 그러나 많은 개발자들의 바람에도 불구하고 우리 주변에서 본격적으로 아키텍트를 길러내는 기관이나 지침이 되는 책은 많이 보지 못했다. 그 이유는 몇 가지로 생각해 볼 수 있다.

첫째, 너무 어려워서일 것이다. 아키텍트를 길러낼 수 있는 사람이나 조직은 누구일 것인가? 마스터 아키텍트(arch-architect)가 있어야 하지 않겠는가? 지금까지 우리 주변에서 그러한 능력을 가진 사람을 본 적이 있는가? 아니, 혹시 스스로를 아키텍트라고 당당하게 말하고 다니는 사람을 몇이나 보았는가? 제다이도 흔치 않은데 제다이 마스터를 어디서 찾겠는가? 감히 제다이 마스터를 자칭할 수 있는 개발자는 정말 드물다.

둘째, 아직 새롭고 복잡한 문제들을 충분히 해결할 만큼의 체계가 잡혀있지 않기 때문이다. 컴퓨팅 역사의 초창기에는 전기공학자로 다 해결됐다. 일이 복잡해지니까 프로그래머가 필요해졌다. 관리자가 나오고, 분석·설계·구현이 나뉘고, 기술 컨설턴트까지 나오고 있지만 아직도 컴퓨팅을 통해 다루는 복잡한 문제를 충분히 해결할 만큼 전문화, 분업화되지 못했다.

우리는 여전히 개발이 잘 안 되어 가면 누군가 능력이 뛰어난 자가 나와서 구원해 주길 바란다. 또 개발 중에 우연히 뭔가 맞아 떨어지는 듯한 분위기(데자뷰)에 전율하기도 한다. 그리고 이런 부분들을 불운이나 행운에 맡기면서 여전히 무의식의 세계에 묻어 버렸다. 앞으로 좀더 체계적인 경험과 지식이 쌓이고 학문적 연구와 결합시켜 '아키텍처적 영역'이라는 이름으로 개발자의 의식 세계로 끌어내 통제할 수 있을 것이다.

셋째, 국내의 특수한 사정도 한몫을 했다. 최근에는 많이 달라졌지만 대부분 기술 위험도가 높거나 매우 복잡한 시스템보다는 단순히 RDBMS를 이용해 데이터를 조작하는 단순한 작업들이었다. 그런 일들은 별 영향함 없이 결정을 해도 탈이 없는 것들이다. 게다가 간혹 있는 초대용량 시스템이나 기술 위험도가 높은 시스템을 개발한다 하더라도 그 개발 사례를 아키텍처적 수준에서 소개하는 경우가 많지 않았다. 그렇기 때문에 이런 영역을 다루는 아키텍트라는 개념에 대한 접근 빈도가 낮았다.



CODE COMPLETE

최근에는 이런 문제가 많이 개선되고 있다. 국내 몇몇 교육기관에서는 특정 분야(예를 들어 J2EE)에 국한해서이긴 하지만 아키텍처 강좌가 개설되어 있고, 몇몇 SI 대기업에서는 내부에 소프트웨어 아키텍처 강좌를 개설해 놓기도 했다. 또 본격적으로 아키텍처를 다룬 여러 가지 글이나 서적이 나오고 있다. 그럼에도 불구하고 여전히 아키텍트를 길러내는 데 대해서는 딱히 어떤 지침이나 조언이 나오지 않고 있다.

그런 의미에서 소프트웨어 아키텍처를 다루는 본 연재에서 아키텍트에 대해 언급해 보는 것이 가치 있을 것 같다. 지난 연재에서 『Code Complete』라는 책에서 소개한 개발자의 태도와 개발자의 자질에 대한 인용을 기억할 것이다(4화의 '게은은 개발자들' 박스기사 참조). 그러나 훌륭한 개발자는 아키텍트의 필요조건이기는 하지만 충분조건은 되지 못한다. 그렇다면 아키텍트가 되기 위해 어떤 요소들이 더 필요할까? 『Applied Software Architecture』라는 책에는 다음과 같이 언급하고 있다.

참고도서 : THE UNIFIED MODELING LANGUAGE USER GUIDE

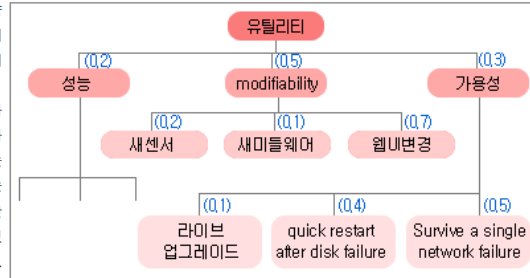
### SAM과 ATAM을 이용한 아키텍처 평가

SEI에서 개발한 SAAM(Software Architecture Analysis Method)(그림)과 같이 우리 시스템이 확보해야 하는 품질 요소를 레벨 2노드들에는 테스트 가능한 시나리오라는 개념을 도입해 변경용이성을 평가시킨다.

가할 수 있는 큰 전기를 마련했다. 이 SAMM을 좀더 발전시켜 모든 품질을 평가할 수 있는 좀더 체계적인 평가 방법인 ATAM은 약 2~3일간 프로젝트 이해당사자들의 대표가 모두 모여 아키텍처의 적합성을 평가하는 방법이다. 자세한 절차는 생략하고 어떤 매커니즘으로 진행되는지 살펴보자.

ATAM은 두 개의 중요한 축을 갖고 있는데 그 중 첫 번째가 SAMM에서 물려받은 시나리오다. 우리는 흔히 '우리 시스템의 아키텍처는 견고하고 유연해서 변화에 잘 대처할 수 있어야 한다'는 문구를 아키텍처 문서에서 보게 된다. ATAM에서는 이런 문구는 평가 불가능하다고 보고 이를 평가 가능한 시나리오로서 정제한다. 예를 들어 '데이터베이스를 A사 제품에서 B사 제품으로 바꿀 때 변경되는 컴포넌트의 개수는 몇 개인가?', '요구사항 2.3.1.항이 변경됐을 때 수정해야 하는 컴포넌트 개수는 몇 개인가?' 등이 이에 해당한다. 물론 이런 시나리오를 추출하는 것과 실제로 평가하는 것 모두 어려운 작업이다. 하지만 아예 평가를 하지 않고 각 이유별로 테스트 가능한 시나리오를 여러 개 기술한다. 각각의 시나리오별로 평가를 하면 현재의 아키텍처가 이 품질 요소들을 얼마나 잘 지원할 수 있는지 가시적으로 볼 수 있게 되는 것이다.

<그림> 품질 요소 유틸리티 트리



“개발할 제품의 비전을 수립하고, 그 비전을 달성하기 위해 어떠한 고난도 감내할 용기를 가지며, 확신을 가지고 전체 개발자들에게 비전을 이룰 수 있을 것이라는 믿음을 줄 수 있도록 설득할 수 있는 능력이 있어야 한다.”

필자는 이 내용을 보면서 삼국지의 제갈공명을 떠올렸다. 하늘에 달을 만한 학문을 가지고 비전을 펼치기 위해 세상에 출사해 대의를 밝혀 우리를 통솔해 훈련시키고, 후계자를 발굴해 교육시키고, 모든 역경을 극복해 나가는 모습을 생각해 보자. 꿈을 가지고 어떠한 고난이 있더라도 주위 사람들을 복돋우면서 목표를 향해 나가는 영웅! 아키텍트를 꿈꾸는 개발자는 스스로 우리 세계의 제갈공명이 되고 싶어 하는지도 모르겠다. 삼국지의 제갈공명처럼 개발자 세계의 아키텍트도 기술적 능력(technical skills), 영도력(leadership), 소통 능력(communication skills), 인사 능력(people skills) 등의 능력을 요구하고 있다.

아키텍트가 갖출 일곱 가지 자질

앞서 언급한 「Applied Software Architecture」에서는 아키텍트가 수행해야 하는 여러 가지 역할을 좀더 정리해서 '아키텍트의 일곱 가지 역할(7 Roles of Architect)'이라는 이름으로 소개하고 있다. 아키텍트에게는 어떤 자질이 있어야 하는지는 일곱 가지 역할을 통해 짐작해 볼 수 있다. 우리 시대, 우리 세계의 영웅이 되고 싶다면, 개발자들의 제갈공명이 되고 싶다면, 여기 소개하는 일곱 가지 역할을 잘 수행하기 위한 자질을 갖추도록 알고 닦아야 한다.

**1. 비전 제시자(creating vision) :** 시대의 기술 혁신 동향을 잘 읽어내야 한다. 시스템의 전체적 인 요구사항과 제약조건을 파악해 비전을 만들어 내고, 그 비전을 효과적으로 전달할 수 있어야 한다. 만들고자 하는 제품의 요구사항과 제약사항을 잘 파악하고 이를 바탕으로 시스템을 바라 보는 눈이 있어야 한다. 시장 동향도 파악해야 한다. 무엇보다도 창의력이 있어야 한다. 이런 역할은 차라리 기획이나 영업 부분에 가까워 보인다. 대부분의 개발자들이 이런 부분에 대해 가장 막막하게 느낄 것이다. 그러나 진정한 아키텍트라는 이런 역할을 잘 수행하기 위한 자질을 갖춰야 한다. 그것이 개발자가 단순 노동자 이상의 그 무엇이 되는 길이다. 제갈공명이 천하상 분지계, 즉 천하를 위·촉·오의 셋으로 나누어 한 왕조 재건을 도모한다는 큰 비전을 제시하지 못한다면 다른 모든 능력은 잡기술에 지나지 않는 것이다. 아키텍트의 나머지 역할과 자질, 즉 가르치고 조정하고 의사 결정하고 구현하는 데 필요한 역할과 자질은 이 비전을 효과적으로 실현하기 위해 필요한 것들이다. 비전 제시자로서의 자질을 키우기 위해 어떤 방법으로 알고 닦아야 하는지 고민해야 한다.

**2. 핵심 기술 조언자(the architect as a key technical consultant) :** 프로젝트 관리자와 아키텍트의 역할은 (표2)와 같이 구분된다. 아키텍트는 관리자에게 기술적인 영역에서 조언을 주는 존재다. 마치 군주에게 군사에 관한 부분에 대해 조언을 하는 군사(軍師)와 같다. 훌륭한 군사 조언을 위해서는 현실에 바탕을 두고 각종 병법과 이론에 기반해야 한다. 물론 작은 나라에서는 군주가 직접 군사 역할까지 하지만 좀더 체계가 필요한 규모에서는 역할 분담이 필요하다.

표2 관리자 와 아키텍트의 역할

항 목	프로젝트 관리자	소프트웨어 아키텍트
소프트웨어 개발	프로젝트 조직구성, 자원, 예산, 일정관리	설계와 관련된 팀 구성, 의존관계관리
요구사항	마케팅 부서와 협상	요구사항을 검토하고 협상
개인사항	고용, 고과, 임금 등 조직원들에게 동기부여	지원자들과의 면접, 구성원들의 기술능력 제고, 개발팀에 동기부여
기술	아키텍트의 권고에 따라 새로운 기술 도입	기술, 훈련, 도구 등을 권고
품질	제품의 품질 보장	설계의 품질 유지
측정	생산성, 크기, 품질	설계 목표에 부합 여부

**3. 의사 결정자(The architect makes decisions) :** 설계 팀을 이끌고 전체 설계에 영향을 미치는 초기 단계의 중요한 설계 결정을 한다. 설계할 시스템에서 중요한 트레이드오프들을 파악해 내고 아키텍처를 구성할 때 이를 반영해야 한다. 결정을 내리기 위해 도메인 지식도 어느 정도 필요하다. 의사 결정시 위험 요소와 시간적 제약을 고려해야 한다. 전쟁 중에 군사의 의사 결정에 대한 책임은 매우 막중하다. 한 번의 실수로 국가 전체가 멸망할 수도 있기 때문이다. 이런 실수를 막기 위해 전쟁 전에 미리 작전을 세우고 군대 편제와 규모, 공격력과 동원 무기체계 등에 대한 결정들을 신중하게 내려야 한다. 물론 미리 이런 계획과 의사 결정을 내릴 뿐만 아니라 진행 중에도 상황에 따라 제때에 적절한 의사 결정을 내려야 한다. 심지어 XP 같은 프로세스에서는 '계획은 무의미하다. 대신 계획하는 과정은 대단히 의미 있다'라고까지 말하면서 제때에 상황에 맞게 과거의 결정사항을 수정하고 새로운 의사 결정을 내려야 한다. 다양한 종류의 사안 에 대해 논리적 근거를 밝히고, 그런 사안들끼리의 상관관계를 고려하기 위해서는 다양한 경험 과 이론적 기반을 필요로 한다. 이를 기반으로 즉흥적이고 임의적인 의사 결정을 배제하고 논리적 근거와 타당성을 바탕으로 예측 가능한 결론을 끌어낼 수 있는 의사 결정을 할 수 있어야 한다.

**4. 코치(The architect coaches) :** 팀 구성원들과 대화의 통로를 열고 이를 통해 설계한 아키텍처를 가르친다. 아키텍처를 설계한 내용을 이해시켜 주고 그들의 의견을 수렴, 반영한다. 팀원들이 아키텍처에 설계해 놓은 아키텍처의 틀 안에서 세부 설계를 해 보면서 설계 능력을 키우도록 지도해야 한다. 삼국지의 제갈공명은 모두 처음부터 큰 우리를 거느리지 않는다. 처음에는 핵심 멤버 몇몇과 함께 전체 조직의 틀을 짜고, 새로운 참가자가 생길 때마다 그들에게 기존 조직체 계를 이해시키고 의견을 반영해 나가면서 조직을 점점 더 탄탄하게 만들어 간다. 군사는 휘하의 우리들에게 전투기술과 조직 운용을 가르쳐 좀더 성장할 수 있도록 해 준다. 누군가를 성장시켜 줌으로써 강력한 리더십을 발휘할 수 있게 된다. 이런 리더십을 통해 의사 결정자로서 입지도 강화된다.

**5. 조정자(The architect coordinates) :** 아키텍처에 영향을 미칠 수 있는 사람들의 활동을 조정, 중재한다. 아키텍처에 의해 영향을 받을 사람들의 활동을 조정한다. 설계 통합성을 유지한다. 아키텍처에 따라 설계가 진행됨을 보장한다. 많은 프로젝트에 불화가 존재한다. 그리고 때로는 이런 불화가 프로젝트 성패를 좌우하는 경우까지 있다. 서로 충돌하는 의견들을 기술적 입장에서 중재할 사람이 필요하다. 기술적인 문제에서 나온 불화를 해결하기 위해서는 기술적 지 도력 이 있는 사람의 권위 있는 중재가 필요하다. 지도력이나 중재력 같은 것들이 단순히 기술만 뛰 어난 개발자에 대해서 아키텍트에게 요구되는 '훌륭한 인격과 원만하게 사람 대하는 기술'을 가진 대인 능력적 자질이다.

**6. 구현 능력(The architect implements) :** 새로운 기술을 도입하면 설계에 어떤 영향을 미칠지 고 려한다. 자차원의 세부사항을 보면서 초기 개념을 확인할 수 있다. 프로토타입을 만들어서 설계 결정을 평가할 수 있다. 구현 위험을 최소화하기 위해 가능 한 덩어리를 완전히 구현할 수도 있다. 개발자들에게 구현 모 델을 제시하기 위해 샘플 컴포넌트 하나를 구현할 수도 있다. 칼잡이들에게 말이 먹히기 위해서는 기막히게 칼질하는 모습을 직접 보여주는 수밖에 없다. 아 우리 공 부를 많이 하고 이론에 해박해도 최고로 구현을 잘 하는 사람의 한마디 말 앞에 무너진다. 최고의 개발자가 최고의 아키텍트가 될 가능성이 높을 수 밖에 없다. 물론 여기에 과도하게 집착한 나머지 칼질만 배우고 있으면 곤란하다. 칼잡이는 언제나 칼잡이일 뿐이다. 아더 왕은 랜슬롯보 다 칼을 잘 쓸 필요는 없다. 단지 랜슬롯이 무시하지 못할 만큼의 무공을 갖추면 된다.

**7. 대변자(The architect advocates) :** 소프트웨어 아키텍처에 대한 투자를 대변한다. 소프트웨어 공정에 소프트웨어 아키텍처를 포함하도록 한다. 계속해서 새로운 소프트웨어 아키텍처 기술을 평가하고 도입하도록 주장한다. 많은 고승들이 계율을 파괴하고, '부처는 똥 막대기' 식의 언 사를 해 왔으나 그것은 어 디까지나 위대이지 않는 공극의 진리를 표현하기 위한 방법이었을 뿐, 실제로는 계율을 존중하고 깨달은 자들에 대한 존경심을 버리지 않았다. 아키텍트는 아키텍처에 대해 끊임없이 투자할 것을 주장해야 한다. 아키텍처에 대한 투자가 개발자들이 처한 절박한 현실 모순을 해소해 줄 것이기 때문이다(박스 기사 참조).

이렇게 아키텍트의 일곱 가지 역할을 통해 아키텍처가 가져야 할 자질에 대해 살펴봤다. 이 자질들을 어떤 순서로 어떻게 닦아 나가야 하는지에 대해 필자들은 잘 알지 못한다. 많은 개발자들이 사업적인 요소와 교육자적인 요소를 무시하고 구현 능력만을 최고의 가치로 추구하는 경향이 있다. 그러나 이것만으로 자기 만족을 이룰 수 있을지는 모르겠으나 아키텍트로 대성하기는 힘들다고 본다.

다음에 소개할 내용은 아키텍트로 가고자 하는 초급 개발자들에게 도움이 될 만한 과정이라 생각한다. 이 과정은 아키텍트의 한 단면인 개발자와 기술 전문가 정도의 능력에 대해서만 언급할 뿐, 좀더 어렵고 고차원적인 분야에 대해서는 역시 필자들이 다루기 어려운 부분이다. 일곱 가지 자질 중에서 어느 한 가지도 만족스럽게 연마하지 못한 필자들이지만 이 부분은 특히 어려운 부분이다. 이 부분에 대한 개발 과정은 본지와 같은 기술 잡지보다는 다른 곳에서 좀더 쉽게 찾을 수 있을 것이다.

## 아키텍트의 이데올로기

어떤 관념형태를 본인의 사회적 기반과 관련시켜 이해하는 것을 이데올로기라 한다. 이런 이데올로기가 등장하는 이유는 그 사회의 당면 문제를 해결하고자 하는 간절한 바람 때문이다. 성리학 이데올로기가 성립한 이유는 송대의 혼란한 사회상황을 극복하기 위한 간절한 바람 때문이었다. 우리나라에 성리학 이데올로기가 도입된 것도 마찬가지로 이유에서였다. 마르크스주의가 등장한 이유는 산업화 과정에서 빚어진 자본주의의 모순을 해결하고자 하는 간절한 바람 때문이었다. 소프트웨어 개발자들도 간절하게 해결을 바라는 문제의식을 가지고 있다. 새로운 기술이 하루가 다르게 등장하고 있지만, 실제 개발 현장에서는 그 많은 기술을 적용하기는 커녕 내용을 파악하기도 힘들다. 개발을 하면 할수록 전체 기술 발전에는 소외돼 같은 일만 반복적으로 하게 되고, 결국 연륜이 미덕이 아니라 부담으로 치부된다. 이렇게 새로운 기술과 엄청난 요구사항 속에서 35세를 전후해 개발 일선에서 은퇴한다. 그리고 싫든 좋든 뒤늦게 별다른 교육도 받지 못하고 관리자나 영업 쪽으로 방향을 돌리게 된다. 이런 암울한 미래는 소프트웨어 개발자들에게 절박함으로 다가온다.

이런 시점에서 아키텍트의 일꾼이자 자질과 같은 개발자 역할론은 우리에게 이데올로기로서 다가온다. 아키텍트의 역할에 충실하기 위해서는 구현만이 아니라 시장 상황을 제대로 파악하고, 그에 맞는 제품을 개발할 수 있도록 제안하는 능력을 가져야 한다. 아주 높은 수준의 구현 능력과 기술 우위를 가지기 위해 끊임없이 갈고 닦아야 한다. 자신의 성취를 다른 개발자들에게 편하게 나눠주고, 좀더 높은 수준의 개발자가 되도록 이끌어주면서 서로 서로 개발자로서의 완성을 향한 수행 공동체를 만들어 간다. 최종적으로는 아키텍처와 아키텍트에 대한 관심과 투자를 끊임없이 주장함으로써 스스로의 가치를 높이고 이 이데올로기 자체를 지켜 나가도록 만든다. 이 이데올로기를 통한 개발자들은 관리자나 영업조직에 편입되지 않고 스스로의 길로 계속 발전해 나갈 수 있다는 희망을 가지게 된다. 대형 프로젝트에서 관리자와 아키텍트의 역할이 명확히 분리되고, 기술적인 문제에 대한 권한과 권위를 가질 수 있을 것이라 말하고 있다. 아키텍트 이데올로기는 우리 개발자들이 계속 갈고 닦아 3D 노동자에서 탈출하고 마침내 정치 지도자에게 가르침을 주는 고고한 종교 지도자가 될 것을 부추기고 있다.

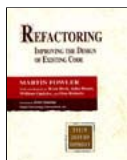
## 아키텍트로 가는 길 : 학습 가이드라인

필자들이 소프트웨어 아키텍처에 대해 본격적으로 공부를 시작한 것은 1999년 3월 필자들이 첫 조우하게 된 스터디를 통해서다. 현재 그 스터디는 계속 발전해서 오브젝트월드(ObjectWorld, <http://www.objectworld.org>)라는 4개 스터디 연합체로 발전했다. 여기에서 소개할 서적들의 평균 학습 순서는 주로 스터디 멤버들이 '술터디(!)'라는 소리를 들어가며 수차례 토론한 내용을 토대로 작성했다.

이후의 내용은 어느 정도의 개발 능력을 갖추고 있다는 전제 하에 작성됐다. 지금까지의 연재를 계속 관심 있게 봐온 사람이라면 그러리라고 생각한다. 만약 그렇지 않다면 먼저 구현 능력을 키우자. 몇 안 라인 이상 구현한 경험이 없으면서 이 과정을 수행하면 많은 부작용이 생길 수 있으므로 추천하고 싶지 않다. 끊임없이 도에 대해 고민한 사람은 붓다와 1분만 대화를 해도 커다란 깨달음을 얻을 것이다. 하지만 필자와 같은 사람은 하루 동안 대화를 해도 지루하기만 하고 얻는 것이 많지 않을 것이다. 심지어 붓다의 비유적인 말씀을 잘못 해석해 그릇된 길로 들어설 가능성도 배제하지 못한다.

## 설계를 익히자

소프트웨어 아키텍처를 공부하려면 먼저 설계를 할 수 있어야 한다. 그러기 위해 익혀야 할 내용은 리팩토링(refactoring), 디자인 패턴, UML 등이다. 공부하는 순서는 나열된 대로가 적절한 듯 하다. 사실 어떤 순서로 읽어도 상관 없지만 경험 상 이 순서가 가장 효율적이다. UML은 표기법이기 때문에 가장 재미가 없다. 그렇기 때문에 UML부터 공부하면 이 분야에 흥미를 잃을 수 있다. 먼저 리팩토링과 패턴을 통해 설계에 흥미를 느끼는 것이 필요하다. 리팩토링은 패턴보다 나중에 등장했지만 순서상으로는 먼저 보는 것이 좋다. 둘 다 독립적인 서적이기 때문에 패턴을 보기 위해 리팩토링을 봐야 한다거나 하는 규칙은 없다. 단지 리팩토링이 디자인 패턴에 비해 훨씬 쉽고 개발자에게 직접 와 닿는 내용이 많기 때문에 먼저 보는 것이 효과적이다.



Refactoring

리팩토링은 한 가지 책-「Refactoring」-밖에 없으니까 서적에 대해서는 가이드할 내용은 없다. 한글판이 있으니 한글판을 보는 것도 좋겠다. 이 책은 자바월드(<http://www.javaworld.com>) 2000년 9월에 자바 개발자가 봐야 할 네 권의 바이블 중 한 권으로 꼽힐 정도로 평이 좋은 책이다.

이 책을 읽는 가장 좋은 방법은 순서대로 읽되, 3장에서 나오는 개념을 공부하면서 각 개념에서 해결책으로 제시한 리팩토링 카탈로그(6~11장)를 참조하면서 동시에 읽는 것이다. 개념과 리팩토링 카탈로그를 분리해서 읽지 말고 반드시 둘을 연결시켜 읽기를 추천한다.

이 책을 다 읽었으면 실제로 리팩토링을 여럿이 실습해 보는 것도 좋다. 답이 없는 문제인 만큼 격렬한 토론이 벌어질 텐데 그 때 얻어진 내용은 소중한 경험이 될 것이다.

필자들이 실습했던 리팩토링 코드는 <http://www.objectworld.org/uml2/RefactoringExample.zip>에서 받을 수 있다.

디자인 패턴을 볼 때는 반드시 「GoF의 디자인 패턴」을 보도록 하자. 한글판이 출판됐으니 한글판을 보는 것도 좋을 것이다. 최근 패턴의 열풍을 타고 비슷한 책이 많이 나와 있다. 특히 자바를 예제로 해서 쉽게 풀이해 놓은 책이 많이 등장했다. 그 모든 책을 필자들이 다 보진 못했지만 여러 스터디들이 원전 대신 그런 책들을 사용해 공부하는 경우를 보았다. 결론은 1편만 2편이 없다는 영화의 법칙이 여기서도 맞는다는 것이다. 이미 GoF에서 너무나 훌륭한 얘기들을 거의 다 해 버렸기 때문에 다른 책들은 아무리 잘 해도 해설서 정도에 머물고 만다.



Design Patterns

아키텍트가 되길 바란다면 패턴의 원전인 GoF에서 소개하는 설계철학과 23가지 패턴을 충분히 숙지해서 언제든 출수(出手)할 수 있을 정도가 되어야 한다. 독학이 어렵다면 스터디를 통해 공부하는 것을 추천한다. 구체적인 방법은 <http://www.industriallogic.com/papers/learning.html>을 참고하라.



Pattern Hatching

스터디를 통해 공부했을 때의 또 다른 효과는 각각의 패턴에 대한 이해도가 훨씬 깊어진다는 것이다. 필자들은 모두 스터디를 통해 만나기 전에 이미 이 책을 다 읽은 상태였다. 그리고 스터디를 통해 다시 공부했다. 스터디를 통해 공부한 게 최소 세 배는 이해도가 높아졌다고 느꼈다. 마지막으로 한 가지 더 조언을 하자면 패턴 책에는 대부분 패턴의 부작용이나 패턴을 올바르게 적용하는 방법에 대해서는 나와 있지 않다. 그래서 필자들도 패턴을 오용하고 남용한 기억이 꽤 많다. 이에 대한 방법이 리팩토링과 XP의 간략한 디자인(simple design)에 있으니 반드시 같이 공부하기를 권한다.

「Pattern Hatching」은 GoF의 활용서다. 두께가 얇으니까 복습 삼아 보면 GoF를 좀더 깊이 있게 음미할 수 있다. GoF를 봤다면 「PloPD」 시리즈도 GoF에 뒤지지 않는 훌륭한 책이니 여유가 되면 보는 것도 좋을 것이다.

## 참고도서

Pattern-Oriented Software Architecture, Volume 1  
Pattern-Oriented Software Architecture, Volume 2

UML을 처음 공부한다면 『UML Distilled』가 가장 좋다. 다른 좋은 책도 많지만 표기법을 익히는 것이니만큼 책이 두꺼우면 금방 재미를 잃게 된다.

『UML Distilled』는 얇아서 가볍게 읽을 수 있고 내용도 흥미를 잃지 않게 잘 썼다. 특히 저자의 파울러가 UML 이외의 내용을 간간히 섞어 넣는데 그 내용도 읽다 보면 시간 가는 줄 모르게 재밌는 주제가 많다. 이 책으로 전체적인 감을 잡았으면 그 다음에는 두 가지 선택이 있다. 뒤에서 소개할 OOAD를 공부하거나 UML에 대해 좀더 깊이 보는 것이다. UML을 실무에서 사용할 수 있을 정도로 깊이 보려면 『UML User Guide』를 추천한다.



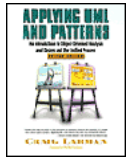
UML Distilled 2/e

그 외에도 설계에 대해 좋은 서적이 많이 있다. 여기 나열한 내용은 설계를 익히기 위한 필수조건이지, 충분조건은 아니다. 충분조건까지 만족하려면 Agile Modeling, XP, 여러 기본 원칙에 대한 서적들, CRC 카드 등 더 많은 것을 공부해야 한다. 하지만 이런 내용은 개개인마다 다르고 앞에서 열거한 내용을 공부하고 나면 이런 가이드 없이 본인 스스로 어떤 라인으로 공부할 것인지 판단할 수 있을 것이다.

### 좀더 체계적인 방법을 익히자

앞에서 언급한 것들을 공부하고 나면 대부분의 사람들이 “좀더 구체적이고 체계적인 방법이 없을까?”라는 질문을 던지게 된다. 이 때 공부할 것이 객체지향 분석/설계(OOAD)를 익히는 것이다. 앞에서 익힌 것은 좋은 설계에 대해 배운 것이고, OOAD에서는 이를 더 체계적으로 접근하는 법을 배우게 된다. OOAD는 아키텍트에겐 또 다른 의미로 중요하다. 보통 개발 절차에서 먼저 아키텍처를 설계하고 그 다음에 상세 설계를 하게 된다. 물론 이 과정이 여러 번의 반복을 거치다보면 그 경계가 모호해지지만 이 둘의 역할을 명확히 이해하는 것은 아키텍트의 필수 자질이다.

OOAD에서 익힐 것은 단순히 절차뿐만이 아니다. 앞에서 공부한 UML을 실제로 도구를 사용해서 쓰는 것 또한 익혀야 하기 때문에 책만으로는 공부하기엔 무리가 있다. 추천하고 싶은 방법은 먼저 도구를 선택하고 그 도구를 이용해 OOAD를 강의하는 교육기관에서 체계적으로 익히는 것이다. 장점은 전체적인 그림을 쉽게 잡을 수 있다는 것과 도구를 익히는 데 걸리는 시간을 최소화할 수 있다는 것이다. 필자들의 OOAD 강의 경험에 의하면 깊은 지식은 5일에 전달하는 것은 불가능하다. 따라서 아키텍트를 희망하는 사람이라면 교육 후에 좀더 깊이 공부하는 것이 반드시 필요하다. 개인적으로는 레노탈의 OOAD 교육 교재가 가장 좋다고 생각한다. 하지만 교육을 받지 않으면 구할 수 없는 점이 아쉽다.



APPLYING UML AND PATTERNS(2/E)

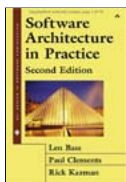
좀더 학문적인 차원에서 OOAD를 공부해 보고자 하는 사람들에게는 『Applying UML and Patterns』를 추천한다. 이 책은 개발방법론, 특히 Unified Process를 축으로 요구사항 추출, 객체지향 분석, 객체지향 설계, 이들 간의 통합을 주 내용으로 하는데 여러 대학에서 강의교재로 채택해 사용할 만큼 쉽게 쓰여 있다. 요구사항 추출에 유스케이스를 어떻게 사용하는지, 객체지향 분석에서 각 객체는 어떻게 뽑아내는 지, 각 객체에 행위를 할당할 때 어떠한 기준을 사용해야 하는지 등 여러 기법을 자세히 쉽게 설명하고 있다. 보통 UML 같은 설계 표기법을 공부하고 나서 분석/설계를 해보려고 하면 막막해짐을 느끼게 되는데, 이는 단순히 문장을 안다고 좋은 문장을 만들어 낼 수 없는 것과 같은 이치다. 『Applying UML and Patterns』는 바로 문장을 만드는 방법을 알려주는 책으로 객체지향 분석/설계에 대한 입문 혹은 수련 수단으로 적합하다고 할 수 있다. 그리고 저자의 사이트에는 대학 강의시 사용할 목적으로 만든 시험 문제들도 있으니 학창 시절로 돌아간 기분으로 풀어보는 것도 나쁘지 않으리라 본다.

OOAD를 익힌 다음에는 RUP와 XP 같은 소프트웨어 개발 프로세스를 익히는 것이다. 엄밀히 말해 이 부분은 아키텍트의 필수 지식 영역에 해당하지는 않는다. 하지만 소프트웨어 아키텍트에 대한 권위 가 명확하지 않고 방법론 전체를 꿰뚫어서 적재적소·적시에 아키텍트를 배치하지 못하는 경우가 많은 것이 국내의 현실이다. 이런 상황에서는 아키텍트 스스로가 자신이 나서야 할 때와 책임져야 할 때를 확신을 가지고 판단할 수 있는 능력이 필요하다. 그러기 위해서는 아키텍트에 어느 정도까지는 개발 프로세스에 대한 지식과 경험이 필요하다. 적절한 책의 선택은 본인이 속한 조직에서 주로 사용하는 방법론이 무엇인가에 달려있다. 각 방법론별로 바이블로 꼽히는 책이 있을 것이다. 그 책을 공부하고 프로세스는 책만 보서는 익히기 힘들다 반드시 파일럿 프로젝트(소규모로 시험 삼아 해보는 프로젝트)를 병행하기 바란다.

이렇게 해서 점점 더 높은 추상 수준을 다루게 되는 것은 좋은 일이다. 그러나 언제나 잊지 말자. 우리는 개발자이다. 개발자들이 가장 싫어하는 사람은 아무 것도 모르면서 말만 떠드는 사람이다. 아키텍처가 가장 적나라하게 반영된 모습이 바로 코드라는 점을 잊지 말자. 그런 점에서 코드 구축에 대해 실증적이고도 체계적으로 다루고 있는 『Code Complete』의 일독을 권한다. 아키텍처에 대한 다른 어떤 책들보다 좋은 아키텍처에 대해 더 깊이 음미해 볼 수 있을 것이다.

### 아키텍처의 기본 소양을 익히자

이제 본격적으로 아키텍처란 단어가 들어간 책들을 살펴 볼 차례다. 가장 먼저 공부할 것은 아키텍처의 기본 소양을 전반적으로 다룬 책들의 지식을 흡수하는 것이다. 본인의 내공이 어느 정도 된다 면-지금까지의 연재 내용 정도는 이미 알고 있었던 독자라면 이 단계는 뛰어 넘을 수 있다.



Software Architecture in Practice

추천하고 싶은 책은 『Software Architecture in Practice』다. SEI에서 나온 서적답게 적절히 실용적이면서 적절히 이론적이다. 난이도도 그렇게 높지 않고 [http://www.sei.cmu.edu/ata/products\\_services/sap\\_tutorial.html](http://www.sei.cmu.edu/ata/products_services/sap_tutorial.html)에서 이를 분량의 강의 교재도 받을 수 있다. 내년 봄에 2판이 출간 예정이니가 구입은 잠시 뒤로 미루자. 학구적인 내용을 필요로 한다면 『Software Architecture』를 추천한다. 카네기 멜론 대학의 교수들답게 전반적인 이론의 기반을 200쪽이 약간 넘는 얇은 책에 잘 담아내고 있지만 후반부는 지나치게 학구적인 듯해서 아쉬움이 남는다.

참고도서 : PATTERN LANGUAGES OF PROGRAM DESIGN

이 단계에서 기본적인 이론보다는 좀더 실제적인 내용을 보고 싶다면 『Applied Software Architecture』를 추천한다. 이 책은 아키텍처를 수립하기 위한 아키텍팅에 관한 내용을 좋은 예제들과 함께 잘 다루고 있다. 하지만 이 책은 아키텍처에 관한 기본적인 소양을 쌓기 위한 책이 아니다. 책의 저자들이 소속된 조직에서 사용하는 독특한 아키텍팅 방법을 소개하는 책이다. 앞에서 소개한 ‘아키텍처의 자질’은 이 책 마지막 장인 ‘소프트웨어 아키텍처의 역할’에서 발췌한 것이다. 여기서 아키텍처가 어떤 소양을 갖춰야 하고 프로젝트에서 어떤 역할을 수행해야 하는지, 개발자의 길을 가는 나는 어떻게 성장해 나가야 할지에 대해 무언의 가르침을 주고 있다. 이 마지막 장을 읽으면 피가 뜨거워짐을 느낄 것이다!

### 좋은 아키텍처를 살펴보자

설계도 마찬가지로 좋은 아키텍처를 만들어 내기 위해서는 먼저 선배 아키텍트들은 어떻게 했는지를 살펴봐야 한다. 바로 패턴이다. 디자인 패턴에 GoF가 기준으로 군림 하듯 아키텍처 패턴에서는 절대적인 바이블이 두 권 있다. 『POSA 1』과 『POSA 2』이다. 그 외에 『PLoPD』 시리즈에도 좋은 아키텍처 패턴이 많이 나온다.

『POSA 1』은 계층, MVC, 리플렉션(reflection) 등과 같이 특정 문제 영역에 국한되지 않은 아키텍처 패턴들과 이를 지원하는 디자인 패턴들로 구성되어 있다. 모두 필수적으

로 봐야 하는 내용이니깐 하나도 빼놓지 말고 다 읽기를 바란다.



Core J2EE Patterns

『POSA 2』는 오픈소스 네트워크 프레임워크인 ACE와 역시 오픈소스 실시간 CORBA ORB인 TAO를 구축하면서 발견되거나 사용된 아키텍처 패턴, 디자인 패턴, 이디엄(Idiom : 특정 언어에서만 사용되는 디자인 패턴)을 모은 책이다. 네트워크와 동시성을 다루는 도메인에서 사용되는 패턴들을 엮어 하나의 거대한 패턴 언어를 구성했다. 이 도메인이 주 업무라면 아키텍트에 관심 없더라도 반드시 읽어보라고 권하고 싶다.

여기까지 봤으면 그 다음에는 본인이 속한 업무와 관련된 아키텍처를 다룬 책을 보는 것이 좋을 것 이다. J2EE 개발자라면 『Core J2EE Patterns』와 같은 책이 좋겠다. 엔터프라이즈 환경 개발자라면 파울러의 『Patterns of Enterprise Application Architecture』를 강력히 추천한다. 아쉽게도 J2EE 와 같이 광범위하게 널리 쓰이는 기술이 아니면 하나의 도메인만을 다룬 책은 거의 없다. 그런 경우 라면 『PLoPD』 시리즈에 있는 패턴들을 선별해 읽거나 직접 논문들을 읽는 수밖에 없다. 패턴과 관련된 논문이 발표되는 학회는 산하리 그룹 (<http://hillside.net>)에서 확인할 수 있다.

참고도서 : Evaluating Software Architectures

### 경험을 쌓자!

현재 필자들이 이 단계에 머물러 있기 때문에 지금부터의 글은 불완전할 수 있고 어느 정도의 추측 이 포함되어 있다는 점을 미리 밝혀둔다.

한 때 유행한 모 CF 중에 열심히 진군한 군대의 장군이 “여기가 아닌가봐”하는 코믹한 장면이 있었다. 아키텍트가 그러면 주변 사람들은 상당히 괴롭다. 아이러니하게도 아키텍트는 필연적으로 이 실수를 많이 한다. 아키텍트의 본질적인 특징인 선형적(a priori) 성격 때문이다. 선형적이란 것은 마치 선생이가 호흡법을 배우지 않았어도 스스로 호흡하는 방법을 이미 터득하고 있는 것처럼 경험 하지 않고도 이미 알고 있는 것을 말한다. 아키텍트는 자신이 설계한 아키텍처가 세부 설계에 어떤 영향을 미쳐서 최종적으로 품질 요소들이 어떤 미묘한 균형상태를 이룰 것인지 알고 있어야 한다. 당연히 이는 매우 난해하다.

아키텍트의 능력을 키우는 방법은 많은 경험을 쌓는 것이다. 비슷한 문제와 상황에 대해 더 많은 경험이 있을수록 대자부 현상의 정확도는 높아진다. 주의할 점은 여기서 말하는 경험이란 아키텍팅의 경험이 아니라서 것이다! 흔히 컨설턴트들이 그렇듯이 아키텍처 수립 단계에 투입되어 아키텍팅 을 하고 빠지는 것은 배우는 입장에서 절대 권하고 싶지 않다(돈을 벌고 싶다면 권하고 싶다). Robert C. Martin이 인터뷰에서 말했듯이 자신이 한 작업에 대해 피드백을 받지 않으면 그 사람의 발전은 기대하기 어렵다. 고등학교 때 수학 문제를 푼 다음에 답을 확인하지 않으면서 공부한 학생 을 상상해보면 피드백 없는 학습의 폐단을 쉽게 알 수 있다.

이러한 경험을 여럿이 같이 공유하고 토론할 수 있는 공간이 있다면 학습 범위가 훨씬 넓어지고 효 율성과 정확성도 상승할 것이다. 아키텍처만을 위해 마련된 이런 공간은 드물지만 비슷한 성격의 커 뮤니티들이 앙양리에 여러 곳에서 활동 중이다. 필자들이 활동하는 오브젝트월드 는 주로 스터디 중심으로 그런 활동을 하고 있다. 르네상스 클럽(juneafn@hanmail.net에게 문의)은 아키텍처가 주요 주제는 아니지만 특정한 형식에 얽매이지 않고 활발하게 활동하고 있다. 그 외에 필자들이 모르는 여러 커뮤니티가 이와 비슷한 활동을 하고 있으리라 추측한다.

### 아키텍팅 프로세스를 익히자

필자들이 ‘우리 정도면 이걸 공부해도 되겠지’하고 생각하고 시도했다가 포기한 분야다. 100% 추 측에 근거한 글이니깐 새겨서 읽기를 바란다. 개발을 어느 정도 하고 디자인 패턴 등을 공부하면 좀 더 체계적인 방법이 없을까 하는 고민을 하게 된다. 그 때 보통 공부하게 되는 것이 OOAD이다. 마찬 가지로 아키텍처 수립에 대해 공부하게 되면 좀더 체계화된 방법이 있지 않을까 하는 고민에 빠진다. 이 때 도움이 될만한 자료가 아키텍팅 프로세스다. 구체적으로는 소프트웨어 설계 공정에 소개 한 섀론, TOGAF 등이 있다.

사실 이 단계에 오면 실력자라면 ‘나는 TOGAF 같은 거 없어도 나 혼자 잘 할 수 있어’라고 생각 할지 모른다. 맞는 말일 수도 있다. 그 정도의 실력자라면 TOGAF 없어도 잘 하는 경우를 종종 봤다. 하지만 조직은 개인 플레이를 하는 곳이 아니다. 그 사람이 없어도 동일한 성취를 이룰 수 있어야 한다. 프로젝트가 한 사람에게 의존하기 시작하면 그 프 로젝트는 이미 거실에 지뢰를 두고 생활하는 것이 된다.

붓다가 도를 터득한 다음 한 것이 교리를 설파하고 교단을 만든 것이듯, 아키텍트 또한 자신이 없어도 조직이 동일한 성취를 만들 수 있는 터전을 마련해야 한다. 자기 밥그릇 만 챙기고 있어서는 곤 란하다(박스 기사 ‘아키텍트의 이데올로기’ 참조). 이런 모든 것을 떠나더라도 이 정도 높은 성취 가 있는 아키텍트라면 허우적대는 조직 내 다른 개발자들 을 위해 아키텍팅 프로세스를 익히고 자기 만의 노하우를 담아 자신이 속한 조직에 뿌리 내리게 해야 할 것이다. 아키텍트는 다른 개발자들에 대한 코치이고 아키텍처에 대 한 대변자이어야 하기 때문이다.

처음 프로그래밍을 할 때는 새로운 API를 익히는 것 자체가 공부다. 하지만 점점 프로그래밍에 속 숙해지면 그 이상의 법칙을 찾기 시작한다. 함수의 크기는 어느 정도가 적당 할까? 서로간의 인터페 이스 불일치는 어떻게 막을 수 있지? 등의 법칙을 찾기 위해 여러 공부를 해 나가는 것이다. 이렇게 프로그래밍의 메타적인 법칙을 찾기 위해 계속 공부를 해 가면서 프로그래머의 끝에 있다고 느끼는 것이 아키텍트인 것이다. 그래서 필자들은 지난 몇 년간 계속 아키텍트가 되기 위해 노력하 고 있다. 하지만 여전히 아키텍트 지망생에 불과할 뿐이다. 그렇기 때문에 이번 5회 기사는 틀린 내 용일 수 있다. 우리가 성취한 다음에 결과를 적은 것이 아니라 그냥 지나간 길을 기록한 것이기 때 문이다. 다만 2~300명의 인원이 거쳐 갔고 2~30명의 인원이 현재도 지속적으로 만나서 토론하고 궤 도를 수정해가며 지나온 길이기 때문에 필자들만의 독단적인 생각은 아니라고 생각 한다.

### 중심의 경지를 펼치는 아키텍트

흔히 아키텍트라고 하면 컨설턴트라고 생각한다. 많은 사람이 프로그래머로서 경력을 쌓게 되면 관리자라 빠진다. 필자들도 10년 후쯤은 그렇게 생각이 바뀌어 있을지 모르겠다. 하지만 지금은 개발 자의 정점, 아키텍트의 길을 걸어가고자 한다. 그렇다고 관리자와 컨설턴트의 길이 프로그래머의 길 이 아니라는 것은 아니다. 그들은 분명히 필요한 직책이고 우리에게 많은 도움을 준다. 우리가 우려 하는 것은 많은 사람들이 ‘프로그래머→관리자 또는 컨설턴트’의 캐리어를 당연하게 여기는 풍조 다.

연재 처음에 공자의 ‘칠십이 중심소욕 불유구(七十而 從心所欲 不踰矩 : 나이 칠십이 되니 마음 가는 대로 해도 법도에 어긋남이 없다)’를 강히 얘기했었다. 개발자로서 존엄하 게 늘어가기 위한 길로 아키텍트라는 역할을 생각해 본다. 개발자들에게 있어 35세는 중심의 나이이다. 우리 개발자들이 무슨 프로그래밍 선수도 아닌데 왜 여기서 생명이 다 해야 한단 말인가? 나이가 들수록 임공에 비해 생산량이 떨어져 가기만 하는 개발자가 아니라 경험이 쌓일수록 양질의 산출을 해 내는 존재로서의 개발자가 많아지기를 기대한다. 진짜 중심의 나이까지 중심의 경지를 펼치는 아키텍트들을 친견(親見)하고 싶다.

정리 : 송우일wooil@korea.cnet.com

..... 월간 마이크로소프트웨어