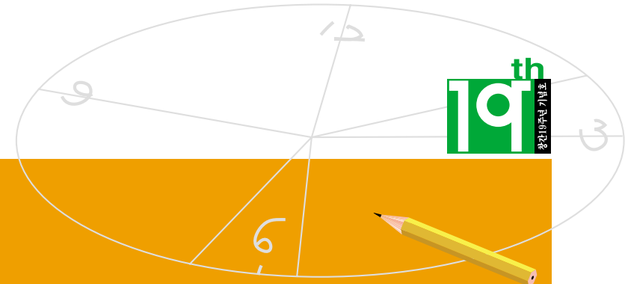
특집 1부



소프트웨어 개발 속도와 개발 기간의 상관 관계

개발 속도 향상은 영원한 숙제로 남을 것인가

개발자라면 한 번쯤 '소프트웨어 개발 속도가 느려지는 이유는 무엇일까'라는 물음을 스스로 던져 보지 않은 이가 없을 것이다. 이 글에서는 과거부터 현재까지 소프트웨어 개발이 계속 '느리기만 한' 이유는 무엇인지에 대해 개발 속도와 개발 기간의 상관 관계를 토대로 접근해 봤다. 소프트웨어 개발 속도가 느린 이유와 속도를 향상시키기 위한 노력들, 그리고 해결되지 않고서는 결코 개발 속도를 끌어올릴 수 없는 문제점은 무엇인지 하나씩 살펴보겠다.

처음 마소 편집팀으로부터 '왜 소프트웨어 개발이 느려지는가'에 대한 글을 청탁받으면서 기사의 기획 의도는 현 시점에 적절하고 연구 대상으로도 흥미로운 주제라고 여겼다. 물론 이번 기사는 매우 통계학적인 접근이나 학문적인 접근이 필요하다는 점이 부담됐지만 도전적이고 의미있는 기사가 되리라 생각하고 기꺼이 시작했다.

기획 단계에서도 아직 국내에서는 이런 분야에 대한 자료를 구하기 어려운 것이라고 지레 짐작했지만 여기저기 자료를 얻기 위한 조사를 진행하면서 이 분야에 관련된 원천 자료를 구하려는 시도는 사실상 포기하고 말았다. 스스로 연구하기보다는 외국의 소프트웨어 품질 프로세스를 답습해서 가르치거나 컨설팅하는 풍토도 한 몫 했고, 널리 알려진 바와 같이 실패에 너무 인색한 나머지 '실패에 가까운 성공'이라는 새로운 사고의 전환(?)을 통해서 모든 프로젝트는 성공적으로 끝났다고 발표하는 국내 정서 때문에 국내에 대한 연구는 찾기 어려웠다. 가장 가슴 아픈 일은 실패한 프로젝트는 단 한 개도 없고 모두 성공한 프로젝트뿐이라는 점이다. 우리는 혹시 '실패를 통해서 배운다'는 진리를 잊어버리고 성공 지상주의에 물들어 있는 건 아닐까. 있는 것도 없는 척, 없는 것도 있는 척하는 문화 속에서 탄생한 우리만의 프로젝트 성공기가 아닌가 생각하면 우울해지기도 한다.

실패한 프로젝트는 없다??

정확한 데이터를 구할 수 없는 국내 환경에서 통계학적인 접근은 거의 가능성이 없어 보였다. 주제가 주제인만큼 한 달 사이에 설문조사를 해서 정보를 모으는 것은 아예 꿈도 꾸지 못했다. 마소에서 설문조사를 통해서 자료를 어느 정도 모은 다음 이 주제를 다룰걸 하는 아쉬운 심정도 들었다.

한편 소프트웨어 공학적인 접근에 있어서 수 백 개의 프로젝트 분석을 통해 적절한 비용 산출법이나 어떤 환경에서 프로젝트 성공률을 가늠할 수 있는지 연구할 수 있는 외국의 문화가 새삼 부러워졌다. 마케팅에 관한 원서들만 보더라도 성공사례나 실패사례 모두 기업의 실명과 담당자명이 기재될 수 있는 것은 서구 문화에서만 가능한 새로운 장벽처럼 느껴졌다. 외부 정보 제공에 인색한 것은 남들의 이야기가 아닌 어쩌면 내 자신의 이야기인지도 모른다. 그것은 익명성을 보호받지 못한다는 선입관과 실패를 공개하는 것은 '누워서 침뱉기'라는 사고가 우리에게 공개할 수 있는 것도 공개하지 못하도록 강요하는 것이다.

신승근 Laborer@dev.co.kr

숭실대학교 동호회 초대 대표 시삽과 비주얼 툴 사용자 동호회 초대 · 2대 회장, 마이크로소프트 디벨로퍼 저널 기술 편집장 등을 역임했다. 개발 과정과 컨설팅의 결과를 각종 저널에 기고했으며, 현재 국내 소프트웨어 업계 최초로 TL 9000 인증을 받은 데브의 CEO 겸 CSA로 활동하고 있고 마소의 자문위원이기도 하다.

고민 끝에 펜을 들고…

지금까지 기사를 쓰면서 이렇게 고민을 한 기사는 없을 정도로 많은 고민을 했다. 뒹굴뒹굴 방바닥에서 굴러다니면서(집에서는 동굴이라는 별명을 얻었다) 이번 기사에 대한 맥을 잡아가려고 애를 썼지만 진행하는 컨설팅이 여러 개가 되어서 생각만큼 맥을 잡는 일도 시간조차 내기가 힘들었다. 거의 원고 마감일까지 기자를 애태우게 하리라는 필승의 각오(?)를 다시 다져보면 작업을 해나갔다. 몇 번이고 기사 제목과 목차를 잡아보았지만, 역시 컨설팅 경험을 바탕으로 기술할 수밖에 없는 점이 아쉽다. 물론 몇몇 외국 자료를 언급하기는 했지만, 아주 특별한 자료들은 아니고 소프트웨어 공학에서 약방의 감초처럼 다루어졌던 자료들이다.

이번 기사는 크게 세 가지 방향으로 가닥을 잡았다. 소프트웨어 개발이 느려진 이유와 그것을 극복하기 위한 노력들, 소프트웨어 개발을 지연시키는 요소에 대해서 살펴보기로 했다. 각각을 독립적으로 다룬다기보다는 연계해서 설명하겠다.

소프트웨어 개발은 옛날부터 느려터졌다?

우선 주제의 범위를 정확히 정하는 일부터 시작했다. 업계에서 느끼는 체감은 최초의 기획 의도였던 '소프트웨어가 개발이 느려진다'보다는 '소프트웨어 개발이 제때 끝나지 않는 경우가 점점 많아진다'는 것에 가까웠다. 그러나 이것은 너무 오래된 소프트웨어 개발의 이슈이자 진부하기까지 한 주제이다.

이 범위를 좀더 넓히고 유식하게 설명하면 소프트웨어의 위기(Software Crisis) 혹은 소프트웨어 생산성의 위기(Software Productivity Crisis)라는 표현을 사용하게 된다. 문제는 소프트웨어 위기는 이미 35년 전부터 주장되어오던 이야기라는 점이다. 본격적인 소프트웨어 개발 역사를 1960~70년대 초라고 볼 때 근 35년 후인 현재까지 생산성을 극대화하거나 개발 방법론상에서 큰 변화가 없었다는 점이다. 차이가 있다면 키 펀치를 눌러서 천공 카드로 프로그래밍하던 것을 CRT에서 직접 코딩을 한다는 점이라고 하면 너무 극단적인 표현일까!

비록 1990년대 이후 정보공학의 발전으로 데이터모델(ER)의 확립, RAD와 프로토타입 기법의 확산을 가져온 비주얼 툴의 출현, 객체지향 관련 모델링 언어인 UML의 제시 등 굵직굵직한 이슈들은 많았지만, 이것들은 모두 획기적인 전환이 아닌 기존의 방법을 좀더 보안해주는 측면을 넘어서지 못했다. 즉 약간 억지스러운 표현이지만, DFD(Data Flow Diagram)로 표현된 개발이 많이 실패했다고 해서 이를 UML 모델로 표현하면 모든 개발이 성공적으로 수행되는 것은 아니라는 점이다.

그런데 소프트웨어 위기는 너무 오래 전부터 제기되던 문제인데 최근에 와서 다시 새삼스럽게 더욱 더 부각되고 있는 이유는 무엇일까? 다음의 세 가지 현상에서 그 원인을 찾을 수 있다.

첫째, 이전과는 전혀 다른 비주얼 개발 툴의 출현

비주얼 툴은 단지 인터페이스를 시각적으로 쉽게 만들 수 있는 것에서 진보해 개발 플랫폼을 제공하고 있었다. 그 정점은 마이크로소프트(이하 MS)의 비주얼 스튜디오 닷넷이다. 이런 비주얼 툴들은 소프트웨어 개발 과정을 어느 정도 사용자에게 이해시키는 데 도움을 줬다. 사용자는 개발자가 개발하기 이전의 화면

'소프트웨어 위기'란?

소프트웨어 위기(Software Crisis)는 1968년 10월 독일에서 개최된 'NATO Science Committee' 국제회의에서 처음 언급됐다. 소프트웨어 위기의 근본적인 이유를 원론적으로 표현하면 소프트웨어 생산성이 고객의 요구를 따라가지 못한다는 것이다. 쉽게 말하면 고객의 컴퓨터 적응성이나 이해도는 높아지고 그에 따른 요구사항은 고급화, 다양화가 되어가는데 개발자가 고객의 요구사항을 제대로 반영해서 개발할 능력은 제자리 걸음이란 뜻이다. 이러한 소프트웨어 위기의 핵심 문제는 네 가지로 요약될 수 있다.

- 개발 예산의 초과
- 개발 일정의 지연
- 소프트웨어 생산성의 저조
- 소프트웨어 품질의 미흡

이 소프트웨어 공학적인 면에서 바라볼 때 이 네 가지 핵심 문제의 원인은 다음에 있다.

- 보이지 않는 논리적 구조의 소프트웨어에 대한 이해의 부재
- 소프트웨어 개발 관리 프로세스의 부재
- 소프트웨어 품질이나 유지보수성을 고려하지 않는 프로그래밍 방식

일반인들이 볼 때 문제 현상도 알고 그에 대한 원인도 아는데 개선법을 모른다고 하니 더 답답할 수밖에 없다.

을 미리 볼 수 있게 됐다. 이전의 사용자는 개발이 완료돼야지만 소프트웨어의 인터페이스(화면)을 확인할 수 있었던 것에 비하면 사용자의 개발의 개입성이 매우 커진 것이다. 또한 개발자가 아니더라도 화면 인터페이스를 작성할 수 있으며 어떤 경우에는 간단한 코딩 정도는 사용자가 할 수 있을 정도가 됐다.

결국 이전과는 달리 비주얼 툴은 사용자가 개발 과정에 어느 정도 적극적으로 가담할 수 있는 기회를 만들어 줌으로써 사용자가 소프트웨어 개발 프로세스의 문제점을 직접 느끼게 했다. 아이러니컬하게도 그 느낌은 소프트웨어를 만드는 것이 어렵지도 않은 것 같은데, 개발자는 매번 지연하는 것처럼 보인다는 점이다.

둘째, 사용자 수준의 극적인 향상

사용자의 수준의 극적인 향상이다. 불과 10년 전만 해도 컴퓨터를 사용하는 이는 소수에 불과했다. 하지만 지금은 많은 사용자들이 인터넷을 통해서 다양한 IT 환경에 노출돼 있다. 결국 개발자가 사용자의 수준만큼 IT 경험이 없다면 사용자의 요구사항을 제대로 이해하지 못하게 된다. 특히 사용자들은 최신의 소프트웨어 개발 프로세스들을 적용하면 개발은 더 빨라지리라고 예측을 하지만 실제로는 그렇지 못하다는 것에 대해서 격분하게 된다. 사용자가 생각하는 소프트웨어 개발과 실제 개발 과정은 매우 큰 격차로 벌어지고 있다.

프로젝트 규모가 커지다 보니 사용자라고 하더라도 입장이 다른 집단이 하나의 조직 안에 있는 경우가 많고 따라서 요구사항은 그 방향성을 잃어버리기 십상이다. 개발하는 입장에서도 이제 개발이 혼자 하는 것이 아니라 팀 단위로 움직이므로 의사소통에 필요한 시간도 꽤 많이 필요하다. 개발자 각 개인의 경험과 능력이 다르므로 보이지 않는 논리를 모두가 정확하게 이해하기란 쉽지 않다. 자기 혼자 하는 일은 쉽게 할 수 있지만 다른 이와 의사소통을 통해서 일한다는 것은 쉬운 일이 아니다.

필자의 경우에도 혼자 자료를 조사해서 쓰는 원고는 단 며칠만에 끝나는 경우가 있지만 지도교수를 통해서 교정을 받은 논문은 목차를 확정짓는 데도 두 달이 넘게 걸리는 경우가 많다. 글을 쓰는 경우에도 커뮤니케이션이 있는냐에 따라서 소요되는 시간의 차이는 매우 커진다.

셋째, 대량으로 양산되는 개발자

또 하나 간과하지 말아야 할 것은 개발자가 대량으로 양산되고 있다는 점이다. 솔직히 현재 학원이나 교육센터에서 쏟아져 나오는 인력은 개발자라기보다는 코더나 프로그래머에 가까운 수준인 경우가 많다. 그럼에도 불구하고 집중화된 교육으로 기간 내에 이것저것 귀동냥으로 들은 것은 많은 반면, 데이터 모델이나 알고리즘조차도 제대로 이해하지 못하고 있다. 이런 개발자들이 비록 자바나 닷넷 프로그래밍을 배워서 코딩을 할 줄 안다고 해도 그것은 언어를 겨우 이해해서 몇 가지 작업을 할 수 있는 수준에 불과하다는 것이다.

개발 회사들은 역량있는 개발자를 구하기가 더 어려워졌다. IT 회사들은 많아진 반면에 역량있는 개발자 수는 한정되어 있기 때문이다. 따라서 초급 수준의 개발자에게 회사의 모든 개발의 책임을 맡기게 됐다. 그 결과는 뻔한 것이다. 프로젝트를 관리할 만한 프로젝트 팀장이 절대적으로 부족한 상황이다.

G사에 컨설팅을 하기 위해서 그 이전 개발사인 E사가 만든 데이터 모델을 살펴보게 되었다. E사의 개발자들은 자바 프로그래밍을 템플릿에 맞춰 그럴싸한 형태로 개발을 했으나 데이터 모델은 엉망이었다. 암호 필드는 255바이트로 잡혀 있고 테이블은 통합되지 않아서 주 키(Primary Key)가 중복되어서 결국 100바이트나 되는 필드끼리 조인을 해야 하는 모델이었다. 학생이 리포트로 제출해도 D학점 이상 평가받기 어려운 수준이다.

'급성장' 한 IT 분야의 이면

이런 최근의 현실들이 우리로 하여금 소프트웨어 개발이 느려진다는 생각을 하게 한다. 사회적 현상에서도 개발자는 너무 많은 것처럼 느껴진다. 전공자보다는 비 전공자인 개발자 수가 급속하게 늘고 있기 때문이다. 이런 상태에서 우리의 개발 방법이나 조직 문화가 상대적으로 선진국에 비해서 매우 떨어져 있기 때문에 소프트웨어 개발은 더 느리게 진행되는 것처럼 보이고 항상 개발 결과는 고객이 원하는 것과 다른 방향으로 구축된다. 실제로 정보통신 분야의 급성장으로 우리는 우리가 소프트웨어 분야에서도 매우 성장한 것처럼 느끼지만 〈표 1〉과 〈표 2〉에서 보듯이 각종 통계 자료의 결과는 이와 매우 다르다.

연구에 의하면 소프트웨어에서 개발 비용이 차지하는 비율이 20%에 지나지 않는다고 한다. 건축이 전체 비용의 80%를 차지하는 것에 비하면 매우 대조적이다. 그것은 소프트웨어는 논리적 비가시적인 산물이며, 실행에 의한 작용의 결과로 확인할 수 있으므로 개발 비용보다는 기획이나 설계 비용이 더 커지게 된다.

한 마디로 소프트웨어는 저작물의 특징을 가진다. 예를 들어서 글을 쓰는 것은 얼마든지 대필시킬 수 있지만, 글의 시나리오를 구성하는 것은 작가만이 가능하며, 글의 시나리오를 가시적으로 보여줄 수 없는 것만 마찬가지로 소프트웨어도 그 안을 보여줄 수 없다(목차가 동일한 두 권의 책이 있을 수 있으나 내용까지 동일하지는 않은 경우는 고객의 요구사항이 비슷하지만, 궁극적으로 고객이 원하는 것은 매우 상이할 수 있다는 점과 비슷하다).

소프트웨어는 개발보다도 기획, 분석, 설계에 치중해야 함에도 불구하고 우리는 개발에 치중한 나머지 기획, 분석, 설계를 대충하게 된다. 결국 우리 스스로 무덤을 파고 있는 셈이다. 소프트웨어는 다른 저작물과의 다른 특성이 유기적(有機的) 성질을 들 수 있다. 소프트웨어의 유기적 성질이란, 소프트웨어는 환경의 변화에 적응하도록 개선해서 사용해야 한다는 뜻과 일맥상통한다. 일반적인 저작물은 이전 저작물과 유기성을 가진다기보다는 시나리오의 연속성을 가진다고 하는 편이 맞다. 실제로 소프트웨어의 전체 비용 중 개발 비용이 33%, 유지보수 비용이 67%를 차지할 정도로 변경이 많다는 점을 고려한다면 처음부터 잘 만들어야 한다. 소프트웨어 유기성의 향상이란 소프트웨어가 가지는 가장 큰 부분이 유지보수성을 좋도록 구성해야 한다는 것과 동일하기 때문이다.

최근 두 가지 사건에 대한 다른 평가

최근 우리에게 소프트웨어 개발의 비 생산성이나 품질평가 등에 대해 관심을 기울이게 하는 두 가지 사건이 있었다. 대표적인 두 가지 사건은 국민은행과 주택은행의 통합에 따른 불안한 서비스와 MS의 닷넷 서버 출시 연기이다. 요약해 정리한 기사를 살펴보자.

국민은행과 주택은행 통합에 대한 오류 또는 불안한 서비스에 대한 기사는 지난 10월초에 여기저기서 쏟아져 나왔다. 개선요구사항이 1084건이라면 적은 숫자가 아니다. 미리 구축될 시스템에 대한 기획안조차 제대로 리뷰되지 않았기 때문에 발생한 일처럼 느껴진다. 특히 3개월 동안이나 개선해야 할 항목이 30%라니 놀랍기만 하다.

옛 국민은행 노조가 "통합시스템 오류 인정하라"고 요구하고 나와서 화제이다. 국민은행 전산부가 2주간에 의견수렴을 통해서 총 1084건의 'IT통합 개선 · 건의 내용'을 작성했다. 이를 국민은행 '경영진이 즉시 개선 가능', '1개월 이내 개선 가능', '3개월 이내 개선 가능'으로 분류하고 전담팀을 통해서 수정해 나가기로 했다. 문제는 3개월 동안 개선해야 할 부분이 30%에 가깝다는 것이고, 1084건의 개선요구사항이 대부분 옛 국민은행 노조에 의해 건의돼서 통합 시스템은 옛국민은행 시스템에 좀더 가까워질 수밖에 없다. 서재인 부행장을 비롯한 IT 담당자들은 "기능 개선은 언제나 있을 수 있는 일이다. 개선이 필요한 부분도 시스템 자체에 대한 문제보다는 업무 프로세스나 특정 기능에 대한 것이 대부분"이라며 시스템 선정 오류 인정설을 일축했다.

- 전자신문, 2002년 10월 9일

다음으로 닷넷의 출시 연기에 대한 장황한 기사를 정리해 봤다.

"'MS 닷넷 서버 출시 세 번째 연기 감관성?' 이라는 제목의 기사가 언론에 실린 것은 최근의 일이다. 언론에 따르면 MS는 2000 서버의 후속 제품명을 닷넷 서버 2003으로 바꾸었다. 이 운영체제의 코드명은 휘슬러였고 2001년 4월에 MS는 윈도우 2002 서버로 명명했었다. 이렇게 제품명을 변경한 것은 이미 두 차례나 출시가 연기된 닷넷 서버가 2003년 초까지는 시장에 확산되지 못할 것이라는

소프트웨어 개발의 몇 가지 법칙

1. 전체 프로젝트에서 소프트웨어 개발비가 차지하는 비용은 20%에 불과
2. 전체 소프트웨어 비용 중에서 소프트웨어의 유지보수 비용이 차지하는 비용은 67%나 됨
3. **브룩스의 법칙(Brooks's Law)** : 브룩스는 그의 저서 'Man-Month의 신화'에서 "소프트웨어 개발 중간에 개발 인력을 보강하는 것은 소프트웨어 개발을 더 늦출 뿐이다(Adding manpower to a late software project makes it later)", 즉 개발자를 더 투입해도 생산성은 증가하지 않는다고 밝혔다.

〈표 1〉 국가별 소프트웨어 총생산액 대비 수출액 비중(출처 : 전자신문)

국가	아일랜드	인도	이스라엘	한국
SW 총생산액 대비수출액 비중	88.5%	68% (총생산액 57억 달러)	46.7%	2%

〈표 2〉 국내 소프트웨어 시장 규모(출처 : 전자신문)

SW 시장 규모	한국 IT 산업에서의 비중	GDP에서 SW 산업의 비중
3694백만 달러 (1996)	8.2%(1997)	0.95%(1996)
7005백만 달러 (2000)	23% 예상(2002)	1.61%(2000)

불안감을 반영했기 때문으로 분석가들은 평하고 있다.

2000년 10월 MS는 2001년 하반기에 출시될 것이라 말했으나 다시 2001년 4월에 2002년 초로 연기됐고 2002년 3월에는 다시 2002년 하반기로 연기했다. 이는 MS의 예상과 달리 윈도우 2000 서버 보급이 아직 진행중이라는 것을 나타낸다. 2001년말을 기준으로 윈도우 NT 4가 전체 윈도우 서버 시장의 61%를 차지하고 있고, 2002년말을 기준으로 윈도우 NT 4 서버는 32%를 차지할 것으로 예측되고 있다.

또한 MS는 아직도 닷넷 웹 서비스 전략을 수정중이고 이 작업이 예상보다 느리게 진척되고 있다고 표현한 바 있다. 결국 닷넷 서버 개발의 지연은 웹 서비스 전략 수정에 어느정도 영향을 미쳤을 것이라고 분석가들은 보고 있다.

- Joe Wilcox, Special to ZDNet News, 2002년 9월 2일

이 글을 읽고 무엇을 느낄 수 있는가? 전략이 수정되면 개발은 다시 지연될 수밖에 없다. 이것은 지극히 당연한 현상이다. 전략이 수정되고 시장 환경이 바뀌면 개발 기간은 느려진다. 한국의 모든 (갑)들이여, 그대들의 전략이 변경돼도 쌩쌩하게 돌아갈 시스템은 없다는 것을 아는가? IT 업계의 최고 회사인 MS도 지연하는데, 중소 소프트웨어 개발 회사는 그대들의 말 한 마디에 수많은 밤을 새고 또 새고, 날이 새고 낮과 밤이 바뀐다. 래쇼날의 백서 중에서 'Straight Answers to Tough Questions about Team-Based Development'를 보면 다음과 같은 문구가 나온다.

"I can build you the perfect product. Just don't ask me to ship it."
- Anonymous

의미 심장하지 않는가? 고객의 요구가 확정되지 않는다면 우리는 아무 것도 할 수 없다.

소프트웨어 위기를 극복하려는 노력들

소프트웨어 위기를 극복하려는 노력은 많이 진행돼 왔다. 이런 노력들은 다양한 방향에서 추구해 왔다. Dijkstra가 Goto문 없는 구조적 프로그램 개발이 가능하다고 하면서 소프트웨어 공학은 시작됐다. 지난 1979년에 Boehm은 소프트웨어 생명주기 모델(Waterfall형)을 제시했고 그것을 기점으로 개발 프로세스 관리가 하나의 대안으로 연구되기 시작했다. 1980년대 후반에 접어들면서 4GL, 객체지향, 정보공학, 형상관리, 프로토타이핑 등의 기술적 진보들이 일어나기 시작했고 1990년에는 객체지향이 IT 업계 전반으로 퍼져서 상용화된 제품들이 쏟아져 나왔다. 2000년도에는 XML을 기반으로 하는 웹 서비스가 태동하고 있다.

이런 노력들이 전혀 의미가 없었던 것은 아니지만 확신하건데 소프트웨어 위기를 극복할 수 있는 해법은 아니었다는 것이다.

CASE 도구에서 RAD 툴까지

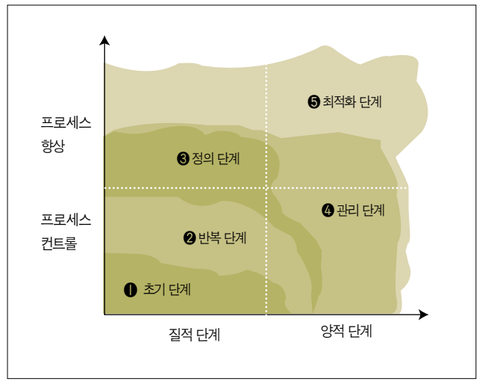
CASE 도구나 4세대 언어의 사용으로 생산성 향상을 꾀하거나 프로토타이핑을 통해서 고객과의 대화 채널을 만들어서 요구사항의 반영을 정확히 할 수 있는 기법들, 소프트웨어를 빨리 개발할 수 있도록 하는 RAD 툴은 어느정도 개발 시간을 단축시키는데 일조를 했다. 하지만 그만큼 개발의 범위와 규모가 커지고 있어서 이전보다 쉽게 소프트웨어 개발에 참여할 수 있는 것처럼 보일 뿐 결국 단축된 시간만큼 더 복잡한 소프트웨어를 만들어야 한다.

객체지향 기술과 컴포넌트 컴퓨팅

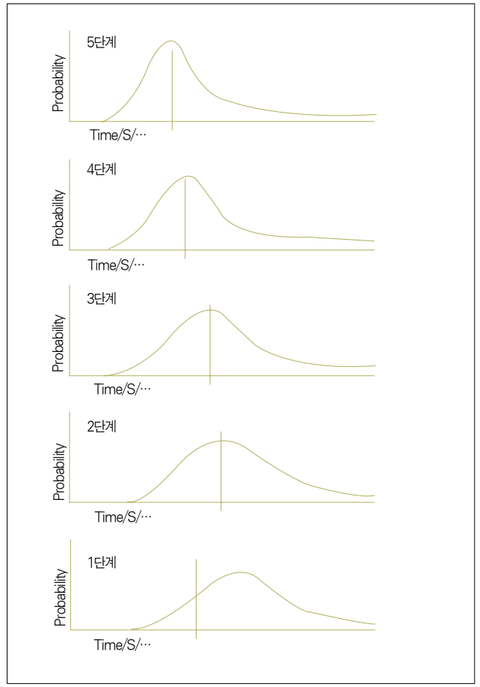
가장 활발한 움직임은 객체지향 쪽에서 일어나고 있다고 본다. 객체지향 기술을 바탕으로 소프트웨어를 부품(component)처럼 조립하는 공장(factory)를 만들겠다는 취지이다. 객체지향 프로그래밍 기술은 사람의 사고를 모델링한다는 의미에서 자연적 모델링(natural modeling)이라고 한다. 그러나 여기서 한 가지 생각해야 할 것이 있다. 객체지향은 사람의 사고를 모델링하는 방법 중 하나이지만 실제 세계에서도 매우 힘든 결정들을 해야 한다는 것이다. 모델을 만드는 것은 결코 쉬운 일이 아니며 잘못 만든 모델은 재사용성에 있어서도 회의적인 반응이 나오고 있다.

재사용성은 크게 코드의 재사용성과 기능의 재사용성으로 나누어지는데 객체지향으로 잘 만들어진 모델이라고 하더라도 급격한 비즈니스 모델의 변경에서는 잘 구성된 상속관계가 방해가

〈그림 1〉 CMM의 프로세스 평가 레벨(SEI 자료)



〈그림 2〉 CMM 각 단계별 성과 분포도(SEI 자료)



되는 경우가 많다. 재사용성에 있어서 객체지향이 생각하지 못한 것은 부품의 단위이다. 소프트웨어에 있어서 인터페이스 설계는 얼마든지 자유롭게 진행될 수 있으므로 각 회사가 만든 동일한 기능의 컴포넌트라고 할지라도 인터페이스는 매우 상이하다. 결국 인터페이스가 거의 달라지지 않을 단위의 라이브러리 수준의 부품은 재사용성이 매우 커지지만 복합적인 기능 또는 비즈니스 로직을 내재하고 있는 컴포넌트나 클래스는 재사용되지 못할 가능성이 커진다.

소프트웨어 개발에 있어서 객체지향 프로그래밍이나 컴포넌트 프로그래밍은 최소한 개발자에게 인터페이스에 대한 개념과 재사용성에 대한 인식을 심어 줬으나 이 역시 소프트웨어 위기를 극복할 수 있는 정답이라고 하기에는 많은 의문점을 남기고 있다. 물론 옹호론자 입장에서는 어설프게 배운 이들의 잘못된 적용 사례들이 문제를 발생시킨다고 이야기할 수 도 있지만, 제조업체 수준의 품질을 유지하기 위해서는 소정의 프로그래밍 지식이 있는 자가 품질을 유지할 수 있는 방법이 필요하다. 그러나 현실은 그렇지 못하다. 장기간의 학습과 연습, 경험이 필요하므로 이것은 하나의 대안이나 시대의 흐름이 될 수는 있지만 소프트웨어 위기를 극복하기 위해서 추구해야 할 전부가 될 수는 없다.

컴포넌트 컴퓨팅은 우리에게 환상만 심어주고 있다. 컴포넌트를 이용한 조립식 소프트웨어는 허공을 떠도는 메아리 같다. 컴포넌트가 가져다 준 점은 냉철하게 현실적으로 말한다면 프로그램간의 인터페이스가 개방화될 수 있다는 점과 별도의 인스턴스 생성으로 라이브러리보다는 쓰기 쉽다는 점뿐이다. 컴포넌트가 상업적으로 라이브러리보다는 더 많은 시장성을 가진 것은 인정할 수 있으나, 능력있는 자들을 위한 기회의 시장일 뿐 수백, 수천 만 명의 일반 개발자에게는 이해하기 어려운 메커니즘이다.

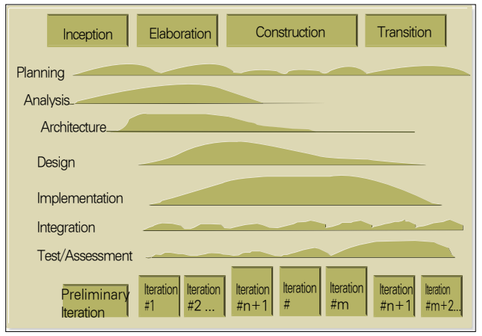
오히려 웹 서비스와 같은 서비스의 이용이 더 승산이 있어 보인다. 물론 이것도 현재의 소프트웨어의 위기를 해소시켜주지 못한다.

CMM의 도입

CMM(Capability Maturity Model)은 1986년 미국 소프트웨어 공학연구소에서 개발된 절차이다. 이 절차는 다섯 개의 잘 정의된 일련의 개발 절차, 즉 초기 단계, 반복 단계, 정의 단계, 관리 단계 및 최적화 단계 등으로 나뉘어진다(〈그림 1〉). SEI(Software Engineering Institute)의 W. Humphrey가 1989년에 발표한 소프트웨어 개발 조직(프로세스)의 '능력'을 판정하는 척도이다.

프로세스를 평가하는 것은 CMM뿐만 아니라 SPICE, TL 9000 등이 있다. 이것도 CMM과 거의 유사한 프로세스 평가법이다. 대상 규모나 적용법, 평가 결과가 다를 뿐이다. CMM이나 SPICE가 조직의 개발 성숙도나 조직의 프로세스 능력을 평가하

〈그림 3〉 점진적 개발 프로세스(래쇼날 자료)



는 데는 유용하지만 적은 기업에서 적용하기에는 많은 무리가 있고, 프로젝트의 성격을 바꾸면 평가는 원점으로 돌아갈 수도 있다. 그럼에도 불구하고 CMM과 같은 프로세스 능력을 평가하는 것은 매우 의미가 있다. 어떤 조직이 프로젝트를 어느 정도 능력으로 수행할 수 있는지 평가할 수 있는 객관성을 만들어 냈다는 것은 발주사 입장에서 보면 상당히 획기적인 전기가 된다.

〈그림 2〉는 매우 의미심장한 그래프를 보여준다. 프로세스 능력에 따라서 개발이 종료되는 시점이 단축될 뿐 아니라 예정 기간에 거의 근사해서 종료된다는 것을 알 수 있다. 모든 조직이 항상 정해진 개발 시간 내에 프로젝트를 종료할 수 있는 것은 아니다. 그것은 개발업체의 능력 밖의 사항도 많기 때문이다. 그러나 잘 조직화된 회사는 개발시 발생하는 문제들을 간과하지 않고 문제 발생 이전에 파악하고 대처할 수 있다.

개발 프로세스도 점점 현실화되고 있다. 〈그림 3〉의 래쇼날의 자료를 보면 각 개발 단계는 지속적으로 평가되고 수정되어진다. 일반적인 개발 방법과 다를 바 없는 것처럼 느껴진다. 그러나 핵심은 반복한다는 것에 있지 않다. 반복의 근거가 되는 문서화에 있다. 문서화 없는 반복이란 인생이 끝나지 않는다고 생각하는 것과 같다.

소프트웨어 생산성은 어떻게 측정하는가

높은 품질의 소프트웨어는 항상 높은 생산성을 보장받아 왔는지를 살펴볼 필요가 있다. 개발 과정에서 발견되는 오류는 1000줄당 약 50~60개이고 개발해서 설치한 후에는 평균 4개 이하라고 한다. 일반적으로 생산성이란 단위 시간당 내놓는 결과물의 양이다. 소프트웨어에 있어서 생산성이란 단위 시간당 프로그래머가 작성하는 프로그램의 양인가? 소프트웨어는 지적 저작물이므로 프로그램의 양에 비교할 수만은 없다.

언젠가 S사의 프로젝트에서 한 개발자가 하루에 4개의 화면 분량의 코딩을 맡아서 한 적이 있다. 그는 많은 소스를 만들어 냈지만, 결국 통합 테스트에서 그 개발자가 한 개발 모듈은 전면적으로 다시 개발해야 했다. 그는 동일한 소스 코드를 조금씩 바꾸어서 하루 할당량을 채워놓았을 뿐이었다. 또한 동일한 프로젝트를 할 때 한 쪽은 통합해서 많은 기능을 분석하고 공통 모듈을 만들어서 소스의 양을 줄이고 다른 한 팀은 화면별로 업무량을 나누어 모두 개발을 해서 3~4배의 소스로 만들어 냈다고 할 때 누가 더 생산성이 좋은 것일까? 다른 경우로 에러 처리나 경우의 수 처리를 아주 많이 해서 소스가 늘어난 경우와 해당 기능만 겨우 개발해서 소스의 양이 적은 경우 어떤 경우가 더 생산성이 높은 것일까?

기간 내에 요구사항을 얼마만큼 만족했는가

일반적으로 생각하는 프로그램의 생산성은 총 프로젝트 기간 동안 투입된 인력과 들어간 비용과의 관계일 것이다. 소프트웨어의 생산성을 다른 각도에서 본다면 정해진 기간 동안 요구된 요구사항을 얼마만큼 만족했는가의 정도라고 표현할 수 있다. 그래서 다음과 같은 공식을 연구해 봤다. 아직 다양한 프로젝트에 적용해보지 않았지만 시뮬레이션을 통해서 그 결과를 〈표 3〉과 같이 측정할 수 있었다.

◆ **소프트웨어 생산성 지수**

S : 소프트웨어 생산성 지수(software production rate)
A : 최초 요구사항 대비 선정된 요구사항 비율(adaptive request rate)
C : 선정된 요구사항 중에서 실제 개발된 비율 (complete request rate)
P : 예정기간 대비 소요된 개발 기간 비율 (developed period rate)
1.2 : 경험적 상수

$$S = \frac{A \times C}{P \times 1.2}$$

소프트웨어 생산성 지수에 대해 다음과 같이 적용해 봤다. 요구사항을 전부 이해(분석율 100%)해서 정해진 기간 내(일정 기간 100%)에 100%를 구현(구현율 100%)하면 생산성은 100%가 된다. 그러나 분석된 요구사항이 전체 요구사항 중 80% 정도이고 그 중에서도 정해진 기간 동안에 80%만 개발했다면 생산성은 64%이다. 이것은 전체 요구사항을 모두 개발하기 위해서는 최소한 '지금까지 투입된 개발 기간×0.36(=예정 기간×P×(100%-S))' 의 시간이 더 필요하다는 것을 알려주기도 한다.

아직 이 부분에 대한 연구가 끝난 것이 아님에도 이렇게 공개를 하는 것은 어떤 방법으로라도 우리는 소프트웨어의 생산성에 대해서 규명을 해야 했다. 이것은 그 많은 방법 중에서 하나가 될 것이다.

이 공식에서는 투입 인력이 반영돼 있지 않다. 그것은 소프트웨어 개발시 단순히 사람의 수로 그 생산성을 향상하는 데에는 한계가 있기 때문이다. 공동 저작물이라고 더 빨리 원고가 써지는 것이 아니라는 것은 쉽게 생각할 수 있다. 그래서 여기서는 주어진 기간을 하나의 기준으로 생각을 했다. 비용 역시 제외했다.

개발이 지연되는 진짜 이유는 무엇인가

개발이 느리다는 것을 다른 입장에서 생각해 보자. 개발 기간 내에 개발된 것이 고객이 요구한 것과 달라서 결국 재개발에 소요된 기간까지 포함하면 개발은 매우 더디게 느껴질 수 있다. 결국 개발은 영원히 끝나지 않는 숙제처럼 쳇바퀴를 돌게 된다. 여러 경험적인 사례를 통해서도 개발이 지연되는 이유는 소프트웨어의 위기에서 지목되는 원인과 어느 정도 유사하다. 차이가 있다면 동전의 양면과 같이 다른 입장에서 접근의 차이이다. 개발 현장에서 느끼는 소프트웨어 개발 지연의 첫 단추는 계약부터 일어난다.

D사가 회사 사이트를 개편하려고 공고를 냈다. 3개 회사가 입찰을 했는데, A사는 2500만원, B사는 5000만원, C사는 1억원에 입찰을 했다. 당연히 D사는 A사와 계약을 했으나 A사는 한 달도 채 안되어서 개발을 포기했다. D사가 공고를 낸 5가지 업무가 공고 내용과는 달리 하나 하나가 엄청나게 방대하다는 이유다. 다시 D사는 B사와 계약을 했다. B사는 업무 범위보다는 기간과 투입 인력에 따른 비용 기준으로 청구를 했다. 즉 D사가 요구사항을 많이 늘리면 비용은 더 청구하거나 요구사항을 줄일 수 있도록 계약을 했다.

B사는 D사의 요구사항을 모두 문서로 받아보니, 3건의 요구사항은 약 300개의 화면이 필요한 개발이고 2건의 요구사항은 최소 3~4명이 4~5개월은 개발 후에도 지속적인 테스트와 수정이 필요한 내용이었다. B사는 개발 기간을 연장하고 인력을 더 투입해야 한다는 결론을 내리고 필요한 계약 조정에 들어갔다. 결국 총 개발비용은 1억 5000만원으로 불어났다. D사는 당황스럽다는 생각을 했으나, 유사한 솔루션이 1~5억원 대에 판매된다는 것을 감안할 때 D사가 원하는 내용은 최고 수준이므로 개발비용은 오히려 적어 보였다.

A, B, C 회사가 처음 산정한 비용과 실제 개발비용의 차이는 무엇인가? 분석설계비용과 개발비를 혼동하기 때문이 아닐까? 알지도 못하는 개발내역에 대해서 비용을 계산한다는 것은 어불성설(語不成說)이다.

명확치 않은 요구사항이 개발을 지연시킨다

그 다음은 전산화에 대한 몰이해이다. 최근에 컨설팅을 진행하며 제품 개발에 몰두했을 때에는 몰랐던 사실들을 알게 되면서 놀라는 일이 많아졌다. 전산화의 의미가 그렇다. 기업을 방문해 보면 전산화가 단순히 업무를 컴퓨터로 진행하는 것이고 그것을 위해서 필요한 소프트웨어를 개발하는 것이라고 생각한다. 기업을 전산화한다는 것은 소프트웨어 개발을 한다는 의미가 아니라 기업의 업무 모델을 시스템화 혹은 조직화한다는 것이다. 시스템화에서 프로세스 확립은 필수적이다. 리엔지니어링 혹은 BPR(Business Process Restructuring)란 어려운 용어를 사용하지 않더라도 기업 업무 전산화를 위한 프로세스 확립은 가장 핵심적인 부분이다.

전산화를 위한 가장 첫걸음은 비즈니스 프로세스 확립이다. 업무 매뉴얼도 제대로 갖추어져 있지 않은 회사의 전산화란 의미가 없는 것이다. 의미가 없을 뿐 아니라 전산화를 해도 실패할 수밖에 없다. 요구사항이 명확하지 않은데 개발이란 있을 수 없는 것이다.

〈표 3〉 소프트웨어 생산성 지수의 시뮬레이션 결과

분석율(A)	구현율(C)	일정기간(P)	생산성(S)
100%	100%	100%	100%
100%	80%	120%	64%
100%	80%	100%	80%
100%	80%	80%	105%
80%	100%	100%	80%
80%	80%	120%	51%
80%	80%	100%	64%
80%	80%	80%	84%
50%	100%	100%	50%
50%	80%	120%	32%
50%	80%	80%	52%

개발 경력자가 생각하는 개발의 어려움

개발 경력(흔히 전산 짬밥이나 내공의 수준)에 따른 소프트웨어 개발의 핵심 문제점에 대한 입장을 살펴보자. 몇몇 문제점들이 프로젝트를 지연시키고 있으므로 핵심 문제점을 제거하면 프로젝트가 제때 끝나리라는 생각이 짧은 생각이라 할 수도 있지만, 업계의 경력자들의 생각을 조명하는 것으로 지연의 문제점에 대한 새로운 전기를 맞이할 수 있으리라 본다.

개발 경험이 별로 없는 상태에서는 대부분 개발자는 개발자간의 커뮤니케이션이 가장 문제라고 생각한다. 개발 경험이 없으니 개발팀 자체를 이끌어 나가는 것이 가장 큰 문제이고, 개발자간의 격차도 심하니 개발자간의 커뮤니케이션이 가장 큰 문제가 될 수밖에 없다. 일단 사용자의 요구가 어떻든 간에 개발자는 개발 그 자체에만 집중하게 된다. 대개 개발을 시작한지 2~3년 지난 개발자가 이런 이야기를 핏대 올려가며 말하곤 한다.

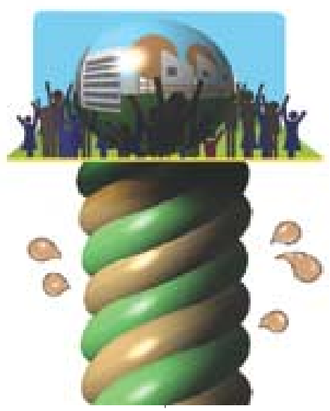
개발경력 4~6년이 되면 개발에서 가장 어려운 것이 요구사항을 파악하는 것이라고 답한다. 이때부터는 설계된 문서가 아닌 직접 고객을 상대로 분석, 설계를 하는 시기이므로 고객과 만남에서 고객이 원하는 것을 파악하는 것이 어려울 수 밖에 없다. 요구사항을 파악하는 법칙이 있는 것도 아니니 저마다 타고난 소질을 발휘해서 고객의 요구사항을 파악하는 것은 마치 숨바꼭질을 하는 것과 같다.

문제는 '의사소통'의 부재에서 비롯

개발경력이 7~8년 이상 되면 고객의 문화 파악이 어렵다고 한다. 고객이 술을 좋아하는지 고기를 좋아하는지 아니면 다른 무언가를 좋아하는지 파악해야 하는 위치에 있는 차장급 개발자로서는 발주사의 기업 문화 파악이 가장 어렵다. 기업 문화에 따라서 전혀 다른 방식으로 프로젝트를 진행해 나간다. 상당한 수준의 프로젝트 매니저급 개발자들은 개발을 하지 않고도 기업 문화를 파악해서 프로젝트를 완료하기도 한다. 이 정도 수준의 개발자를 만나면 내공의 극치를 보여준다고 생각하기 쉽다.

그러나 근 10년이 다 되어가는 필자가 느끼는 소프트웨어 개발에 있어서 가장 큰 어려움은 믿음과 자신감의 결여다. 믿음과 자신감이 없는 개발자나 발주사와 함께 일하는 것보다 더 고통스러운 일은 없을 정도이다. 안타깝게도 대다수 프로젝트가 기술이나 조직 문화가 따라오지 못해서가 아니라, 하고자 하는 마음과 희생이 없어서 제 길을 가지 못하고 있다.

누군가 개발에 있어서 가장 큰 문제를 비용으로 뽑는다면 개발자가 아니라 영업이나 마케팅 혹은 관리부서 소속일 것이다. 대부분의 개발자는 비용보다는 기술을 우선시하는 경향이 있다. 개발을 제때 끝내기 위해서는 위에서 말한 부분의 것을 모두 해결해야만 한다. 앞의 이야기에서 무엇을 어떻게 해결할 수 있을까? 아마 가장 중요한 키워드는 '의사소통'일 것이다. 개발자간의 의사소통과 개발자와 고객간의 의사소통, 그리고 고객의 조직에 의한 이해 등, 이보다 더 큰 해결책은 없을 것이다.



피히테와 셰익스피어를 거론하는 이유는

독일의 철학자이면서 관념론자의 한 사람인 피히테의 강연중 '독일국민에게 고함'은 매우 유명하다. 200여년 전의 한 대학교수인 피히테가 쓴 '독일국민에게 고함'을 읽고 있노라면 200여년 전이나 지금이나 교육 현실에 차이가 없음에 놀라게 된다. 몇 세대가 흐른 지금에도 사회 구조적인 면은 쉽게 바뀌지 않고 있는 것이다. '인생은 짧고 예술은 길다'고 말하지만 예술은 변해도 문화는 변하지 않는다고 말하고 싶다. 그것은 유구한 인생이 단기적으로 어떤 성과를 만들어 낼 수 있으나 그것은 한 시대에서 다음 세대로 전해질 때에는 오직 보편적 진리(universal truth)만이 전해지는 것이다.

반면에 기술들은 매우 빠르게 보급되는 것처럼 느껴진다. 그러나 기술도 혁신적 기술이 아니라면 보급되는 데 꽤 오랜 시간이 걸렸다. 미국에서 라디오가 발명되고 인구 5000만명에게 보급되는데 걸린 시간은 무려 38년, 텔레비전은 16년이나 걸렸다. 물론 인터넷은 불과 4년밖에 걸리지 않았고 그래서 혁신적인 기술이라고 불리우지만.

소프트웨어 개발이란 기술 이전에 예술이고 예술 이전에 문화이다. 외부 프로젝트를 하면서 회사와 회사의 문화가 달라서 고생한 적이 한 두 번이 아니다. 소프트웨어 위기를 극복하기 위해서 역량있는 개발자를 양성한다는 것은 어쩌면 400여년 전에 영국이 나은 세계적 작가인 셰익스피어나 19세기의 대문호인 톨스토이와 같은 작가를 만드는 것처럼 어려운 일이 아닌가 한다. 문호는 만들어지는 것이 아니라 태어나는 것이라고 한다면 소프트웨어가 갈 길은 더 멀어져만 간다.

예술성과 공학성 모두 만족시켜야

소프트웨어 개발은 어떤 면에서 예술에 속할 수 있다. 그 만드는 기법이나 사상은 하루 아침에 습득될 수 있는 것이 아니며 지속적인 관심과 노력이 필요하다. 소프트웨어가 가지는 예술성과 공학성의 두 부분을 모두 만족시킬 수 있는 방안을 찾는다면 좀더 현실적인 대안들이 나오리라 생각한다. 소프트웨어의 저작물적인 성격과 사용자를 가지는 제품이라는 점을 현실적으로 고려한 방법들과 사회적인 문화 성숙, 기업의 조직 문화의 성숙 없이는 실제적인 대안이 없는 것처럼 느껴진다. 소프트웨어의 위기는 고작 35년밖에 되지 않은 것이라고 생각하면 마음이 편해지지 않겠는가?

처음 기업용 소프트웨어 개발에 몸을 담은 지난 9년 전이나 지금이나 개발자의 의식이나 개발의 능력이 제자리라는 점에 가끔 실망하게 된다. 한 발 먼저 발을 들여 놓았음에도 불구하고 후배들에게 물려줄 문화를 만들지 못한 선배들의 책임도 있지만, 개발 맨 밑바닥의 코딩을 한낱 저급한 노동으로 치부하는 사회 문화적 분위기에 휩쓸려서 후배들이 제자리 걸음만 하는 것 같아 슬퍼진다. 이제는 우리도 외국에 판매할 수 있는 소프트웨어를 만들 수 있는 문화를 창출해야 한다.

끝으로 이 글에서 사용된 용어 중 낯설은 용어는 필자가 설명을 위해서 만들어 낸 용어들이다. 한 마디로 꼭 집어서 이야기를 해야 하는데 마땅한 용어가 없어서 과감하게 용어를 정의했다. 주제넘은 시도일 수도 있으나, 이런 시도들을 통해서 우리의 소프트웨어 개발 문화가 한층 더 성숙해지기를 바라고 이런 분야에 대한 실질적인 토론들도 많이 오고 갔으면 한다.

정리 : 이종림 nowhere@sbmedia.co.kr

참고 자료

1. http://ivory.snu.ac.kr/Sebase/Seminar/97Winter/Improvement/Tutorial/sld001.htm
2. http://se.dongguk.ac.kr/edu/edu2002_2_scd.htm
3. http://salmosa.kaist.ac.kr/~cs550/
4. http://ithink.pe.kr/kimka/index.cgi/Software_20Engineering
5. http://oopsla.snu.ac.kr/oocasetool/SOFT/CH-3/sld006.htm
6. http://www.nanet.go.kr/nal/3/3-1-4/nal97021.htm
7. http://www.love2u.pe.kr/soft01.htm
8. http://redwood.snu.ac.kr/~ksaehwa/misc/uml/MASO9812UML.htm
9. http://my.netian.com/~withwish/cathedral-bazaar-10.html
10. http://www.omg.org/technology/documents/formal/uml.htm
11. http://www.cup.org/
12. http://www.rational.com/products/rose/whitepapers.jsp
13. http://www.kisa.or.kr/K_trend/KisaNews/199911/4_6SSE-CMM.htm
14. http://zeropage.cse.cau.ac.kr/moin/moin.cgi/CMM
15. http://soomsori.net/moa/moin.cgi/CrcCard
16. http://mail.nl.linux.org/alliance/2002-02/msg00000.html
17. http://selab.sogang.ac.kr/~cmm/
18. http://www.sitri.org/cmm/main.htm
19. http://www.component.or.kr/index_06.htm