공학 트렌드

SW 프로세스 개선 통찰 :
애자일로부터 얻은 것, 놓은 것 그리고 간직하는 것 Part 2
- 프로세스 개선원리와 고정관념,
애자일에서 해결해야할 몇 가지 과제

2012. 11. 5. (제31호)

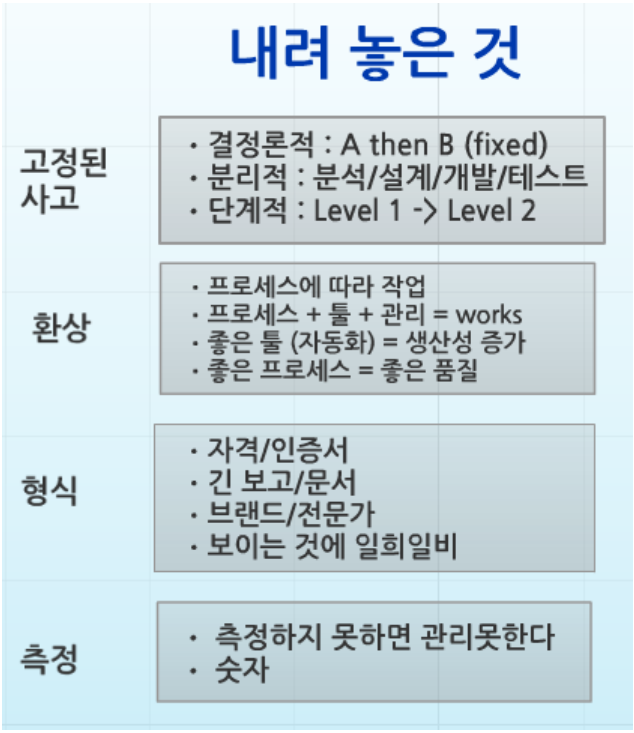
목 차

소프트웨어공학센터
경영지원TF팀

Ⅲ. 내려놓은 것

만약 수술을 마치고 의사가 나와서 수술은 성공했지만 환자는 죽었다고 한다면 그 수술은 성공한 것일까 실패한 것일까? 즉, 기술적으로는 성공적이지만 비즈니스적으로는 실패한 모순적 결과가 프로젝트에서 발생할 수 있다. 소프트웨어 공학의 원칙이나 베스트 프랙티스를 충실하게 적용하였는데도 결과적으로 실패하는 경우, 방법론적으로는 완벽한데 사람들이 잘못 적용했다거나 혹은 애시 당초 적용한 방법론이 잘못이라는 식의 오류에 빠지기 쉽다. 본 장에서는 기존의 소프트웨어 공학적 접근원칙들 중에서 애자일 관점에서 잘 맞지 않았던 사항들을 중심으로 설명해보고자 한다.

<그림 5> 내려놓은 것



접근방법

선형적 절차에 따라 A가 끝나야만 B를 시작하게 만들어 놓는 경우에 프로젝트에서 병목이 발생하기 쉽다. 실제 요구사항이 완료되어야 설계를 시작하고 설계가 완료되어야 개발을 하는 폭포수적 접근방법에서 요구사항이나 설계와 같은 상위단계 작업이 지연되어 본래 계획보다 개발 기간이 부족해지는 경우가 많다. 모호한

요구사항이 간단한 프로토타입 개발을 통해 더욱 명확해질 수도 있고, 설계를 통해 놓쳤던 요구사항을 발견할 수 있다.

프로젝트를 분석, 설계, 개발, 테스트와 같이 분리하여 접근하는 경우 전체 시스템을 놓치고 각 단계별로 부분 최적화되는 문제점을 낳기 쉽다. 소위, 요소환원주의(Reductionism)의 기본 전제인 "전체는 부분의 합과 같다"는 것은 적어도 소프트웨어 개발과 같이 복잡한 협업기반의 지식 작업에는 맞지 않는 말이다. 보고서 하나를 쓰더라도 서론, 본론, 결론을 나눈 뒤 각자 최적의 내용을 만들어 붙여보면 전체적으로 따로 노는 듯한 느낌을 받게 된다. 부분 최적화로는 전체 최적화에 도달하지 못하기 때문이다.

한 단계 올라가기 위해 반드시 단계적으로 접근해야만 하는 것도 아니다. CMMi(Capability Maturity Model Integration)에서 제시하는 능력성숙도 모델에 따르면 관리가 완성되어야 기술적 엔지니어링이 가능해지고 엔지니어링 표준화가 이루어져야 측정이 가능해지고 정량적 측정이 완성되어야 최적화 단계에 도달할 수 있다. 사칙연산이 가능해야 곱셈이 가능하고 곱셈을 알아야 분수로 나갈 수 있다는 것이다. 논리적이고 유용한 접근방법이지만 동시에 성장을 가로막는 제약사항이 될 수 있다. 실제 프로젝트에서 관리와 엔지니어링, 측정과 개선은 따로 떨어져 있지 않다. 조직적, 장기적 관점에서 특정 단계에 집중할 필요성은 있겠지만 프로세스 접근방법이 통합적이고 유연성을 갖출 때 프로젝트가 기술적으로 성공하고 결과적으로 실패하는 경우가 줄어들 수 있을 것이다.

환상

프로젝트 팀에서 실제 하는 일과 적용하는 프로세스가 일치하지 않는 경우가 많다. 모든 프로젝트마다 상황이 다르고 예상하지 못했던 일들이 발생하게 되는데 이런 경우를 방법론에 모두 대응한다는 것은 불가능하다. 그럼에도 프로세스 완전성을 위하여 빠진 부분을 채우고 도구를 추가하여 자동화시키고 관리를 강화하여 프로세스를 강제로 적용하게 하려는 경우를 종종 보게 된다. 소프트웨어 공학의 좋은 소프트웨어를 개발하려는 목적을 놓쳐버리는 것이다.

1991년 JSS(Journal of Systems and Software)에 실린 "Controllable factors for programmer productivity: A statistical study" 논문에서는 이런 내용이 있다. 소프트웨어 업계에서는 프로그래밍 도구를 사용해도 노동생산성이 별로 달라지지 않는다는 것이다.

좋은 자동화 도구를 갖추면 생산성이 향상될 것이라는 기대와는 다른 결과이다. 소프트웨어 공학에서 자동화도구(Computer Aided Software Engineering)를 통한 생산성 극대화의 노력은 꽤 오래전부터 이루어져왔지만 아직까지도 획기적인 성과를 만들어내지 못하였다. 이는 소프트웨어 개발이라는 것이 사람이 만드는 것이고 창의적이고 복잡한 사고와 여러 사람간의 상호작용이 이루어지는 활동이라는 점을 다시 고려해본다면 놀라운 일도 아닐 것이다. 베리 보엠(Barry Boehm)은 저서인 Software Engineering Economics에서 프로세스가 아니라 사람이 좋은 제품과 생산성을 높이는 열쇠라고 지적하였다. 프로세스와 도구를 통해 생산성과 좋은 품질을 얻을 수 있다고 믿는 것은 아직 환상에 가깝다.

형식

소프트웨어 개발과 관련된 무수한 암묵적 과정들을 관리하기 위해서 뭔가 눈에 보이는 형식지로 만들 필요가 있다. 따라서 프로젝트 개발 과정에서 많은 보고서를 만들어 내야 한다. 특정한 지식과 기술력을 보유하고 있다는 것을 입증하기 위해 개인도 회사도 자격증이 있어야만 한다. 똑같은 상품이라도 브랜드가 있는 경우 가치가 달라지고 같은 말이라도 전문가가 하는 경우 힘이 실린다. 뭔가 형식을 빌려 표현하고 이를 통하여 차별화하기 위함이다. 그러나 이러한 형식물이 과연 본질을 잘 대변할 수 있는지를 판단하기는 쉬운 일이 아니다. 이를 구분할 정도의 판단력이 있다는 것은 형식을 꿰뚫어 볼 수 있는 전문적 지식을 요구하기 때문이다.

CMMi에서 최고 단계를 받은 업체가 과연 최고 품질의 제품을 만들어내는지, 프로젝트에서 보고한 테스트 결과서대로 제품의 품질이 보장되는지, 전문가의 의견이 정말로 맞는 것인지를 알기는 매우 어려운 일이다. 즉 그럴 수도 그렇지 않을 수도 있는 일이다. 따라서 보이는 것을 그대로 믿고 좋아하거나 슬퍼할 필요가 있는지 곰곰이 따져볼 필요가 있다.

측정

측정하지 못하면 관리하지 못한다는 말을 사람들은 자연스럽게 받아들인다. 소프트웨어 분야에서 뚜렷한 측정 지표나 성과가 별로 없음에도 불구하고 그동안 소프트웨어는 어떻게 발전되고 개선되어 올 수 있었을까? 부모에게 아이를 키울 때 모든 것을 측정해야만 그 아이를 관리할 수 있다고 얘기하지는 않는다. 관리자

입장에서 관리를 위해 지표가 필요하다는 점은 이해하지만 측정이 가져다주는 득과 실을 따져보는 것이 더욱 중요하다. 생산성을 측정하기 위해 개발자에게 작업시간을 입력시키는 활동이 개발 생산성을 낮추는 결과를 가져온다면 무엇을 위해서 측정을 해야 하는 것일까. 정량화된 수치가 사실을 왜곡시키거나 원하지 않는 결과를 가져올 수도 있다. 가령, 코드 라인 수로 개발자의 생산성을 측정하는 경우 개발자가 불필요한 코드량만 늘려 인당 생산성을 향상되었음에도 성능과 품질이 떨어질 수 도 있는 것이다. 무엇인가를 측정하기에 앞서 왜 측정하려고 하는지 그리고 측정 지표를 통해서 원하는 것을 정말로 볼 수 있는 것인지 생각해보자.

Ⅲ. 간직하고 있는 것

소프트웨어 개발 프로세스라는 것이 이론적인 영역뿐만 아니라 전문가의 경험과 노하우가 담고 있는 이상 공통적으로 유용하다고 인정되는 영역이 존재하기 마련이다. 가벼운 개발 프로세스이건 무거운 개발 프로세스이건 공통적으로 주장하는 프랙티스라는 것은 있다. 개인적으로 두 가지 프로세스를 실천하여 공통적이었다고 생각되는 것을 관리, 품질 그리고 개선으로 나눠서 설명하고자 한다.

<그림 6> 간직하는 것

관리

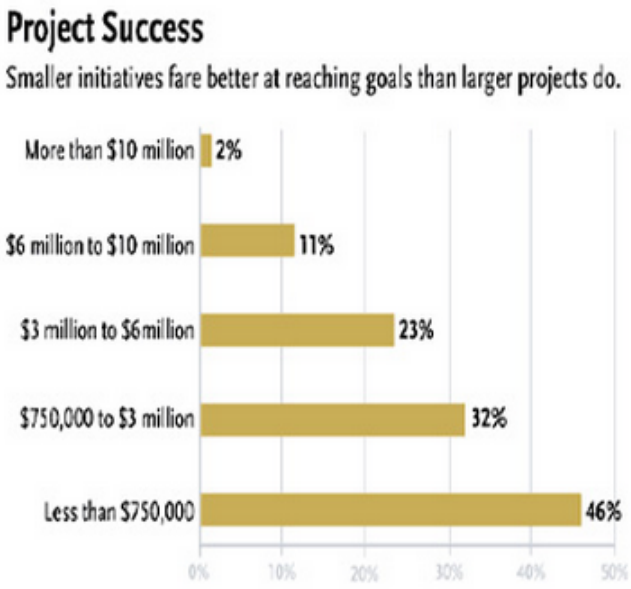
일반적으로 상위 단계에서 요구사항과 설계를 강조하는 무거운 개발 프로세스에 아키텍처는 매우 중요하다. 1:10:100의 원리에 따라 상위단계에서 Design-Up-Front를 통해서 오류나 요구사항을 검증하는 것이 하위단계에서 이를 수정하거나 변경하는 것보다 효율적이기 때문이다. 애자일과 같은 가벼운 프로세스에서는 점진적 반복 개발방식에서 아키텍처의 역할은 어떻게 바뀔까? 개념 아키텍처가 타당한가를 간단한 프로토타입을 통해 검증하고 개발팀원과의 지속적인 커뮤니케이션을 통해 이슈를 제거하며 시스템을 완성해 나간다. 상위단계에서 전체 시스템 아키텍처를 딱 정의하지 않다보니 개발팀과의 의사소통과 코드를 통제하며 점진적 통합을 이루어나가는 아키텍처의 역할은 매우 중요해질 수밖에 없다.

아무리 좋은 프로세스와 도구를 갖추더라도 결국 소프트웨어 개발은 사람이 하는 일이다. 최고와 최악의 프로그래머 간에 생산성 차이는 약 10배에서 30배까지 차이가 나며, 100명이 1년 걸릴 개발할 프로젝트를 200명을 넣는다고 6개월로 단축시킬 수 없다는 사실을 기억하자. 완벽한 프로세스에 대한 고민보다 좋은 인재를 어떻게 얻을 수 있을까를 고민해야 한다.

스탠디쉬 그룹(Standish Group)은 1994년부터 소프트웨어 개발 프로젝트의 성공 비율과 성공 요인을 분석하여 Chaos Report를 발행하고 있다. 일정, 비용, 품질을 충족시키는 프로젝트 성공률은 1994년 16%에서 2008년 32%까지 증가하였다. 그런데 성공한 프로젝트를 규모에 따라 분류해보면 예산이 $750,000 이하에서는 46%였던 것이 $10 million을 넘어가면 2% 이하로 급격하게 줄어든다. 프로젝트 규모가 증가하면 복잡성과 위험이 지수적으로 증가하게 된다. 애자일에서는 프로젝트 규모와 기간이 15명, 6개월 이상인 경우 프로젝트를 나누어 진행하는 것을 추천한다. 기간과 규모가 크다면 프로세스에 상관없이 프로젝트를 쪼개서 복잡성과 위험을 관리하는 것이 바람직하다.

<그림 7> 프로젝트 성공률과 프로젝트 규모

Benchmark/year	1994	1996	1998	2000	2004	2006	2008
Succeeded (%)	16	27	26	28	29	35	32
Challenged (%)	53	33	46	49	53	46	44
Failed (%)	31	40	28	23	18	19	24



(자료 인용: Standish Group Chaos Report)

품질

Quality Assurance를 우리말로 품질보증으로 번역하는데, 보증이라는 어감 때문인지 품질에 대한 책임을 품질보증부서 탓으로 생각하는 경우가 있다. Assurance는 보험처럼 확률상의 개념이지 뭔가를 책임지는 의미는 아니다. 품질은 제품 개발하는 개발자에게 1차적인 책임이 있다. 개발자끼리 서로의 코드를 검토하고 문제점을 찾고 해결책을 토론하는 코드 리뷰 활동은 프로세스에 상관없이 필요하다. 무거운 프로세스에서는 개발팀에서 정기적으로 코드 리뷰를 수행하거나 애자일 프로세스에서는 짝 프로그래밍(Pair Programming)을 통해 2명의 개발자가 하나의 개발 장비를 공유하며 실시간으로 개발과 검토를 수행해 나갈 수 있다.

막판 개발 일정에 쫓기게 되면 테스트 활동을 줄이는 것을 대안으로 쉽게 떠올린다. 과연 테스트를 줄이면 개발 일정이 줄어들까? 테스트를 줄이고 제품을 출시하는 경우 결함 수정비용뿐만 아니라 기업이미지와 신뢰도 상실에 따른 비즈니스 측면의 손해를 감안한다면 결코 대안이 되어서는 안 된다. 개발 일정을 줄이기 위해서는 무거운 프로세스는 요구사항, 설계, 코드 리뷰 등을 통해 테스트 단계 이전에 결함을 발견할 수 있도록 하고, 가벼운 프로세스에서는 테스트 주도개발(Test Driven Development)을 통해 테스트를 초기부터 시작해보자.

<그림 8> 짝 프로그래밍

개선

애자일 프로세스에 대한 많은 오해 중에 가장 흔한 것이 애자일에서는 문서를 작성하지 않는다는 것이다. 애자일이 문서를 통한 의사소통을 지양할 뿐이지 문서를 만들지 말라는 것은 아니다. 문서가 아니더라도 간단한 포스트 잇, 회의 과정에 칠판에 쓴 글을 사진을 찍거나 코드에 달아놓은 주석 등 소통에 필요한 내용을 담고 있다면 형식을 갖추지 못했더라도 문서는 남겨두도록 하자. 언젠가 사람은 조직을 떠나게 마련이다. 내가 보지 않더라도 남을 위해서 문서를 만들어두는 것이다.

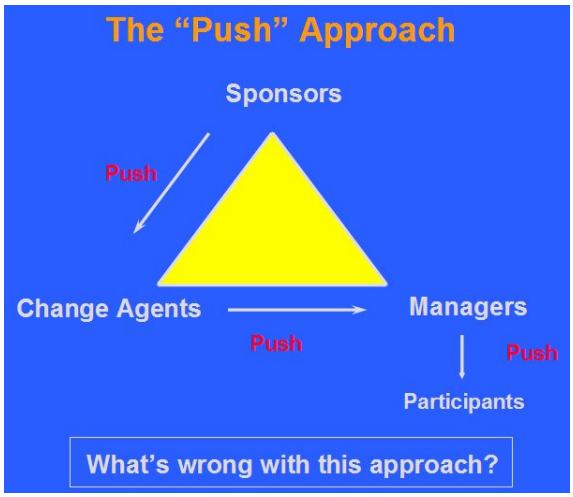
CMMi와 같은 무거운 프로세스를 그대로 가져다쓸 수 있는 조직은 이 세상 어디에도 없을 것이다. 모든 프로세스는 조직과 상황에 따라 맞춤질(Tailoring)을 해야만 한다. 애자일 프랙티스도 처음부터 모든 것을 다 수용할 수는 없다. 처음에는 쉬운 프랙티스를 도입하는 것도 괜찮다. 상황에 맞춰 새로운 것을 배우고 시도해보는 노력이 중요하다.

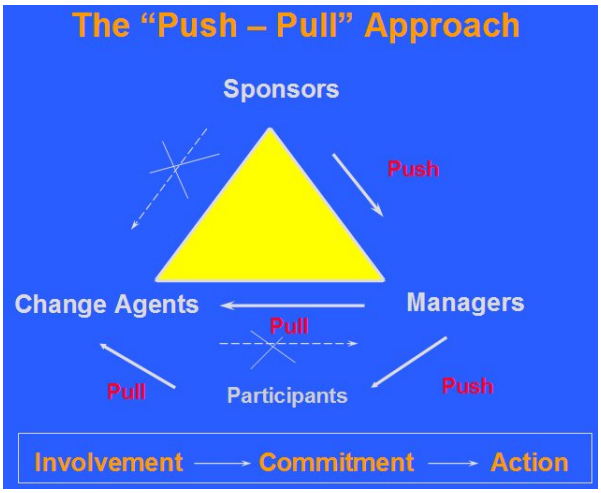
조직구조가 복잡하고 계층화되어 있다면 애자일과 같은 가벼운 프로세스를 도입하기도 어렵고 도입을 해도 복잡한 조직구조를 따르기 때문에 성공하기 어렵다. 역으로 무거운 프로세스라도 프로젝트 팀이 멀티 플레이어처럼 움직이고 수평적 구조로 의사결정이 이루어지면 프로세스를 심플하게 만들 수 있다.

조직에서 소프트웨어 개선을 담당하는 전사품질조직이 있고 경영목표에 따라 정해진 개선목표를 팀에게 부과합니다. 그리고 팀장은 이를 실무자에게 전달합니다. 이렇게 되면 프로세스 개선은 팀의 목표가 아닙니다. 개선담당 조직의 업무에 그저 동원되었을 뿐입니다. 따라서 프로세스 개선은 수동적이고 목표를 그저 밀어내는 것에 불과합니다.

이러한 푸시 접근(push approach)으로는 개선이 이루어질 수 없습니다. 개선 목표를 팀의 목표로 상황을 만들어 주어야 합니다. 즉, 개선조직의 목표가 아니라 팀의 목표가 되어 팀장과 실무자가 개선부서에 도움을 받아 품질을 개선할 때 변화가 이루질 수 있습니다. 이를 푸시-풀(push-pull approach)라고 합니다. 중요한 것은 개선을 수행하는 주체를 관여시키고 합의를 통해 행동을 만들어내는 것입니다.

<그림 9> 푸시 접근방법과 푸시-풀 접근방법





<참고 자료>

1. "Enough Process – Let's Do Practices", Journal of Object Technology, Ivar Jacobson, 2007
2. The Standish Group CHAOS report (프로젝트 성공요인 분석 보고서), http://blog.standishgroup.com/pmresearch
3. The Brain Has a Mind of Its Own, Richard Restak, Crown Publishers, 1993
4. 미국 프로젝트 관리자 협회, http://www.pmi.org/PMBOK-Guide-and-Standards.aspx
5. 애자일 이야기, http://agile.egloos.com
6. 소프트웨어 프로세스 이야기, http://swprocess.egloos.com.
7. Controllable factors for programmer productivity: A statistical study, L. Kemayel, Ali Mili, and I. Ouederni, Journal of Systems and Software, Vol. 16, No.2, 1991
8. Chaos report, http://blog.standishgroup.com/pmresearch
9. Software Engineering Economics, Barry Boehm, Prentice Hall, 1981