2 ▪ 원 칙 | *Principles*

원칙과 실천법

원칙principle은 시공을 초월하여 변하지 않는 근원적인 진실이며, 실천법practice은 원칙을 특정 상황에 맞게 적용한 것이다. 실천법은 처한 환경에 따라 달라질 수 있고 달라져야 하며, 상황 전개에 따라서도 변해야 한다.

어떤 팀에서 사용중인 소프트웨어 개발 실천법이 별로 효과가 없어서 이를 바꾸려 하는데, 어떤 것을 선택해야 할지 잘 모른다고 가정해보자. 다시 말해 어딘가에서 벗어나야 하는데 어느 쪽으로 가야 할지 모르는 상황이다. 소프트웨어 개발의 새로운 접근법을 찾을 때, 원칙을 이해하는 것과 실천법을 살펴보는 것 중 어디에 시간을 쓰는 것이 더 좋은 방법일까?

실행하면서 배우는learn by doing 접근법이 있다. 결국에는 그 기저에 깔린 원칙을 이해하게 될 것이라는 확신을 가지고 서로 밀접하게 연관된 실천법들을 먼저 도입하는 방식이 여기에 해당한다. 반면에 실행하기 전에 이해하는understand-before-doing 접근법이 있다. 근원적인 원칙을 먼저 이해한 뒤 원칙으로부터 특정 상황에 맞는 실천법을 개발하는 방식이다. 우리는 두 가지 접근법을 결합했을 때 가장 좋은 결과를 얻을 수 있음을 깨달았다. 기저에 깔린 원칙을 이해하지 못하고 실천법만 그대로 가져다 실행하는 것은 오랫동안 평범한 결과만 낳았다. 그러나 근원적인 원칙을 이해하고 나면, 유사 조직에서 좋은 성과를 낸 실천법을 가져와 자신의 환경에 맞도록 수정함으로써 유용한 것으로 만들 수 있다. 이렇게 하면 원칙들을 구현하기 위한 활동을 쉽고 빠르게 시작할 수 있다.

예를 들어, 메리의 비디오 테이프 공장에서 적시 생산방식(Just-in-Time, JIT)을 도입했을 때 그녀의 공장은 조립 공정이 아니라 가공 공정이었기 때문에 도요타의 실천법을 그대로 따라하기가 불가능했다. 경영진은 그들의 분야에서 JIT가 의미하는 것이 무엇인지 곰곰이 생각해야 했다. 그들은 도요타의 간판 시스템과 당김 스케줄링pull scheduling과 같은 실천법을 도입하기로 결정했다. 그러나 재고 감축만으로는 비약적인 비용 절감을 이루기에 충분하지 않다는 것을 알았고, 그래서 결국 제조 현장의 작업자들을 참여시키는 독특한 방법을 개발하여 라인정지stop-the-line 문화를 만들어 냈다. 그들은 원칙을 이해하는 것과 실천법을 적응시키는 것을 잘 조합하여 극적인 성공을 일궈냈다.

90년대에는 많은 회사들이 도요타의 간판 시스템을 따라했지만 그저 그런 결과밖에 얻지 못했다. 그 회사들은 린이 낭비 제거를 목표로 한 경영 시스템이라는 것을 깨닫지 못했다. 아마도 간판 시스템을 도입했던 많은 회사들이 그들의 가치흐름value stream상에 많은 낭비를 남겨두었을 것이라고 추측된다. 마찬가지로 많은 회사들이 이 책에서 소개하는 애자일 소프트웨어 개발 실천법들을 적용하려 하겠지만, 낭비를 인식하지 못하고, 그것을 제거하기 위해 가치흐름을 관리하지 않고, 직원과 파트너들을 존중하지 않는다면 그 결과는 또다시 그저 그런 것이 되고 말 것이다.

소프트웨어 개발

제조업과 공급망관리supply chain management에서의 린 실천법들은 소프트웨어 개발로 쉽게 변환되지 않는다. 왜냐하면 소프트웨어와 개발은 둘 다 각각 공정 운영이나 물류와는 사뭇 다르기 때문이다. '소프트웨어'와 '개발'이라는 두 단어를 각각 살펴보고 그 독특함이 어디에 있는가를 분석해보기로 하자.

소프트웨어

임베디드 소프트웨어는 제품에서 변경이 가능한 부품이다. 만약 변경할 필요가 없다면 그것은 하드웨어나 마찬가지다. 기업 소프트웨어는 그 복잡성을 감당해야 하는 비즈니스 프로세스의 일부분이다. 만약 까다로운 계산이나 복잡한 정보의 흐름을 처리할 일이 있으면 그것은 소프트웨어의 몫이 된다. 소프트웨어는 사용자 상호작용이나 마지막 순간에 해야 할 작업 또는 '나중에 신경 쓸' 작업을 담당하게 된다. 이런 이유로 훌륭한 소프트웨어 아키텍처의 특징이라고 할 만한 것들은 거의 모두가 소프트웨어를 쉽게 변경할 수 있도록 하는 것과 관련있다.[1] 그리고 이것은 놀랄 일이 아니다. 왜냐하면, 전체 소프트웨어의 절반 이상이 첫 출시 이후에 개발되기 때문이다.[2]

시간이 지남에 따라 양산 소프트웨어를 수정하는 것은 점점 더 어려워지고 비용은 더 많이 소요되게 마련이다. 수정은 복잡도를 증가시킨다. 이러한 복잡함은 대개 코드베이스code base를 경화시켜서 깨지기 쉽게 만든다. 너무나 많은 기업들이 자사의 소프트웨어에 많은 투자를 했음에도 불구하고 결국 손댈 수 없이 뒤엉켜버린 코드를 만들어냈음을 발견하곤 깜짝 놀란다. 그러나 모든 소프트웨어가 항상 비극으로 끝나는 것은 아니다. 최고의 소프트웨어 제품들은 10년 또는 그 이상 사용되기도 하는데, 그 정도의 긴 시간 동안 사용되는 모든 제품은 출시 이후에도 정기적인 수정이 이루어진다. 이러한 제품들은 변화를 허용하는 코드를 만들어내는 아키텍처와 개발 프로세스를 갖고 있다. 소프트웨어라고 부를 만한 모든 코드는 반드시 변화허용성을 염두에 두고 설계하고 구현해야 한다.

개발

개발은 아이디어를 제품으로 변환하는 과정이다. 이러한 변환의 방법에 대한 두 가지 부류의 사고가 있는데 하나는 결정론적인 사고이고 다른 하나는 경험론적인 사고이다. 결정론적 부류는 제품에 대한 완벽한 정의를 먼저 만들고 그 정의를 실현한다. 경험론적 부류는 상위 수준의 제품 컨셉으로 시작하여 잘 정의된 피드백 루프를 통해 그 컨셉에 대한 최적의 해석을 이루어낸다.

도요타 제품개발방식Toyota Product Development System은 정확히 경험론적 부류에 속하며, 차량에 대한 정의보다는 컨셉으로 시작하여, 개발 프로세스를 거치면서 경험적으로 그 컨셉을 제품화해 나간다. 예를 들어 프리우스Prius의 제품 컨셉을 살펴보면 하이브리드 엔진을 언급하지 않고, 단지 연비 목표를 리터 당 20km로 설정했을 뿐이었다. 또한, 프리우스의 제품

1 예를 들어 짐 쇼어(Jim Shore)는 'Quality With a Name'에서 고품질 소프트웨어 아키텍처를 다음과 같이 정의했다. "좋은 소프트웨어 설계는 만족할 만한 실행 성능을 가지면서 소프트웨어의 생성과 수정 그리고 유지보수 시간을 최소로 하는 설계이다." www.jamesshore.com/Articles/Quality-With-a-Name.html 참조.

2 소프트웨어 생명주기에서 '유지보수'에 들어가는 비용은 40%에서 90%에 이른다. 참조 논문: Kajko-Mattsson, Mira, Ulf Westblom, Stefan Forssander, Gunnar Andersson, Mats Medin, Sari Ebarasi, Tord Fahlgren, Sven-Erik Johansson, Stefan Törnquist, and Margareta Holmgren, Taxonomy of Problem Management Activities, Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, March 2001, 1 – 10

컨셉은 승객을 위한 실내공간이 넓어야 한다고만 하고 차량의 치수를 지정하지는 않았다. 단지 그 개발 과정에서 도전적인 연비 목표를 달성하기 위한 최선의 방법으로 실험실에서 갓 나온 하이브리드 엔진을 얹었던 것이다.[3]

우리는 변화하는 환경을 다루는 개발 프로세스라면 경험론적 프로세스여야 한다고 믿는다. 왜냐하면, 그것이 변화에 적응하기 위한 최선의 방법을 제공하기 때문이다. 위에서 언급한 것처럼 소프트웨어는 그 본질상 초기 개발 기간뿐만 아니라 수명이 다할 때까지 변화에 적응할 수 있도록 설계되어야 한다. 따라서 소프트웨어 개발 프로세스는 반드시 경험론적 프로세스여야 한다.

Story '폭포수'란 도대체 무엇인가?

정부 프로젝트에서 일하기 시작한 해인 1999년이 될 때까지 나는 폭포수waterfall란 말과 마주칠 일이 없었다. 나는 70년대에 자동제어 소프트웨어를 개발하는 (꽤 훌륭한) 프로그래머였다. 지금이라면 그것을 임베디드 소프트웨어라고 부르겠지만 당시의 컴퓨터는 끼워넣기에는 embedded 너무 컸다. 나는 비디오 테이프를 만드는 매우 복잡한 공정라인을 짓고 가동시키는 대규모 프로젝트에서 일하고 있었다.[4] 이 프로젝트에 참여한 나의 동료들은 대규모의 복잡한 테이프 생산라인을 개발하는데 다년간의 경험을 가진 엔지니어들이었다. 프로젝트 관리는 숙련된 전문가들이 맡았다. 그들은 전체 예산 및 일정에 대한 승인을 얻는 방법이나, 필요에 따라 계획을 조정할 수 있도록 전문가들에게 업무를 이관하는 방법을 잘 알고 있었다. 예산과 일정도 중요하지만, 무엇보다도 중요한 목표는 최고의 제품을 생산해내는 라인을 구축하는 것이었다. 우리 모두는 그 점을 잘 알고 있었다. 그리고, 항상 그렇게 했다.

비디오 카세트 공장의 정보 시스템 관리자가 되었을 때, 나는 엔지니어 시절에 배운 것들을 이용하여 신규 소프트웨어 개발을 관리했다. 이 때도 우리 부서의 가장 중요한 업무는 생산을 지원하는 것이란 사실을 잊지 않았다. 후에 나는 제품 개발 부서로 이동하였다. 거기서는 새로운 제품을 상품화하기 위해 엄격하지만 매우 유연한 단계별 심의stage-gate 프로세스를 사용했다.

그렇게 함으로써, 나는 1999년 정부 프로젝트에 우연히 투입되기 전까지는 폭포수 방법론으로 일하는 것에서 벗어날 수 있었다. 후에 폭포수 접근법을 접하고는 무척 당황스러웠다. 대체 그런 방법으로 어떻게 성공할 수 있는지 도저히 이해할 수 없었기 때문이다. 그리고 사실상 그 방법은 성공하지 못했다. 복잡했지만 성공적이었던 프로젝트 경험과 비교적 작은 프로젝트에서도 실패한 폭포수 방법을 비교해보고 나는 정말 효과가 있는 것이 무엇인가에 관한 책을 쓰기로 결심했다.[5] 그 책에서 우리는 소프트웨어 개발의 7가지 원칙을 개략적으로 설명했는데 바로 다음 절에 요약해 두었다.

3 Jeffrey Liker, The Toyota Way, McGraw-Hill, 2004. 6장의 프리우스 개발 부분 참조.

4 테이프를 만드는데 필요한 장비들은 1920년대 도요타 자동 직조기 社에서 만든 자동 직조기와 유사한 면이 있다.

— 메리 이야기

린 소프트웨어 개발의 7가지 원칙

이번 절에서는 앞서 출간한 책의 핵심인 소프트웨어 개발의 7가지 원칙을 요약하였다. 일부 독자들은 몇 가지 원칙에 대한 용어가 달라졌음을 발견할 수도 있을 것이다. 하지만 그 의도는 같다. 각각의 원칙을 설명하고 나서 그 원칙과 관련하여 널리 퍼져있는 잘못된 통념을 덧붙였다. 통념을 사실로 믿고 있는 사람들에게는 그 원칙이 직관에 위배된다고 느껴질 것이다.

원칙 1: 낭비를 제거하라

오노 다이이치(大野 耐一)는 도요타 생산방식Toyota Production System을 '낭비를 완전히 제거하기 위한' 경영 시스템이라고 불렀다.[6] 그것이 어떻게 운영되는가를 물었을 때 그는 다음과 같이 말했다. "우리가 하는 모든 것은 고객이 우리에게 주문을 내릴 때부터 우리가 현금을 손에 넣을 때까지의 시각표timeline를 살펴보는 것이다. 그리고 가치를 더하지 못하는 낭비들을 제거함으로써 그 기간을 단축하는 것이다."[7] 이것은 간단히 말해 린 생산Lean Production에 관한 것이다. 린 소프트웨어 개발에서도 낭비를 제거한다는 목표는 같지만 시각표의 시작과 끝은 수정되어야 한다. (조직마다 의미는 다르겠지만) 시각표는 고객 요구를 반영한 주문을 받을 때 시작되며 요구를 만족시키는 소프트웨어를 배포할 때 끝난다. 린 소프트웨어 개발

5 Mary and Tom Poppendieck, "Lean Software Development: An Agile Toolkit", Addison-Wesley, 2003.

6 Taiichi Ohno, Toyota Production System: Beyond Large Scale Production, Productivity Press, 1988, p. 4.

7 6)번 각주와 동일함, p. ix.

은 가치를 더하지 않는 모든 낭비를 제거하여 그 기간을 단축하는데 중점을 두고 있다.

낭비를 제거하기 위해서는 먼저 낭비를 인지해야 한다. 가치를 더하지 못하는 어떤 것도 낭비이므로 낭비를 제거하기 위한 첫 단계는, 가치가 정말 무엇인지를 감지할 수 있는 예리한 감각을 개발하는 것이다. 고객이 소프트웨어를 사용하기 시작했을 때 그들이 무엇을 가치있다고 평가할 것인지를 깊이 이해하는 것 외에 다른 방법은 없다. 소프트웨어 업계에서는 가치 자체가 변하는 특성을 갖고 있다. 고객들은 그들이 진정 원하는 것이 무엇인지 모르고 있는 경우가 너무도 많기 때문이다. 더구나 새로 만들어진 소프트웨어가 돌아가는 것을 보고 나면 원하는 것이 언제나 변하게 마련이다. 그럼에도 불구하고 훌륭한 소프트웨어 개발 조직은 고객가치에 대한 깊은 통찰력을 개발하여 계속해서 그들의 고객을 기쁘게 한다. 구글을 보라. 매번, 어김없이 전 세계의 고객을 기쁘게 하고 있다.

일단 가치를 잘 이해하고 나면, 다음 단계는 현실에서 낭비를 보는 능력을 개발하는 것이다. 낭비란 고객에게 가치를 적시 적소에 전달하는 것을 방해하는 모든 것을 의미한다. 고객 가치를 더하지 않는 모든 것은 낭비이며, 고객이 원하는 순간에 가치를 제공받지 못하게 하는 모든 지연도 낭비다.

제조 분야에서 재고는 낭비다. 재고가 있으면 이를 통제하고, 옮기고, 저장하고, 추적하고, 꺼내와야만 한다. 이것은 시간과 노력을 빼앗아 갈 뿐만 아니라 복잡도를 증가시켜 비용을 몇 곱절로 늘어나게 한다. 재고는 분실되거나 못 쓰게 되고, 품질 문제를 가리고, 돈을 묶어놓는다. 따라서 제조에서의 한 가지 목표는 가능한 한 재고를 최소한으로 가져가는 것이다.

소프트웨어 개발에서의 재고는 미완성 작업partially done work이다. 미완성 소프트웨어는 제조에서의 재고가 가지는 해악을 모두 갖고 있다. 그것은 분실되거나 못 쓰게 되고, 품질 문제를 가리고, 돈을 묶어놓는다. 거기다가 소프트웨어 개발의 리스크 중 상당수가 미완성 작업에서 비롯된다.

소프트웨어 개발에서 낭비의 전형적인 형태는 '혼란'[8]이다. 교육에 참가했던 사람들을 보면 30%에서 50% 정도의 참가자들이 '요구사항 혼란'을 경험했고 테스트하고 고치는test-and-fix 기간이 초기 개발보다 2배 이상 걸리는 경우도 많이 보았다. 우리는 소프트웨어 개발에

8 churn. 버터 제조용 우유 교반기를 말하는데 여기서는 개발 팀이 내외부적 요인에 의해 야기된 혼란스러운 상황을 의미한다. 흔히 이러한 형태의 혼란을 '삽질'이라고 표현한다.

서의 혼란이 언제나 잔뜩 쌓인 미완성 작업과 관련되어 있다는 것을 발견했다. 요구사항이 코딩보다 훨씬 먼저 정의된다면 요구사항은 당연히 변할 것이다. 테스팅이 코딩보다 한참 후에 이뤄진다면 테스트하고 고치는 혼란은 피할 수 없다. 불행히도 이런 종류의 혼란은 향후 지연된 통합(소위 빅뱅 통합이라고 불린다)에 의해 발생할 더 큰 혼란에 대한 예비조짐일 뿐이다.

그러나 소프트웨어 개발에서 무엇보다 가장 큰 낭비는 가외 기능extra feature이다. 일반적인 커스텀 소프트웨어(고객 주문 소프트웨어)의 경우 대개는 20% 정도의 기능만이 일상적으로 사용된다. 다시 말해서, 약 3분의 2에 달하는 기능이 거의 사용되지 않는다는 얘기다.[9] 직접 사용하진 않지만 꼭 필요한 안전이나 보안 등의 기능을 말하는 것이 아니다. 애초부터 필요하지 않았던 기능들을 말하고 있는 것이다. 소프트웨어의 가외 기능을 개발하는데만 막대한 비용을 지불하고 있는 셈이다. 가외 기능은 코드베이스를 더 복잡하게 만들고 이는 다시 놀라운 속도로 비용을 증가시키고, 유지보수 비용도 훨씬 더 비싸게 만든다. 결국 소프트웨어의 수명을 현저히 감소시킨다.

잘못된 통념: 스펙 조기 확정이 낭비를 줄인다

이러한 가외적인 코드를 개발하게 되는 것은 고객과 벌이는 게임 방식에서 비롯된다. 우리는 흔히 이런 식의 규칙을 세운다.

> 존경하는 고객 님, 소프트웨어에 원하는 기능을 빠트리지 않고 목록으로 만들어 주십시오. 그러면 저희가 그것을 문서로 만들어 고객 님의 승인을 요청하겠습니다. 승인 후 변경이나 추가를 하고 싶으면 '변경관리'라고 불리는 매우 복잡한 절차를 밟아서 변경승인을 얻으셔야 합니다. 그러니 고객님이 바라는 모든 것을 지금 당장 생각해내야 합니다. 왜냐하면 좋은 소프트웨어를 만들어내기 위해서는 개발을 시작할 때 모든 요구사항을 알아야 하기 때문입니다.

이런 상황에서 고객이 요구사항 리스트에 이것저것 온갖 것을 다 집어넣는 것이 놀랄 만한 일인가? 이러한 방식의 범위 조정 게임은 너무나 자주 정반대의 효과를 낳는다. 즉, 쓸데없는 범위 팽창을 야기하게 되는 것이다. 오노 다이이치가 과잉생산을 제조에서의 가장

9 스탠디쉬(Standish) 그룹의 회장인 짐 존슨(Jim Johnson)이 사르디니아 공국에서 열린 XP 2002에서 보고한 이 비율은 제한된 연구결과에서 나온 것이었다. 그 후로 우리는 만나는 그룹(특히 커스텀 소프트웨어를 개발하는 그룹)들마다 이 숫자가 그들의 경험에 비추어 맞는 것인지를 물었다. 그들의 응답은 그 숫자가 완전히 똑같지는 않더라도 그 언저리에 있음을 확인시켜 주었다.

큰 낭비라고 불렀듯이 사용되지 않는 기능은 소프트웨어 개발에서도 최악의 낭비이다. 필요하지 않으면서 존재하는 모든 코드 조각이 복잡도를 높여서 코드베이스의 남은 일생을 계속 괴롭힐 것이다. 사용되지 않는 코드로 인해 쓸데없이 테스팅, 문서화, 유지보수 업무가 늘어난다. 이것은 시간이 지남에 따라 코드베이스의 유연성을 갉아먹고 코드를 이해하거나 수정하기가 점점 더 어렵게 만든다. 코드 복잡도에 따른 비용은 다른 모든 비용을 압도한다. 가외 기능들은 소프트웨어 생산성을 가장 크게 저해하는 요인중 하나다.

우리에게 필요한 프로세스는 가치의 80%를 제공하는 20%의 코드를 먼저 개발하고 다음 단계에는 그 다음으로 가장 중요한 기능을 개발하도록 하는 프로세스이다. 어쩌면 필요할 수도 있는 모든 것들을 포함하는 목록을 개발 대상으로 잡아서는 절대로 안된다. 특히나 그 목록이 아직 자신이 정확히 무엇을 원하는지 잘 모르는 고객으로부터 나온 것이라면 더더욱 그렇다.

원칙 2 : 품질을 내재화하라

회의적인 관리자들은 "여기서는 규율이 더 많이 필요합니다. 규율을 줄여서는 안됩니다."라고 말하곤 한다. 그럴 때마다 우리는 린 소프트웨어 개발이 바로 규율적인 방법이라고 대답한다. 단지 사람마다 규율을 보는 시각이 좀 다를 뿐이다. 여러분의 목표는 시작부터 코드에 품질을 내재화하는 것이지 나중에 품질을 테스트하는 것이 아니다. 결함 추적 시스템에 결함을 기록하는 데 집중하지 말고, 결함이 처음부터 생기지 않도록 해야 한다. 그렇게 하려면 고도로 규율화된 조직이 되어야 한다.

Story 결함 대기열

이미 CMM 레벨 4를 획득했고 레벨 5에 근접한 조직을 방문한 적이 있다. 우리는 그 사람들과 규율에 감명을 받았지만 그들의 제품에 대해서는 별로 놀라지 않았다.

그러나 사건이 발생했다. 최근 들어 릴리스 주기가 6주에서 4개월로 길어진 것이다. 그들은 우리에게 그 원인을 찾는 것을 도와달라고 요청했다. 우리는 칠판에 릴리스 주기의 일정표를 그렸는데, 거기서 주기 마지막에 4주간의 테스팅 기간이 있는 것을 발견했다. 나는 "6주 안에 릴리스를 못하는 것은 당연하네요. 여러분들은 지금 테스팅에만 4주를 쓰고 있습니다."라고 말했다.

품질 담당자는 이렇게 대답했다. "압니다. 테스팅을 자동화해야 합니다. 하지만 문제가 그렇게 간단하지는 않은 것 같습니다." 나도 동의했다. 이 조직은 최고의 조직이었다. 아마도 자동화된 테스팅만으로 해결할 수 없는 무언가가 더 있을 것 같았다.

"4주간의 테스팅에 대한 파레토 분석[10]을 해 봅시다." 라고 제안하면서 "4주간의 테스팅 기간 동안 가장 많은 시간이 드는 곳은 어디인가요?"라고 물었다.

"결함 수정요."라는 즉각적인 대답이 돌아왔다.

"그렇다면 여러분들은 그 4주 동안 테스팅을 하는 것이 아니라 사실은 결함을 수정하는 것이로군요! 그러면서 테스팅이라고 부르는 것은 적절하지 않습니다. 만약 고쳐야할 결함이 전혀 없다면 그때는 테스팅이 얼마나 걸릴까요?"

"아마도 2, 3일 정도 걸리겠죠." 라는 대답이 돌아왔다.

"만약 테스팅을 2, 3일 내에 끝낼 수 있다면, 그때는 다시 예전처럼 6주마다 릴리스를 할 수 있을까요?" 라고 물었다.

"물론이죠." 라는 즉각적인 대답이 돌아왔다.

"여러분들은 개발 프로세스 상에서 너무 늦게 테스팅을 하는 것으로 보입니다. 이런 결함들은 훨씬 더 일찍 발견되어야 합니다." 라고 제안했다.

그러자 그는 "아, 우리는 벌써 그렇게 하고 있어요! 결함 추적 시스템에 다 기록해 두거든요." 라고 말했다.

"그럼 그 결함들을 이미 알고 있으면서 마지막까지 아무 조치를 취하지 않는다는 말씀이세요?" 나는 놀라서 물었다.

"네…" 라는 소심한 대답이 돌아왔다.

"좋습니다. 여러분도 이미 무엇을 해야 할지 알고 있는 것 같습니다. 결함을 발견하면 그것을 기록하지 말고 그냥 고치세요! 최종 검증에서 코드가 잘 동작할 거라는 기대를 갖고 마지막 테스팅을 2일 정도로 단축하세요." 이것이 중요하다고 결정하기만 하면 이 뛰어난 조직은 내가 제안한 목표를 달성할 수 있을 것이라고 나는 확신했다.

— 메리 이야기

10 파레토 분석은 '80/20'의 법칙으로도 불리는 '중요한 소수와 대수롭지 않은 다수'의 법칙을 응용한 것으로 품질의 대가인 조지프 주란(J.M. Juran)에 의해 처음으로 널리 알려지게 되었다. 그 분석은 따라서 다음과 같이 진행된다. 문제의 범주를 나누고 가장 많은 문제에 해당하는 범주를 찾아서 그 범주를 만들어낸 문제의 근본원인을 찾아 이를 해결한다. 가장 큰 문제의 원인이 제거되면 이제 남아 있는 문제들에 대해 다시 파레토 분석을 반복한다. 7장에는 이 기법을 사용하는 좀 더 자세한 예제가 담겨 있다.

신고 시게오(新郷 重雄)에 의하면 검사inspection에는 두 가지가 있다. 하나는 결함이 발생한 후에 하는 검사이고 다른 하나는 결함이 발생하기 전에 예방하기 위한 검사이다.[11] 만약 여러분이 정말 고품질을 추구한다면 결함이 발생한 후 검사할 것이 아니라 처음부터 결함이 발생하지 않도록 조건들을 통제해야 한다. 만약 이것이 불가능하다면 각각의 작은 단계가 끝날 때마다 검사하여 결함이 발생하자마자 즉시 잡아내도록 해야 한다. 결함이 발견되면 라인을 멈추고, 원인을 찾아서, 즉시 고쳐야 한다.

결함 추적 시스템은 미완성 작업의 대기열이다. 만약 여러분이 결함을 수정하려고 덤비면 그때부터는 재작업의 대기열이다. 우리는 너무나 자주 결함이 대기열에 들어가 있다는 것만으로, 이제 그 결함을 빼먹지 않고 추적할 수 있을 테니 문제가 없을 것이라고 생각한다. 그러나 린의 관점에서 보면 대기열은 낭비의 집합소다. 목표는 대기열에 결함이 없도록 하는 것이고, 더 궁극적인 목표는 결함을 추적하는 대기열까지 제거하는 것이다. 만약 이것이 불가능하다고 생각한다면 낸시 반 슈엔더뵈르트Nancy Van Schooenderwoert의 경험을 살펴보자.[12] 그녀는 복잡하고 변경이 잦은 임베디드 소프트웨어를 개발하는 3년 동안 단위 테스트 후 발생한 결함이 모두 합쳐 51개 뿐이었으며 어느 순간도 결함이 최대 2개를 넘지 않았다. 누가 2개의 결함을 기록하기 위해 결함 추적 시스템을 사용하겠는가?

오늘날 우리는 개발 중인 코드 결함의 대부분을 걸러낼 수 있는 도구를 갖추고 있다. 테스트 주도 개발test-driven development 방식을 따르는 개발자들은 코드를 작성하기 전에 단위 테스트와 인수 테스트를 먼저 작성한다. 그들은 최대한 자주 (대략 1시간마다) 코드와 테스트를 함께 통합하여 테스트 하니스[13]를 실행하고 결함이 없음을 확인한다. 만약 테스트가 실패하면 문제를 해결하거나 오류를 일으킨 코드를 지우기 전까지 새로운 코드를 추가하지 않는다. 퇴근 무렵에는 더 길고 완전한 테스트 하니스를 실행하고, 매 주말에는 시스템에 더욱 더 포괄적인 테스트 하니스를 붙여 실행한다. 이러한 프로세스를 사용하는 조직은 발견된 결함의 원인을 매우 짧은 시간에 찾아낼 뿐만 아니라 믿을 수 없을 정도로 낮은 결함율을 나타낸다.

11 Shigeo Shingo, Study of 'Toyota' Production System, Productivity Press, 1981, 2.3장 참조.

12 Nancy Van Schooenderwoert and Ron Morsicato, "Taming the Embedded Tiger–Agile Test Techniques for Embedded Software," Proceedings, Agile Development Conference, Salt Lake City, June 2004, 그리고 Nancy Van Schooenderwoert, "Embedded Agile Project by the Numbers with Newbies," Proceedings, Agile 2006 Conference, Minneapolis, July 2006 참조

13 Test Harness – 시스템 및 시스템 컴포넌트를 시험하는 환경의 일부분으로 시험을 지원하는 목적하에 생성된 코드와 데이터. 시험 드라이버 라고도 하며 일반적으로 단위 시험이나 모듈 시험에 사용하기 위해 코드 개발자가 만든다.

Story 생산성 급등[14]

테스트 주도 개발TDD은 코드 품질을 개선하는데 놀라울 정도로 효과적인 방법이다. 2004년 2월, 나는 품질 문제로 씨름하고 있는 한 회사에 이 기법을 소개했다. 모든 코드가 기껏해야 5년밖에 안 되었지만 (Java로 작성된 시스템이었다.) 그들은 레거시 애플리케이션으로 간주하고 있었다. 출시 후 6개월 동안 주석을 제외한 순수 실행코드 기준으로 1,000라인 당 평균 10개의 결함이 보고되고 있었다.

열악한 품질은 그 회사의 명성과 고객만족에 타격을 주었으며, 적시에 새로운 기능을 추가하는 능력을 저하시키고 있었다.

이 문제를 해결하기 위해 그 팀은 스크럼[15]이라는 애자일 개발 프로세스를 사용하였고 곧 TDD를 채택하였다. TDD를 채택한 이후 보고된 결함 수는 1,000라인 당 3개 이하로 줄어들었다. 사실은 70%의 결함 감소보다도 더 큰 개선을 이루어냈다. 왜냐하면 보고된 결함 중에서 TDD를 적용하기 전에 작성된 코드에서 나온 것을 따로 추적하지 않았기 때문이다. 즉 보고된 결함 중 일부는 TDD를 적용하기 전에 작성한 코드의 결과이다. 보수적으로 보더라도 그 팀은 대략 80~90%의 개선을 이루었다고 말할 수 있다.

품질이 향상되자 효과는 여러 형태로 나타났다. 결함(개발 중에 발생한 것과 사용자가 보고한 것까지 모두 포함)을 잡아내는데 훨씬 더 적은 시간을 사용했기 때문에 생산성이 급등했다. TDD 적용 전에는 한 사람이 한 달에 기능 점수function point로 7점을 개발했는데, TDD 적용 후에는 23점이나 개발하게 되었다. 그 팀은 결함을 훨씬 적게 내면서도 3배 이상의 생산성 향상을 이룬 것이다.

– 마이크 콘Mike Cohn, 마운틴 고트 소프트웨어Mountain Goat Software 社 대표

잘못된 통념: 테스팅의 역할은 결함을 발견하는 것이다

테스트의 역할 그리고 테스트를 개발하고 실행하는 사람들의 역할은 결함을 '예방'하는 것이지 발견하는 것이 아니다. 품질보증 조직은 개발이 끝난 뒤에 품질을 테스트하는 것보다, 시작부터 품질을 코드에 내재화하는 프로세스를 구현해야 한다. 검증이 불필요하다고 말하는 것이 아니다. 최종 검증은 좋은 생각이다. 다만 검증하는 동안 결함을 발견하는 것은 예

14 경험을 들려준 마이크 콘에게 감사한다. 허가 받아 인용함.

15 스크럼(Scrum)은 잘 알려진 반복적 개발방법론이다. Ken Schwaber and Mike Beedle, Agile Software Development with SCRUM, Prentice Hall, 2001, and Ken Schwaber, Agile Project Management with Scrum, Microsoft, 2004. 참조.

외사항이지 규칙이 되어서는 안된다. 만약 검증이 일상적으로 테스트와 수정의 사이클을 낳는다면, 개발 프로세스 자체에 결함이 있는 것이다.

메리의 공장에서는 사람의 실수에 의해 발생한다고 생각되는 결함의 80%가 사실은 그런 실수를 저지를 수 있게 한 시스템 때문에 발생한다는 것이 상식으로 되어 있다. 따라서, 대부분의 결함은 사실상 관리상의 문제이다. "처음부터 똑바로 하라" 라는 슬로건은 사후 검사를 없애기 위한 표어였다. 이것은 각 제조 단계에서 실수 방지mistake-proofing를 함으로써 결함 유발을 쉽게 피하도록 하는 것이었다.

우리가 소프트웨어 개발에 "처음부터 똑바로 하라"라는 슬로건을 사용했을 때는 테스트 주도 개발과 지속적인 통합continuous integration[16]을 사용하여 코드가 항상 의도한 대로 정확히 동작해야 한다는 의미였다. 그러나 불행히도 그 슬로건은 전혀 다른 의미로 사용되었다. "처음부터 똑바로 하라"는 말이 코드가 한번 쓰여지면 절대 변경되어서는 안된다는 것을 의미하는 것으로 해석되었던 것이다. 이러한 해석은 개발자들로 하여금 복잡한 시스템을 설계하고 개발하는데 최악의 실천법들을 사용하는 것을 조장하였다. 소프트웨어가 한번 작성되면 절대 바뀌어서는 안된다는 생각은 매우 위험한 잘못된 통념이다.

소프트웨어 개발에서 낭비를 제거하는 최고의 방법을 다시 살펴보자. "코드를 더 적게 짜라." 코드를 더 적게 짜기 위해서는 가치의 80%를 제공하는 20%의 코드를 발견하여 그것을 먼저 개발해야 한다. 이런 식으로 다음 기능들을 추가하다가 기능 추가에 따른 비용보다 얻게 되는 가치가 더 적은 경우에 작업을 멈춘다. 기능을 추가할 때는 코드를 간결하고 결함이 없게 유지해야 하며 그렇지 못하면 복잡도가 곧 여러분을 위협할만한 수준으로 높아질 것이다. 그래서 우리는 새로운 기능을 추가하기 위해 설계를 리팩터링하여 코드베이스를 간결하게 유지하고, 소프트웨어의 가장 큰 적인 '중복'을 제거한다. 이것은 기존 코드를 늘 수정해야 한다는 것을 의미한다.

원칙 3: 지식을 창출하라

'폭포수 모델'이 우리를 곤혹스럽게 만드는 측면 중 하나는, 지식이 '요구사항'이라는 형태로 코딩과 분리되어 존재한다는 생각이다. 소프트웨어 개발은 지식을 창출해가는 프로세스다.

16 Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007

전체적인 아키텍처 개념은 코딩 전에 개략적으로 그려질 수 있지만 그 아키텍처의 검증은 코드가 작성될 때 이루어진다. 소프트웨어의 상세 설계는 설령 그것이 문서로 먼저 작성된다고 하더라도 실제로는 코딩 중에 이뤄진다. 초기 설계에서는 구현 중에 마주치는 복잡성을 충분히 예측할 수 없을 뿐만 아니라 실제 소프트웨어를 개발하면서 얻게 되는 피드백을 고려할 수도 없다. 더 나쁜 것은 초기의 상세 설계는 이해 당사자나 고객으로부터 피드백을 받기가 힘들다는 것이다. 지식을 창출하는 데 중점을 두는 개발 프로세스는 설계가 코딩 중에 진화할 것이라고 기대하고, 설계를 너무 일찍 확정하는 데 시간을 낭비하지 않을 것이다.

하버드 비즈니스 스쿨의 앨런 맥코맥Alan MacCormack은 조직이 어떻게 학습하는가를 연구하는데 평생을 보냈다. 예전에 그가 소프트웨어 개발 실천법을 공부하고 있을 때 한 회사로부터 2개의 프로젝트('좋은' 프로젝트와 '나쁜' 프로젝트)를 평가해 달라는 부탁을 받았다.[17] '좋은' 프로젝트는 교과서대로 운영되었다. 그것은 최적의 설계를 갖고 있었고 그렇게 만들어진 시스템은 초기의 명세와 매우 가까웠다. '나쁜' 프로젝트는 개발 팀이 시장의 변화를 이해하고 대응하기 위해 싸우는 동안 계속되는 변화를 겪었다.

맥코맥의 기준에 따라 평가한 '좋은' 프로젝트는 품질, 생산성, 시장 수용성 측면에서 매우 낮은 점수를 받은 반면 '나쁜' 프로젝트는 시장에서 성공했다. 개발기간 내내 시장을 통해 학습을 해온 팀이 더 좋은 제품을 만들어낸 것은 놀랄 일이 아니다. 정말 눈이 휘둥그레질 만한 사실은 그 회사의 관리자들은 시장에 대해 학습하고 시장을 만족시키는 제품을 만들어내는 것을 '나쁜' 것으로 생각했다는 점이다.

맥코맥은 성공적인 소프트웨어 개발을 위한 실천법을 다음과 같이 4가지로 정리하였다.[18]

1. 고객의 평가와 피드백을 위한 최소 기능 집합의 조기 출시
2. 일일 빌드와 통합 테스트를 통한 빠른 피드백
3. 좋은 판단을 할 수 있는 경험과 직관을 갖춘 팀 그리고/또는 리더
4. 새로운 기능을 쉽게 추가할 수 있는 모듈화된 아키텍처

17 이 이야기는 앨런 맥코맥의 "Creating a Fast and Flexible Process: Research Suggests Keys to Success" 에 실려 있다. www.roundtable.com/MRTIndex/FFPD/ARTmaccormack.html, www.agiledevelopmentconference.com/2003/files/AlanAgileSoftwareJun03.ppt 참조.

18 Alan MacCormack, "Product-Development Practices That Work: How Internet Companies Build Software," MIT Sloan Management Review, Winter 2001, Vol. 40 number 2 참조.

제품 개발에서 오랜 기간 동안 뛰어난 실적을 보여온 회사들은 공통적인 특징이 있다. 그들은 규율화된 실험을 통해 새로운 지식을 창출하며, 그 지식을 간결하게 정리하여 상위 조직이 그것을 얻기 쉽게 만든다. 그리고 명확한 자료를 수집할 뿐만 아니라 암묵적인 지식을 명확하게 만들고 그것을 조직의 지식 기반으로 만드는 방법을 모색한다.[19] 이런 회사들은 개발 중인 제품에 대해 학습하는 것도 중요하지만, 그 지식을 집대성하여 향후의 제품에 사용할 수 있게 하는 것도 필수적이라는 것을 알고 있다.

개발주기 동안 체계적인 학습을 독려하는 개발 프로세스를 갖는 것도 중요하지만, 개발 프로세스 자체를 체계적으로 개선시킬 필요도 있다. 때때로 우리는, '표준' 프로세스를 찾는 과정에서 우리 자신의 프로세스를 문서 속에 가둬버려서, 개발 팀이 스스로의 프로세스를 지속적으로 개선하는 것을 어렵게 만든다. 린 조직은 복잡한 환경에서는 항상 문제가 발생한다는 것을 알고 있고, 따라서 조직이 지속적으로 자체 프로세스를 개선해야 한다는 것도 안다. 일의 크고 작음을 떠나, 모든 문제에 대해 그 근본 원인을 찾고 해결하기 위한 최선의 방법을 찾아 실험 하며, 다시 같은 문제가 생기지 않도록 프로세스를 개선한다. 개발 팀은 의무적으로 프로세스 개선에 노력을 기울여야 하며, 모든 팀은 정기적으로 프로세스 개선을 위한 시간을 따로 확보해야 한다.

잘못된 통념: 예측을 해야 예측이 가능해진다.

회사와 고위 경영진이 가장 바라는 것 중의 하나는 바로 예측 가능한 결과다. 이러한 기대는 결국 소프트웨어 개발에까지 영향을 미친다. 불행히도 소프트웨어 개발은 예측 불가능한 것으로 악명이 높고, 그래서 이것을 더 예측 가능하게 해달라는 엄청난 압력이 가해진다. 여기서의 모순은 소프트웨어 개발의 예측 가능성을 개선하려는 열망이 오히려 반대 효과를 갖는 실천법들을 제도화했다는 것이다. 우리는 계획을 세우고, 마치 그것이 미래에 대한 정확한 예측을 구체화한 것처럼 그 계획에 따라 행동한다. 예측이 사실이라고 가정해버리기 때문에 의사결정을 일찍 내리는 경향이 있고, 그것은 결국 우리 스스로를 꽉 짜여진 틀 안에 가둬버리는 결과를 낳는다. 그로 인해, 예측이 어긋났을 때 변화에 대응할 능력을 잃어버린다. 이 문제에 대한 해결책은 더 정확히 예측하는 것이라고 주장할 수도 있을 것이다.

19 Ikujiro Nonaka and Hirotaka Takeuchi in The Knowledge Creating Company: How Japanese Companies Create the Dynamics of Innovation, Oxford University Press, 1995, p. 225. 참조. 번역서로는『지식창조기업』(장은영 역, 세종서적, 2002)이 있다.

우리는 만약 예측이 1) 복잡하거나, 2) 자세하거나, 3) 먼 미래에 대한 것이거나, 또는 4) 불확실한 환경에 대한 것이라면, 그 예측은 언제나 부정확할 수밖에 없다는 사실을 잊고 있다. 이러한 종류의 예측을 더 정확하게 하기 위해 아무리 많은 노력을 해도 좋은 결과를 기대하기는 힘들다. 그러나 우리가 정확한 예측으로부터 시작할 수 없다고 해도 신뢰할 만한 결과를 낳을 수 있는 잘 검증된 방법들이 있다. 그것은 바로 미래에 대한 우리의 예측을 마치 사실인양 인식하고 행동하는 것을 멈추는 것이다. 예측은 단지 예측일 뿐이다. 대신에 어떤 사건이 발생하면 그에 빠르게 대응할 수 있도록 반응 시간을 줄여야 한다.

결과에 대한 예측 가능성을 높이기 위해서는 의사 결정에 들어가는 추측의 양을 줄여야 한다. 예상이 아닌 사실에 근거한 결정이 가장 예측 가능한 결과를 낳는다. 메리는 제어 엔지니어로서 경험론적 제어 시스템(피드백 기반의 제어 시스템)이 결정론적 제어 시스템보다 더 예측 가능한 결과를 낳는다는 것을 안다. 근본적으로는 사건이 일어나기를 기다리고 거기에 빨리 그리고 정확히 대응하는 능력을 잘 개발한 조직이 미래를 예측하려 하는 조직보다 훨씬 더 예측 가능한 결과를 낳는다.

원칙 4: 확정을 늦춰라

긴급 상황에 대처하는 사람들은 어렵고 예측 불가능하며 위험한 상황을 다루는 훈련을 받는다. 그들은 어려운 상황을 평가하여 중대한 결정을 내리기까지 얼마 동안 결정을 늦출 수 있는지를 결정하는 방법을 배운다. 그러한 결정을 위한 타임박스[20]를 설정하고 그 시간이 다 될 때까지 기다리도록 교육받는다. 바로 그 순간이 결정을 내리기 위해 필요한 정보를 가장 많이 확보하는 때이기 때문이다.

소프트웨어를 개발하는 과정에서도 돌이킬 수 없는 결정을 내려야 할 때와 같은 논리를 적용해야 한다. 돌이킬 수 없는 결정은 마지막 결정의 순간last responsible moment까지 미뤄야 한다. 즉, 그 순간을 지나면 너무 늦어져버리는 마지막 순간에 결정을 내려야 하는 것이다. 모든 종류의 의사 결정을 미뤄야 한다는 의미가 아니다. 무엇보다도 먼저, 대부분의 결정을 돌이킬 수 있게 만들어서 결정을 내리고 나서도 쉽게 바꿀 수 있게 해야 한다. 반복적 개발의 목표 중에서도 가장 유용한 것이 바로 '분석 불능' 상태에서 시작하여 무언가 구체적인 방향으로 이동하는 것이다. 그러나 시스템의 초기 기능을 구현하는 동안에는 나중에 변

20 시작과 종료시점이 정해진 일정.

경하기 어렵게 만들 만한 중대한 설계 결정을 가급적 피해야 한다. 소프트웨어 시스템이 완전히 유연할 필요는 없지만 변화가 일어날 만한 부분에는 옵션을 유지할 필요가 있다. 해당 분야와 기술에 대해 상당한 경험을 쌓은 팀이나 리더는 어느 곳에 옵션을 유지해야 하는지를 본능적으로 알고 있을 것이다.

많은 사람들이 까다로운 결정도 일단 내리고, 리스크 요인들을 정면으로 공략하며, 알 수 없는 것의 개수를 줄이는 것을 좋아한다. 그러나 불확실성이 있고 특히나 복잡성을 동반한 경우, 더 성공적인 접근방법은 어려운 문제들에 대한 다양한 해결책을 시도하고 중대한 옵션들은 결정을 더 이상 늦출 수 없는 순간까지 보류하는 것이다. 사실 최고의 소프트웨어 설계 전략이라고 하는 것들의 대부분은, 옵션을 결정되지 않은 채로 남겨두었다가 가능한 가장 늦은 시점에 돌이킬 수 없는 결정을 내릴 수 있도록 하는 것이다.

잘못된 통념: 계획을 세우는 것은 확정하는 것이다

"나는 전투를 준비함에 있어 계획plan은 언제나 쓸모가 없다는 것을 발견했다, 그러나 계획을 세우는 것planning은 필수불가결하다." 아이젠하워의 이 유명한 말은 계획을 세우는 것과 확정하는 것의 차이에 대한 올바른 통찰을 제공한다. 계획을 세우는 것은 중요한 학습 훈련이고 조직의 올바른 반사신경을 개발하는데 매우 중요하며, 복잡한 시스템에서 상위 아키텍처 설계를 확립하는데도 필요하다.

반면에 계획은 과대평가되고 있다. 오노 다이이치의 이야기를 들어보자.[21]

> 계획은 아주 쉽게 변한다. 이 세상의 일들은 항상 계획대로 진행되지는 않기 때문에, 상황의 변화에 대응하여 업무 지시도 빠르게 변해야 한다. 만약 한번 결정한 아이디어를 계속 고집한다면 계획은 변하지 않을 것이며 사업은 오래 지속될 수 없을 것이다.
>
> 사람의 척추는 튼튼할수록 더 잘 휘어진다고 한다. 이러한 탄성elasticity이 중요하다. 만약 뭔가가 잘못되어 척추에 깁스를 한다면 이 중요한 부위가 뻣뻣해지고 기능을 멈추게 된다. 한번 세워진 계획에 집착하는 것은 사람의 몸에 깁스를 하는 것과 같다. 그것은 건강하지 않다.

21 Taiichi Ohno, Toyota Production System: Beyond Large Scale Production, Productivity Press, 1988, p. 46.

베리 뵘Barry Boehm과 리차드 터너Richard Turner[22]는 계획이 곧 확정이라는 기대에 근간을 둔 소프트웨어 개발 방법론을 설명하기 위해 '계획 주도 방법plan-driven method'이라는 용어를 만들어냈다. 그들은 유능한 계획 주도 프로세스를 "계획된 결과를 만들어 내는 고유한 능력"을 갖고 있는 프로세스라고 정의했다.[23] 그들은 더 나아가 계획 주도 방법은 정부기관과의 계약에서 비롯된 것이라고 하였다.[24] 사실상 약속으로 간주되는 상세 계획을 만들지 않고서, 정부기관에서 요구하는 계약을 대등한 입장에서 체결하는 것은 매우 어려운 일이다.

그러나 민간 분야의 현명한 회사(도요타와 같은)들은 상세 계획에 집착하는 것이 건강하지 않으며, 특히 상세 계획을 따라잡는 능력에 따라 프로세스 역량을 평가하는 것이 잘못이라는 사실을 알고 있다. 계획을 세우는 것이 확정하는 것이라는 잘못된 통념에 굴복하지 말자. 계획은 사려 깊게 하되 확정은 인색하게 해야 한다.

원칙 5: 빨리 인도하라

우리의 자녀들이 너무 어려서 의자에 앉으면 그들의 턱이 테이블 높이 정도 되던 시절에는, 어느 집을 방문하건 항상 두꺼운 시어스[25] 카탈로그 한두 권쯤은 볼 수 있었고 우리는 그것을 의자 높이를 높이는데 사용했다. 누구나 그 카탈로그의 페이지들을 넘겨보는 것을 좋아했고 거기서 주문을 하곤 했다. 배송까지는 2~3주가 걸렸기 때문에 근처 상점에서 구할 수 있는 물건을 일부러 주문하지는 않았다. 1980년대 중반쯤 메인Maine 주의 통신 판매 회사인 L.L. 빈L.L. Bean은 시간을 무기로 시어스와 경쟁하기로 결정했다. 그들의 목표는 모든 주문에 대해 접수 후 24시간 이내에 물품을 배송하는 것이었다. 이것은 너무나 놀라운 개념이었기 때문에 다른 회사들은 어떻게 그것이 가능한지 보기 위해 L.L. 빈의 물류 센터를 방문하곤 했다.

얼마 지나지 않아 사람들은 2~3주의 배송 기간이 끔찍하게 느리다고 느끼게 되었다. 창업 100년을 향해 가던 전통 깊은 시어스 카탈로그로는 더 이상 경쟁력이 없게 되었고 결국 1993년에 문을 닫았다. L.L. 빈은 훨씬 더 적은 비용을 쓰고 있었는데 그 이유는 그들이

22 Barry Boehm and Richard Turner, Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley, 2004.

23 22)번 각주와 동일, p. 12.

24 22)번 각주와 동일, p. 10.

25 Sears. 미국의 거대 통신판매 회사.

주문 취소나 변경으로 인한 비용을 줄일 수 있었던 것에서도 찾을 수 있다. 다음을 살펴보자. 시어스에서 노란색 셔츠를 주문했는데 1, 2주 후에 전화를 걸어 "있잖아요, 생각이 바뀌었어요. 파란색 셔츠로 바꿔주실 수 있나요?"라고 말한다면 그들은 아마도 주문을 바꿔줄 것이다. 그러나 만약 L.L.빈에 똑같은 요청을 했다고 한다면, 그들은 아마도 이렇게 말할 것이다. "현관 앞을 보셨나요? 오늘 도착했을 겁니다."

이야기의 교훈은 어떻게 하면 고객이 그들의 생각을 바꿀 시간이 없을 정도로 빨리 소프트웨어를 인도할 수 있을지 생각해 보아야 한다는 것이다.

페덱스FedEx는 하룻밤 배송을 가능하게 했다. 사우스웨스트 항공은 공항 게이트에서의 신속한 턴어라운드[26]를 최초로 실현했다. 두 회사는 모두 다른 회사들이 그들을 널리 모방한 후에도 각각의 산업에서 주도권을 유지하고 있다. 델이 만들어낸 속도에 기반을 둔 주문 조립 생산 모형은 모방이 매우 어려운 것으로 이미 판명되었다. 도요타는 혁신적인 하이브리드 엔진을 갖춘 프리우스 모델을 15개월만에 출시했는데, 이 정도의 제품 개발 속도에 근접하는 회사는 거의 없다.

시간을 기반으로 경쟁력을 갖춘 회사는 경쟁사에 비해 현저한 비용 우위를 가질 수 있다. 그들은 비용을 초래하는 엄청난 양의 낭비를 제거하였다. 더구나 그들의 결점율은 극단적으로 낮다. 반복가능하고 안정적인 속도는 뛰어난 품질이 뒷받침되지 않으면 불가능하다. 한발 더 나아가 고객에 대한 깊은 이해를 발전시켰다. 그들은 너무나 빨라서, 제품 개발에 새로운 생각을 시도하고 실제로 먹히는 것이 무엇인지 배우는 실험적인 접근을 하는 것이 부담스럽지 않다.

잘못된 통념: 서두르는 것은 낭비를 낳는다

소프트웨어 업계에는 높은 품질을 얻으려면 "속도를 늦추고 조심해야 한다" 라는 생각이 지배적이었다. 그러나 업계가 그러한 타협을 고객에게 강요하고 있을 때, 그 타협을 깨트리는 회사는 현저한 경쟁 우위를 점하게 된다.[27] 이 책의 뒤에서 소개되는 구글과 페이션트 키퍼 社가 바로 고품질의 소프트웨어를 신속하게 공급하는 회사들 중 하나다. 속도와 품질

26 turnaround. 비행기의 착륙에서 출발까지의 시간을 말한다. 항공사 입장에서는 비용을 줄이고 승객을 신속히 수송할 수 있기 때문에 신속한 턴어라운드가 중요한 의미를 가진다.

27 George Stalk와 Rob Lachenauer, Hardball: Are You Playing to Play or Playing to Win, Harvard Business School Press, 2004.

이 양립할 수 없다는 믿음을 고수하는 소프트웨어 개발 조직은 그리 멀지 않은 미래에 시어스 카탈로그처럼 역사의 뒤안길로 사라질 운명에 처할 것이다.

주의: 빠른 속도와 꼼수를 동일시 하지 마라. 그 둘은 전혀 다르다. 빠르게 움직이는 개발 팀은 뛰어난 반사 신경과 규율에 따른 라인정지stop-the-line 문화를 갖추어야 한다. 그 이유는 명백하다. '품질을 내재화하지 않으면 빠른 속도를 유지할 수 없다'.

규율을 추구함에 있어 많은 조직이 상세한 프로세스 계획, 표준화된 작업 산출물, 작업 흐름 문서 그리고 명확한 업무 지침서를 개발한다. 게다가 이러한 일은 주로 프로세스 상의 작업자를 훈련시키고 규율 준수 여부를 관찰하는 지원 조직에 의해 이루어진다. 목표는 다른 것들 중에서도 프로젝트 간에 인원을 쉽게 이동할 수 있게 하는 표준화되고 반복 가능한 프로세스를 달성하는 것이다.[28] 그러나 대체 가능한 인력을 만들도록 설계된 프로세스는, 빠르고 유연한 프로세스에 필수적인 사람을 육성해 내지 못한다.

빠른 조직을 만들고 싶으면 옳은 판단을 할 것이라고 신뢰할 수 있고, 서로를 도울 수 있는, 적극적으로 참여하고 연구하는 사람들이 필요하다. 빠르게 움직이는 조직에서는 그 업무를 하는 사람이 지시 받지 않아도 무엇을 해야 하는지 알고 있고, 승인이 없이도 문제를 해결하고 변화에 대응할 수 있도록 업무가 구성되어 있다. 린 조직은 표준에 맞춰 일한다. 그러나 그러한 표준들은 업무를 어떻게 수행할 것인지에 대한 현재 수준에서 최고의 지식을 구체화하고 있기에 존재한다. 그 표준들은 작업자들이 업무를 수행하기 위해 더 좋은 방법을 시험하기 위한 기준선이 된다. 린의 표준은 도전을 받고 개선되기 위해 존재한다.[29]

고품질을 달성하기 위해서는 2가지 방법이 있다. 여전히 속도를 늦추고 조심하는 것과 사람을 육성하는 것이다. 후자는 사람을 통해 계속해서 프로세스를 개선하고 제품에 품질을 내재화하며, 경쟁사보다 몇 배나 빨리 고객에게 반복적이고 안정적으로 대응하는 능력을 개발하는 것을 뜻한다.

28 Barry Boehm and Richard Turner, Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley, 2004, pp. 11-12. 참조

29 Jim Shook "Bringing the Toyota Production System to the United States" in Becoming Lean, Jeffrey Liker, editor, Productivity Press, 2004, pp. 59-60 참조.

원칙 6: 사람을 존중하라

조엘 스폴스키Joel Spolsky가 대학을 갓 졸업하고 마이크로소프트에 신입사원으로 입사했을 때 맡은 일은 엑셀 프로그램에 매크로 언어 전략을 개발하는 것이었다. 그는 얼마간 고객이 매크로에 원하는 기능이 무엇인지 연구하고 나서 명세를 작성했다. 애플리케이션 아키텍처 그룹이 명세 이야기를 듣고서 검토를 해주겠다고 했다. 조엘로서는 반가운 소식이었다. 조엘은 그들로부터 좋은 조언을 들을 수 있을 것이라고 생각했다. 그러나 그 작은 그룹은 매크로에 대해 조엘보다도 더 모르고 있었다. 거기다 그 그룹에는 박사 4명과 매우 높은 경력의 상관(빌게이츠의 친구이자 회사 내 서열이 여섯 번째였다)이 있었다. 애플리케이션 아키텍처 그룹은 고객이 그 매크로를 어떻게 사용할 것인지에 대해 매우 피상적인 아이디어만 갖고 있으면서도, 매크로가 어떻게 구현되어야만 하는지 결정하는 것이 그들의 일이라고 생각했다. 조엘의 관리자들은 그 결정은 조엘의 몫이라고 했고 조엘의 프로그래밍 팀도 그를 지지했다. 그러나 애플리케이션 아키텍처 그룹은 그것을 못마땅해 했으며 그들의 상관은 조엘이 매크로 전략을 엉망으로 만들고 있다고 불만을 토로하기 위해 대규모 회의를 소집했다.

만약 마이크로소프트가 평범한 회사였다면, 여러분은 이 시나리오가 어떻게 마무리 될지 상상할 수 있을 것이다. 애플리케이션 아키텍처 그룹은 제 갈 길을 갔을 것이고 조엘은 마지못해 따르든지 아니면 회사를 떠났을 것이다. 그러나 마이크로소프트에서는 그렇게 되지 않았다. 대규모 회의 다음 날 부사장이 조엘과 같이 식당에 들러 "그 애플리케이션 아키텍처 그룹하고는 어떻게 진행되고 있나요?"라고 물었다. 조엘은 모든 것이 잘 되고 있다고 대답했다. 그러나 그 다음 날 조엘은 애플리케이션 아키텍처 그룹이 해체되었다는 소문을 들었다. 그는 크게 감명을 받았다. 조엘은 이렇게 말한다. "마이크로소프트에서는 여러분이 엑셀 매크로 전략 팀의 일원이라면, 여러분이 입사한지 6개월도 안 되었다 하더라도, 그것은 중요하지 않다. 여러분이 엑셀 매크로 전략에 있어서는 신(神)이며 누구도, 설사 그가 회사 내 서열 6위라 할지라도, 여러분의 길을 가로 막을 수는 없다. 이상 끝."[30]

조엘의 이야기는 '원칙 6: 사람을 존중하라'가 소프트웨어 업계에서 일하는 사람의 관점에서 보았을 때 무엇을 의미하는지를 보여주고 있다.

30 Joel Spolsky, "Two Stories", http://www.joelonsoftware.com/articles/twostories.html. 허가 받아 인용. "Smart and Gets things done", Apress 2007. 번역서로는 『똑똑한, 100배로 일 잘하는 개발자 뽑기: 조엘 온 소프트웨어 시즌 2』 (이석중, 위키북스 2007)가 있다.

그림 1.5[31]의 도요타 제품개발방식의 4개 핵심 요소 중 3개가 제품개발 프로세스에 종사하는 사람들과 관련되어 있다는 것은 주목할 만하다. 이 3가지를 살펴보면 사람을 존중한다는 것이 어떤 의미인지에 대한 폭넓은 아이디어를 얻을 수 있을 것이다.

1. **기업가적 정신을 가진 리더**: 사람들은 성공적인 제품을 만드는 곳에서 일하고 싶어 하는데, 매우 성공적인 제품은 대개 뛰어난 리더에 기인하는 경우가 많다. 조직의 구성원을 존중하는 회사는 훌륭한 리더를 양성하고, 적극적으로 참여하고 연구하는 사람들을 육성하며, 위대한 제품을 창조하는데 그들의 노력을 집중하는 리더십을 가지도록 한다.

2. **전문 기술 인력**: 특정 분야에서 경쟁 우위를 지속하고자 하는 회사라면 그 분야의 기술적 전문 지식을 개발하고 키워야 한다. 필요한 모든 전문 지식을 구매하는 회사는 경쟁사 역시 그것을 구매할 수 있다는 것을 깨닫게 될 것이다. 전문 지식에 대한 필요성을 느끼지 못하는 회사는 자신들에게 지속 가능한 경쟁 우위가 없다는 것을 머지않아 발견할 것이다. 현명한 회사는 적절한 기술적 전문 지식을 육성하고, 모든 팀이 목표를 달성하기 위해 필요한 전문 지식을 갖추는 데 집중한다.

3. **책임 기반의 계획과 통제**: 사람을 존중한다는 것은 팀에 전반적인 계획과 합리적인 목표만을 부여하면, 그 팀이 목표를 달성하기 위해 스스로 노력할 것이라고 신뢰한다는 뜻이다. 존중은 사람들에게 무엇을 어떻게 하라고 지시하는 대신 그들 스스로 머리를 굴려 생각해내는 반사신경을 갖춘 조직을 개발하는 것을 의미한다.

잘못된 통념: 유일한 최선의 방법이 존재한다

『Cheaper by the Dozen』[32]이라는 책은 효율성 전문가인 아버지와 12명의 자녀로 이뤄진 가족에 관한 매우 재미있는 실제 이야기이다. 11명의 형제 중 넷째로 자란 메리는 어린 시절 이 책을 아주 열심히 읽었다. 최근 메리는 이 책을 다시 읽으면서 여태까지 미처 깨닫지 못했던 것을 깨달았다. 『Cheaper by the Dozen』이 과학적 관리scientific management의 기원에 관한 독특한 시각을 보여준다는 것이다. 12명의 아버지 프랭크 길브레스Frank Gilbreth는 어떤 일을

31 Michael Kennedy, Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It, Oaklea Press, 2003, p. 120. 허가 받아 인용.

32 Frank B. Gilbreth Jr. and Ernestine Gilbreth Carey, Cheaper by the Dozen, T.Y. Crowell Co., 1948. 같은 제목의 영화는 책과는 많이 동떨어진 내용이다. 번역서로 『아빠, 고생하신거 우리 다 알아요: 아빠, 그만 좀 웃기세요』(장석영 역, 현실과미래, 2000)가 있다.

하더라도 거기에는 항상 '유일한 최선의 방법one best way'이 존재한다고 주장하는 세계적으로 유명한 컨설턴트였다. 길브레스의 조직화된 가족들을 생각하며 행간을 읽어보면, 그의 효율성 향상 노력의 결과물을 대부분의 작업자들이 받아들이고 싶지 않았을 것이라고 추측할 수 있다. 그러나 프랭크 길브레스의 사람에 대한 공경심 부족이 익살스러운 수준인 반면, 그의 친구인 프레드릭 윈슬로우 테일러Frederick Winslow Taylor가 그의 저서를 통해 보여준 것은 '노동자'에 대한 숨김 없는 멸시였다.

프랭크 길브레스가 일찍 사망한 후 그의 부인이자 파트너인 릴라이언 길브레스Lillian Gilbreth가 그 일을 물려 받았는데, 그녀는 훗날 그 시대의 가장 훌륭한 산업공학자 중 한 사람이 되었다. 그녀는 가족을 돌보는 일[33]에서도, 그녀의 직업에서도, '유일한 최선의 방법'을 찾는 대신 작업자의 동기부여에 초점을 맞추었다. 그러나 상황은 이미 돌이킬 수 없었다. 모든 일에 대해 '유일한 최선의 방법'을 찾고 제도화하는 접근방식이 대다수 미국 산업계의 산업공학자들에게 각인되었다. 소위 프로세스 경찰이라 불리던 사람들의 선조격인 이 산업공학자들은, 주어진 일을 제대로 이해하지도 않고 표준부터 만들어내곤 했다. 더구나 더 나은 방법에 대한 가능성을 생각하지도 않고 그 표준들을 강요했다.

'유일한 최선의 방법'같은 것은 없다. 개선될 수 없는 프로세스는 없다. 이것을 여러분 스스로 증명하고자 한다면, 그저 시간을 좀 내어 사람들이 일하는 모습을 조용히 지켜보라. 잠깐만 지켜보더라도 개선 대상을 많이 발견할 수 있을 것이다. 프로세스는 반드시 그 일을 직접 수행하는 작업 팀에 의해 개선되어야 한다. 그들은 주어진 문제를 가장 큰 것부터, 한 번에 하나씩 해결하기 위한 시간, 권한 그리고 적절한 도움을 필요로 한다. 이러한 영속적인 개선 프로세스가 모든 소프트웨어 개발 조직에 자리잡고 있어야 한다.

원칙 7: 전체를 최적화하라

소프트웨어 개발은 부분 최적화로 유명하다.

- **악순환 1번 (물론 이런 일은 여러분 회사에서는 절대 일어나지 않아야 한다.)**
  - 고객이 '어제' 느닷없이 새로운 기능을 이야기한다

33 속편, Frank B. Gilbreth Jr. and Ernestine Gilbreth Carey, Belles on Their Toes, T.Y. Crowell Co., 1950. 번역서로는 『천사표 우리 엄마』(장석영 역, 현실과미래, 2000)이 있다,

  - 개발자에게 이렇게 전달된다. "비용이 얼마가 들더라도 빨리 끝내도록!"
  - 결과: 코드베이스에 조잡한 변경이 가해진다.
  - 결과: 코드베이스의 복잡도가 증가한다.
  - 결과: 코드베이스의 결함수가 증가한다.
  - 결과: 기능을 추가하는데 들어가는 시간이 기하급수적으로 증가한다.
- **악순환 2번 (이것 또한 여러분 회사에서는 일어나지 않아야 한다)**
  - 테스팅 업무가 과중하다.
  - 결과: 테스팅이 코딩보다 한참 뒤에 이뤄진다.
  - 결과: 개발자가 피드백을 즉시 얻지 못한다.
  - 결과: 개발자가 결함을 더 많이 발생시킨다.
  - 결과: 테스팅 업무가 더 많아진다. 시스템에 결함이 더 많아진다.
  - 결과: 피드백이 더 지연된다. 이것이 반복된다.

린 조직은 전체 가치흐름을 최적화한다. 전체 가치흐름은 고객 요구를 반영한 주문을 받았을 때부터 소프트웨어가 배치되어 고객 요구가 만족될 때까지를 포함한다. 만약 어떤 조직이 전체 가치흐름보다 작은 것을 최적화하는데 집중한다면, 전체 가치흐름이 악화될 것이라고 말할 수 있다. 우리는 그동안 수업중에 다루었던 가치흐름도value stream map에서 이런 경우를 자주 보았다. 커다란 지연이 발생하는 것은 대부분 한 부서에서 다음 부서로 책임을 넘기면서 그 사이에 정작 고객의 관심사가 제대로 넘어가도록 돌보는 책임은 누구도 지지 않는 경우였다.

조직은 대개 그들이 전체를 최적화하고 있다고 생각한다. 어찌 됐든 누구나 부분 최적화가 나쁘다는 것을 알고 있으며 아무도 그것을 받아들이려 하지 않는다. 그러나 측정 시스템상에 부분 최적화가 제도화된 경우는 놀라울 정도로 많다. 후지쯔Fujitsu의 사례를 살펴보자.[34] 2001년 후지쯔는 영국 항공사인 BMI의 헬프 데스크를 맡게 되었다. 후지쯔는 BMI 종업원들로부터 받은 요청들을 분석했고 그 결과 요청 중 26%가 항공사 탑승 수속 카운터의 프

34 James P. Womack and Daniel T. Jones, Lean Consumption: How Companies and Customers Can Create Value and Wealth Together, Free Press, 2005, pp. 58–63.

린터 오동작 때문이었다. 후지쯔는 프린터가 얼마 동안 동작불능이었는지 측정하고 BMI가 그것에 치르는 전체 비용을 산정했다. 그러고 나서 후지쯔는 그 직무사례를 보고함으로써 BMI 경영진이 프린터를 더 튼튼한 것으로 바꾸도록 했다. 그 후 후지쯔는 헬프 데스크에 걸려오는 요청의 근본원인을 밝혀내고 이를 제거하기 위한 활동을 계속하여 18개월 후에는 전체 요청 수의 40%가 감소하였다.

이것은 좋은 이야기처럼 들린다. 그러나 뭔가 잘못된 것이 있다. 대부분의 헬프 데스크는 요청처리 건수에 근거하여 보수를 받는다. 요청 건수를 40% 감소시킴으로써 후지쯔의 수입도 같은 양만큼 줄어들었다. 다행히도 후지쯔의 성과가 너무 극적이었으므로 BMI와의 재협상을 실시하여 실제 요청이 아닌 '잠재적' 요청에 근거하여 보수를 받을 수 있었다. 새로운 협의하에서 후지쯔는 BMI가 사업상의 문제를 해결할 수 있도록 근본적으로 지원하는 것을 계속할 수 있는 재정적인 인센티브를 갖게 되었다.

이 이야기는 전통적인 수익 모델하에서 운영되는 헬프 데스크 외주는 고객 문제에 대한 원인을 찾고 해결하기 위한 동기부여가 없음을 보여준다.

조직적인 경계를 넘나드는 것은 비용이 많이 든다. 피터 드러커Peter Drucker는 가치흐름을 관통하는 단일 관리시스템을 유지하는 회사는 경쟁사에 비해 25~30%의 비용 우위를 보인다고 하였다.[35] 이것이 사실이라면 구성원 모두가 전체를 최적화하도록 만드는 올바른 인센티브에 의해 계약, 외주 협약, 그리고 업무에서 상호 협력 체계를 구축함으로써 상당한 양의 비용 절감이 가능하다.

잘못된 통념: 부분으로 나누어 최적화하라

P. 슬론P. Sloan은 복잡도를 다루기 위한 조직 구조를 창안했다. 그는 분권화된 조직을 만들고 관리자의 성과를 판단하는데 재정적 지표를 이용했다. 이것은 중역이 직접 통제하던 포드Ford 방식에 비해 크게 개선된 것이었다. 이것으로 인해 제너럴 모터스General Motors는 다양한 종류의 차를 생산하고 포드를 앞질러 시장의 선두를 차지하게 되었다. 그 이후 회사들은 성과를 관리하기 위한 올바른 지표를 찾고자 애쓰고 있다.

35 Peter Drucker, Management Challenges for the 21st Century, Harper Business, 1999, p. 33. 번역서로는 『21세기 지식경영(지식근로자의 자기개발법)』 (이재규 역, 한국경제신문사, 2002)가 있다.

복잡도를 다루기 위해 지표를 사용한 슬론의 생각은 훌륭했지만 이 같은 개념은 오용되기 쉬운 것이다. 사람들은 복잡한 상황을 작은 조각들로 나누려는 경향이 있는데, 이는 아마도 일을 매우 작은 작업단위로 나누었던 효율성 전문가들에게서 기인한 것이 아닌가 한다. 잘게 나눈 뒤에는 각각의 조각들을 측정하고 그 측정치를 최적화한다. 여러분은 각각의 측정치를 최적화하면 전체 시스템도 최적화될 것이라고 기대하겠지만, 그것은 틀린 생각이다. 만약 여러분이 가치흐름을 여러 조각으로 나누어 창고Silo에 넣고 각각을 최적화하면[36], 그동안의 경험으로 볼 때 전체 시스템은 부분 최적화될 것이 거의 확실하다.

모든 것을 측정할 수는 없다. 그러므로 측정을 제한해야 하는데 그렇게 되면 측정의 틈사이로 빠지는 것이 있게 마련이다.[37] 예를 들어 프로젝트 성과를 비용, 일정, 목표 범위를 기준으로 측정하려고 한다. 그러나 이러한 측정만으로는 품질과 고객 만족을 고려하지 못한다. 만약 서로 대립하게 되면 품질과 고객 만족을 포기하게 된다. 그렇다면 어떻게 해야 하는가? 부분으로 나누는 방법을 따르자면 측정 지표를 2개 추가하여 이제는 비용, 일정, 목표 범위와 함께 품질과 고객 만족까지 측정하게 된다. 이것 보라! 우리는 방금 철의 3각지대를 철의 5각지대로 바꿔놓았다.

측정 시스템이 너무 많은 것을 측정하게 되면, 너무 많은 지표로 인해 진정한 목적을 상실하게 되고, 지표들끼리 대립하는 상황에서 효과적인 결정을 내릴 수 없게 된다. 해결책은 **"더 높은 것을 측정하라"**이다. 즉, 측정을 한 수준 위로 올려서 측정의 수를 '줄이는' 것이다. 하위 수준의 지표들이 올바른 값이 나오도록 인도할 상위 수준의 지표를 찾고, 지표들 간에 대립이 있을 경우 올바른 절충안을 찾도록 하는 기반을 확립하라. 위의 프로젝트 지표 예에서는 지표를 2개 추가하는 대신 지표를 하나로 줄여 정말 중요한 하나만을 측정해야 한다. 프로젝트 평가에서는 투자수익률이 후보가 될 수 있다. 손익 모델은 제품 평가에 잘 맞는다.[38] 정말 중요한 오직 하나의 지표를 최적화하면 나머지 숫자들은 스스로 관리될 것이다.

36 사일로 효과(silo effect) 참조. 조직 내에서 부서 간의 협력 없이 내부의 이익만 추구함을 의미한다.

37 Rob Austin, Measuring and Managing Performance in Organizations, Dorset House, 1996. 이 주제를 훌륭하게 설명하고 있다.

38 Mary and Tom Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley, 2003의 83-92쪽 참조. 번역서로는 『린(Lean) 소프트웨어 개발 : 애자일 실천 도구 22가지』(김정민, 김현덕, 김혜원, 박영주 공역, 인사이트, 2007)가 있다.

Story **중고차**[39]

우리가 하는 수업 중에서 투자수익률 실습은 사람들에게 인기가 좋다. 고객이 정말 만족하는 의사결정을 내릴 수 있도록 도와주기 때문이다. 한 수업에서 이안Ian과 그의 그룹이 중고차 재판매 회사의 투자수익률을 산출했다. 차량의 재고 시간을 5일에서 4일로 줄였더니 매우 높은 재정 수익이 발생하였다.

투자수익률 뒤에 숨어있는 이야기는 더욱 흥미롭다. 이안은 스크럼을 막 배우고 나서 중고차 재판매 회사에서 매우 오래된 컴퓨터 시스템을 대체하는 일을 맡았다. 그는 고객에게 그의 팀이 앞으로 한달 분량씩 점진적으로 소프트웨어를 개발하여 배포할 것이며 만약 그 중 한 번이라도 만족하지 못한다면, 계약을 취소해도 좋다고 말했다.

"뭐라고 말했다고?" 이안의 경영진은 불만을 나타냈다. 만약 그 중고차 재판매 회사의 계약이 파기되면 그 컨설팅 회사에서 이안에게 더 이상 일을 주지 않을지도 모른다는 암시가 있었다. 그래서 이안은 매월 고객을 기쁘게 할 방법을 찾는데 더 매달리게 되었다.

여기저기 문의한 끝에 이안은 차량 재고 시간을 줄이는 것이 재정에 긍정적인 효과가 있을 것이라는 사실을 알아냈고, 우리 수업에서 그 효과를 정량화하는 실천법을 익혔다. 그는 고객을 찾아가 실제 데이터를 얻어서 간단한 재정 모델에 입력했다. 그 결과를 토대로 이안과 고객은 매달 가장 가치있는 기능을 개발하는데 집중할 수 있었다.

몇 개월 후 이안은 그의 고객이 결과에 매우 만족하여 그의 컨설팅 회사가 더 많은 사업을 따내게 될 것이고, 경영진이 매우 기뻐했다는 소식을 우리에게 알려주었다. 이안의 경영진은 애자일 소프트웨어 개발 방법론을 받아들였고 현재 자신들의 회사를 영국에서 최고의 애자일 회사 중 하나라고 생각하고 있다.

– 메리 이야기

39 이 이야기를 사용하게 해준 이안 쉬밍즈(Ian Shimmings)에게 감사한다.

시도해 볼 것

1. 아래 7가지 잘못된 통념 중 여러분이 처한 상황에서 특별히 가슴에 사무치는 것이 있는가? 왜 그런가?
   - ○ 스펙 조기 확정이 낭비를 줄인다.
   - ○ 테스팅의 역할은 결함을 발견하는 것이다.
   - ○ 예측을 해야 예측이 가능해진다.
   - ○ 계획을 세우는 것은 확정하는 것이다.
   - ○ 서두르는 것은 낭비를 낳는다.
   - ○ 유일한 최선의 방법이 존재한다.
   - ○ 부분으로 나누어 최적화하라.
2. 여러분의 조직에서는 가치흐름 시각표가 언제 시작되는가? "주문을 받는다"는 것이 여러분의 환경에서는 어떤 의미인가? 어떤 회사에서는 개발을 위한 제품 컨셉이 승인될 때(주문은 승인 프로세스에서 온다) 제품 개발의 시각표가 시작된다. 마케팅 팀이 고객 요구를 인지하고 새로운 기능을 요청할 때 시작되기도 한다. 어떤 것이 여러분의 상황에 가장 적합한가?
3. 혼란: 여러분 회사에서,
   a. 요구사항이 문서화된 후 몇 %가 변경되는가?
   b. 개발 주기 끝에서 '테스트하고 수정하기' 또는 '안정화' 활동을 하는데 개발 기간의 몇 %가 소요되는가?
   c. 이 같은 비율을 줄이기 위해 무엇을 할 수 있는가?
4. 사람: 일선 감독자들은 부서를 어떻게 이끌고 갈 것인지에 대해 얼마 만큼의 교육을 받는가? 그들을 평가할 때 강조하는 항목은 무엇인가? 일선 프로젝트 관리자들은 팀을 어떻게 이끌고 갈 것인지에 대해 얼마 만큼의 교육을 받는가? 그들을 평가할 때 강조하는 항목은 무엇인가? 두 경우에 차이가 있는가? 왜 그런가?

5. 측정: 여러분 조직에서는 사람들마다 성과 측정을 위한 핵심 '성과 지표'를 종이 한 장에 정리하여 가지고 있는가? 그렇다면 모든 지표를 (가능하다면 익명으로) 모아 하나의 목록을 만들어 보라. 지표가 얼마나 되는가? 그것들은 올바른 것들인가? 놓친 핵심 지표는 없는가? "더 높은 것을 측정하라"를 실천할 방법을 생각해낼 수 있겠는가?