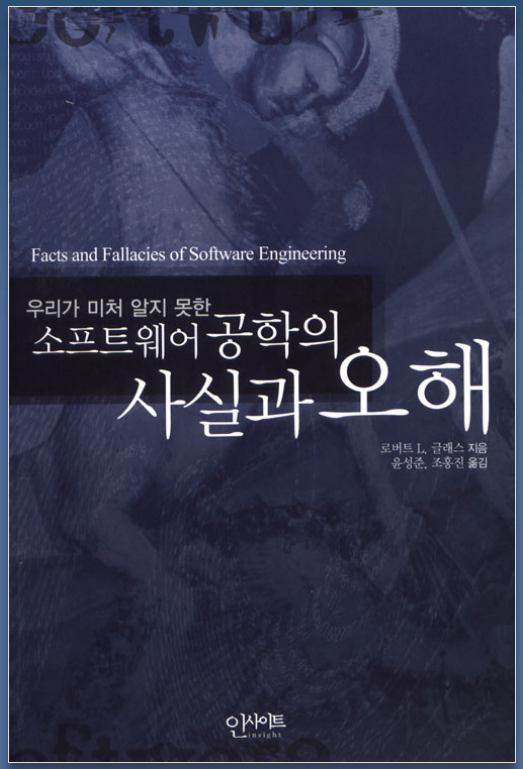

우리가 미처 알지 못한

소프트웨어 공학의 사실과 오해

목 차

1부. 사실

I. 관리

- 사람
- 도구와 기술
- 추정
- 재사용
- 복잡성

II. 생명주기

III. 품질

IV. 연구

1장. 관리

1. 관리

- 관리는 항상 지루한 주제
- 소프트웨어 분야에서 가장 영향력이 높고 가시성이 높은 것들은 관리에 대한 것이다.
- 대부분의 실패는 관리자의 책임으로 돌아간다. 대부분의 성공 또한 관리의 덕이라고 할 수 있다.
- Al Davis의 책: 뛰어난 관리가 기술보다 중요하다.

2. 사람

- 정말 중요한 것은 사람이다.
- 어떤 사람은 다른 사람보다 놀라울 정도로 뛰어나다.
- 많은 프로젝트의 성공과 실패는 어떻게 수행하는가 보다 누가 수행하는 가에 따라 결정된다.

3. 도구와 기술

- 과대광고는 득보다 해가 된다.
- 새로운 접근방법으로 전환하면, 처음에는 효율이 향상되기 보다 저하된다.
- 최신의 도구나 기술이 실제로 사용되는 경우는 드물다.

4. 추정

- 우리의 추정은 부정확한 경우가 많다.
- 추정작업을 위한 프로세스 또한 형편없다.
- 이런 허술한 추정목표를 달성하는데 실패한 것과, 이보다 훨씬 중요한 프로젝트의 실패를 동일시 한다.
- 추정을 사이에 두고 경영진과 기술자 사이에 단절이 존재한다.

5. 복잡성

- 소프트웨어 구축의 많은 문제는 복잡성에 기인한다.
- 복잡성은 매우 빠르게 증가한다.
- 이 복잡성을 극복하기 위해서는 매우 뛰어난 사람들이 필요하다.

사실 1: 소프트웨어 작업에서 가장 중요한 요소는 프로그래머가 사용하는 도구가 아니라, 프로그래머의 자질이다.

✓ 사람이 중요

도구, 기술, 프로세스도 중요하지만 더 중요한 것은 사람이다.

✓ 사람보다 도구(?)

프로세스를 소프트웨어 개발에서 가장 중요한 것이라 주장하고, 도구를 소프트웨어 제작 능력의 돌파구라 선전
잡다한 기법을 모아놓고는 그것을 **방법론**이라 하며, 수천 명의 프로그래머에게 읽히고, 교육을 듣게 하고, 훈련 연습하도록 하며, 이를 중요한 프로젝트에 적용한다.
도구/기술/프로세스라는 이름 하에 이 모든 것이 이루어지고 사람은 뒷전으로 밀려나 있다.
아직도 많은 소프트웨어 공학 전문가들의 마음속에는 사람보다 프로세스가 중요하다는, 때에 따라서는 훨씬 더 중요하다는 생각이 자리잡고 있다.

✓ 사람이 접근하기 어려운 문제

거의 모두가 표면적으로는 사람이 도구나 기술, 프로세스 보다 중요하다는 것에 동의하지만, 행동은 이와 다르게 한다. 아마도 **사람**이 도구나 기술, 프로세스보다 **접근하기 어려운 문제**이기 때문일 것이다.

사실 2: 최고의 프로그래머는 최하의 프로그래머보다 28배 더 뛰어나다.

✓ 개발자 별 개인차

다른 이들보다 **뛰어난** 소프트웨어 실무자들이 있고, 그들이 **5배**에서 **28배**까지 뛰어나다.
가장 뛰어난 사람들을 잘 돌보는 것이 소프트웨어 관리자의 가장 중요한 업무이다.
사실 28배 뛰어난 사람들(그러면서 임금은 보통의 다른 동료들과 비교해 2배도 안 되는)은 소프트웨어 분야에서의 '봉'이다.

✓ 참고문헌

'우리의 연구에서 가장 중요한 실질적인 발견은 프로그래머 실문 능력의 현저한 개인차다' (Sackman 1968)
'한 그룹 내에서도, 프로그래머 간의 능력 차가 10배 이상 날 수 있다' (Schwartz 1968)
'개인간에 5배 정도의 생산성 차이는 흔한 것이다' (Boehm 1975)
Al Davis의 (1995)의 인용
원리 132 - 기술이 부족한 다수의 사람 보다는 뛰어난 소수의 사람이 낫다.
원리 141 - 소프트웨어 엔지니어 들도 개인에 따라 커다란 차이가 있다.

사실 3: 지체된 프로젝트에 사람을 추가 투입하면 프로젝트가 더 늦어진다.

✓ 프로젝트 일정 지연 시 추가인원 투입은 잘못된 일

프로젝트에 인원이 투입되면 제 속도로 일하도록 하는데 시간이 필요
새로운 인원은 생산성을 제대로 내기 까지 많은 것을 배워야 한다.
그들이 배워야 하는 것들은 주로 프로젝트 팀 내의 **다른 사람으로부터 얻어야** 한다는 것이다.
새로운 팀 멤버가 충분한 생산성을 내기 전까지는 기존 프로젝트 팀의 시간과 주의를 뺏는 요인이 된다.
프로젝트에 사람이 많을수록 커뮤니케이션은 더욱 복잡해진다.

✓ 논쟁

이미 애플리케이션 도메인에 대해 잘 알고 있다면 어떨까?
프로젝트가 거의 시작도 하지 않았다면 어떨까?

사실 4: 작업환경은 생산성과 품질에 지대한 영향을 미친다.

✓ 작업환경

소프트웨어 작업은 사고 집약적이므로, 작업환경은 생각을 촉진하는 곳이어야 한다. 사람이 붐벼서 자꾸 방해를 받는다면 거의 일을 진행할 수 없게 된다.

사실 5: 소프트웨어 업계에는 과대선전(도구와 기술에 대한)이 만연해 있다.

✓ 도구와 기술의 진보로 10~35%정도의 향상만 된다

프로그래머 없이 프로그래밍 할 수 있는 4세대 언어, 프로그래밍을 자동화하는 CASE도구, 소프트웨어를 구축하는 최상의 방법인 객체지향 기술, 차세대 방법론인 XP 등 오늘날 도약이라 부를만한 어떤 것을 가지고 있을 수 있다. 그러나 이런 것들이 우리의 소프트웨어 구축 능력을 **극적으로 향상시키지는** 못한다.

이 내용은 소프트웨어 프로세스 개선에 대한 최상의 관례집에 있는 도표에서 나타나는데, 거기서 가장 큰 이득이 큰 프로세스 변경은 재사용이었다. 재사용으로 인한 이득은 10~35% 정도로, 10배 이상의 이득을 얻을 수 있다는 최근 컴포넌트 열광자들의 터무니없는 주장과 배치된다.

사실 6: 새로운 도구와 기술은 도입 초기에 생산성/품질 저하를 초래한다.

✓ 학습곡선을 극복한 이후에 이득을 얻을 수 있다.

새로운 도구나 기술을 배우는 데는 비용이 든다. 어떻게 적용할 지 결정하고, 언제 사용하고 언제 사용하지 말지를 고려해야 한다.
새로운 두구나 기술을 적용한 첫 번째 프로젝트는 보통의 경우보다 빠르지 못하고 더 느리게 진행될 것임을 의미한다.

생산성과 품질 향상을 위해서 새로운 도구와 기술을 채택하는데 뭔가 새로운 것을 시도할 때 처음에는 생산성과 품질이 저하되는 것은 프로세스의 아이러니다.

'얼마나 오래 걸릴 것인가?' 라는 질문은 새로운 것을 익히는데 필요한 학습곡선을 말하는 것이다.
'얼마나 오랫동안 생산성이 저하될 것인가? 얼마나 빨리 일반수준에 도달할 것인가? 궁극적인 이득을 얻는 데는 얼마나 걸릴 것인가?

학습곡선의 길이는 상황과 환경에 종속적이다.
객체지향 기술을 피상적으로 배우는 데는 석 달이면 되겠지만, 능숙해지기 위해서는 2년 내낸 몰두해야 할 것이다.

새로운 아이디어가 얼마나 많은 이득을 가져올 것인가? 이 또한 알 수 없다. 아마 이전 보다 5~35%의 이득을 얻을 것이다.

사실 7: 소프트웨어 개발자는 도구에 대해 많은 말을 하지만, 별로 사용하지 않는다.

✓ 소프트웨어 도구는 거의 사용되지 않는다

소프트웨어 개발자들에게 도구는 장난감이다. 그들은 새로운 도구에 대해 배우고, 사용해보고, 구입하는 것을 좋아한다. 그러나 이런 도구들이 거의 사용되지 않는다.

10여 년 전에 CASE도구가 한참 주가를 올리던 때는 CASE도구가 소프트웨어 개발의 미래 소프트웨어 개발 프로세스를 자동화시킬 방법인 것처럼 보였고, 엄청나게 많은 도구가 판매되었다.

그러나 대부분의 CASE도구는 선반(shelf)에 놓인 다음 전혀 사용되지 않았고, 이런 현상을 묘사하는 용어로 셸프웨어(shelfware)란 말이 생겨났다.

✓ 왜 이런 도구들이 사용되지 않을까?

새로운 도구나 기술을 시도할 때, 즉각적인 생산성 향상은 없고 실제로는 초기에 생산성이 감소한다. 그리고 개발자들은 너무도 자주 자신들이 가장 잘 아는 항상 사용하던 도구로 되돌아간다.

잘 알려진 프로그래밍 언어를 위한 컴파일러, 이 언어를 위한 디버거, 자신들이 가장 좋아하는 텍스트 에디터, 생각할 필요가 없을 정도로 익숙한 링커(linker)와 로더(loader), 작년(또는 10년 전)부터 사용하던 형상관리 도구

사실 8: 폭주하는 프로젝트의 가장 흔한 원인 두 가지 중 하나는 부정확한 추정이다.

✓ 소프트웨어 공학 분야에서 폭주의 원인이 무엇일까?

어떤 사람들은 적절한 방법론의 부재가 폭주를 야기한다고 말한다. 이런 사람들은 보통 방법론을 파는 사람들이다. 어떤 사람들은 좋은 도구가 없기 때문이라고 말한다. 어떤 사람들은 프로그래머들에게 훈련과 주의가 부족하기 때문이라고 말한다.

✓ 추정 작업은 매우 부정확하다

추정은 프로젝트의 기간이 얼마나 걸릴지, 비용은 얼마나 들지를 결정하는 프로세스다.

소프트웨어 분야의 추정작업은 매우 부정확하다. 대부분의 추정이 현실적인 목표라기 보다는 **희망**에 가깝다.

그 결과 사람들은 불가능한 **추정 목표**를 맞추기 위해 노력한다. 지름길이 선택되고, 좋은 관례는 생략된다.

추정 능력을 개선하기 위해 모든 종류의 접근방법을 시도했다.

1. '전문가들', 즉 예전에 비슷한 프로젝트를 경험한 개발자들에게 의지했다. 이 접근방법의 문제는 너무 주관적이라는 것이다.
2. 알고리즘적 접근방법을 시도했다. 추정에 대한 답을 줄 수 있는 매개변수 방정식을 만든다. 프로젝트 관련 데이터를 넣어 주면 알고리즘이 작동하고 추정을 산출한다. 그러나 이 방법은 완전히 다른 결과(2배에서 8배)를 산출한다는 것이 밝혀졌다.
3. 코드의 라인 수(LOC, line of Code)와 같은 데이터를 근거로 추정하는 것을 지지한다. 그러나 시스템 얼마나 많은 LOC를 가질지 알아내는 것은 기간과 비용이 얼마나 들지를 아는 것보다 더 어려울 것이다.
4. 입력 수와 출력 수 같은 핵심 파라미터(FP, 기능점수, Function Point)를 살펴보고 이에 근거해 추정해야 한다고 말한다. 이런 방법도 문제가 있다. 첫째, 무엇을 세야 하는지, 어떻게 세야 하는지에 대한 의견이 다르다. 둘째는 어떤 어플리케이션에는 FP가 의미를 가지지만, 어떤 경우(기능의 개념이 충분하지 않는 애플리케이션)는 아무런 의미가 없다.

사실 9: 소프트웨어 추정은 보통 부적절한 시기에 수행된다.

✓ 추정은 부정확한 것이 추정의 시기이다

추정의 시기에 대한 것이다. 보통 프로젝트를 **시작할 때** 소프트웨어 추정을 한다.

의미 있는 추정을 얻기 위해서는 해당 프로젝트에 대해 상당히 많이 알아야 한다. 그러나 요구사항을 결정하는 단계는 생명주기의 첫 단계, 바로 프로젝트 시작 부분이다. 요구사항 정의는 무슨 문제들 풀지에 대해 생각하는 것이다. 그런데 풀고자 하는 문제가 무엇인지 모른다면 어떻게 솔루션을 만드는 데 드는 비용과 시간을 추정할 수 있겠는가?

그러나 이런 사실이 '분명 잘못된 것이다'라는 말이 나올것 같지만 지금까지 누구도 그러지 않다. 대신 머리를 끄떡이며 이해나 동의를 할 뿐이었다.

사실 10: 소프트웨어 추정은 보통 부적절한 사람들에 의해 수행된다.

✓ 추정이 부정확한 이유는 누가 추정을 하느냐이다

소프트웨어 프로젝트를 추정하는 사람들은 소프트웨어 구축에 대해 뭔가를 아는 사람들, 즉 **소프트웨어 엔지니어**나 그들의 **프로젝트 리더** 또는 **관리자**여야 한다. 그러나 대부분의 경우 소프트웨어 추정은 소프트웨어 제품을 원하는 사람들, 즉 경영진이나 마케팅, 고객, 사용자들에 의해 이루어진다.

소프트웨어 추정은 현재로선 현실이라기보다는 희망사항에 가깝다. 경영진이나 마케팅은 소프트웨어 제품이 내년 1사분기에 출시되기를 원한다. 따라서 이는 맞춰야 하는 일정이다. 이런 상황에서는 추정이 실제로 거의 또는 전혀 일어나지 않음에 주목하지 바란다.

사실 11: 프로젝트가 진행되면서 소프트웨어 추정을 수정하는 경우는 거의 없다.

✓ 추정이 부정확하더라도 현실에 맞게 조정되지 않는다

소프트웨어 분야 사람들은 처음의 잘못된 추정에 죽고 살아야 하는 경우가 많다. 고위 경영진은 추정을 수정하는 데 별로 관심이 없다. 현실적 추정 대신 희망사항에 대해 그렇게도 자주 표명을 했는데, 왜 그런 희망사항을 함부로 변경하도록 하겠는가?

프로젝트에서 작업이 어떻게 진척되는지, 어떤 마일스톤들을 거쳐야 하는지, 심지어는 최종 마일스톤이 어디까지 연장되어야 하는지 까지도 추적할 수 있다. 그러나 프로젝트가 최종시한에 이르렀을 때 그 성패 여부는 대개 처음에 추정했던 수치에 의해 판단된다. 하지만 그 수치는 **부적절한 시점**에 **부적절한 사람들**이 만들어낸 수치가 아닌가?

사실 12: 소프트웨어 추정이 부정확한 것은 별로 놀라운 일이 아니다. 그러나 우리는 추정에 죽고 산다!

✓ 소프트웨어 프로젝트는 항상 일정에 의해 관리된다

부적절한 시점에 부적절한 사람에 의해 만들어지고 잘 변경되지 않는, 정말 그렇게도 부적절한 소프트웨어 추정이라면, 상대적으로 덜 중요하게 다뤄질 것이라 생각되지 않는가? 틀렸다! 사실 소프트웨어 프로젝트는 거의 항상 **일정**에 의해 관리된다. 그 때문에 적어도 고위 경영진은 소프트웨어에서 일정을 가장 중요한 요소로 생각한다.

일정을 관리한다는 것은 여러 개의 장기, 단기 마일스톤(아주 작은 것은 인치페블(inch-pebbles)이라 불리기도 한다)을 구성하고 그 일정상의 어느 시점에 발생하는 사황에 따라 프로젝트가 성공적으로 진행되고 있는지 아닌지 결정하는 것을 의미한다. 만약 예정보다 늦게 마일스톤 26에 도달했다면, 프로젝트에 문제가 생긴 것을 의미한다.

✓ 소프트웨어 프로젝트를 관리할 수 있는 방법

- 제품에 의한 관리: 최종 제품을 어느 정도 사용할 수 있고 어느 정도 작동하는가에 따라 프로젝트의 성공 또는 실패를 말할 수 있을 것이다.
- 이슈에 의한 관리: 프로젝트를 수행하는 동안 항상 발생하는 이슈를 얼마나 빠르게 잘 해결하는가에 따라 프로젝트의 성공 또는 실패를 말할 수 있을 것이다.
- 위험요소에 의한 관리: 프로젝트 초기에 확인된 위험요소가 적절히 해결되었는지를 보여주는 일련의 시험을 통해 프로젝트의 성공 또는 실패를 말할 수 있을 것이다.
- 비즈니스 목표에 의한 관리: 해당 소프트웨어가 비즈니스의 능률을 얼마나 향상시켰는가에 따라 프로젝트의 성공 또는 실패를 말할 수 있을 것이다.
- 품질에 의한 관리: 제품의 품질 속성을 얼마나 많이 달성했는가에 따라 프로젝트의 성공 또는 실패를 말할 수 있을 것이다.

사실 13: 경영진과 프로그래머 사이에는 단절이 있다.

✓ 경영진이 실패라고 여기는 프로젝트를 개발자들은 가장 성공적이었다고 생각한다

예산은 419%, 일정은 194% 초과했고, 원래의 추정 규모보다 소프트웨어는 130%, 펌웨어는 800% 초과했다. 그러나 프로젝트는 성공적으로 완료됐다.

그러나 잘 작동하는 유용한 제품을 만들어냈다는 사실만으로 '가장 성공적인' 프로젝트라고 하는 것은 여전히 이상하지 않은가?

기술자들이 그 프로젝트를 성공적으로 보는 데는 몇 가지 다른 이유가 있었다.

- 제품은 의도한 대로 잘 작동했다.
- 제품을 개발하는 것은 기술적 도전이었다.
- 팀의 규모는 작았지만 높은 성취도를 보여주었다.
- 지금까지 경영진 중 가장 좋았다. 범위 변경도 없었으며 일정으로 압박하지도 않았다.

프로젝트가 늦어진 이유에 대한 견해를 물었는데, 다음과 같이 답했다.

- 일정에 관한 추정이 비현실적이었다.
- 자원, 특히 전문적인 조언이 부족했다.
- 처음에는 범위가 제대로 파악되지 않았다.
- 프로젝트가 늦게 시작됐다.

이 모든 것이 프로젝트 초기에 대한 사실이라는 것이다.

기술자들이 아무리 열심히, 훌륭하게 일했다 하더라도 경영진의 기대를 만족시킬 수는 없을 것 같았던 것이다.

Jeffery와 Lawrence(1985)는 '추정을 전혀 하지 않은 프로젝트가 생산성 측면에서 가장 성공적이었다'는 사실을 발견했다. 바로 다음은 기술자들이 추정을 수행한 프로젝트가 차지했고, 관리자가 추정을 수행한 프로젝트는 최악이었다.

사실 14: 타당성 조사에 대한 대답은 항상 '타당하다'이다.

✓ 새로 이사 온 아이 (new kid on the block)

➔ 장소나 조직에 새로 와서 익숙하지 않기 때문에 새로 배울 것이 많은 사람을 뜻한다.

'새로 이사 온 아이' 현상이 우리에게 영향을 미치는 다른 방법은 우리가 너무도 자주 구제불능의 낙관론을 가진다는 것이다. 마치 아무도 풀 수 없었던 문제를 우리가 풀 수 있기 때문에, 어떤 새로운 문제도 너무 어려워서 풀지 못할 정도는 아니라고 믿는다. 이 말이 맞을 때도 있고 그렇지 않을 때도 있다.
프로젝트를 내일이나 또는 수일 내로 끝낼 수 있다고 믿을 때, 아무런 오류 없이 소프트웨어를 즉각적으로 만들어 낼 수 있다고 믿을 때, 오류 제거 단계가 종종 시스템 분석, 설계, 코딩 단계를 합친 것보다 더 오래 걸리는 경우도 있다.

낙관론은 기술적 타당성이 이슈일 때 정말 문제가 된다. 실제 시스템 구축에 앞서 타당성 조사를 하는 프로젝트에서(이런 경우는 거의 없지만), 타당성 조사에 대한 대답은 거의 항상 '타당하다, 우리는 할 수 있다'이다. 그리고 프로젝트가 한참 진행된 어느 시점에서 그 대답이 잘못됐다는 것이 밝혀진다.

사실 15: 소규모 재사용은 잘 해결된 문제다.

✓ 재사용의 시초

1950년대 중반에 IBM 메인프레임의 과학용 애플리케이션을 위한 사용자 조직이 형성됐는데, 이 조직의 중요한 기능 중 하나는 소프트웨어 서브루틴을 제공받아 이에 대한 정보를 공유하는 것이었다. 조직의 이름도 그에 걸맞게 Share라 불렸으며, 제공 받은 루틴은 재사용 가능한 소프트웨어 라이브러리의 시초가 되었다.

초기의 소프트웨어 루틴 라이브러리는 오늘날 우리가 소규모 재사용 루틴이라 부르기도 하는 수학 함수, 정렬과 병합, 제한된 범위의 디버거, 문자열 처리기와 같이 프로그래머라면 적어도 한번쯤은 필요로 했던 매우 유용한 기능을 모두 포함하고 있었다.

그 당시 재사용은 라이브러리 안에 들어 있는 것에 대한 질적인 관리 없이 무조건 넣고 쓰는 식이었다. Share 라이브러리 루틴을 재사용 하는데 있어 아무런 질적 문제도 없었던 것으로 기억한다.

사실 16: 대규모 재사용은 여전히 해결되지 않은 어려운 문제다.

사실 17: 대규모 재사용은 서로 관련 있는 시스템 사이에서 가장 잘 적용된다.

사실 18: 재사용 가능 컴포넌트는 만들기가 3배 어렵고, 3곳에 적용해봐야 한다.

사실 19: 재사용된 코드를 수정하는 것은 특히 오류를 범하기 쉽다.

✓ 기존 소프트웨어를 수정하는 것은 상당히 어렵다.

시스템을 구축하거나 유지보수하는 경우 그 복잡성 때문에 기존의 소프트웨어를 수정하는 것은 상당히 어려울 것이다. 보통 소프트웨어 시스템은 어떤 설계 계층 위에 **설계 철학**을 가지고 구축된다. 소프트웨어를 수정하려는 사람이 그 계층을 이해하지 못하거나 설계 철학을 받아들이지 않는 다면 수정 작업을 성공적으로 마무리하는 것은 매우 어려울 것이다.

✓ 유지보수 문서는 여러 이유 때문에 무시가 된다.

기존 소프트웨어를 수정하는 것이 어려운 데는 또 다른 근본적인 문제가 있다.
그것은 '**기존 솔루션을 이해하는 것**'이다. 원래 그 솔루션을 구축한 프로그래머 조차도 몇 달이 지난 후에는 그 코드를 수정하는 데 어려움을 느낀다는 것은 소프트웨어 분야에서 잘 알려져 있다.

이 문제를 해결하기 위해 유지보수 문서라는 개념을 생각했다. 유지보수 문서는 처음에 소프트웨어 설계 문서로 시작한다. 모두가 유지보수 문서의 필요성을 인정하지만, 소프트웨어 프로젝트에 **비용이나 일정 문제**가 발생했을 때 대개는 이런 문서 작성 작업이 **제일 먼저 무시**된다. 그 결과, 적절한 유지보수 문서를 가진 소프트웨어 시스템은 거의 없다.
설상가상으로, 유지보수 중에도 소프트웨어는 수정되지만 유지보수 문서는 그에 맞게 수정되는 경우는 별로 없다.

사실 20: 디자인 패턴 재사용은 코드 재사용 문제에 대한 해결책이다.

✓ 디자인 패턴

새로운 문제가 발생할 때 마다 처음부터 다시 하지 않으려면 프로그래머는 어떻게 해야 할까?

➔ 개발자들이 항상 해온 방법 중 하나는 **과거의 솔루션을 기억해 현재의 문제를 해결**하는 것이다.

'디자인 패턴'의 4가지 필수 요소

- 패턴의 이름
- 어떤 경우에 솔루션을 적용해야 하는가에 대한 설명
- 솔루션 그 자체에 대한 설명
- 그 솔루션을 사용함으로써 얻는 결과

패턴이 소프트웨어 분야에서 빠르게 채택된 이유는 무엇일까?
실무자들은 이던 일을 예전에도 해왔는데 그것이 새로운 구조와 관심으로 가져진 것일 뿐이라고 생각한다. 학자들은 패턴이 설계와 관련되고, 코드보다 추상적이고 개념적이라는 측면에서 코드 재사용보다 흥미로운 개념이라 생각한다.

사실 21: 문제의 복잡성이 20% 증가하면 솔루션의 복잡성은 100% 증가한다.

✓ 왜 그렇게 사람이 중요한가?

복잡성을 극복하는 데는 상당한 사고력과 기술이 필요하기 때문이다.

✓ 왜 그렇게 추정이 어려운가?

단순해 보이는 문제도 그 솔루션은 훨씬 복잡할 수 있기 때문이다.

✓ 왜 대규모 재사용은 성과가 좋지 않을까?

복잡성이 다양성을 증대시키기 때문이다.

✓ 왜 요구사항은 폭발적으로 증가하는가?

25% 부분에서 100% 부분으로 옮겨가는 중이기 때문이다.

✓ 왜 하나의 문제의 솔루션을 설계하는 데 설계방법이 그렇게도 많이 존재하는가?

솔루션 공간(solution space)은 매우 복잡하기 때문이다.

✓ 왜 최고의 설계자는 반복적, 경험적 접근방법을 사용하는가?

단순하고 명확한 설계 솔루션이 있는 경우는 거의 없기 때문이다.

✓ 왜 설계는 최적화가 되는 경우가 거의 없는가?

상당한 복잡성 때문에 최적화는 거의 불가능하기 때문이다.

사실 21: 문제의 복잡성이 20% 증가하면 솔루션의 복잡성은 100% 증가한다.

- ✓ 왜 100%의 테스트 커버리지는 불가능하고, 테스트는 어떤 경우든 부족한 것인가?
  프로그램 내의 많은 실행경로와 복잡성으로 인해 테스트 커버리지가 잡아내지 못하는 오류가 있기 때문이다.
- ✓ 왜 검사(inspection)가 오류 제거에 대한 가장 효과적, 효율적인 접근 방법인가?
  그 모든 복잡성을 걸러내고 오류의 위치를 찾는 데는 결국 사람의 노력이 필요하기 때문이다.
- ✓ 왜 소프트웨어 유지보수에 그렇게도 많은 시간이 소모되는가?
  초기단계에 문제의 솔루션에서 파생될 모든 가능성을 결정하는 것은 거의 불가능하기 때문이다.
- ✓ 왜 '기존의 제품을 이해하는 것'이 유지보수에서 가장 중요하고 어려운 작업인가?
  하나의 문제를 해결하는 데도 적용할 수 있는 접근방법이 매우 많기 때문이다.
- ✓ 왜 소프트웨어에는 그렇게 많은 오류가 있는가?
  처음부터 소프트웨어를 올바르게 이해하는 것은 매우 어렵기 때문이다.
- ✓ 왜 소프트웨어 연구자들은 옹호에 치중하는가?
  복잡한 소프트웨어 세계에서는, 옹호보다 우선해 필요한 평가적 연구를 수행하는 것이 매우 어렵기 때문이다.

사실 22: 소프트웨어 작업의 80%가 지적인 작업이다. 그 중 상당부분이 창조적인 작업이다. 사무적인 일은 거의 없다.

✓ 소프트웨어는 80%의 지적인 작업, 20%의 사무적인 작업이다

컴퓨터 작업이 하찮고 자동화할 수 있는 것인지 아니면 아주 복잡한 것인지 알아내려면 어떻게 해야 할까?
➔ 작업중인 프로그래머를 자세히 관찰해보면 된다.

시스템 분석작업(요구사항 정의)을 수행하는 시스템 분석가를 비디오 테이프에 녹화했다.
뙈 오랜 시간 동안 시스템 분석가들은 아무것도 하지 않았다. 그 뒤에 주기적으로 뭔가를 메모하곤 했다.
이런 패턴을 일정 시간 관찰한 후, 아무것도 하지 않고 앉아 있는 것은 곰곰이 생각하는 것이고, 뭔가 메모하는 것은 그 생각의 결과를 기록하는 것임을 알게 되었다.

시스템 분석가들을 녹화한 비디오 테이프를 분석한 결과 시스템 분석가들은 대략 시간의 80%를 생각하는 데 사용했고 20%는 메모하는 데 사용했다. 다른 식으로 말하면 시스템 분석 작업의 80%는 지적인 작업이었고 나머지 20%는 사무적인 작업이었다.

1부. 사실

I. 관리

II. 생명주기

- 요구사항
- 설계
- 코딩
- 오류제거
- 테스트
- 검토와 검사
- 유지보수

III. 품질

IV. 연구

2장. 생명주기

요구사항(requirement)

요구사항이 빈번하게 바뀐다고 말했던 것을 기억하는가? 바로 이 현상에 대해 매우 중요한 사실이 몇 가지 있다. 그리고 요구사항과 관련된 문제를 어떻게 다루는지 살펴볼 것이다.

설계(design)

설계는 생명주기 중 가장 지적이고 가장 창조적인 부분으로, 소프트웨어 개발 프로세스가 실제 얼마나 복잡한지를 보여준다.

코딩 (coding)

코딩은 소프트웨어 개발 프로세스에서 가장 기본적인 부분이다.

오류제거(error removing)

생명주기 앞부분의 지적 도전과 복잡성을 지나 마침내 제품이 탄생하지만 제대로 동작하는 경우는 거의 없다. 오류 제거에 대한 도전이 여기 사실에 반영되어 있다.

테스트(test)

테스트가 오류제거 활동에서 가장 중요한 것은 분명하지만, 모든 오류를 발견할 정도로 완전하게 소프트웨어를 테스트하는 것은 거의 불가능하다.

검토(review) 와 검사(inspection)

테스트로 오류를 완전하게 제거할 수는 없으므로, 검토나 검사와 같은 약간의 정적인 접근방법으로 보완해야 한다. 그러나 테스트에 검토와 검사를 추가한다 해도 소프트웨어 제품에는 여전히 오류가 많이 포함되어 있을 것이다. 이 두 가지 활동에 대한 사실에서 높은 품질의 소프트웨어 제품을 만드는 것이 얼마나 어려운 일인지 살펴볼 것이다.

유지보수(maintenance)

유지보수는 가장 적게 이해된 부분이고, 많은 면에서 볼 때 생명주기 중 가장 중요한 단계다.

사실 23: 폭주하는 프로젝트에서 가장 흔한 원인 두 가지 중 하나는 불안정한 요구사항이다.

✓ 폭주하는 프로젝트

프로젝트 통제에서 벗어난 프로젝트를 말한다.

✓ 부정확한 추정

부정확한 또는 낙관적인 추정은 폭주하는 프로젝트의 두 가지 주요 원인 중 하나다.

✓ 불안정한 요구사항

소프트웨어 솔루션의 고객과 사용자가 어떤 문제를 해결해야 하는지에 대해 확실하게 알지 못한다는 사실에서 기인한다.

사실 24: 요구사항의 오류는 생산 단계에서 수정하는데 가장 비용이 많이 든다.

✓ 소프트웨어 수정 비용

소프트웨어 출시 이후 오류를 수정하는 것은 개발 단계의 초기에 수정하는 것보다 100배 많은 비용이 든다.

소프트웨어 제품에서 이런 오류는 가능한 생명주기의 초기 단계에서 제거하라는 것이다.
그런데 이 사실을 자주 잊어버리는 이유는 무엇일까?
➔이 사실을 인정하긴 하지만 아무런 조치도 취하지 않기 때문인 것 같다.
➔초기에 요구사항 오류를 제거하기 위해서는 요구사항 정제기법을 비중 있게 사용해야 하는데 일정에 맞추기 위해 허둥지둥 서두르느라 미쳐 챙기지 못한다.

요구사항 정제기법
- 요구사항 일관성 검사
- 요구사항 검토
- 요구사항에서 파생되는 초기 테스트 케이스 설계
- 테스트 가능한 요구사항 고려 등

✓ 대처방법

개발자는 검토(review)
테스터와 품질보증 쪽 사람들은 테스트 가능한 요구사항과 초기 테스트 케이스 구축
시스템 분석가는 모델링을 통한 접근법
XP 지지자들은 고객 대표를 개발팀에 참여시키려 할 것

사실 25: 누락된 요구사항은 가장 수정하기 힘든 오류다.

✓ 요구사항

풀어야 할 문제를 정의하는 것이다.
요구사항이 어떻게 수집되는가? ➔ 사람간의 상호작용이 필요한 프로세스다.

소프트웨어 요구사항 수집 프로세스는 1) **반복적**이어야 하고 2) 많은 **상호작용**이 필요하다.

✓ 누락된 요구사항이 엄청난 파괴력을 지닐까?

문제 해결의 난이도에 영향을 주고, 요구사항 간의 상호 작용은 문제에 대한 솔루션의 복잡성을 급속히 증가시키기 때문이다.

✓ 누락된 요구사항은 발견하기도 어렵고 수정하기도 어려울까?

그건 소프트웨어 오류 제거 프로세스의 가장 기본적인 부분이 요구사항으로부터 유도되기 때문이다.
테스트케이스 정의 > 요구사항이 없으면 명세서에 나타나지 않음 > 검토나 검사에도 확인되지 않음 > 요구사항을 만족하는지 검증하기 위한 테스트 케이스도 작성되지 않음

사실 26: 명시적 요구사항을 설계로 옮겨갈 때 '파생 요구사항'이 폭발적으로 증가한다.

✓ 추적용이성

요구사항이 수정 또는 개선됐을 때 변경해야 할 모든 소프트웨어 구성 요소를 확인할 수 있다는 것이다.
소프트웨어를 유지보수 하는 사람들에게는 엄청난 도움이 된다.

✓ 추적용이성의 실현 가능성

추적용이성이 바람직하긴 해도, 실현하기 어렵다는 것이 밝혀졌다.
왜 그럴까? 단순 연결리스트? ➔ 이유는 요구사항의 폭발적 증가 때문이다.
하나의 요구사항이 50개 이상의 설계 요구사항으로 연결되고 더 많은 수의 코딩 요소와 연결된다.

사실 27: 소프트웨어 문제에서 최적의 솔루션이 하나 존재하는 경우는 거의 없다.

✓ 요구사항의 폭발적 증가로 인해 최상의 솔루션은 어렵다.

이 사실에는 두 개의 키워드가 있다. '하나'와 '최상'이 그것이다.
대부분의 소프트웨어 문제는 다양한 방법으로 풀 수 있다. 이것은 '**하나**'에 대한 것이다. 그리고 최상의 솔루션이 어떤 것인지를 알아내는 것은 매우 어렵다. 이것은 '**최상**'에 대한 것이다.

프로세스의 복잡성과 요구사항의 폭발적 증가는 소프트웨어 설계가 단순히 최상의 솔루션을 찾는 것과는 거리가 먼 어렵고 복잡한 프로세스임을 나타낸다.

사실 28: 설계는 복잡하고 반복적인 과정이다. 초기 설계 솔루션은 보통 잘못 되었거나, 최적이 아닌 경우가 많다.

✓ 최적의 설계 솔루션은 없다.

최고의 설계자들은 설계 솔루션을 찾으려 할 때 순차적인 상향식(bottom-up) 또는 하향식(top-down) 방법으로 작업하지 않고, 중요 상황에 중점을 둔다. 이런 중요 상황은 보통 설계자가 즉각적으로 솔루션을 떠올릴 수 없는 어려운 문제를 의미한다.

전문 설계자들은 설계 작업에서 핵심을 파악할 때 경험적, 시행착오적 접근방법을 사용한다는 것이다.
설계는 예측 가능한, 구조화할 수 있는, 규격화할 수 있는 프로세스와는 거리가 멀고, 너저분하고 시행착오적인 것임을 알 수 있다.
대부분의 설계 초보자들이 적용하고 싶은 접근방법은 '쉬운 부분을 먼저 처리하는' 방법일 것이다.

설계는 소프트웨어 개발 프로세스에서 가장 지적인 활동일 것이다. 초기 설계 솔루션이 잘못될 가능성이 높다는 것도 쉽게 알 수 있다.

복잡한 프로세스로 인해 최적의 설계는 보통 불가능하므로, 우리는 '**만족스러운**' 솔루션을 위해 노력해야 한다.
만족스러운 솔루션은 훌륭한 설계에 대한 조건을 충분히 만족시켜, 그 접근방법을 선택해 작업을 계속하는 데 따른 위험을 감수할만한 가치가 있는 것을 의미한다.

사실 29: 설계자의 기본단위와 프로그래머의 기본단위가 일치하는 경우는 거의 없다.

✓ 설계에서 코딩으로의 전환 시 문제를 초래할 수 있다.

설계에서 코딩으로의 전환은 보통 매끄러운 작업이라고 생각한다. 설계하는 사람과 코딩하는 사람이 같다면 맞는 말이다.
그러나 작업을 분할하는 경우, 설계를 코드로 옮길 수 있는 최적의 방법을 이야기하는 것이 중요해진다.

설계자가 코더보다 높은 수준의 기본단위(추상화 정도)를 사용한다면, 결과로 나오는 설계는 코더가 작업을 시작하는 데 부적절할 것이다. 따라서 코더는 코딩을 하기 전에 적절한 수준의 설계를 추가하기 위해 시간을 소모해야 할 것이다.

반대로 설계자의 경험이 부족하다면 설계자는 매우 상세한 수준의 설계를 만들 것이다. 그러나 이 설계자보다 경험이 많은 코더는 지나치게 상세한 수준의 설계를 받아들이지 않고 자신의 설계 아이디어로 대체할 것이다.

이는 설계자가 코더보다 똑똑하거나 숙련된 사람이어야 하는가에 대한 문제가 아니라, 그들이 같은 경험과 기본 단위를 가지고 있는가에 대한 문제다.

사실 30: COBOL은 별로 훌륭한 언어가 아니지만, (비즈니스 데이터 처리에 대해서는) 다른 언어도 마찬가지다.

?

사실 31: 오류 제거는 생명주기에서 가장 많은 시간을 소모하는 단계다.

✓ 오류제거 단계

많은 사람들은 이 단계를 '테스트 단계' 혹은 '검증과 확인(verification & validation)'이라 불렀다.
대부분의 소프트웨어 제품에 대해 요구사항을 수집하거나 솔루션을 설계하거나 코딩하는 것보다 오류를 제거하는 작업이 더 오래 걸린다는 사실이다.

생명주기의 시작부터 둘러보면 요구분석, 설계, 코딩 작업을 시각화하는 것은 어렵지 않다.
이런 단계에서는 구체적이고 최소한 부분적으로라도 예측 가능한 작업을 한다. 그러나 오류 제거 단계에서 하는 일은 **예측 불가능**하다.

✓ 소모된 시간 비율

요구분석에 20%, 설계에 20%, 코딩에 20%(대부분의 프로그래머는 직관적으로 이 단계에서 가장 많은 시간을 소모할 것으로 생각), 그리고 오류 제거에 40%를 사용한다. (Glass 1992)

사실 32: 프로그래머가 완전하게 테스트했다고 믿는 소프트웨어도 보통은 로직 경로의 55~60%만 테스트된 경우가 많다.

✓ 테스트 방법

- 요구사항 중심의 테스트(requirements-driven testing): 요구사항을 만족시키는지 확인하는 테스트
- 구조적 테스트(structure-driven testing): 소프트웨어를 구성하는 모든 기능이 제대로 동작하는지 확인하는 테스트
- 통계적 테스트(statistics-driven testing): 소프트웨어가 얼마나 오래, 얼마나 잘 동작하는지 랜덤하게 테스트 하는 방법
- 위험요소 중심의 테스트(risk-driven testing): 주요 위험요소가 적절히 처리되는지 확인하는 테스트

✓ 구조적 테스트 커버리지

- 연구에 따르면, 프로그래머가 모든 코드를 완전히 테스트했다고 말할 때, 보통 55~60%의 로직 경로만이 실제로 테스트된 것이라 한다(Glass 1992)
- 이는 소프트웨어 제품의 복잡성을 반증하는 것이기도 하고, 소프트웨어를 작성한 사람조차도 그들이 작성한 것에 대해 완전히 이해하지 못함을 보이는 것이기도 하다.
- 구조적 테스트 커버리지를 찾아내는 도구(테스트 커버리지 분석기라 불린다)를 사용할 경우 테스트되는 로직 경로의 비율을 85~90%까지 끌어올릴 수 있다고 한다(Glass 1992)

✓ 결론

수많은 종류의 테스트 방법이 존재하지만 소프트웨어 제품의 복잡성 때문에 완벽한 테스트는 불가능하다는 것이다.
이 때문에 1) 테스트를 수행하는 데 있어 타협할 수 밖에 없고 2)대부분의 주요 소프트웨어 제품이 오류를 포함한 채 출시되는 것 또한 놀랄만한 일이 아니다(순진한 사람들만이 오류 없는 소프트웨어를 기대한다).

사실 33: 100% 테스트 커버리지도 결코 충분하지 않다.

✓ 오류 없는 소프트웨어는 가능(?)

1. **누락에 의한 오류**로, 필요한 작업을 수행하기 위한 로직이 프로그램에 없는 경우
   - 로직이 존재하지 않으면 로직 경로 커버리지를 아무리 늘려도 해당 오류를 발견할 수 없다.
2. **조합에 의한 오류**로, 특정 조합의 로직 경로가 실행될 때만 나타나는 경우
   - 각각의 로직 경로를 실행시킬 때는 성공적일 수 있으나, 특정 조합일 경우에는 오류가 발생할 수 있다.

✓ 얼마나 자주 발생할까?

이런 종류의 오류가 얼마나 자주 발생할까? ➔ 대답은 '아주 많이'다.
누락된 로직에 대한 오류는 35%
조합에 의한 오류는 40%나 차지했다.

사실 34: 테스트 도구는 꼭 필요하지만, 많은 경우 거의 사용되지 않는다.

✓ 테스트 단계는 인정을 받지 못한다.

요구분석과 설계 같은 생명주기 앞쪽 단계에는 사람들이 많은 관심을 가진다. 그러나 테스트와 유지보수 같은 생명주기의 뒤쪽 단계는 그렇지 못한다.

디버거, 커버리지 분석기, 환경 시뮬레이터, 테스트 관리자, 기록/재생을 자동화 하는 테스터, 표준화 테스터, 테스트 데이터 생성기 등 테스트를 위한 도구는 거의 사용되지 않는다.

✓ 테스트 도구가 널리 사용되지 않는 이유

1. 소프트웨어 분야의 경영진과 학계는 모두 앞쪽 단계를 훨씬 더 중요하게 보이도록 만들었다.
2. 뒤쪽 단계의 작업은 기술적으로 너저분하다. 앞쪽 단계에서는 실제로 분석 또는 설계를 할 수 없다 하더라도 상상해서 채워 넣을 수 있지만, 테스트나 유지보수에서는 그럴 수가 없다.
3. 생명주기의 뒤쪽 단계에는 **일정 압박**이 심하다. 특히 주어진 시간이 너무 짧아 테스트를 철저히 하지 못하는 경우도 있다.

사실 35: 특정 테스트 프로세스는 자동화할 수 있고, 또 자동화해야 한다. 그러나 자동화할 수 없는 테스트 작업도 많다.

✓ 자동화 된 작업

- 테스트 동작을 기록/재생하는 도구는 테스트의 입력을 기록해두었다가 필요할 때 다시 실행시킬 수 있어 매우 편리하다.
- 테스트 관리자는 반복적으로 일련의 테스트 케이스를 실행하고 그 결과를 테스트 오라클과 비교할 때 상당히 편리하다.
- 회귀 테스트를 생성하고 조작하는 것은 코드에 새로운 변경을 가했을 때 예전에 잘 동작하던 코드에서 오류가 생기는지 확인하는 훌륭한 방법이다.

✓ 자동화 되지 않은 작업

- 어떤 것을 어떻게 테스트해야 하는지 선택하는 작업
- 각 테스트가 나타내는 동치류(equivalence class) 수가 최대가 되도록 하는 테스트 케이스 생성
- 테스트 오라클을 만들기 위해 예상되는 테스트 결과를 수집하는 작업
- 테스트 프로세스를 위한 기반 구조 계획 작업
- 어떤 테스트가 수행되어야 하는지, 그리고 나중에는 어떤 테스트가 완료되었다고 생각할 수 있는지를 판단하는 과정
- 소프트웨어 제품의 고객, 사용자와 함께 테스트를 구축, 실행, 결과를 확인하는 것에 대한 조율작업
- 테스트 기법 선정과 테스트 수행 효과의 최대화에 대한 절충안 결정

사실 36: 프로그래머가 작성한 디버그 코드는 테스트 도구에 대한 중요 보완 수단이다.

✓ 디버깅 코드 삽입

프로그램에 별도의 디버깅 코드를 삽입
디버깅 과정은 프로그래밍의 추리 소설과 같다. 여러분은 미꾸라지 같은 소프트웨어 버그를 추적하는 Sherlock Homes역할이다.

프로그램에 코드를 추가하는 것은 비생산적으로 보일지도 모르지만, 이는 대부분의 경우 반드시 필요한 과정이다.

사실 37: 엄격한 검사는 첫 번째 테스트 케이스를 실행시키기도 전에 소프트웨어 제품에 포함된 오류의 90%까지 제거할 수 있다.

✓ 검사(inspection)의 효용성

어떤 것보다도 비약에 가까운 기법이 있는데 그것은 엄격한 검사이며 소프트웨어에 포함된 오류를 찾기 위해 노력을 쏟는 이 기법은 거의 비약이라 할 수 있다.

또한 같은 연구에서, 동일한 오류를 찾는 데 **검사에 드는 비용이 테스트에 드는 비용보다 적다**는 것을 증명해 보였다.

✓ 검사가 case도구나 방법론에 비해 찬사를 받지 못하는 이유

1. 검사로 돈을 버는 메이저 벤더는 거의 없다.
2. 검사에는 새로운 것도 없고 따라서 시장성도 없다.
3. 검사는 소프트웨어 생명주기의 뒤쪽 단계에 있는 보이지 않는 부분으로 간주된다.
4. 검사는 효율적이긴 하지만, 녹초가 될 정도로 정신을 집중해야 하는 고된 작업이다

사실 38: 엄격한 검사도 테스트를 대체할 수는 없다.

✓ 오류 제거단계에서 은탄환(silver bullet)

검사가 강력한 기법이긴 하지만, 오류 제거 작업은 검사만으로 충분하지 않다.

전산학자들은 오랫동안 정형 검증(formal verification)을 충분히 엄격히 수행하면 그것으로 충분하다.
결함 허용(fault-tolerance)을 옹호하는 사람들은 오류를 감지하고 스스로 복구하는 자체 검사 소프트웨어면 충분하다.
테스트 쪽 사람들은 100% 테스트 커버리지면 충분할 것이라고 말해왔다.

✓ 오류 제거단계에서 은탄환은 없다.

오류 제거를 위한 은탄환은 없다.
검사가 아무리 훌륭한 기법이라 하더라도, 그것만으로는 충분하지 않다.

사실 39: 출시 후 검토(회고라 부르는 사람들도 있다)는 중요하지만, 거의 실행되지 않는다.

✓ 프로젝트 현실

알파 테스트, 베타 테스트, 회귀 테스트 같은 방법을 이용해 오류 추적을 계속하고 있지만, 어떤 일이 있었는지 숙고하거나 앞으로 어떻게 개선할 수 있을지에 대해 기록하지는 않는다.
그 결과 소프트웨어 프로젝트에서 배운 모든 교훈은 버려지거나, 프로젝트가 끝날 때 바람과 함께 날아가 버린다.
왜 그럴까? 어제 프로젝트에서 일하던 프로그래머들이 이미 내일 프로젝트로 흩어져 배치돼 버리기 때문이다.

검토 혹은 회고 지지자들은 프로젝트가 끝난 후 1~3주 후에 검토할 것을 권고한다.

✓ 출시 후 검토(postdelivery review)는 무엇으로 구성될 수 있을까?

1. 사용자에 중점을 둔 검토로 사용자의 관점에서 제품에 어떤 일이 일어났는지를 논의하는 것
2. 개발자 중심의 검토로 개발자에게 과거로 돌아가 잘못했던 것, 더 좋게 할 수 있었던 것을 수정할 수 있게 기회를 주는 것

✓ 반복된 실수

소프트웨어 분야는 한자리에 처박혀 매 프로젝트마다 같은 실수를 반복하는 것 같다.

사실 40: 동료 검토(peer review)는 기술적 측면과 사회학적 측면을 모두 가지는데, 어느 쪽도 무시하면 안 된다.

✓ 검토의 엄격함

검토 과정에 참여하는 사람들이 검토에 전적으로 전념하고 집중하는 것은 매우 중요하다.
소프트웨어 제품의 복잡성 때문에, 검토자는 검토하는 동안 소프트웨어 개발 프로세스의 다른 어떤 단계에서보다도 더 집중해야 한다.

✓ 검토의 사회적 측면

대부분의 사회 활동에서 우리는 현재 이야기하는 주제에 일정 비율만 주의를 기울이고, 나머지는 대개 사회적 관계를 원만히 유지하는 데 집중한다.
그러나 소프트웨어 검토에서는 이것이 어려워진다.
엄격함을 달성하기 위한 집중 때문에 사회적 문제를 해결하기 위해 남겨둔 에너지까지 모두 소모돼 버린다.

✓ 검토의 다양한 정형적 접근방법

- 관리자가 검토에 참여하지 못하게 - 관리자는 제품이 아닌 제작자를 검토하려 한다.
- 준비되지 않은 사람은 참석하지 못하게 - 이들은 준비해온 참석자들을 짜증나게 하고, 주제의 탈선을 초래
- 검토 리더의 역할과 제작자의 역할을 분리 - 제작자의 자아가 관여하는 것을 줄이기 위해서

사실 41: 유지보수는 보통 소프트웨어 비용의 40~80%를 차지한다. 따라서, 유지보수는 소프트웨어 생명주기 중 가장 중요한 단계일 것이다.

✓ 소프트웨어에 문외한 사람에게 유지보수란?

첫 번째, 소프트웨어 분야의 '유지보수'란 단어가 무엇을 뜻하는지에 대한 문제
소프트웨어는 오류를 포함할 수 있다. 그리고 새로운 작업을 위해 수정될 수 있다. 이것이 소프트웨어의 소프트(soft)가 가지는 의미다.

두 번째, 소프트웨어 유지보수에 얼마나 많은 비용이 시간이 소모되는지에 대한 것이다.
평균적인 소프트웨어 구축에 드는 시간과 비용은 20~60%정도다. 그 외에 40~80%는 유지보수가 차지한다.
시간과 비용 관점에서 본다면, 유지보수는 소프트웨어들 생명주기 중에서 가장 지배적인 단계다.

사실 42: 유지보수 비용의 60%는 개선 작업에 소요되는 비용이다.

✓ 유지보수의 60%가 개선작업

소프트웨어 비용의 약 60%가 유지보수에 사용되고 유지보수 비용의 60%가 소프트웨어를 보다 유용하게 만들기 위한 수정작업에 든다는 것이다.
이는 개선이라 부르며 비즈니스적 필요에 따른 새로운 요구사항이다.

✓ 유지보수의 17%가 오류 수정

유지보수 비용의 17%만이 오류 수정에 쓰일 뿐이다.

60+17%=77%다.

18%는 환경이 변하더라도 소프트웨어가 계속 작동할 수 있도록 하는 적응성 유지보수에 들어간다.
나머지 5%는 기타 (예방적 유지보수, 리팩토링)

60/60 법칙: 소프트웨어 비용의 60%는 유지보수에 사용되며, 유지보수 비용의 60%는 개선에 사용된다.

사실 43: 유지보수는 문제가 아니라 해결책이다.

✓ 유지보수는 해결책

많은 사람들이 소프트웨어 유지보수를 감소시키거나 아예 제거해야 하는 '문제'로 본다.
소프트웨어 유지보수의 거의 모든 활동이 오류 수정에 대한 것이라면, 유지보수는 '문제'라 할 수 있겠지만, 이는 사실과 거리가 멀다.

유지보수는 '우리는 이런 것을 구축했지만 지금 보니 약간 다른 것을 구축했어야 한다.'와 같은 문제를 해결할 수 있는 유일한 방법이다.

사실 44: 유지보수에서 가장 어려운 작업은 기존 시스템을 이해하는 것이다.

✓ 소프트웨어 유지보수 생명주기

1. 문제에 대한 요구사항 분석, 수정 또는 개선작업을 한다.
2. 기존 시스템의 설계 컨텍스트 안에서 솔루션을 설계한다.
3. 솔루션을 코딩하고 기존 시스템에 맞춰 넣는다.
4. 코딩된 솔루션을 테스트 하면서, 새로 변경한 것이 제대로 동작하는지, 예전에 제대로 작동하던 것들에 영향을 끼쳐 문제를 일으키지는 않는지 확인한다.
5. 그리고 개선된 부분을 기존 시스템에 반영하고 계속 유지보수 한다.

✓ 기존 시스템을 이해하는 것이 가장 어렵다

기존 시스템을 이해하는 것은 소프트웨어 유지보수에서 가장 어려운 작업이라 한다.

왜 그럴까?

- 설계로 전환하는 시점에서 요구사항의 폭발적 증가(사실 26)
- 대부분의 문제에서 설계 솔루션이 하나만 있는 것은 아니라는 사실(사실 27)
- 설계는 복잡하고 반복적인 과정이라는 사실(사실 28)
- 문제의 복잡성이 25% 증가하면 솔루션의 복잡성은 100% 증가한다는 사실(사실 21)
- 이 모든 사실을 조합하면, 설계는 어렵고, 지적이며, 창조적인 작업이란 것을 알 수 있다(사실 22)

사실 45: 더 좋은 소프트웨어 공학 기술로 개발하면 더 많은(더 적은 게 아니라) 유지보수가 필요하다.

✓ 현대적 개발 방법

구조적 또는 프로세스 지향 소프트웨어 공학, 데이터 지향 정보 공학, 프로토타이핑, CASE 도구 등과 같은 전형적인 방법, 방법론, 도구, 기술을 모아놓은 것이다.

이런 접근방법을 이용한 시스템은 예전 방법보다 신뢰성이 높고 수리할 필요성이 적다.
그러나 이런 시스템은 더 오랫동안 유지보수 된다.

✓ 개선이 쉽기 때문에 더 많은 수정이 가해진다

이런 시스템은 더 많은 수정이 가해지기 때문에 다른 시스템보다 더 오래 유지된다.
그리고 이런 잘 구축된 시스템은 개선하기 쉽기 때문에 더 많은 수정이 가해진다.

1부. 사실

I. 관리

II. 생명주기

III. 품질

- 품질
- 신뢰성
- 효율

IV. 연구

3장. 품질

품질은 누구의 책임?

소프트웨어 품질에 대한 대부분의 책과 교육에서는 품질 보증이 관리 업무라고 주장한다.
그러나 품질을 구성하는 속성을 바탕으로 한 내정의를 본다면, 매우 기술적인 내용도 있음을 알 수 있을 것이다.

- 수정용이성 - 소프트웨어를 수비게 수정할 수 있도록 구축하는 방법을 아는 것이 중요하다.
- 신뢰성 – 오류를 포함할 가능성을 최소화하도록 소프트웨어를 구축하고 모든 의미 있는 오류 제거 방법을 동원해 오류 제거 프로세스를 적용해야 한다.
- 이식성 – 한 플랫폼에서 다은 플랫폼으로 쉽게 이식될 수 있도록 소프트웨어를 구축해야 한다.

위와 같이 품질은 소프트웨어 분야에서 가장 기술적인 문제 중 하나라고 단언하고 싶다.

품질은 측정할 수 없는가?

품질 자체의 의미가 모호할 뿐 아니라 품질을 구성하는 속성들 또한 그 의미가 모호하기 때문이다.
이해용이성, 수정용이성, 테스트용이성 및 다른 품질 요소들을 수치화 하는 것은 거의 불가능하다.

품질에 대해 무엇을 다룰 것인가?

- 무엇이 품질이고 무엇이 품질이 아닌지에 대해 올바른 설명을 시도할 것이다.
- 오류의 특성과 원인 같은, 신뢰성에 대한 몇 가지 측면을 간략히 살펴볼 것이다.
- 효율의 몇 가지 측면을 간략히 살펴볼 것이다.

사실 46: 품질은 속성의 집합이다.

✓ 속성들은 무엇인가?

이식성은 다른 플랫폼으로 쉽게 이식할 수 있도록 소프트웨어를 구축하는 것에 대한 것이다.
신뢰성은 소프트웨어 제품이 원래 의도대로 작업을 믿을 수 있게 처리하는가에 대한 것이다.
효율은 소프트웨어 제품이 실행 시간과 사용 공간에 있어 얼마나 효율적인지에 대한 것이다.
인간공학(human engineering, 사용 편의성)은 소프트웨어 제품이 얼마나 사용하기 쉽고 편리한지에 대한 것이다.
테스트 용이성은 소프트웨어 제품을 쉽게 테스트할 수 있는지에 대한 것이다.
이해 용이성은 유지보수하는 사람이 소프트웨어 제품을 이해하기 쉬운지에 대한 것이다.
수정 용이성은 유지보수하는 사람이 소프트웨어 제품을 수정하기 쉬운지에 대한 것이다.

✓ 프로젝트에 대한 일반적인 우선순위

1. 신뢰성 - 만약 제품이 제대로 동작하지 않는다면, 다른 속성은 아무 소용 없다.
2. 인간 공학 - 최근 GUI에 대한 강조는 사용 편의성의 중요성을 보여 준다.
3. 이해 용이성과 수정 용이성 - 소프트웨어로서 존재 가치가 있는 제품은 아마도 오랜 시간 동안 유지보수될 것이다.
4. 효율 - 효율의 우선 순위가 이렇게 낮다는 사실에 약간 당혹스럽다. 어떤 애플리케이션에서는 효율이 가장 중요할 것이다.
5. 테스트 용이성 - 신뢰성에 직접적인 영향을 주기 때문에 끝에서 두 번째에 위치했다고 해서 그 중요성이 줄어들지는 않는다.
6. 이식성 - 많은 경우 이식성은 문제가 되지 않지만 어떤 경우에는 이식성이 가장 중요해질 수도 있다.

사실 47: 품질은 사용자 만족, 요구사항 충족, 비용과 일정 목표 달성, 또는 신뢰성이 아니다.

✓ 사용자 만족에 대한 정의

사용자 만족 = 요구사항 충족 + 납기일 준수 + 적절한 비용 + 좋은 품질의 제품

✓ 품질의 정의?

많은 소프트웨어 전문가들은 소프트웨어 품질을 소프트웨어 제품 내의 오류 존재 여부와 동일시 한다.
사실 46에서 본 것처럼, 품질은 오류에 대한 것이기도 하지만, 훨씬 많은 다른 특성에 대한 것이기도 하다.

사실 48: 대부분의 프로그래머가 흔히 범하는 오류가 있다.

✓ 일반적인 오류

- 하나씩 밀린 인덱스
- 정의/참조의 불일치
- 중요한 설계 항목 누락
- 변수에 대한 초기화 실패
- 일련의 조건 중 하나의 조건 누락

사실 49: 오류는 뭉치는 경향이 있다.

✓ 오류는 뭉친다

- '오류의 반이 모듈의 15%에서 발견된다' - Davis(1995)
- '오류의 80%가 단지 모듈의 2% 이내에서 발견된다.' - Davis(1995)
- '대략 80%의 결함이 모듈의 20%에서 나오고, 모듈의 절반 정도는 오류가 없다.' = Boehm, Basili(2001)

✓ 오류는 왜 뭉칠까?

프로그램의 어떤 부분은 다른 부분보다 훨씬 더 복잡하기 때문에 그 복잡성이 오류를 초래하는 것일까? (나는 이렇게 믿는다.)
프로그램은 보통 여러 프로그래머가 나누어 작성할 수 있는데, 어떤 프로그래머가 다른 사람보다 더 많은 오류를 만들거나 포함된 오류를 적게 발견하기 때문에 그런 것일까? (개인차를 고려한다면 충분히 가능하다.)

➔ 어떤 프로그램 모듈에서 처음 기대했던 것보다 **많은 오류**를 발견하면, 계속 살펴보기 바란다. 바로 거기에 **더 많은 오류**가 숨어 있을 수 있다.

사실 50: 소프트웨어 오류 제거에 있다 단 하나의 최상의 방법은 없다.

✓ 오류 제거에 있어서는 은탄환(silver bullet)이 없다.

- 테스트의 특징이 어떻든 그것만 가지고는 충분하지 않다.
- 검사(inspection)와 검토의 정의가 어떻든, 역시 충분하지 않다.
- 정확성 검증도 충분하지 않다.
- 결함허용도 충분하지 않다.

사실 51: 오류는 항상 남아 있다. 심각한 오류를 제거하거나 최소화 하는 것이 목표가 돼야 한다.

✓ 오류는 항상 남아있다.

가장 엄격한 오류 제거 프로세스를 거쳤다 하더라도 소프트웨어는 항상 결함이 남아 있을 것이다.
목표는 그 수를, 특히 **심각한 오류의 수**를 최소화하는 것이다.

* 오류의 심각성 개념
심각한 오류는 소프트웨어에서 제거돼야 한다.
다른 오류도 모두 제거하면 좋겠지만(예를 들면 문서화 오류, 중복된 코드 오류, 도달할 수 없는 경로 오류, 알고리즘에서 무시할 수 있을 정도의 수치상 오류 등), 항상 그럴 필요는 없다.

사실 52: 효율은 훌륭한 코딩보다는 훌륭한 설계에 더 많은 영향을 받는다.

✓ 일반적인 소프트웨어 제품의 비효율

- 외부 I/O(느린 데이터 접근 같은)
- 너저분한 인터페이스(불필요한 또는 원격의 프로시저 호출 같은)
- 내부적 시간 소모(불필요한 루프 같은)

✓ 설계의 중요성

컴퓨터는 다른 어떤 곳보다도 외부 장치의 데이터에 접근할 때 느리다.
순차적 구조와 해시 데이터 구조와 같은 단순한 것이 있는데, 왜 연결 리스트(linked list), 인덱싱 같은 것을 고안했을까?
이것은 데이터 구조에 로직의 복잡성과 데이터 접근 효율 사이에 트레이드오프가 있음을 보여주는 것이다.

따라서 설계 시 가장 적절한 데이터 구조, 파일 구조, 또는 데이터베이스 접근방법을 선택하기 위해 많은 노력을 해야 한다.

인터페이스와 내부 비효율은 I/O 비효율에 비하면 그 중요성이 덜하다.

➔ 다시 말하지만, **교묘한 코딩**으로 효율적 솔루션을 만드는 것보다 설계할 때 **효율적 알고리즘**에 대해 조금 더 생각하는 것이 훨씬 효과적이다.

사실 53: 고급 언어 코드도 어셈블리어 코드의 90%에 가까운 효율을 낼 수 있다.

사실 54: 크기와 속도 사이에는 트레이드오프가 있다.

✓ 삼각함수

대부분의 삼각함수는 알고리즘으로 코딩돼 있고 그 코드는 매우 적은 공간을 차지하지만, 그 반복적 속성 상 실행하는 데 시간이 걸린다.
이에 대한 대안으로 프로그램에 삼각함수 값을 표에 저장하고 그 사이의 값을 구하는 데는 보간법(interpolation)을 사용할 수 있다. 보간법은 반복 보다는 훨씬 빠르지만 표가 차지하는 공간은 반복적 코드가 차지하는 공간보다 훨씬 클 것이다.

✓ Java

Java 코드는 머신 코드(machine code)로 컴파일 되지 않는다. 대신 바이트 코드(byte code)로 표현된다.
바이트 코드는 동일한 기능을 하는 머신 코드보다 훨씬 간결하고, 그 결과 java 프로그램 크기를 매우 적게 차지한다. 그러나 속도는 빠른가? 절대 아니다.
실행시점에 인터프리트 돼야 하고 실행시간이 100배나 느려지기도 한다.

1부. 사실

I. 관리

II. 생명주기

III. 품질

IV. 연구

- 연구

사실 55: 많은 연구자들이 연구보다는 옹호에 치중한다.

?

2부. 오해

V. 관리

- 관리
- 사람
- 도구와 기술
- 추정

VI. 생명주기

VII. 교육

오해 1: 측정할 수 없는 것은 관리할 수 없다.

✓ 소프트웨어 메트릭(metrics)

새로운 측정 요소에 대한 제안, 오랫동안 알고 있던 요소를 측정하는 새로운 방법
'얼마나 많이', '언제', 또는 '얼마나 잘'같은 질문에 대답을 알 필요가 있다.

메트릭의 가치와 비용에 대한 연구
➔ 전체 프로젝트 비용의 최대 3%(데이터 수집 및 분석) + 4~6%(데이터 처리 및 분석) = 7~9%

메트릭의 오점
별로 중요하지 않거나 얻는데 많은 비용이 드는 데이터를 지나치게 자주 수집했고 나중에 의미도 없었다.

✓ '측정할 수 없는 것은 관리할 수 없다'는 말은 틀리다.

우리는 늘 측정할 수 없는 것을 관리한다. 우리는 암 연구를 관리하고, 소프트웨어 설계를 관리한다.
훌륭한 지식 관리자는 정량적으로 측정하지 않고 정성적으로 측정하는 경향이 있다.

데이터를 가지고 관리하는 것은 데이터 없이 관리하는 것보다 훨씬 쉽고, 효율적이다.
사실, 뭔가를 이해하는 데 숫자의 도움을 활용하는 것은 관리자의 본성이자 인간의 본성이다. (야구의 타율, 방어율 등)
심지어는 피겨 스케이팅이나 다이빙과 같이 데이터화하기 어려운 것들도 데이터로 다루기 위한 방법을 만든다.

오해 2: 소프트웨어의 품질은 관리로 해결할 수 있다.

✓ 품질의 가장 중요한 적은 일정?

관리자들은 기술자들을 압박해 품질적 관점을 주입시키면 품질에만 관심을 가지게 될 것이라고 생각한다.
그러나 기술자들은 이런 접근방법을 멀리하는 경향이 있다.

경영자들은 한 손에는 동기부여와 방법론을 들고 다른 쪽 손에는 일정 압박이란 반 품질적 요소를 적용하려 한다.

✓ 품질이 관리 업무라는 것은 오해다

물론 품질을 달성하는 데 관리는 중요한 역할을 한다.
품질 달성에 높은 우선순위를 두는 문화를 만들 수도 있고, 기술자들의 품질향상 작업을 방해하는 요소를 제거해 줄 수도 있다.

오해 3: 프로그래밍은 비자아적이 될 수 있고, 또 되어야 한다.

✓ 비 자아적 프로그래밍은 어렵다

[The Psychology of Computer Programming] (Weinberg 1971)은 소프트웨어 공학 책으로는 최초의 베스트셀러다.
프로그래머는 그들이 만드는 제품에 자신의 자아(ego)를 투영해서는 안 된다고 했다.
너무도 많은 프로그래머들이 제품에 자신의 자아를 투영하여 객관성을 잃어버린다고 말했다.

자아적 프로그래머의 대안은? 팀 플레이어 프로그래머다.

그러나, 인간의 자아는 매우 자연스런 것이며, 자아와 자신의 일을 분리할 수 있는 사람을 찾는 것 또한 쉽지 않다는 것이다.

비자아적 관리자란 개념에 대해 생각해보자. 보통 관리자의 자아는 자신을 유능하게 만드는 추진력이 된다.

오해 4: 도구와 기술: 한 가지로 모든 문제를 해결할 수 있다.

✓ 소프트웨어 만병통치약

방법론을 파는 사람들, 프로세스적 접근방법을 정의하는 사람들, 도구와 기술을 선전하는 사람들, 컴포넌트 기반의 소프트웨어 구축을 바라는 사람들, 표준을 제정하는 사람들, 차세대 소프트웨어 공학이란 성배를 추구하며 연구하는 사람들, 자신의 작업이 뭐든 간에 그 앞에 메타란 접두사를 붙이는 학자들, 이런 사람들 모두 '**소프트웨어 만병통치약**'을 찾고 있는 것이다.

✓ 만병통치약은 없다

소프트웨어는 매우 다양한 문제를 다루기 때문에, 보편적인 접근방법은 거의 없다는 것이다.
비즈니스 애플리케이션에서 잘 적용되던 것도 실시간 소프트웨어 프로젝트에서는 충분하지 않을 것이다.
시스템 프로그래밍에서 잘 적용되던 것도 과학계산용 애플리케이션에서는 무의미할 것이다.

문제를 바라보는 관점(차원)

- **크기 문제**, 작은 것이 큰 것보다 훨씬 다루기 쉽다.
- **애플리케이션 도메인 문제**. 예를 들면 과학계산용 애플리케이션에서 필요한 것은 비즈니스나 시스템 프로그램에서는 별로 필요하지 않다.
- **중요도 문제**. 생명이나 막대한 양의 돈을 다루는 프로젝트라면 그렇지 않은 경우와는 완전히 다르게 처리할 것이다.
- **혁신성 문제**. 다루는 문제가 예전에 풀었던 것과 다르면, 그 솔루션을 만드는 데 있어 방법론화된 접근법보다는 탐험적 접근법을 많이 사용할 것이다.

오해 5: 소프트웨어 분야에는 더 많은 방법론이 필요하다.

✓ 많은 사람들이 방법론을 만들고 있다

구루(guru), 대학원생, 엄격한 방법론에 반대하는 사람들, 심지어는 교수와 연구자들까지도 방법론을 만들어내고 있다.
이런 방법론을 그대로 사용하는 실무자는 거의 없음을 밝혀냈다. (Hardy, Thompson, Edward 1995). 대신, 대부분의 사람들은 당면한 상황에 맞게 방법론을 변형시켜 적용했다.

✓ 그 모든 방법론이 정말로 필요한 것인가?

'그렇지 않다'라는 것이다.
Karl Wiegers는 더 많은 방법론이 만들어지는 것에 대해 강력히 반대했다.
왜 그렇게 말했을까? 아무도 지금 우리가 가지고 있는 방법론을 사용하지 않기 때문이라 했다.

✓ 그럼 방법론을 무시해야 할까?

너무 많은 방법론이 1) 무지의 소치고(대학원생이나 교수가 소프트웨어 실무의 복잡한 실상에 대해 뭘 알겠는가?), 2) 일종의 감시적 성향의 결과다(너무도 많은 구루가 자신의 방법론이 소프트웨어를 구축하는 데 정말 도움을 준다는 것을 증명할 수 있어서가 아니라, 단지 그것이 옳다는 생각만으로 다른 사람들이 자신의 방법론을 사용하기를 원하고 있다).

오해 6: 비용과 일정을 추정하기 위해서는 먼저 LOC를 추정해야 한다.

✓ 제품의 크기를 라인 수로 추정

우리는 LOC(Lines of Code)를 이용해 비용과 일정을 산정할 수 있다(LOC와 그 LOC의 제품을 만드는데 소요되는 비용, 일정에 대한 과거 데이터에 기초하여).

✓ LOC로 추정하는 것은 아무런 근거가 없다

LOC를 추정하는 것이 비용과 일정을 추정하는 것보다 더 쉽거나 믿을 만한 아무런 근거도 없기 때문이다.

COBOL 코드 한 라인과 C++코드 한 라인이 동일한 수준의 복잡성을 가질까?
수학적인 과학계산용 애플리케이션 한 라인을 비즈니스 시스템에서의 한 라인과 비교할 수 있을까?
초보 프로그래머의 코드 한 라인과 최고 프로그래머의 코드 한 라인이 동일할까? (최고 28배까지 개인의 능력치가 나타날 수 있다고 한 사실을 참조)
주석이 더덕더덕 붙어 있는 프로그램의 LOC와 주석이 없는 프로그램의 LOC를 비교할 수 있을까?

2부. 오해

V. 관리

VI. 생명주기

- 테스트
- 검토
- 유지보수

VII. 교육

오해 7: 랜덤 테스트 입력은 테스트를 최적화하는 좋은 방법이다.

✓ 랜덤 테스트의 장점

모든 테스트 케이스가 사용자의 실제 사용으로부터 유도된다면, 무작위적인 테스트 결과는 실제 사용을 모의 실험하는 데 사용될 수 있다. 통계적 테스트(랜덤 테스트)를 통해 '이 제품은 전체 시간의 99.7% 동안 성공적으로 동작했다'와 같이 말할 수 있는 것이다.

✓ 랜덤 테스트의 단점

- 도박적이라는 것이다.
  테스트가 정말 무작위라면, 어느 부분이 테스트되고 어느 부분이 테스트되지 않을지 전혀 알 수 없다.
  랜덤 테스트는 예외 처리 코드를 실행시킬 것이라 믿을만한 근거가 없다.
- 대부분의 프로그래머가 쉽게 실수하는 '편향적' 오류가 있고, 오류는 뭉치는 경향이 있다.
  대부분의 프로그래머는 솔루션의 어떤 부분이 구현하는데 어려웠고 어떤 부분에 테스트 노력을 집중해야 할지 알고 있다.
- 테스트를 반복하는데 문제가 있다는 것이다.
  회귀테스트와 같은 테스트 접근법에서는 고정된 집합의 테스트를 실행시키므로 같은 집합의 테스트를 반복 실행시켜야 할 필요가 있다.
  랜덤 테스트가 정말로 무작위라면, 동일 집합의 테스트를 여러 번 반복한다는 것은 말이 안 된다.

오해 8: '보는 눈이 많으면, 모든 버그는 그 깊이가 얕다'

✓ '충분히 많은 사람이 코드를 보면, 모든 오류가 발견될 것이다' 라는 뜻

이것을 오해로 분류한 이유와 말꼬리 잡는 것에서부터 중요한 것들이 있다.

- 말꼬리 잡는 이유: 어떤 오류의 깊고 얕음은 그것을 찾고자 하는 사람의 수와는 관계가 없다.
- 직접적인 이유: 검사(inspection)에 대한 연구에 따르면, 검사에 참가자의 수가 늘어남에 따라 발견되는 버그 수는 급격히 줄어든다.
- 극히 중요한 이유: 위의 구호가 옳다는 것을 증명하는 데이터가 없다.

✓ '충분히 많은 사람이 코드를 보면, 모든 오류가 발견될 것이다' 라는 뜻

말꼬리 잡는 이유. 어떤 버그는 다른 버그보다 얕고(덜 심각하고), 그 깊이(심각성)는 버그를 찾으려 하는 사람의 수가 얼마가 되든 변하지 않는다.

직접적인 이유. 도움이 되는 검사 참가자 수에는 상한이 있고, 이를 넘어서면 검사의 성과가 급격히 떨어진다(사실 37 참조). 그 수도 매우 제한적이다(2~4명 정도). 아무리 의욕적인 사람들이 떼로 있다 해도 다른 어떤 오류 제거 방법에서와 마찬가지로, 오류 없는 소프트웨어 제품을 생산할 거라고 생각해서는 안 된다.

극히 중요한 이유. 이번 오해의 배경 사상이 옳다는 증거가 없다. 오픈소스 열광자들이 다른 대안보다 훨씬 신뢰성이 높음을 증명하는 자료가 없다.

오해 9: 과거의 비용 데이터를 살펴봄으로써 미래의 유지보수 비용을 예측할 수 있고 시스템 교체 결정을 내릴 수 있다.

✓ 소프트웨어를 유지보수 하면서 자주 나오는 질문 두 가지

이 시스템을 유지하기 위해 얼마나 많은 비용이 들어갈 것인가?
지금이 이 시스템의 교체를 고려해야 하는 시점인가?

✓ 유지보수 비용 그래프는 욕조 모양이다 (Sullivan 1989)

시스템이 처음 구축될 때는 많은 새로운 문제를 발견하고 많은 오류를 발생하기 때문에 유지보수 작업이 많다. 그러나 시간이 지나면서 안정적이고 유지보수 작업이 적은 중간단계에 접어들어, 개선에 대한 관심은 낮아지고 버그는 적절한 통제하에 있게 된다. 그러나 어느 시점부터는 그 동안 누적된 변경으로 인해 처음 설계의 계층간 경계부분은 누더기가 되고, 간단한 변경도 예전보다 훨씬 어려워지는 욕조의 반대편 오르막 경사에 도달하게 된다. 유지보수 비용이 다 시 높아지는 이 시점에서, 간단히 변경하는 것조차도 많은 비용이 들고, 변경 작업 리스트가 늘어남에 따라, 사용자는 더 이상의 변경 요청을 중지하거나 또는 그들이 원하는 작업을 할 수 없게 되어 그냥 시스템의 사용을 중지해버린다. 유지보수 비용은 급격히 감소한다.

✓ 과거를 기반으로 미래의 유지보수 비용의 예측은 헛된 짓이다

1. 미래의 유지보수 비용을 예측하는 것은 극히 어렵다.
2. 시스템 교체 시점에 대한 의미 있는 결정을 내리는 것 또한 거의 불가능하다.

2부. 오해

V. 관리

VI. 생명주기

VII. 교육

- 테스트

오해 10: 프로그래밍을 가르칠 때 프로그램을 어떻게 작성하는지 보여주며 가르친다.

✓ 프로그래밍을 어떻게 배웠는가?

수업시간에 강사나 교수가 프로그래밍 언어의 규칙에 대해 설명하는 것을 듣고, 약간의 코드를 작성해보는 식으로 배웠을 것이다. 그리고 바로 코딩을 시작한다.

여기에 잘못된 것이 있다.
외국어를 배울 때 우리는 첫 번째로 하는 것은 읽는 방법을 배우는 것이다.

✓ 소프트웨어 분야에 읽기도 전에 작성하는 것에 대한 지적

1. 코드 읽기를 가르치려면 예제 코드를 설정해야 한다. 이런 표본을 찾는 것은 쉽지 않다.소프트웨어에서 [전쟁과 평화] 같은 작품의 코드는 아직 없다는 것을 인정해야 할 것 간다.
2. 코드 읽기를 가르치려면 이를 어떻게 하는지 알려줄 교재가 필요하다. 그러나 이런 교재는 없다. 그 이유 중 하나는 아무도 코드를 읽는 방법에 대한 책을 어떻게 써야 할지 모르기 때문이다.
3. 소프트웨어 커리큘럼 조차 읽는 방법을 가르치기 전에 작성하는 것을 가르친다.
4. 소프트웨어 분야에서 코드를 읽어야 하는 경우는 유지보수 때 뿐이다. 많은 사람들이 유지보수 활동을 싫어한다. 그 이유 중 하나는 코드를 읽기가 매우 어려운 작업이기 때문이다.