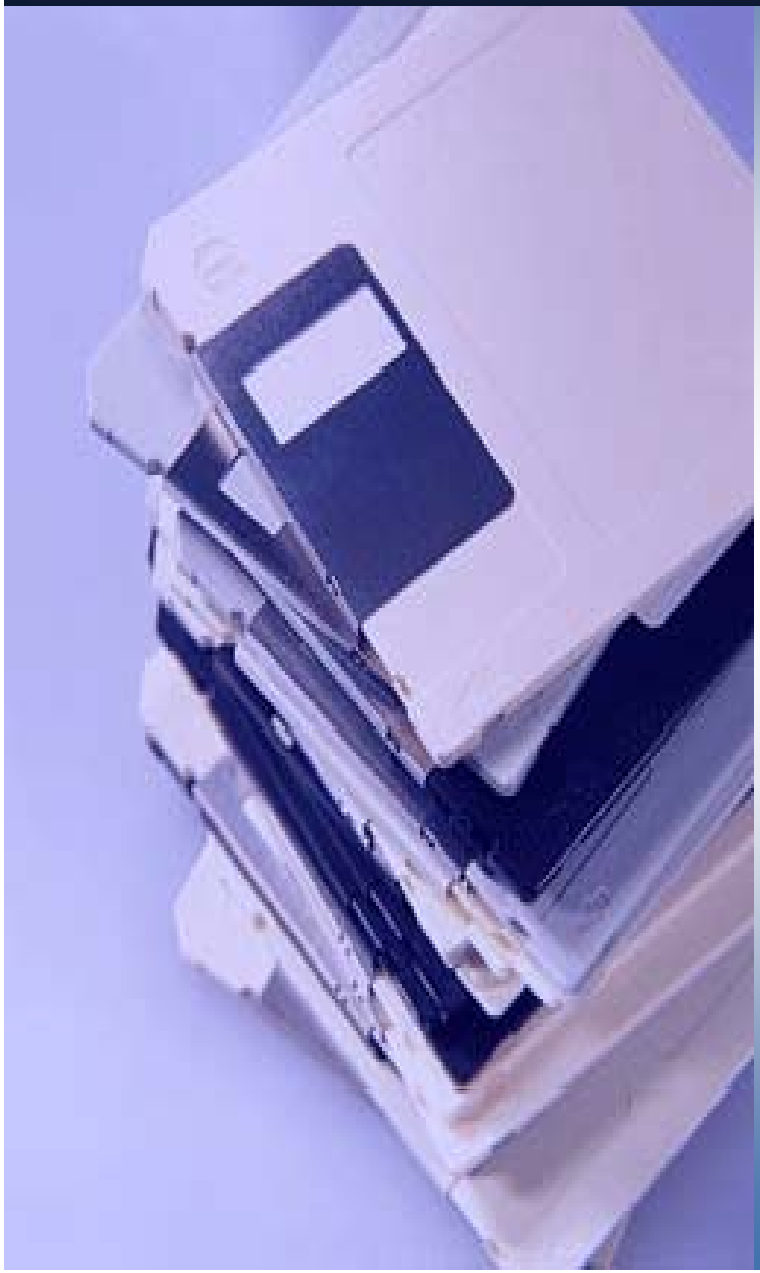
# Software's Chronic Crisis

by W. Wayt Gibbs

# Software's Chronic Crisis



- 소프트웨어의 위기
- 소프트 엔지니어링 배경
- 만성적인 소프트웨어의 위기
- 불안정성
- 고질적 문제
- 공학적 특성을 위한 시도
- 수학적 재 창조
- 만병통치약에 대한 기대
- SE 문제 및 해결책
- 결론

# The Denver Airport Case (1/2)

규모

- 21마일의 트랙
- 4000대의 수하물 운반 차
- 500개의 시각 장비, 400개의 무선 수신기, 56개의 바코드 스캐너

# The Denver Airport Case (2/2)

문제점

- 1년 이상의 일정 지연
- 적자에 대한 이자와 운영비만 하루 110만불
- 시스템 정상 가동이 언제 될지 모름

# Software's Crisis

연구에 따르면..

- 대규모 소프트웨어 시스템 가동 시 1/3은 취소
- 50% 정도는 일정이 지연
- 3/4 정도는 의도한대로 작동되지 않거나 시용되지 않음

>>소프트웨어의 위기

# Software Engineering

Software Engineering

1968년,

NATO과학위원회는 소프트웨어 위기로부터 탈출 할 방법 모색

명확한 방법을 찾지는 못했지만

그 목표를 염두에 두고 '소프트웨어 공학'이라는 명칭을 만듬.

# Software's Chronic Crisis

Software's Chronic Crisis

그 후 25년이 지나도
소프트웨어 공학이라는 말은 영감을 던져주는 용어일 뿐,
여전히 주먹구구식의 코드가 작성되어
계량화할 수도 없고 재사용 할 수 도 없음

# Shifting Sands (1/2)

소프트웨어의 불안정성

소프트웨어는 처음부터 잘 만들기는 어렵다.

소프트웨어의 신뢰성을 보장하기 위하여 테스트를 엄격히 해야 한다.

- 실시간 시스템의 경우 에러 검출이 매우 힘들다.
  발생할 만한 상황이 되어야만 발생하기 때문이다.

# Shifting Sands (2/2)

- 네트워크 상 컴퓨터와 통신해야 하는 분산 시스템의 경우 소프트웨어 개발에 일관성이 없기 때문에 소프트웨어는 더 복잡해 진다.

소프트웨어는 수정이 가능하긴 하지만 복잡하고 까다롭다.

# Mayday, Mayday (1/2)

소프트웨어의 고질적 문제점

- 시스템이 복잡해서 관리자전체를 이해할 수 없을 때
  전통적인 개발 방식은 무력해진다.

- 프로그래밍이 과학과 수학의 견고한 바탕 위에
  공학적인 특성을 가지지 않는다면
  재앙이 점점 늘어날 것이다.

# Mayday, Mayday (1/2)

- 공학적 특성을 가지기 위한 시도는 이미 시작되었다.

업계 선도자들이 일관성 있고 계량적인 측정을 어떻게 할지에 대해,
개발 과정의 무질서에 대해, 상품에 포함된 밀도 높은 오류에 대해,
프로그래머의 생산성이 저하되는 것에 대해 이해하는데
큰 진전을 보이고 있다.

- 이제는 이런 문제들에 대한 실용적이고
  재사용 가능한 해결책을 찾아야 한다.

# Proceeds of Process

- 1991년 SEI 소프트웨어 개발조직이 높은 품질의 소프트웨어를 일관성 있고 예측 가능하게 생산하는 능력을 갖추게 하기 위해 SW Process model CMM개발 하였다.

- CMM은 많은 개발자들을 설득해서 공학적 원칙 적용의 선수조건이 되는 계량화 작업을 하는데 집중하도록 했다.

Ex. Raytheon의 장비부서가 CMM테스트 후, '소프트웨어 공학 개발팀'조직

# Mathmatical Re-creation (1/2)

수학적인 재 창조

- 최고의 디자인도 잘못될 수 있다.
- 에러는 있기 마련이다.
- 버그가 초기에 발견되면 좋지만,
  치명적인 버그일수록 초기에 발견되지 않고 마지막에 발견된다.

엔지니어는 수학적 분석에 기초해서
디자인이 실 세계에서 어떻게 작동할 지 예측해야 한다.

# Mathmatical Re-creation (2/2)

- 스펙의 만족함을 보여줄 뿐
  실 세계의 변화무쌍함을 감당할 수 없고,

- 증명과정에서 실수가 가능하므로 테스트가 필요하다.

※청정접근법

정규적 기법들을 점증적으로 소프트웨어 개발에 융화하는 것으로
기능을 하나씩 더할 때마다
부품의 품질아 완전한가를 확인해야 한다.

# No Silver Bullet

## 만병통치약에 대한 기대

- 60년에는 객체지향 가들은 자신들의 접근법이
  패러다임의 전환이라고 주장 하였다.
- 70년대에는 구조적 기법을 패러다임의 전환으로 여겼다.
- 3세대, 4세대, 5세대 언어들도 기술에 대한 기대는 컸지만
  이렇다 할 결과를 내지 못하였다.

그러는 동안 소프트웨어 공학은 다른 영역에 비해,
특히 하드웨어 공학에 비해 뒤처졌다

일반적이고 우연적인 것을 구분하기 위한 실험적인 분석 결과가 필요하다.
검증된 방법을 기록으로 남겨
그것에 따라 문제를 해결하고 진도된 프로젝트를 만들 수 있게 해야 한다.

# Just Add Water (1/2)

문제점

소프트웨어 부품이 표준화되면서 다양한 용도로 재사용이 가능함
재사용을 위해 라이브러리를 사용해 왔지만,
사용언어에 따라, 플랫폼에 따라, 운영체제에 따라 호환이 어렵다.

현재의 시스템에서는 소프트웨어의 특정부문에 굉장한 능력을 가진
사람이라 할지라도 그 능력을 제대로 발휘하기 어렵다.

# Just Add Water (2/2)

해결책

소프트웨어의 각 부문을 컴포넌트 화 하여서
필요에 따라 각 컴포넌트에 대한 설명서를 제공함으로써
이용하는 사람이 결합하여 사용 가능한 시스템을 만들어야 하며,
이에 따라서 이익구조 창출을 위해
각 컴포넌트 사용에 따라 요금을 부과하는 방식이 필요하며
이를 활용하는 사람의 프로그래밍 능력을 키울 수 있는
인증절차를 도입해야 한다.

# Conclusion (1/2)

소프트웨어 위기로부터 벗어날 가망이 전혀 없는 것은 아니다.

- 계량적인 분석법이 사용되고 있다.
- 수학적 방법을 활용해 치명적 실수를 피할 수 있다.
- 학교에서는 가례 연구를 통한 전문적 교육을 하고 있다.
- 재사용 가능한 소프트웨어 부품을 만드는데 관심을 기울인다.

# Conclusion (2/2)

엔지니어는 저절로 만들어 지지 않는다.

Software Engineering에 대한 전문적인 교육이 필요하다.

# End

Thank you !