7장 실험주의: 과학적 사고와 실천

문제 해결을 위해 실험적인 접근 방식의 선택은 매우 중요하다.

하지만 대부분의 소프트웨어 개발은 아직도 '추측'에 의존한다

- 사용자가 무엇을 좋아할지 **추측**
- 어떤 설계/기술이 맞는지 **추측**
- 코드가 잘 작동할지 **추측**
- 제품이 돈을 벌지 **추측**

→ 이런 추측은 빠르게 잘못된 방향으로 갈 위험이 큼.

실험은 느리고 복잡한 게 아니다—오히려 더 빠르고 효율적이다

실험적 접근은:

- 더 빨리 검증하고,
- 더 빠르게 실패하고,
- 더 낮은 비용으로,
- 더 높은 품질의 제품을 만들 수 있게 해줌.

→ "더 열심히 일하는 것"이 아니라 **"더 현명하게 일하는 방식"**.

그렇다면 실험적인 접근방식으로는 ?

- 문제 해결 시 추측을 사실처럼 다루지 않기
- 가능할 때마다 실험(plan → test → measure → learn)을 설계하기

물리학자 파이먼에게 배우는 '실험주의'

리처드 파인먼의 명언

1. 과학은 전문가들의 무지에 대한 믿음이다.
2. 뭐가 되었든 권위를 존중하지 말고, 누가 말했는지 잊어버리고 그대신 그가 무엇으로 시작해서 어디서 끝나는지 살펴보고 "그것이 합리적인가?"라고 자문해보자

즉 증거에 기반한 의사결정과 선택을 해야만 한다.

ex)

C#보다 클로저Clojure가 더 낫다는 주장을 하고 싶다면, 간단히 실험해서 실험 결과의 안정성과 처리량을 측정해 보면 어떨까? 그러면 누가 더 논쟁 과정에서 설득력이 있느냐에 따라 이런 결정을 내리는 대신, 완벽하지는 않더라도 어느 정도 근거에 기반해서 결정할 수 있다. 그 결과에 동의하지 않는다면 더 나은 실험을 통해 근거를 제시하자.

실험주의자가 되는 접근방식 4가지를 서술한다.

피드백, 가설, 측정, 변수 통제

1. 피드백: 실험주의를 위한 원칙 1

사례

> 한 금융 소프트웨어 회사는 자동화 테스트에 9시간 30분이 걸려 매일 아침 테스트 실패 모듈을 골라 릴리스해야 할 정도로 비효율적이었다. 대부분의 테스트가 매일 실패했고, 3년 동안 모든 테스트가 통과된 적은 단 3번뿐이었다.
>
> 문제 해결의 첫 단계로, 큰 변경 없이 피드백 속도 개선이라는 명확한 목표를 세우고 다양한 실험을 진행했다. 그 결과, 커밋 빌드를 12분, 나머지 테스트를 40분에 실행하도록 개선하는 데 성공했다.
>
> 즉, 실험적 접근을 통해 9시간 30분짜리 빌드를 52분으로 단축하며 개발 효율과 품질을 대폭 향상시킨 사례이다.

1. **문제를 '추측'이 아니라 '관찰된 데이터'로 정의**
2. 하나의 목표에대해 다양한 실험을 반복
3. 실험결과가 명확하고 긍정적 → 피드백

2. 가설: 실험주의를 위한 원칙 2

추측이나 가설은 출발점이다.

과학과 공학은 출발선에서 멈추지 않는다는 사실,

가설의 형태로 추측을 하고 몇 가지 예측 → 예측을 확인할 방법을 찾기 → 추측과 다르다면 틀린것이다.

즉 우리가 하는 일이 추측이 아니라 공학이라고 주장하려면 가설을 검증하기위한 일련의 실험을 진행해야한다.

3. 측정: 실험주의를 위한 원칙 3

측정은 매우 중요하며, 잘못 측정하면 스스로 속게 된다

리처드 파인먼의 유명한 말처럼,

과학적 방법에서 가장 위험한 것은 자기 자신을 속이는 것이다.

데이터를 모을 때는 "이게 정말 의미 있는 측정인가?"를 비판적으로 생각해야 한다.

잘못된 지표는 잘못된 행동을 만든다

첫 번째 사례:

- 한 회사가 테스트 커버리지(코드를 테스트가 얼마나 덮는지)를 늘리면 품질이 좋아질 거라 생각했다.
- 그래서 목표를 "테스트 커버리지 80%"로 잡고, 성과급까지 걸었다.
- 개발팀은 목표를 달성했지만...

테스트의 25%가 assert(검증)이 하나도 없는 빈 껍데기 테스트였다.

즉, 테스트는 많아졌지만 품질은 전혀 개선되지 않았다.

측정 지표가 잘못되자, 사람들은 그 지표를 '달성하기 쉬운 방식'으로 시스템을 해킹한 것.

정작 측정해야 할 건 *테스트 양*이 아니라 코드 품질(안정성)이었다.

평균값 같은 부정확한 측정은 현실을 왜곡한다

두 번째 사례:

- 저지연 금융 시스템에서 성능 목표를 "평균 지연 2ms, 처리량 초당 10만 메시지"로 잡았다.
- 하지만 실제 거래 환경에서는 **이상치(peak)**가 발생해 초당 수백만 메시지가 들어오는 순간이 있었다.
- 평균값은 이런 위험한 순간을 전혀 반영하지 못했다.

평균이 아니라 *최대 지연*, *지연 분포*, *특정 한도 초과 비율* 등을 봐야 하는 것이 맞았다.

실험주의자의 핵심은 학습과 측정의 정교화

처음엔 누구나 잘못된 지표를 선택할 수 있다.

하지만 중요한 건:

- 실험 → 측정 → 문제 발견 → 지표 개선 → 더 나은 실험

이 반복을 통해 지속적으로 배우고 개선하는 것이다.

4. 변수 통제: 실험주의를 위한 원칙 4

버전 관리·자동화 테스트·배포 자동화 같은 기술은 소프트웨어 품질을 안정적으로 통제하게 해주고, 지속적인 배포는 이를 기반으로 작은 변경을 자주 릴리스하여 빠르게 피드백을 얻는 방식이다.

즉, 기술적 불확실성을 줄여서 잘 만들고 있는가? 문제를 해결하고,
그 덕분에 올바른 것을 만들고 있는가?에 집중하며 빠르게 학습할 수 있게 해주는 개발 방식

TDD에서 배우는 자동화 테스트

TDD : 테스트를 시스템 동작에 대한 실행 가능한 명세로 사용하는 전략

소프트웨어는 자동화된 테스트를 통해 계속 실험할 수 있는 최고의 환경을 제공한다.

하지만 코드를 먼저 만들고 나중에 테스트하면 실험(가설 검증) 가치가 떨어진다.

- TDD는 이 상황에서 이렇게 하면 이런 결과가 나와야 한다는 가설을 먼저 세우고,
  그 가설을 테스트로 표현한 뒤 코드를 작성해 검증하는 방식이다.
- BDD·ATDD처럼 높은 수준의 사용자 행동 명세에서 출발해, 더 세세한 단위 테스트로 이어질 수 있다.

TDD로 개발된 소프트웨어는 전통 방식보다 버그가 훨씬 적으며, (연구에서 40~250% 감소) 버그가 줄어든 만큼 개발팀은 버그 처리에 시간을 적게 쓰고 실질적 가치 작업에 더 많은 시간을 투자하게 된다.

그래서 TDD + CI/CD를 사용하는 팀은 더 높은 품질과 더 높은 생산성이라는 두 마리 토끼를 잡을 수 있다.

테스트는 새로운 지식을 끌어내는 원천

소프트웨어 개발은 과학 실험과 같다.
우리는 테스트를 통해 '가설 → 실험 → 검증' 과정을 빠르고 저렴하게 반복할 수 있고,
소프트웨어가 돌아가는 환경(작은 우주)까지 완전히 통제할 수 있다.
테스트 전체는 이 우주에 대한 우리의 지식 체계이며, 새로운 기능은 새로운 실험(테스트)로 추가된다.

품질을 높이는 TDD 적용 사례 하나

TDD에서 테스트를 먼저 만들고 그 테스트가 어떻게 실패할지까지 예상하는 과정 자체가 '작은 과학 실험'이다.

가설(실패 예상) → 예측(어떤 오류가 날지) → 실험(테스트 실행)의 절차를 반복함으로써, 개발은 더 체계적이고 품질이 높아진다.

이 실험적 접근은 어렵지 않지만, 꾸준히 적용하는 것이 중요하다.

정리

실험적으로 개발하려면 변수를 최대한 통제해야 한다. 자동화 테스트, IaC, 지속적 배포 같은 기술이 이러한 통제를 가능하게 해 주며, 이를 통해 실험은 더 신뢰할 수 있고 반복 가능해진다. 그 결과 소프트웨어는 더 결정론적이고 예측 가능하며 품질도 높아진다. 즉, 개발을 작은 실험들의 흐름으로 조직하면 같은 노력으로 훨씬 더 나은 소프트웨어를 만들 수 있다.