5장 우수한 의사결정을 위한 필수 요소

피드백이 없다면 학습할 기회도 없다. 현실에 기반한 결정을 내리는 대신 추측만 가능하기때문이다.
피드백은 의사 결정의 위한 근거의 원천을 확보하게 해준다.

이런 근거의 원천을 확보하면, 의사결정의 품질은 필연적으로 향상된다.
아래의 사례로 피드백의 중요성을 살펴본다.

피드백의 중요성을 보여주는 구체적인 사례

> 빗자루의 균형을 잡는 문제에 직면

1. 빗자루의 균형점을 미리 완벽히 계산후 계획적으로 조정 (폭포수 모델)
- 모든변수를 예측해야 하며, 작은 오차나 변화에도 쉽게 실패한다.
- 계획은 정교하지만 현실에서는 불안정하고 비효율적이다.

2. 빗자루를 실제로 손에 쥐고 기울어짐에 따라 즉각적으로 손을 움직여 조정하는 방식 (애자일 방식)
- 빠르고 정확한 피드백이 핵심이며, 환경 변화나 예기치 않은 상황에도 유연하게 대응 가능하다.
- 겉보기엔 즉흥적이지만 훨씬 더 안정적이고 실용적이다.

이를 보아 피드백은 변화하는 환경에서 운영하는 모든 시스템의 필수 구성 요소다.
고속 고품질 피드백에 대한 요구가 우리의 업무방식에 어떤 영향을 미칠까?
아래의 내용으로 본다.

> 코딩 피드백, 통합 과정 피드백, 설계 피드백, 아키텍처 피드백,

코딩 피드백

[저자의 방식]
새로운 동작 방식 몇가지를 추가 →
먼저 테스트 부터 작성해 내가 만든 테스트가 올바른지 체크 →
코드를 변경할때마다 현재 작업중인 테스트를 다시 실행함으로써 두번째 수준의 피드백을얻음 →
이런 피드백을 통해 변경 이후에도 코드가 계속 동작한다는 사실을 빠르게 확인

이렇게한다면 작업을 일련의 작은단계로 구조화하여 진행상황 반영 및 설계를 더 나은 결과물로 조정하게된다.

통합 과정 피드백

[저자의 방식]

코드 커밋시 지속적인 통합 시스템을 가동해 변경 사안 평가 → 새로운 피드백, 깊은이해

테스트 통과 → 안전하다는 피드백

지속적인 통합 (CI) 와 기능브랜치 (FB) 의 장단점을 구분해보면서 개발현장에서 자주 비교되고 혼동되는 점을 파악해보자

FB는 현실에서 인기 많지만, CI를 방해한다.

- 실제로 많은 팀들이 기능별로 브랜치를 따서 개발(FB) → 기능 완성 후에 한 번에 병합합니다.
- 이 방식이 “깔끔하고 독립적”처럼 느껴지지만,
  CI의 핵심 원칙(“작게, 자주, 빨리 통합”)과 **정반대**예요.
- 결과적으로 FB 위주의 팀은 CI를 “하고 있다”고 착각하지만,
  실제로는 **지속적인 통합이 아니라 지연된 통합**을 하고 있는 셈이죠.

그렇기때문에 지속적인 통합과 지속적 배포는 작은 단계로 변경하고 모든 작은 단계 이후에 사용할 준비를 갖추는게 품질과 작용 적용 가능성에 대해 더 자주 세분화된 피드백을 얻을수있을것이다.

설계 피드백

TDD를 높이 평가하는 이유는 설계 품질에 대한 피드백이기 때문이다.

- 모듈성
- 관심사의 분리
- 높은 응집성
- 정보 은닉(추상화)
- 적절한 결합도

TDD의 정의에 따르면 테스트를 먼저 작성한다. 테스트를 먼저 작성하지 않으면 테스트 주도 개발이 아니다.

1. 테스크코드 작성
2. 자신의 삶을 더욱 어렵게 만들기 위해 이상하고 바보같은 사람이 되어야만 한다.

3. 삶을 더 쉽게 만드는 방법으로 이를 시도

→ 인터페이스 설계도 하게됨, 상호작용하기 위한 방식 정의

TDD는 객관적으로 '더 높은 품질'의 코드를 작성하도록 압력을 가한다.

"나쁜 소프트웨어 개발자"를 더 좋게 만듬

"훌륭한 소프트웨어 개발자"를 더 훌륭하게 만든다.

아키텍처 피드백

피드백 주도 접근 방식을 적용하면 구축하는 시스템의 광범위한 소프트웨어 아키텍처에도 영향을 미친다.

한 시간에 한 번 릴리스 가능한 소프트웨어를 만들어야 한다면, 이는 수십만 개에 이르는 테스트를 수행할 수 있어야 함

이정도 수준을 달성하려면 팀의 노력과 집중, 지속적인 배포라는 아이디어에 대한 개발 조직의 헌신이 필요하지만, 종종 아키텍처적인 사고방식을 요구하기도한다.

구분	**모놀리스(Monolith)**	**마이크로서비스(Microservices)**
구조 특징	하나의 큰 애플리케이션 안에 모든 기능이 포함됨	여러 개의 독립된 서비스로 분리됨
개발 방식	하나의 코드베이스에서 여러 팀이 함께 작업	각 팀이 자신만의 서비스(코드베이스) 개발
배포 단위	전체 시스템을 한 번에 배포	각 서비스 단위로 개별 배포 가능
테스트 방식	전체 통합 테스트 용이	서비스별 테스트 가능하지만 통합 테스트 복잡
장점	- 통합 테스트 간단- 배포 과정 단순- 데이터 일관성 관리 용이	- 팀 독립성 높음- 빠른 배포 가능- 서비스별 기술 선택 자유
단점	- 변경 영향 범위 큼- 대규모 팀 협업 시 충돌 위험- 배포 속도 느림	- 시스템 복잡도 증가- 서비스 간 통신·데이터 일관성 문제- 통합 테스트 어려움
CI(지속적 통합)	변경사항이 많으므로 **자주 병합**해 통합 지옥을 방지해야 함	서비스 간 통합은 적지만, **계약 기반 테스트** 필요
CD(지속적 배포)	가능하지만 전체 시스템을 항상 배포 가능 상태로 유지해야 함	서비스별로 독립적 배포 가능 (자동화 필수)
아키텍처 요구사항	모듈화된 구조, 자동화된 테스트 필수	서비스 간 인터페이스 설계, API 계약 관리 필수
복잡성의 형태	코드 내부 복잡성 ↑	시스템 외부(운영/통합) 복잡성 ↑
피드백 루프	통합 테스트로 피드백 받기 쉬움	각 서비스 단위 피드백은 빠르지만 전체 시스템 피드백은 느림
지속적 배포에 적합성	비교적 단순하지만 대규모에서는 어려움	이상적 구조이지만 관리 복잡도 큼
핵심 성공 요인	좋은 설계 + 자주 통합(CI)	강력한 자동화 + 서비스 간 테스트 체계

이 두 아키텍처에서도 지속적인 배포를 채택하면 더 모듈화되고, 더 추상화되며 , 더 느슨하게 결합된 설계를 촉진한다.

피드백은 빠를수록 좋다

가능한 한 초기에 확실한 피드백을 받으려고 노력하는 편이 효과적이다.

먼저 컴파일러 수준에서 결함을 식별한 다음 단위 테스트에서 식별하고, 두가지 형태의 검증이 성공한 다음에 다른 형태의 상위 수준 테스트에서 결함을 식별하는 방식을 선호

제품 설계 피드백

제품 아이디어 창출에 대한 피드백 루프를 닫고 상용 서비스에 가치를 제공하는 것이 바로 지속적인 배포의 진정한 가치다.

예)

시스템에 원격 측정 기능을 추가해 시스템의 어떤 기능이 어떻게 사용되는지에 대해 데이터를 수집하는 방식

비즈니스 + IT의 디지털 비즈니스 제공

→ 고객스스로가 의식하지 못하는 자신의 욕구, 필요, 행동에 대한 통찰을 제공

조직과 문화 피드백

그렇다면 조직은 어떻게 성공 척도 없이 유용한 피드백을 확보할 수 있을까?

1. 업무에 참여하는 사람들을 피드백 루프에 참여시킴으로서 자신의 행동 결과를 관찰하고, 반서앟며, 시간이 지남에 따라 상황을 개선하기 위해 선택을 구체화 할 수 있도록 만들었다.
2. 단계를 수행하면서 목표에 더 가까워졌는지 아니면 더 멀어졌는지를 확인하고 목표에 도달할 때까지 반복한다.

안정성과 처리량 수치가 좋으면 기술 배포가 잘되고 있는 것이다.

정리

피드백은 우리의 학습 능력에 필수적인 요소다.

피드백의 속도와 품질이 모두 중요한만큼, 지속적인 배포와 지속적인 통합은 둘 다 근본적으로 개발 프로세스를 최적화해 수집하는 피드백의 품질과 속도를 극대화한다는 아이디어에 기반을 둔다.