📖 2025-W08

2025-W08

카카오 코칭

시작하며

jobs

how when what

lincorn

실무에서 TDD, OOP, DDD를 어느수준까지 잘할 수 있는지 궁금합니다. 세가지 다 이상과 현실의 괴리 때문에 쉽게 못하는 경우가 많았던 거 같아서, 기억에 남는 best practice 같은게 있을지 궁금하네요~

- 주어진 시간 내에 동작하도록 하는 것이 가장 중요
  - 1시간짜리 시험을 본다면 어떻게 해야 하나 ?
  - 100점을 맞자고 1번부터 답을 모든 문제를 공들여 푸나 ?
  - 100점을 확신한다면 그렇게 해도 되지만 그렇지 않다면 정해진 시간에 최대한 많은 문제를 푸는게 맞다.
  - 일단 주어진 시간 내에 요구되는 기능을 동작하도록 만들고 남은 시간에 잘하려고 노력하는 게 맞다.
  - 얼마나 잘할 수 있나가 자신의 실력인 것이다.
  - 아주 잘해먄 기능을 구현하고 배포할 수 있다면 우리팀 막내는 언제 배포하여 서비스에 기여하나 ?
- API, SDK 등을 처음 사용할 때 메뉴얼을 다 읽고 사용하나 ? 아님 예제를 먼저 보고 사용하나 ?
  - 사실은 이게 TDD와 유사하다고 생각함
  - 테스트 코드에서 예제를 돌리듯 구현하고 리팩터링을 통해서 모델 코드로 추출해 나가는 것이다.
  - 예전에는 이렇게 구현하기가 어려웠지만 지금은 좋은 툴들이 많아서 이렇게 개발하기 수월해 졌다.
  - 테스트에 절차적으로 구현하고 테스트를 안전망 삼아서 리팩터링을 통해 모델 코드로 추출해 가면서 OOP, DDD를 적용해 나가는 것이다.

TDD, OOP, DDD 관련해서 각각 어떤 생각을 가지고 계신지 궁금합니다.(ex. 적용하기 힘들지만 해볼 만 하다. 적용하기 위해서 거쳐야 하는 단계 등)

- *설계를 하나, 안하나*의 문제가 아니다.
  - 언제 하냐의 문제다.
- 리팩터링을 통해서 필요한만큼 지속적으로 설계를 해 나가는 것이다.
- 그러다 보면 OOP, DDD 원칙을 준수할 수 있게되어 *향후 변경 비용*을 낮출 수 있는 고품질의 코드를 만들 수 있다.
  - *비직관적인 품질과 비용의 관계*에 주의해야 한다.
- 이제 *MVP 수준의 코드는 AI*를 활용해서 비개발자도 만들 수 있는 세상이다.
- 개발자인 우리의 *차별점*은 무엇인가 ?
- 향후 변경 가능성을 높이는 품질이 필요하다고 생각한다.
- 수파리([[tdd-by-example#TDD 수련법]])
  - 아는 것, 할 수 있는 것, 하는 것이 차이를 알아야 한다.
  - 1만 시간의 법칙 = `4시간/일 * 5일/주 * 50주/년 * 10년 = 10,000시간`

신규 마이크로서비스에 Hexagonal architecture 를 시범적으로 적용했다가 overkill 이라는 느낌이 들었고, 협업하던 동료들도 비슷한 생각이어서 걷어낸 적이 있습니다. 실무에서 hexagonal architecture 로 득을 볼 수 있는 케이스는 어떤 상황일지 궁금합니다. (심플한 MSA 보다 거대한 monolith legacy app 이 hexagonal architecture 를 적용하기 더 적절한 것인지)

- hexagonal에서 불필요한 인터페이스가 있다 ?
- read 기능은 hexagonal을 적용하기에 너무 오버스럽다.
- VSA로 시작해서
  - Application Service, Domain Service로 분리해 나가는 것이 좋다.
  - 여기서 문제는 리팩터링 역량이 있어야 한다는 것
  - 그렇지 않으면 BBoM이 됨
- 하지만
  - 참여하는 구성원의 수가 많고,
  - 인원의 교체가 많고,
  - 시스템이 복잡하다면 hexagonal은 지도를 제공할 수 있다.

촉박한 일정하에서 개발을 진행할 때에도 test-first 전략을 지향하는 것이 현명한 것인지 궁금합니다. 현재는 메인 로직 구현 후 unit

test or integration test 코드를 작성하거나, 테스트 없이 코드 작성 후 버그가 발견되었을 때 regression 용도나 리팩토링시 안전망 용도로 테스트를 추가하는 경우가 더 많은 것 같습니다

- 비용이 언제 많이 발생하나 ?
- 라이프 사이클의 후반부에서의 수정 비용은 전반부에 비해 많게는 100배 이상이라고 한다.
- test first는 어쩌면 초기엔 비용이 든다고 생각할 수 있지만 후반부의 비용을 압도적으로 줄여줄 수 있는 무기일지도 모른다.
- 테스트만 있다면 공격적으로 리팩터링을 하거나 기능을 수정하는 것이 두렵지 않고, 안전할 수 있다.

우리나라 IT 업계에서 백발의 개발자가 늘어날 것으로 보이시는지, 그리고 그렇게 되기 위해 어떤 커리어를 밟아야 할지 궁금합니다

- 업무 수행을 통해서 서비스와 회사에 기여하고 지속적으로 배워나간다면 나이와 무관하게 대우받으며 개발자로 살아갈 수 있다고 생각합니다.

> 성장은 직접적인 목표가 아닌 생존을 위한 노력의 결과물입니다, *현장에서의 경험이 가장 큰 배움의 기회*를 제공합니다. - 박웅현교수님(https://www.youtube.com/watch?v=kPyCgEnM8SQ)

제안하는 개발방식

- TDD에서 말하는 단위 테스트에 대한 오해
  - 고객에게 가치를 제공하는 기능에 대한 테스트여야 함
  - Blackbox test임. 구현의 구조에 커플링된 Whitebox test면 안됨(리팩터링 내성. Mock Roles not objects)
  - Use Case, Port In 에 대해서 테스트를 작성하라 - Sociable Test
  - 구현, 리팩터링을 위해 Solitary Test를 작성할 수도 있지만 나중에 지우는 것을 고려해라
- VSA로 시작해서 split by abstraction layer, split unrelated complexity
  - 리팩터링으로 새로운 메소드, 클래스가 생겨도 테스트 추가 말아야.
  - 리팩터링은 외부 행위의 변경 없이 구조만 변경하는 것임
- 한번에 잘 할 수 없다
  - add fSM MD 2차ailing test -> make it pass -> make it right
  - make it pass as soon as possible - duct tape programming([[How to fall in love with TDD#Be a Duct Tape Programmer]])
  - why
    - 우리는 요구사항을 정확히 이해할 수 없고, 추정도 정확할 수 없다. 마법의 수정구가 없다.
    - 끝까지 구현해봐야만 발생 가능한 이슈를 발견할 수 있다.

- JPA Repository는 인터페이스가 아니다 Interface를 통해 도메인 로직을 보호하라.

성장

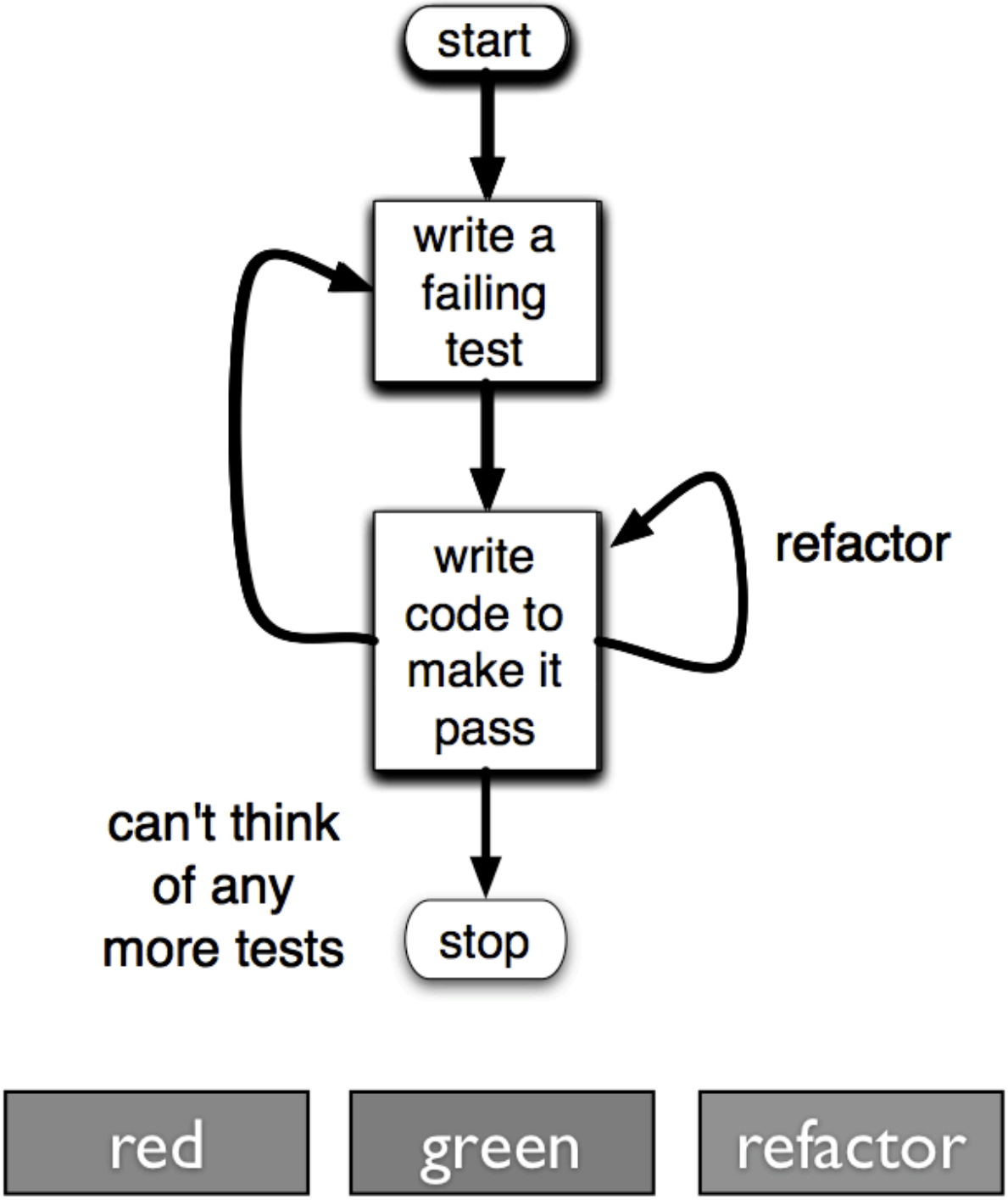
- *성장은 직접적인 목표*가 아닌, *생존을 위한 노력의 결과물*
- *현장에서의 경험이 가장 큰 배움의 기회*
  - book smart, street smart
  - resume(RDD)

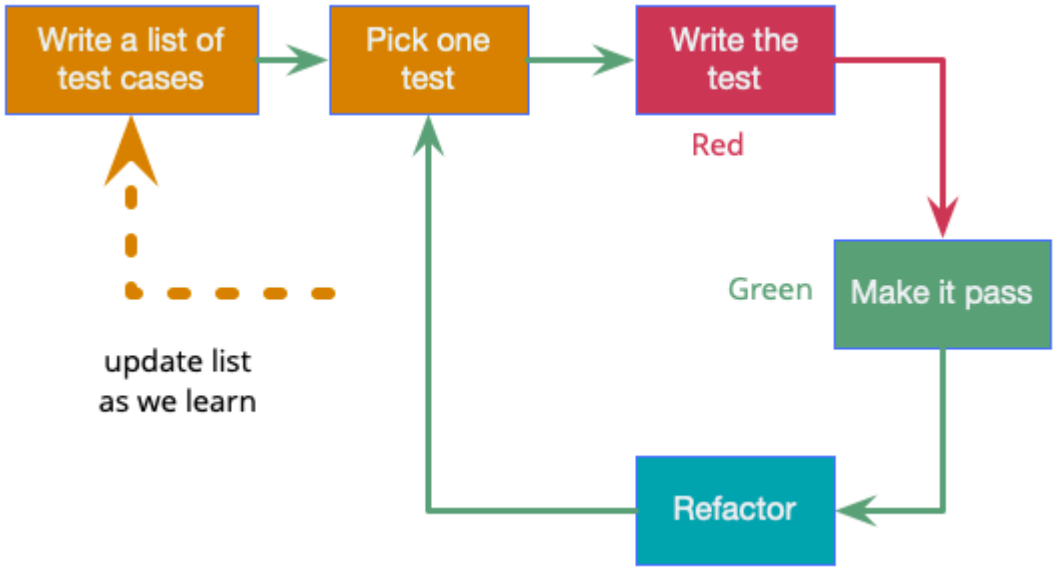
TDD에 대한 오해

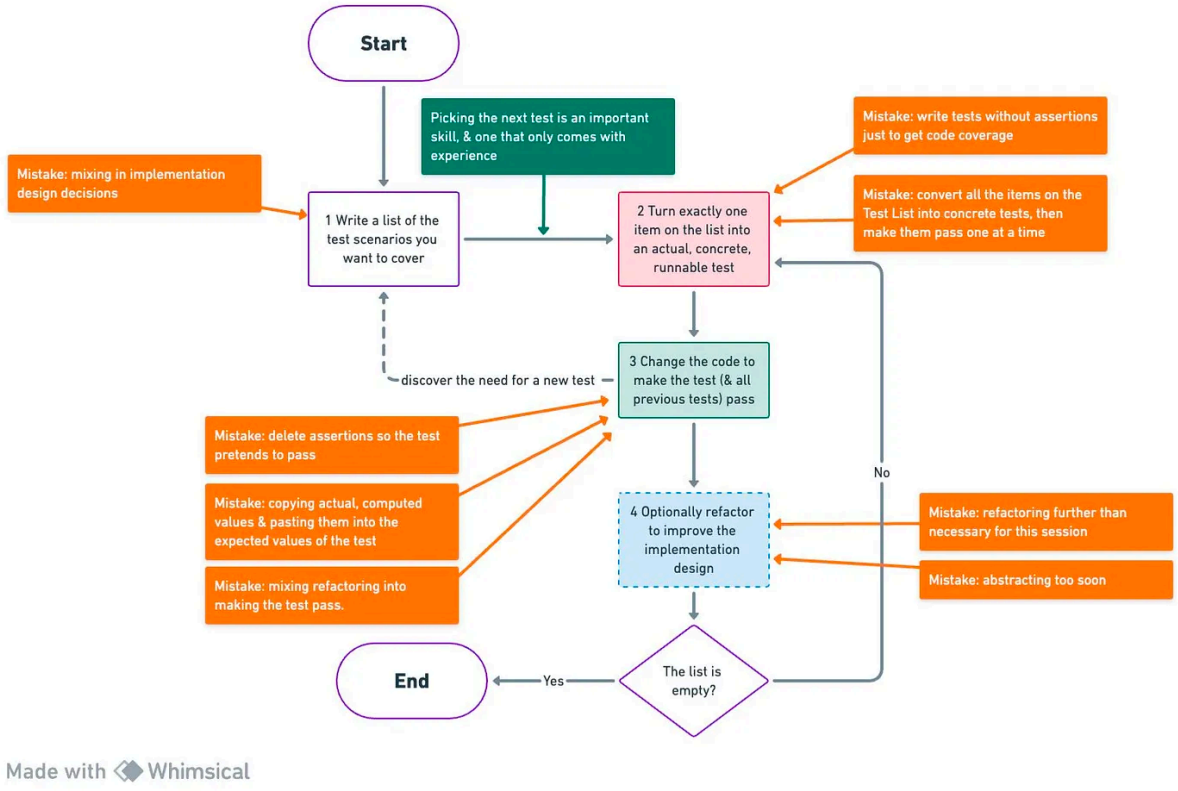
테스트 작성의 트리거

- 구현의 구조에 커플링

절차


start
write a failing test
write code to make it pass
refactor
can't think of any more tests
stop
red
green
refactor

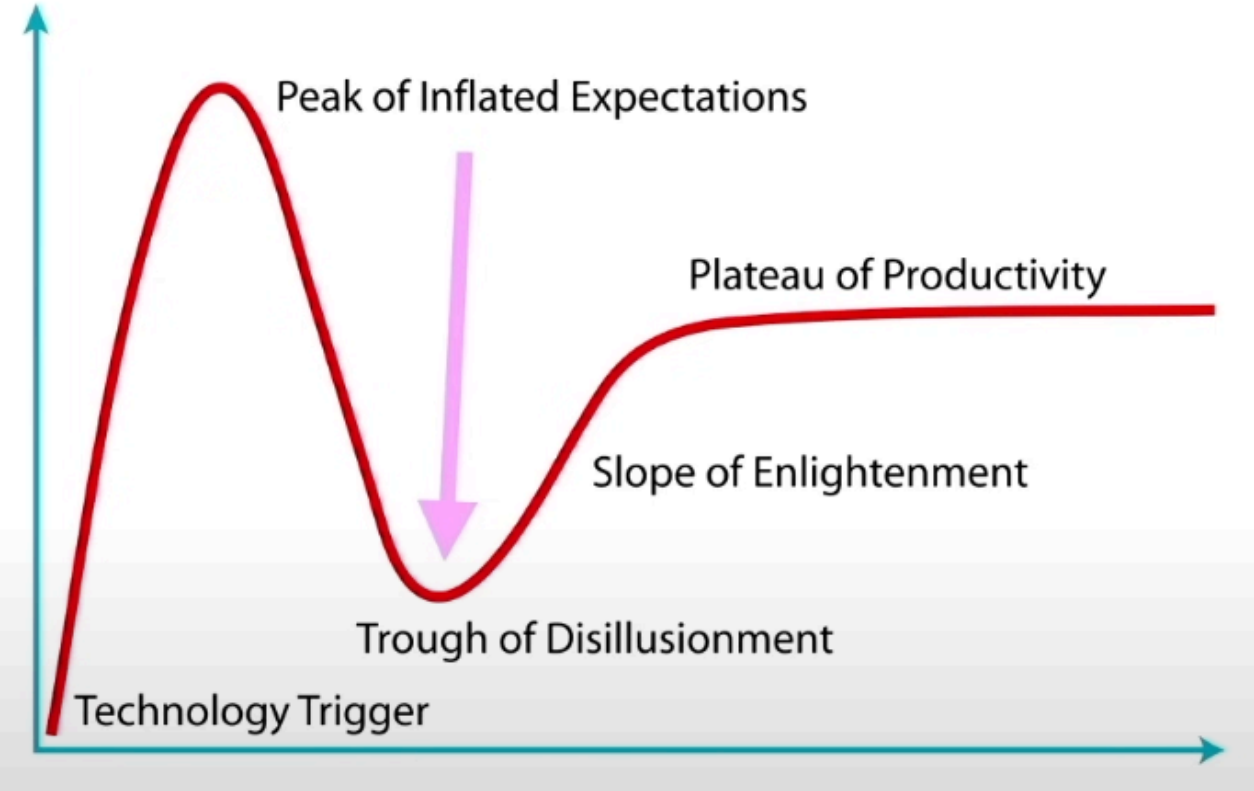




be a duct tape programmer

- 2가지 이유
  - 요구사항을 정확히 이해할 수 없다.
  - 끝까지 구현해봐야만 발생 가능한 이슈를 발견할 수 있다.

Gartner hype cycle



습관 변화의 어려움

- staggering vs column staggering keyboard
- 손깍지 방법
- 하지만 계속해서 연습하다 보면, 양쪽 방식 모두 쉽게 할 수 있게 된다는 것을 알게 될 것