TDD(Test Driven Development) 란?

TDD란 Test Driven Development의 약자로 **'테스트 주도 개발'**이라고 한다.

반복 테스트를 이용한 소프트웨어 방법론으로,
작은 단위의 테스트 케이스를 작성하고 이를 통과하는 코드를 추가하는 단계를 반복하여 구현한다.

짧은 개발 주기의 반복에 의존하는 개발 프로세스이며,
애자일 방법론 중 하나인 **eXtream Programming(XP)의 'Test-First'** 개념에 기반을 둔 단순한 설계를 중요시한다.

> **Tip**
>
> eXtream Programming(XP)
>
> 미래에 대한 예측을 최대한 하지 않고, 지속적으로 프로토타입을 완성하는 애자일 방법론 중 하나이다.
> 이 방법론은 추가 요구사항이 생기더라도, 실시간으로 반영할 수 있다.

TDD, 즉 **테스트 주도 개발(Test Driven Development)**에 대한 프로그래머들의 의견은 늘 엇갈린다.

TDD의 실효성을 업무로 경험한 사람들은 TDD를 더 효과적으로 실무에 적용하기 위해 고민한다.

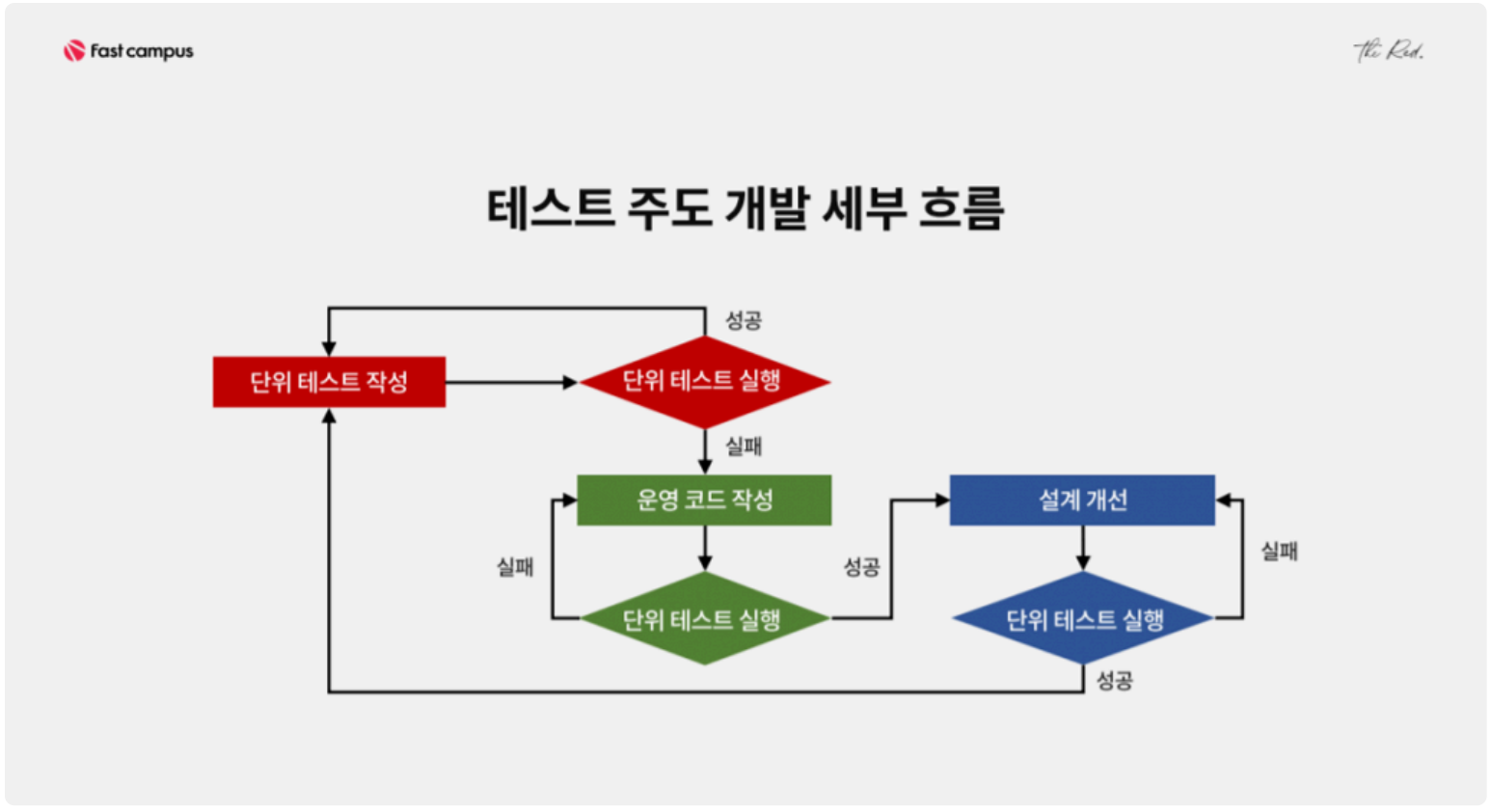
반면, 회사마다 일하는 방식이나 처한 업무 환경에 편차가 있다보니 일각에서는 실무에서 TDD를 사용하는 건 사실상 현실과 괴리감이 크다는 의견도 있다.

TDD 개발주기

Red 단계에서는 실패하는 테스트 코드를 먼저 작성한다.
Green 단계에서는 테스트 코드를 성공시키기 위한 실제 코드를 작성한다.
Blue 단계에서는 중복 코드 제거, 일반화 등의 리팩토링을 수행한다.

중요한 것은 **실패하는 테스트 코드를 작성**할 때까지 **실제 코드를 작성하지 않는 것**과, 실패하는 테스트를 통과할 정도의 **최소 실제 코드를 작성**해야 하는 것이다.

이를 통해, 실제 코드에 대해 기대되는 바를 보다 명확하게 정의함으로써 **불필요한 설계를 피할 수** 있고, **정확한 요구 사항**에 집중할 수 있다.



Info

TDD를 적용한 사례 :
생년월일(input)을 입력받으면 현재 나이(output)를 출력하는 프로그램

1) 처음에는 간단한 것으로 목표를 정한다. (태어난 해와 올해의 연도를 입력해서 연도 뺄셈을 통해 나이 계산)
  - 2015, 2018 -> (만)3살 우선 이것을 만들겠다는 생각을 한다.

2) 만들기도 전에 **만든 후에 무엇을 테스트할지를 설계**한다. (실패하는 테스트)
  - 2015, 2018를 입력하면 2가 나오는 테스트 프로그램(장차 만들 프로그램을 테스트할 코드)를 만든다.

3) 그 다음에 그 **테스트를 통과할 프로그램**(1.을 목표로 작성한 코드)를 만든다.
  - 2018 - 2015 (올해의 연도 - 태어난 해)

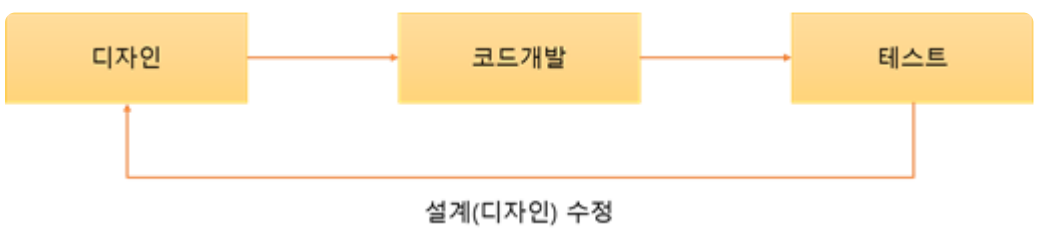
4) 테스트 프로그램으로 이 프로그램(3.에 해당하는 코드)을 실행한다.

5) 통과했으면 새로운 테스트를 추가한다.
  - 이번에는 생월을 추가했을 때 계산하는 프로그램

6)위와 같은 작업을 계속 왔다갔다 수행한다.

일반 개발 방식 vs TDD 개발 방식

일반적인 개발



보통의 개발 방식은 **'요구사항 분석 -> 설계 -> 개발 -> 테스트 -> 배포'**의 형태의 개발 주기를 갖는데 이러한 방식은 소프트웨어 **개발을 느리게 하는 잠재적 위험**이 존재한다.

그 이유로는,

소비자의 요구사항이 **처음부터 명확하지 않을 수 있다.**

따라서 처음부터 완벽한 설계는 어렵다.

자체 버그 검출 능력 저하 또는 **소스코드의 품질이 저하**될 수 있다.

자체 **테스트 비용이 증가**할 수 있다.

이러한 문제점들이 발생되는 이유는 어느 프로젝트든 **초기 설계가 완벽하다고 말할 수 없기 때문**이다.
보통 고객의 요구사항 또는 디자인의 오류 등 많은 외부 또는 내부 조건에 의해, **재설계하여 점진적으로 완벽한 설계**로 나아간다.
하지만, **재설계로 인해** 개발자는 코드를 삽입, 수정, 삭제 하는 과정에서 **불필요한 코드**가 남거나 중복처리 될 가능성이 크다.

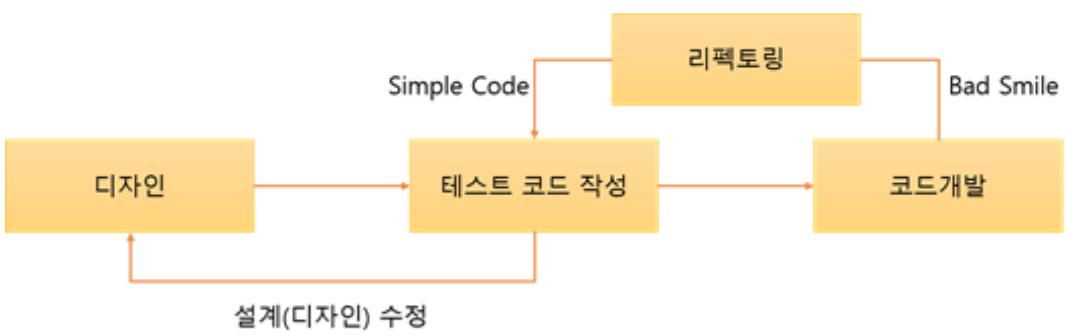
> 결론적으로 이러한 코드들은 재사용이 어렵고 관리가 어려워져 유지보수를 어렵게 만든다.

작은 부분의 기능 수정에도 **모든 부분을 테스트**해야 하므로 **전체적인 버그**를 검출하기 어려워진다.

따라서 자체 버그 검출 능력이 저하된다. 그 결과 어디서 버그가 발생할지 모르기 때문에 잘못된 코드도 고치지 않으려 하는 현상이 나타나고 이 현상은 소스코드의 품질 저하과 직결된다.

이렇게 작은 수정에도 모든 기능을 다시 테스트해야 하는 문제가 발생하여 자체 테스트 비용이 증가된다.

TDD 개발



TDD와 일반적인 개발 방식의 가장 큰 차이점은 **테스트 코드를 작성한 뒤에 실제 코드를 작성**한다는 점이다.

디자인(설계) 단계에서 프로그래밍 목적을 반드시 **미리 정의**해야만 하고,

또 무엇을 테스트해야 할지 **미리 정의(테스트 케이스 작성)**해야만 한다.

테스트 코드를 작성하는 도중에 발생하는 예외 사항(버그, 수정사항)들은 테스트 케이스에 추가하고 설계를 개선한다.

이후 테스트가 통과된 코드만을 코드 개발 단계에서 실제 코드로 작성한다.

이러한 반복적인 단계가 진행되면서 자연스럽게 코드의 버그가 줄어들고, 소스코드는 간결해진다.
또한, 테스트 케이스 작성으로 인해 자연스럽게 설계가 개선됨으로 재설계 시간이 절감된다.

Tip

TDD의 대표적인 Tool 'JUnit'

JUnit은 현재 전 세계적으로 가장 널리 사용되는 'Java 단위 테스트 프레임워크' 이다.
JUnit을 시작으로 CUnit(C언어), CppUnit(C++), PyUnit(Python) 등 xUnit 프레임워크가 탄생하게 되었다.

TDD의 효과

1. 디버깅 시간을 단축 할 수 있다.

이는 유닛 테스팅을 하는 이점이기도 하다.

예를 들면 사용자의 데이터가 잘못 나온다면 DB의 문제인지, 비즈니스 레이어의 문제인지 UI의 문제인지 실제 모든 레이러들을 전부 디버깅 해야하지만, TDD의 경우 자동화 된 유닛테스팅을 전재하므로 특정 버그를 손 쉽게 찾아낼 수 있다.

2. 코드가 내 손을 벗어나기 전에 가장 빠르게 피드백 받을 수 있다.

개발 프로세스에서는 보통 '인수 테스트'를 한다. 이미 배치된 시스템을 대상으로 클라이언트가 의뢰한 소프트웨어가 사용자 관점에서 사용할 수 있는 수준인지 체크하는 과정이다.

이미 90% 이상 완성된 코드를 가지고 테스트하기 때문에, **문제를 발견해도, 정확하게 원인이 무엇인지 진단**하기는 힘들다.

하지만 TDD를 사용하면 **기능 단위로 테스트**를 진행하기 때문에 코드가 모두 완성되어 프로그래머의 손을 떠나기 전에 피드백을 받는 것이 가능하다.

3. 작성한 코드가 가지는 불안정성을 개선하여 생산성을 높일 수 있다.

켄트 백은 TDD는 불안함을 지루함으로 바꾸는 마법의 돌이라고 말했다.

앞서 말한 것처럼 TDD를 사용하면, 코드가 내 손을 떠나 사용자에게 도달하기 전에 **문제가 없는지 먼저 진단 받을 수 있다.** 그러므로 코드가 지닌 불안정성과 불확실성을 지속적으로 해소해준다.

4. 재설계 시간을 단축 할 수 있다.

테스트 코드를 먼저 작성하기 때문에 개발자가 지금 무엇을 해야하는지 분명히 정의하고 개발을 시작하게 된다.

또한 테스트 시나리오를 작성하면서 **다양한 예외사항에 대해 생각**해볼 수 있다.

이는 개발 진행 중 소프트웨어의 전반적인 설계가 변경되는 일을 방지할 수 있다.

5. 추가 구현이 용이하다.

개발이 완료된 소프트웨어에 **어떤 기능을 추가**할 때 가장 우려되는 점은 해당 기능이 **기존 코드에 어떤 영향**을 미칠지 알지 못한다는 것이다.

하지만 TDD의 경우 자동화된 유닛 테스팅을 전제하므로 **테스트 기간을 획기적으로 단축**시킬 수 있다.

> **이러한 TDD의 장점에도 불구하고 모두가 이 개발 프로세스를 따르지 않는다.**
> **그 이유는 무엇일까?**

TDD의 단점

1. 가장 큰 단점은 바로 생산성의 저하이다.

개발 속도가 느려진다고 생각하는 사람이 많기 때문에 TDD에 대해 반신반의 한다.
왜냐하면 처음부터 2개의 코드를 짜야하고, 중간중간 테스트를 하면서 고쳐나가야 하기 때문이다.
TDD 방식의 개발 시간은 일반적인 개발 방식에 비해 대략 10~30% 정도로 늘어난다.
SI 프로젝트에서는 소프트웨어의 품질보다 납기일 준수가 훨씬 중요하기 때문에 TDD 방식을 잘 사용하지 않는다.

2. 이제까지 자신이 개발하던 방식을 많이 바꿔야 한다.

몸에 체득한 것이 많을 수록 바꾸기가 어렵다. 오히려 개발을 별로 해보지 않은 사람들에겐 적용하기가 쉽다.

3. 구조에 얽매힌다.

TDD로 프로젝트를 진행하면서 어려운 예외가 생길 수 있는데 그것 때문에 고민하는 순간이 찾아오게 된다.

원칙을 깰 수는 없고 꼼수가 있기는 한데 그 꼼수를 위해서 구조를 바꾸자니 이건 아무래도 아닌 것 같고, 테스트는 말 그대로 테스트일 뿐 실제 코드가 더 중요한 상황인데도 불구하고 테스트 원칙 때문에 쉽게 넘어가지 못하는 그런 경우다.