

프로그래밍 역량 강화 전문기관, 민코딩

TDD 개요



목차

1. TDD의 시작 : 테스트 코드 먼저 작성하기
2. TDD 세부 규칙 1 : Baby Step
3. TDD 세부 규칙 2 : 설계는 리팩토링 단계에서
4. TDD 장점
5. TDD 단점
6. TDD 직접해보기 : 은행계좌 KATA

테스트 코드 먼저 작성하기

TDD의 시작

TDD

Test Driven Development

- 테스트 기반 개발 방법
- 켄트백 창시 (in XP 개발방법론)

켄트 벅

미국 소프트웨어 엔지니어



켄트 벅은 미국의 소프트웨어 엔지니어이자, 협업적이고 반복적인 디자인 프로세스의 엄격한 사상을 삼가는 소프트웨어 개발 방법론인 익스트림 프로그래밍의 개발자이다. 애자일 소프트웨어 개발의 창설 문서인 애자일 선언문의 17명의 오리지널 서명인 가운데 한 명이였다. 위키백과

출생: 1961년 3월 31일 (60세)

국적: 미국

알려진 분야: 익스트림 프로그래밍, 소프트웨어 디자인 패턴, JUnit

학력: 오레곤 대학, UO Computer Science Department, 아즈 앤드 사이언스 대학

저서

5개 이상 항목 더보기



익스트림 프로그래밍
1999년



테스트 주도 개발
2000년



켄트 벅의 구현 패턴
2007년



리팩토링
1999년

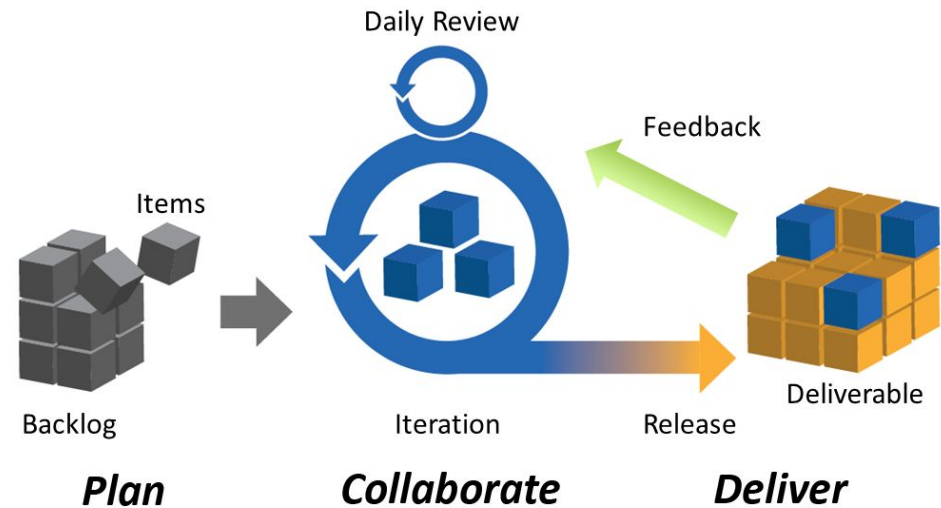
[참고] Agile

Agile 은 개발자가 일하는 방식, 즉 개발 문화이다.

- 업무를 작은 공약 단위로 정하고,
이를 자주 반복하는 방식으로 개발을 하는 접근 방식 (from Atlassian)

Agile 여러 방법론이 존재한다.

- Scrum
- XP
- LEAN 등등



Agile Project Management: Iteration

TDD 방법 - 기본 규칙 1

테스트 코드를 먼저 만든 후, 구현을 시작한다.

- 테스트케이스를 모두 만든다.
 - sum 모듈에 (1, 1) 넣는다면, 2 가 출력되어야 함을 검사하는 Test Case
 - minus 모듈에 (100, 70) 을 넣는다면, 30이 나와야 함을 검사하는 Test Case

순서

1. 요구사항 파악
2. 테스트 코드 개발 : 테스트 케이스
3. Production 코드 개발 : 실제 코드 개발
4. 리팩토링 과정을 거쳐, 간결하고 읽기 쉽게 만든다.
5. 1 ~ 4 번 과정을 반복

TDD Cycle

TDD의 정신

- 레드, 그린, 리팩터

TDD Cycle 4 단계

1. Red
 - Fail되는 테스트 작성
(아직 코드 작성 안했기에 FAIL)
2. Green
 - 동작하는 코드 작성
3. Refactor
 - Clean Code
4. 이를 반복한다.

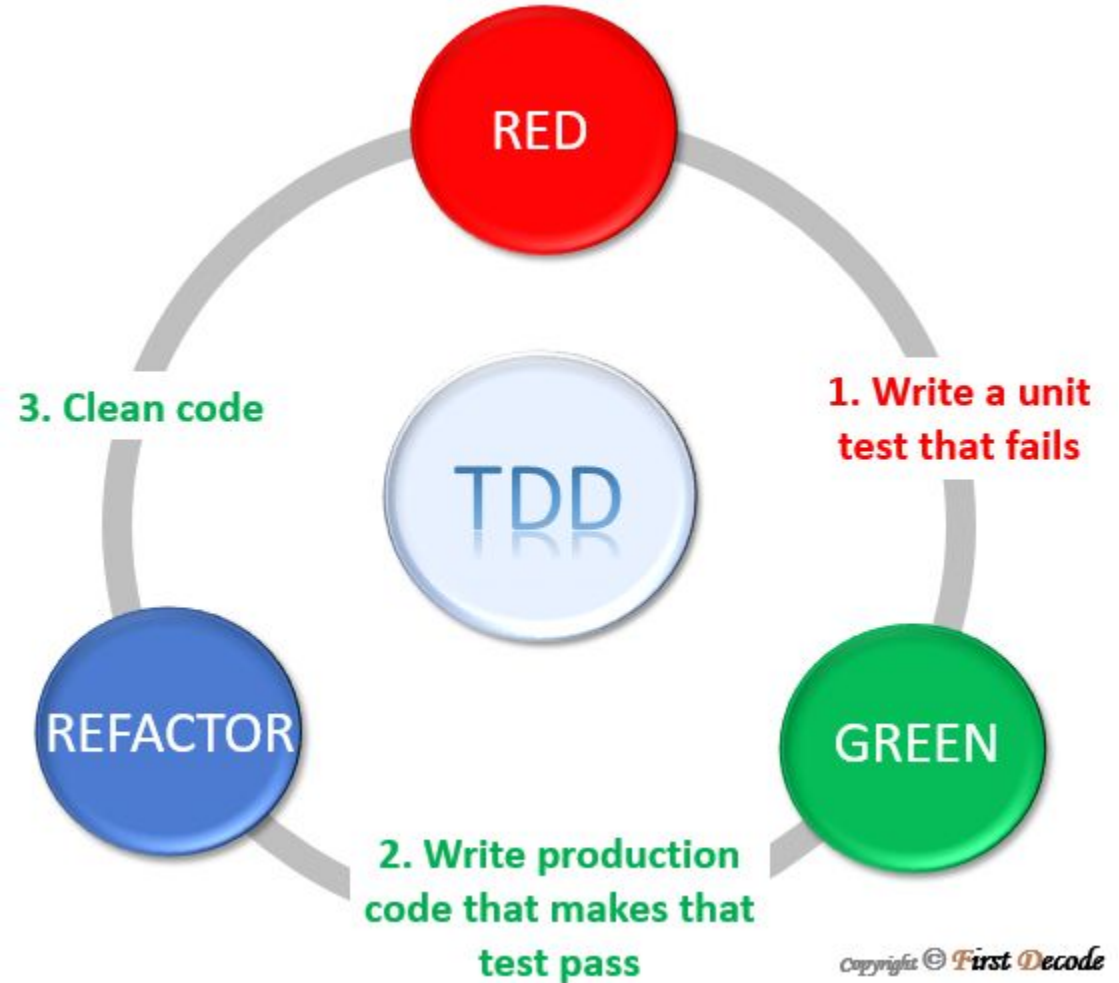


그림 출처 : <https://docs.firstdecode.com/tdd/>

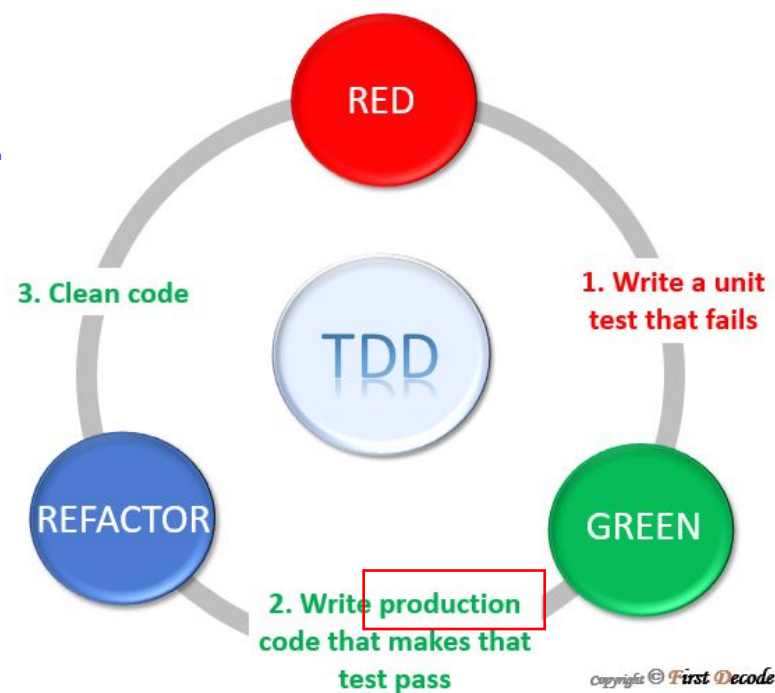
Production Code 의미

Production Code의 의미 1

- Production 환경에 놓여지는 코드
- 완성도 높은 안정적인 코드

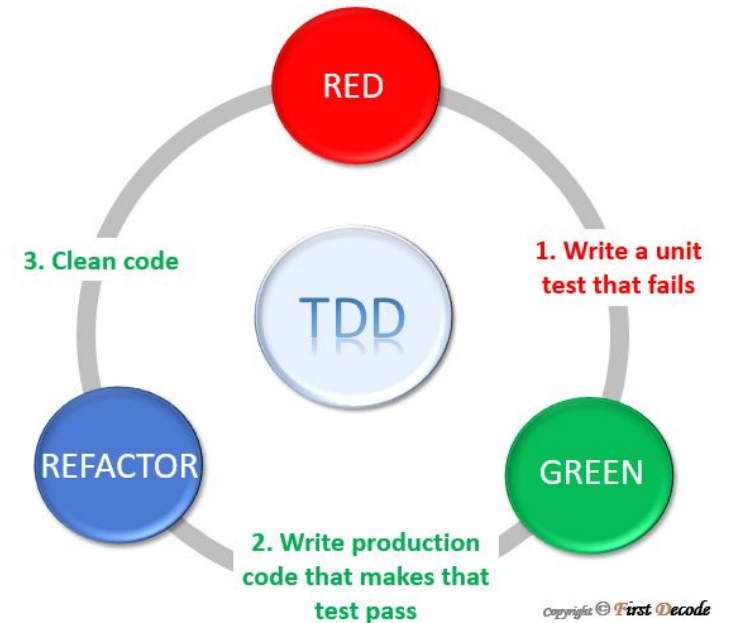
Production Code의 의미 2 □ TDD 에서 사용하는

- 생산중인 코드 (Code in production)
- 만들고 있는 코드
- TDD 에서 Production Code의 의미는 생산중인 코드를 뜻한다.

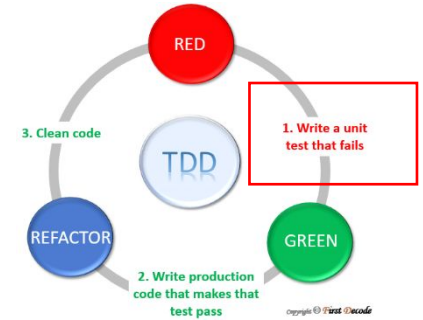


테스트 코드를 먼저 작성해보자.

1. Fail 나는 Test Case 작성한다.
 - 어쩔수 없이, 요구조건이 명확해진다.
2. Production Code를 작성한다.
3. Unit Test에 pass를 확인하여 구현의 자신감을 얻는다.
4. Refactoring을 통해 clean code를 만들어낸다.



1 단계 : Fail 나는 Test Case 제작



다음과 같이, Build만 되도록 구현한다.

그리고 Test Case 를 제작하여, **Fail**을 눈으로 확인한다.

```
class Cal {  
public:  
    int calMinus(int a, int b) {  
        return 0;  
    }  
};
```

```
TEST(CalTest, MinusTest) {  
    EXPECT_EQ(2, Cal().calMinus(4, 2));  
    EXPECT_EQ(4, Cal().calMinus(5, 1));  
}
```

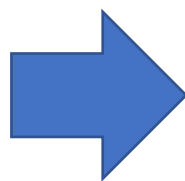
2 단계 : Production Code 작성

cal_minus(int a, int b) 메서드 제작

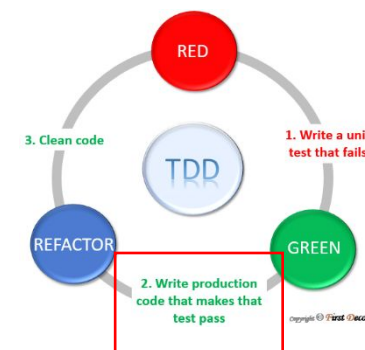
- 리팩토링을 할 것이므로, 가볍게 작성한다.
- 코드 가독성은 신경쓰지 않고 구현을 한다.

작성 후 Unit Test를 돌려 PASS를 확인한다.

```
int calMinus(int a, int b) {  
    return 0;  
}
```



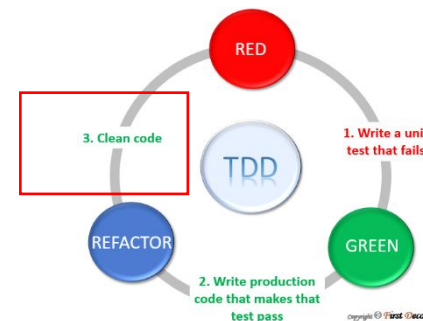
```
int calMinus(int a, int b) {  
    int val = 0;  
    if (a > b) {  
        val = a - b;  
    }  
    else {  
        val = b - a;  
    }  
    return val;  
}
```



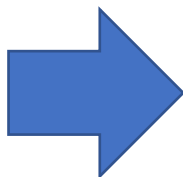
3 단계 : 리팩토링 시작 1

else 를 제거한다.

- 리팩토링을 한 단계 할 때 마다 Unit Test에서 PASS를 확인한다.



```
int calMinus(int a, int b) {  
    int val = 0;  
    if (a > b) {  
        val = a - b;  
    }  
    else {  
        val = b - a;  
    }  
    return val;  
}
```



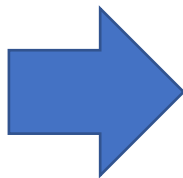
```
int calMinus(int a, int b) {  
    int val = 0;  
    if (a > b) {  
        val = a - b;  
        return val;  
    }  
    val = b - a;  
    return val;  
}
```

3 단계 : 리팩토링 시작 2

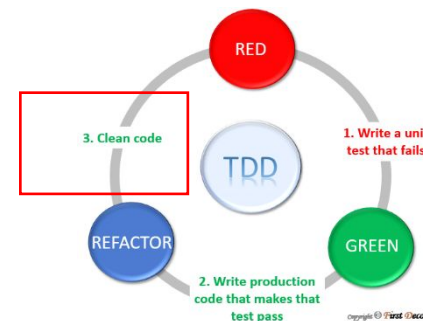
임시변수 제거

- 변수를 inline 시켜, 가독성을 높인다.
- 다시 Unit Test 에서 PASS를 확인한다.

```
int calMinus(int a, int b) {  
    int val = 0;  
    if (a > b) {  
        val = a - b;  
        return val;  
    }  
    val = b - a;  
    return val;  
}
```



```
int calMinus(int a, int b) {  
    if (a > b) {  
        return a - b;  
    }  
    return b - a;  
}
```

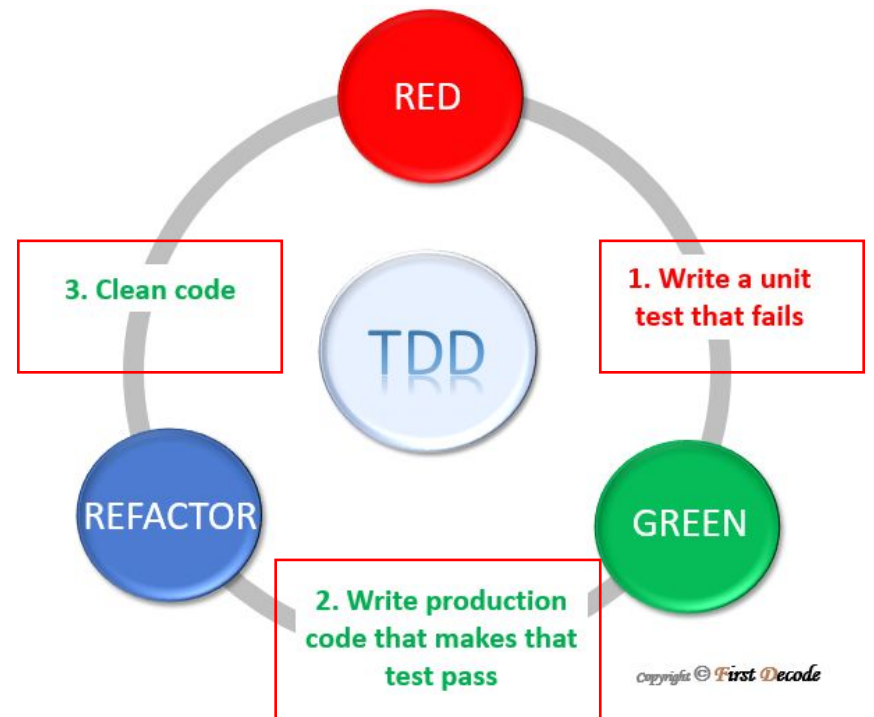


켄트백에 말하는 TDD 실행 규칙! 작은 Cycle을 자주 반복한다.

TDD 세부 규칙 1 : Baby Step

TDD의 두 가지 단순한 규칙

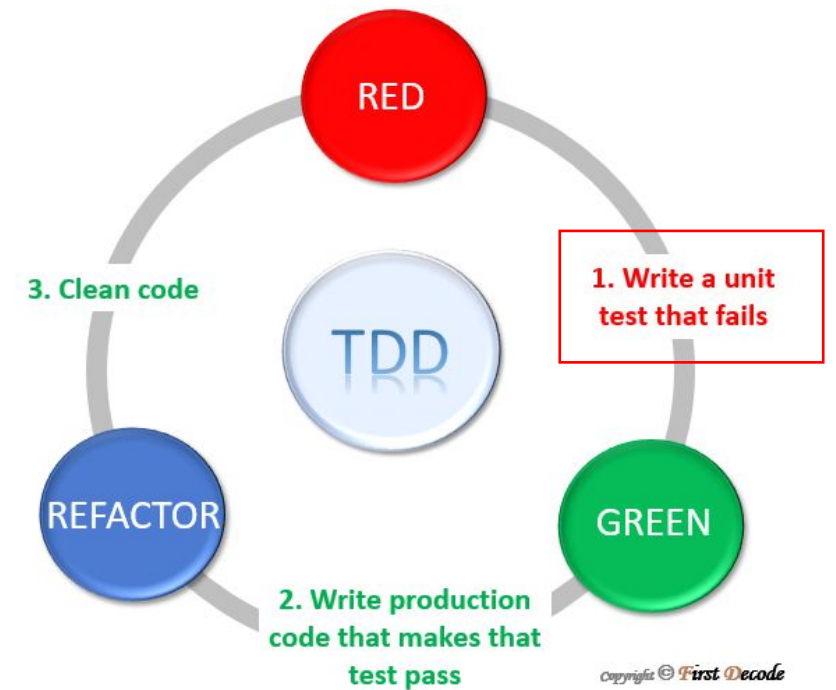
1. 오직, 테스트가 실패할 경우에만 새로운 코드 작성
2. 중복을 제거한다.



Write a unit test that fails

실패하는 Unit Test 만들기

- 작은 테스트를 만든다.
- 컴파일조차 안되는 것도 Fail로 취급한다.
- fail을 눈으로 확인한다.

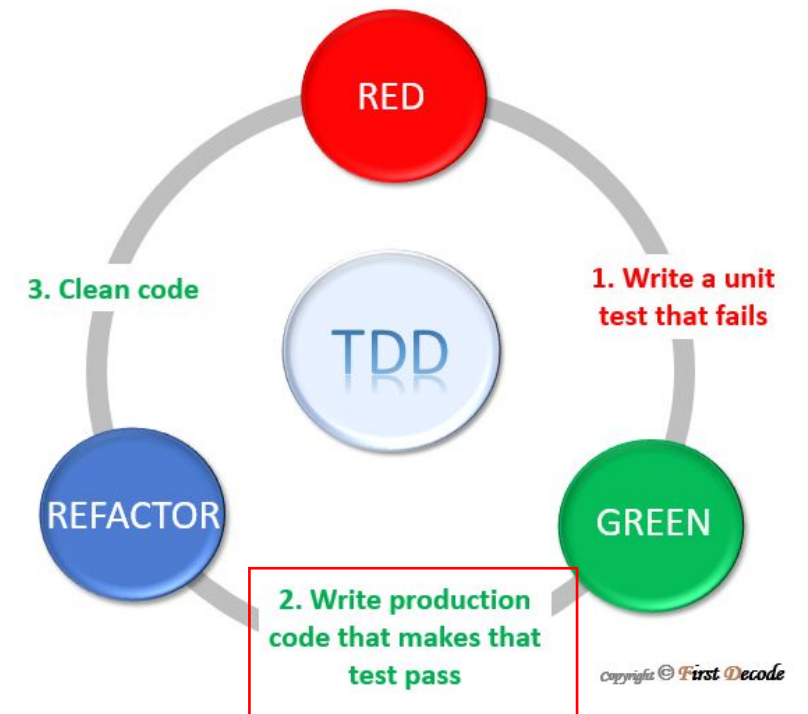


Pass만 뜨면 된다.

통과되게끔, 빠르게 구현을 한다.

이러한 행동을 해도 된다.

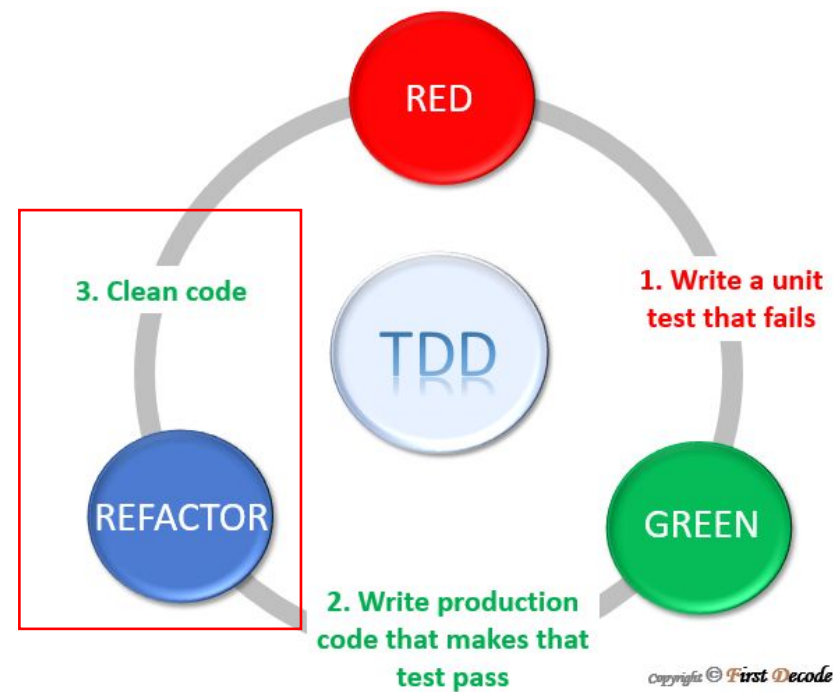
- 기존 코드를 복사 붙여넣기
- 함수가 무조건 특정 상수만 return 하도록 하기
- 특정 입력 값에, 정답을 그대로 return 하도록 하기



Refactoring

TDD의 세번째 단계이다.

- 무작정 Pass만 뜨도록 만든 코드에서
발생된 중복을 제거하여,
Clean 한 코드로 만드는 일.



0 단계 : Test 기본 세팅

Production Code와 Test Code 준비

```
class Cal {  
public:  
    int calMinus(int a, int b) {  
        return 0;  
    }  
};
```

Cal.cpp

```
TEST(CalTest, MinusTest) {  
  
}
```

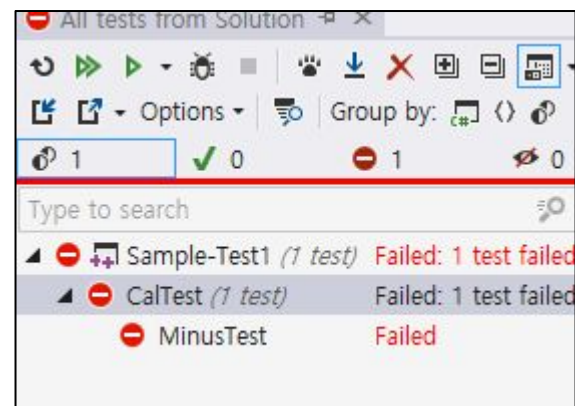
Cal_test.cpp

1 단계 : 실패하는 테스트 만들기

RED 단계

실패를 눈으로 확인한다.

```
TEST(CalTest, MinusTest) {  
    EXPECT_EQ(2, Cal().calMinus(4, 2));  
}
```



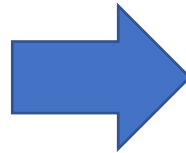
2 단계 : Pass 되도록 최소 구현하기

Green 단계

Fail 에 Pass 되도록 구현하기

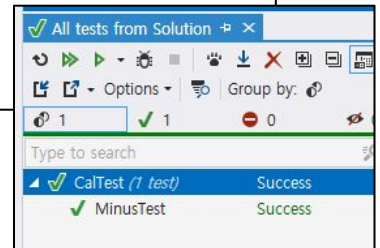
- 더 Baby Step을 하려면, 심지어 하드코딩 해도 좋다.

```
class Cal {  
public:  
    int calMinus(int a, int b) {  
        return 0;  
    }  
};
```



```
class Cal {  
public:  
    int calMinus(int a, int b) {  
        return a - b;  
    }  
};
```

잘 동작함을 눈으로 확인



3 단계 : Refactoring!

Clean 한 코드가 되도록 수정한다.

- Pass만 되게끔 작성한 코드를 깔끔한 코드로 수정한다.
- 중복이 있다면, 중복이 없도록 노력한다.

```
class Cal {  
public:  
    int calMinus(int param1, int param2) {  
        return param1 - param2;  
    }  
};
```

Parameter 명 변경



4 단계 : Test Code 추가

오직 테스트가 실패할 때만, 새로운 코드를 추가한다라는 규칙을 지키기 위해, 테스트 코드 부터 작성한다.

```
TEST(CalTest, MinusTest) {  
    EXPECT_EQ(2, Cal().calMinus(4, 2));  
}  
  
TEST(CalTest, SmallToBig) {  
    EXPECT_EQ(4, Cal().calMinus(1, 5));  
    EXPECT_EQ(3, Cal().calMinus(3, 6));  
    EXPECT_EQ(9, Cal().calMinus(1, 10));  
}
```

작은수에서 큰 수를 빼는 테스트 코드 추가.

5 단계 : Production Code 작성

Green 단계

‘최악’ 을 저질러도 좋다. Pass만 되도록 구현을 하면 된다.
Pass가 되도록 빠르게 구현한다. (by 켄트백)

```
class Cal {  
public:  
    int calMinus(int param1, int param2) {  
        if (param1 == 1 && param2 == 5) return 4;  
        if (param1 == 3 && param2 == 6) return 3;  
        if (param1 == 1 && param2 == 10) return 9;  
        return param1 - param2;  
    }  
};
```

✓ CalTest (2 tests)	Success
✓ MinusTest	Success
✓ SmallToBig	Success

6 단계 : Refactoring

일단 테스트만 통과되도록 만든 코드에서 생겨난 중복 코드들을 모두 제거한다.

```
class Cal {  
public:  
    int calMinus(int param1, int param2) {  
        if (param1 < param2) return param2 - param1;  
        return param1 - param2;  
    }  
};
```

✓ CalTest (2 tests)	Success
✓ MinusTest	Success
✓ SmallToBig	Success

아기 발걸음 부터 (Baby Step)

- ✓ 실습을 하다 보면, 너무 작은 스텝이라고 느껴진다.
- ✓ 아주 작은 단계로 TDD 연습을 하는 이유는 다음과 같다.

이 단계가 너무 작게 느껴지는가? 하지만 기억하기 바란다.
**TDD의 핵심은 이런 작은 단계를 밟아야 한다는 것이 아니라,
이런 작은 단계를 밟을 능력을 갖추어야 한다는 것이다.**

내가 일상에서 항상 이런 식으로 작업하는지 궁금한가? 그렇지 않다.

만약 정말 작은 단계로 작업하는 방법을 배우면,
저절로 적절한 크기의 단계로 작업할 수 있게 될 것이다.
그러나 큰 단계로만 작업했다면,
더 작은 단계가 적절한 경우에 대해 결코 알지 못하게 된다.

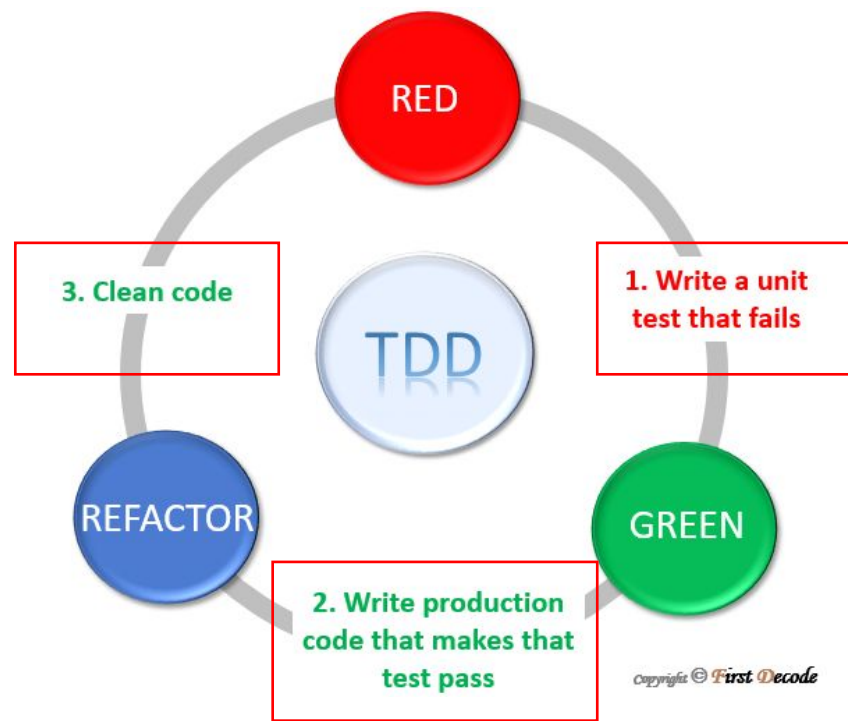
[정리] TDD 세부 규칙

오직, 테스트가 실패할 경우에만 새로운 코드 작성

- 조금만 테스트하고, 조금만 코딩하고.
켄트백은 이를 Baby Step 이라고 표현 했다.
- Baby Step이 익숙해지면,
자신에게 맞는 보폭을 찾아
조율하면 된다.

중복을 제거한다.

- Pass만 되도록 구현한 코드에
중복을 제거하여 Clean code로 만든다.



마틴파울러가 말하는 TDD 세부 규칙

TDD 세부 규칙 2 : **설계는 리팩토링 단계에서**

한 번에 하나만 집중하자. 1

테스트를 작성할 때는

- Unit Test는 To Do List 이며, 뭘 할지에 대해서만 집중하자!
- 그 모듈이 어떤 입출력을 가져야 할 지에 대해서만 생각한다.

구현 모드일 때는

- 어떤 설계가 가장 좋을지 고민하지 않는다.
- 테스트에만 통과할 수 있도록 한다. To Do List 항목을 체크하는데만 신경을 쓴다.
(ex. 이거 해결 했음)
- 모듈을 완성하기 위한 코드 구현 보다는,
테스트에 통과 하려는 목적의 코드가 주 구현이어야 한다.

한 번에 하나만 집중하자. 2

리팩토링 모드일 때는

- 리팩토링 모드에서 올바른 설계를 얻는 데만 신경을 쓴다.
- 더 좋은 To Do 해결방법을 고안한다.
- 완성된 설계를 위해,
리팩토링이 아니라 구현이 더 필요하다면
Unit Test를 추가하는 Red 단계로 넘어간다.

한 작업에 하나에만 집중하는 것은,
뇌의 가해지는 스트레스를 줄일 수 있다.
- 마틴파울러

만들고자 하는 모듈 : Fibonacci

켄트백의 TDD Example

Fibo Class의 fibo 메서드 : $\text{fibo}(n) = \text{fibo}(n - 2) + \text{fibo}(n - 1)$

- $\text{fibo}(0) \square 0$
- $\text{fibo}(1) \square 1$
- $\text{fibo}(2) \square 1$
- $\text{fibo}(3) \square 2$
- $\text{fibo}(4) \square 3$
- $\text{fibo}(5) \square 5$
- $\text{fibo}(6) \square 8$
- $\text{fibo}(7) \square 13$
- $\text{fibo}(8) \square 21$
- ...

테스트코드 작성, 시작 코드 작성하기

여기서 부터 TDD를 시작해보자.

```
TEST(FiboTest, OrderTest) {  
    EXPECT_EQ(0, Fibo().fibonacci(0));  
}
```

Fibo_test.cpp

```
class Fibo {  
public:  
    int fibonacci(int n) {  
        return 0;  
    }  
};
```

Fibo.cpp

창 배치

- ✓ 왼쪽 : Test Code
- ✓ 오른쪽 : Production Code

```
Fibo_test.cpp
1 #include "pch.h"
2 #include "../Project9/Fibo.cpp"
3
4 using namespace std;
5 TEST(FiboTest, OrderTest) {
6     EXPECT_EQ(0, Fibo().fibonacci(0));
7 }
8
9

Fibo.cpp
1 class Fibo {
2 public:
3     int fibonacci(int n) {
4         return 0;
5     }
6 };
7
8
9
```

Fail 확인

RED 단계

Fail을 눈으로 확인한다.

```
TEST(FiboTest, OrderTest) {  
    EXPECT_EQ(0, Fibo().fibonacci(0));  
    EXPECT_EQ(1, Fibo().fibonacci(1));  
}
```

▲ - FiboTest (1 test)	Failed: 1 test failed
- OrderTest	Failed

Production Code 구현

Green 단계

Green을 눈으로 확인한다.

```
class Fibo {  
public:  
    int fibo(int n) {  
        if (n == 0) return 0;  
        return 1;  
    }  
};
```

▲ ✓ FiboTest (1 test)	Success
✓ OrderTest	Success

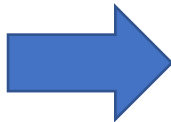
Refactor

Refactor 단계

테스트 코드의 중복을 제거하자.

테스트 코드 구현을 더 편리하게 만들

```
TEST(FiboTest, OrderTest) {  
    EXPECT_EQ(0, Fibo().fibo(0));  
    EXPECT_EQ(1, Fibo().fibo(1));  
}
```



```
TEST(FiboTest, OrderTest) {  
    vector<vector<int>> cases = { {0, 0}, {1, 1}};  
  
    for (int i = 0; i < cases.size(); i++) {  
        int input = cases[i][0];  
        int expected = cases[i][1];  
        EXPECT_EQ(expected, Fibo().fibo(input));  
    }  
}
```


Fail 확인

RED 단계

Fail을 눈으로 확인한다.

- $\text{fibonacci}(3) = \text{fibonacci}(1) + \text{fibonacci}(2)$ 이며, 정답은 $1 + 1 = 2$ 이다.

```
vector<vector<int>> cases = { {0, 0}, {1, 1}, {2, 1} };  
  
for (int i = 0; i < cases.size(); i++) {  
    int input = cases[i][0];  
    int expected = cases[i][1];  
    EXPECT_EQ(expected, Fibo().fibonacci(input));  
}
```



▲	✓ FiboTest (1 test)	Success
	✓ OrderTest	Success

{2, 1} 을 추가하고
Test 했을 때 Pass 이므로
다른 값을 넣도록 하자.

```
vector<vector<int>> cases = { {0, 0}, {1, 1}, {2, 1}, {3, 2} };  
  
for (int i = 0; i < cases.size(); i++) {  
    int input = cases[i][0];  
    int expected = cases[i][1];  
    EXPECT_EQ(expected, Fibo().fibonacci(input));  
}
```

▲	✗ FiboTest (1 test)	Failed: 1 test failed
	✗ OrderTest	Failed

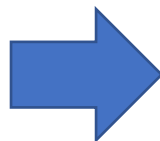
{3, 2}를 넣었을 때
Fail이 발생했다.

Production Code 구현

Green 단계

Green을 눈으로 확인한다.

```
class Fibo {  
public:  
    int fibo(int n) {  
        if (n == 0) return 0;  
        return 1;  
    }  
};
```



```
class Fibo {  
public:  
    int fibo(int n) {  
        if (n == 0) return 0;  
        if (n <= 2) return 1;  
        return 2;  
    }  
};
```

Refactor

n = 3 일 때,

정답을 정확하게 1 + 1 로 표현했다.

- n = 0 일때, 0
- n = 1 일때, 1
- n = 2 일때, 0 + 1 = 1
- n = 3 일때, 1 + 1 = 2

```
class Fibo {  
public:  
    int fibonacci(int n) {  
        if (n == 0) return 0;  
        if (n <= 2) return 1;  
        return 1 + 1;  
    }  
};
```

Refactor

Refactor 단계

모든 조건에서 동작되게끔 리팩토링

```
class Fibo {  
public:  
    int fibo(int n) {  
        if (n == 0) return 0;  
        if (n <= 2) return 1;  
        return 1 + 1;  
    }  
};
```



```
class Fibo {  
public:  
    int fibo(int n) {  
        if (n == 0) return 0;  
        if (n <= 2) return 1;  
        return fibo(n - 2) + fibo(n - 1);  
    }  
};
```


우리가 TDD를 해야하는 이유

TDD 장점

테스트 코드를 먼저 작성하는 것의 의미

테스트 코드를 작성을 위해서는,

- 무엇을 테스트할지 결정해야 한다.
- 프로그래밍의 목적을 명확히 한다.

즉 Test를 먼저 만들게 되면

- 만들고자 하는 모듈을 개발하기 전, 입출력 요구사항이 아주 명확해진다.

모듈화

개발 시 Unit 테스트가 가능하도록 모듈 단위로 개발하게 된다.
이는 자연스럽게 테스트가 쉽도록 모듈을 만들게 된다.

테스트가 쉽도록 모듈을 만들려면 다음 조건을 갖는다.

- Loose Coupling을 갖는 모듈이어야 한다. (독립성이 큰 모듈)
- Cohesion이 높은 모듈이어야 한다. (한 가지 목적을 갖는 모듈)

즉, 자연스럽게 TDD를 하게 되면,
개발자가 테스트가 쉽도록 모듈화를 하게 되고,
이 모듈은 자연스럽게 낮은 결합도, 높은 응집도를 갖게 된다. (by 켄트백)

버그율 감소

작은 테스트 코드를 더 많이 생성하고, 돌리기에

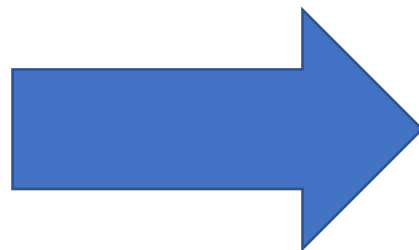
- 쉬운 버그를 초기에 잡을 수 있다.
- 팀 전체 버그율이 감소

Code Coverage 확보

TDD를 통해 약 80% 이상의 코드 커버리지 확보 가능하다.

Code Coverage

- 테스트 프로그램을 돌렸을 때,
코드 라인 별, 몇 % 정도 테스트가
되었는가 ?



```
        lstrcat(arExt, ",");
        if (strstr(arExt, ext+1) != 0) {
            ID=i;
            break;
        }
    }

    Ae.SetParser(ID);
    Option.SetStyleColor(Ae.GetParser());
} = end SelectParser =

void Relayout()
{
    RECT crt;
    int t,s,f,o,w,ch;
    int fh;

    GetClientRect(g_hFrameWnd, &crt);
    GetChildSize(t,s,f,o,w,ch);

    if (Option.bShowToolBar) {
        SendMessage(hToolBar, TB_AUTOSIZE, 0, 0);
        ShowWindow(hToolBar, SW_SHOW);
    } else {
        ShowWindow(hToolBar, SW_HIDE);
    }

    if (Option.bShowStatus) {
        SendMessage(hStatus, WM_SIZE, 0, 0);
        ShowWindow(hStatus, SW_SHOW);
        SetStatusPart();
    } else {
        ShowWindow(hStatus, SW_HIDE);
    }

    if (Option.bShowFileTab) {
        MoveWindow(hTabFrame, 0, t, crt.right, f, TRUE);
        ShowWindow(hTabFrame, SW_SHOW);
    } else {
```

TDD를 하면 프로그래밍이 재미있어진다.

작은 코드를 구현하고, PASS도 눈으로 보고
재미있다. 성취감도 느낀다.

그리고 PASS들을 보면,
불안감이 줄어들면서 자신감이 더 생긴다.

코딩이 재밌어진다.

그리고 심리적 안정감을 얻는다.

TDD에 부정적인 입장.

TDD 단점

TDD 단점

TDD에 대한 부정적인 입장

테스트 자체는 중요하나,
무리한 Unit Test 작성에 걸리는 비용이 크다. 라는 의견이다.

이는 두 가지로 나눌 수 있다.

1. Unit Test 만드는데 어려운 경우
2. 작은 Unit Test 가 너무 많은 경우

Unit Test 만들기 어려운 모듈이 있다.

GUI 개발 시,
병렬 프로그래밍 개발 시,
네트워크 관련 프로그래밍 개발 시,
외부 API 다루는 프로그래밍 개발 시,

TDD를 위해 Unit Test를 억지로 만드는데, 그 비용이 너무 크다.

반론의견

Unit Test 노하우가 쌓이면,
그 비용은 점차 줄어든 수 있다.

Unit Test가 너무 많다.

Baby Step 으로 이뤄진 Unit Test가 너무 많다.

- Unit Test 만드는 시간이 아깝다고 느껴진다.

변경으로 인한, Unit Test 유지보수 시간이 아깝다.

- 3,000개 Unit Test 중, Production Code가 크게 변경되어
500개 Unit Test에서 Fail이 발생할 경우, 500개 Unit Test 유지보수 해주어야 한다.

반론의견

한 프로젝트가 3,000 개의 Unit Test를 보유했다고 한 경우,
객관적으로 품질을 위한 안전 장치를 가졌다고 할 수 있다.
□ 따라서 유지보수 시간을 들일만하다고 생각할 수 있다.

TDD 단점이면서, 동시에 장점

TDD 결과로 수 많은 Unit Test가 만들어진다.

- 개발 비용이 많이 더 투입이 된다.
- 대신, 품질에 대한 안정성을 더 확보할 수 있다.

매우 높은 품질 & 신뢰성이 요구되는 대규모 프로젝트에는
TDD의 단점보다 장점이 더 돋보인다.

TDD 기본 예제, TDD 한번 더 해보기

TDD 해보기 : 은행계좌 KATA

은행 계좌 클래스

기능

1. 입금 / 출금
2. 잔고 조회

TDD 첫 단계

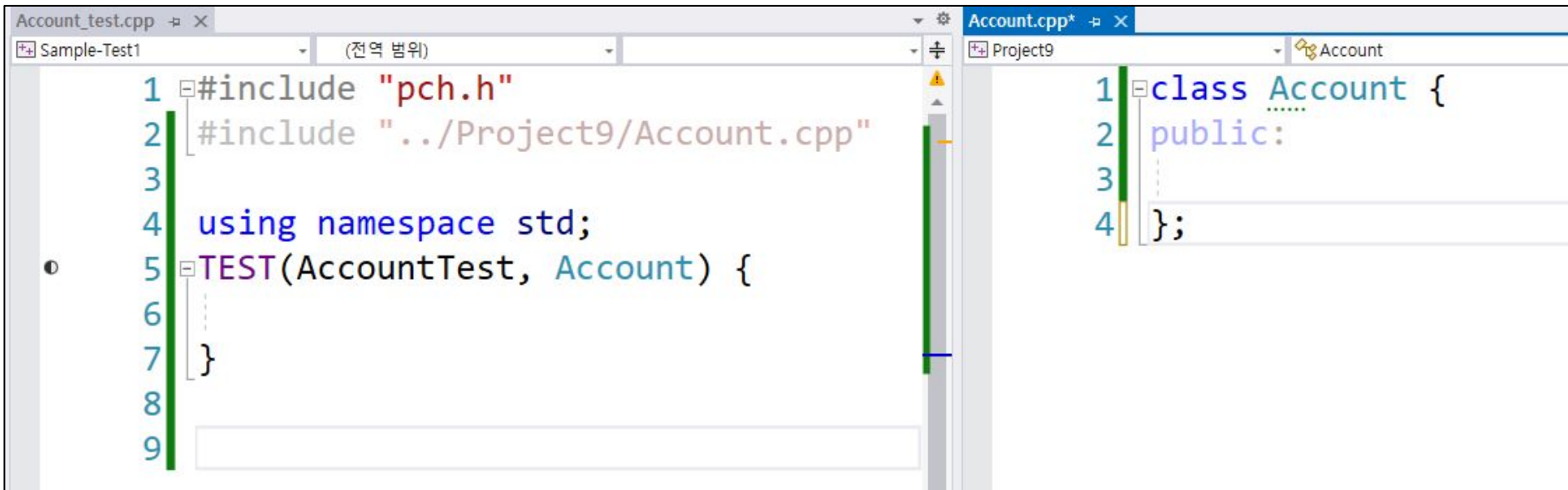
실패하기

- 내가 무엇을 할지 고민한다.
- 작성하고자 하는 메서드 이름을 결정한다.
- 모듈에 Input(Parameter)값과 Output(Return) 값을 결정한다.
- 스켈레톤 코드를 구현하고 시작해도 좋다.

기본코드 작성 & 창 배치

왼쪽 : Account_test.cpp

오른쪽 : Account.cpp



The screenshot shows a code editor with two files open. The left file, `Account_test.cpp`, contains the following code:

```
1 #include "pch.h"
2 #include "../Project9/Account.cpp"
3
4 using namespace std;
5 TEST(AccountTest, Account) {
6     ...
7 }
8
9
```

The right file, `Account.cpp`, contains the following code:

```
1 class Account {
2 public:
3     ...
4 };
```

Step 1. 어떤 작업을 할까? 를 결정

Red 단계는 내가 지금 당장 할 일(Todo)를 기록하는 것이다.

- 계좌를 생성하면, 10000원이 기본적으로 생성된다.
- `getBalance()` 으로 현재 계좌 값을 확인할 수 있다.

```
TEST(TestCaseName, TestName) {  
    Account account;  
    int ret = account.getBalance();  
    EXPECT_EQ(10000, ret);  
}
```

빌드가 안되어도, Fail 단계가 맞다.

Green

ToDo 작업 완료

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

```
Account_test.cpp
5 TEST(AccountTest, Account) {
6     Account account;
7     int ret = account.getBalance();
8     T_EQ(ret, 10000);
9 }
10
```

Create member function 'getBalance'

Split declaration and assignment

```
Account.cpp
class Account {
public:
    int getBalance() {
        return 10000;
    }
};
```

✓ AccountTest (1 test)	Success
✓ Account	Success

리팩토링 단계

이제 더 깔끔한 코드를 위해 고민을 한다.

좋은 네이밍으로 변경하였다.

- val □ balance
- private 적용

```
class Account {  
public:  
    int getBalance() {  
        return 10000;  
    }  
};
```

Refactor This
Extract Method Ctrl+R, Ctrl+M
Introduce Variable
Introduce Field



ReSharper – Introduce Field

Specify name
of the introduced field and define

Name: **balance**

Initialize field in:

- ☐ Current member
- ☒ Field initializer
- ☐ Constructor(s)



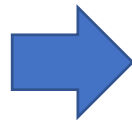
```
class Account {  
public:  
    int getBalance() {  
        return balance;  
    }  
private:  
    int balance = 10000;  
};
```

✓ AccountTest (1 test)	Success
✓ Account	Success

리팩토링 단계

- ✓ 더 가독성 좋은 테스트 케이스명으로 변경하였다.
- ✓ 더 이상 리팩토링 할 것이 없으면,
다시 다음 작업을 위해 **ToDo (Red)** 단계로 넘어간다.

```
TEST(AccountTest, Account) {  
    Account account;  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10000);  
}
```



```
TEST(AccountTest, CreateAccountInit10000won) {  
    Account account;  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10000);  
}
```

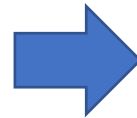
▲ ✓ AccountTest (1 test)	Success
✓ CreateAccountInit10000won	Success

Step 2. 다음 할일 정하기

계좌 생성시, 초기 입금 비용을 결정하도록 한다.

- 생성자에 초기 값을 넣을 수 있도록 한다.

```
TEST(AccountTest, CreateAccountInit10000won) {  
    Account account;  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10000);  
}
```



```
TEST(AccountTest, CreateAccountInit10000won) {  
    Account account(10000);  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10000);  
}
```

기존 작성했던 Unit Test 생성자에 10000 을 대입한다.

Green

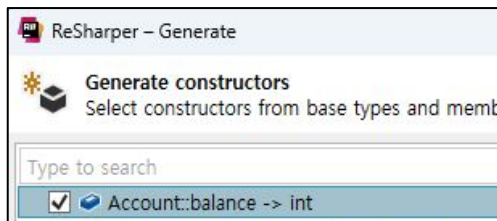
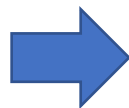
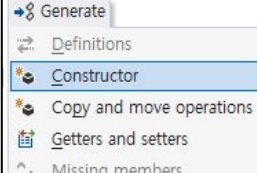
ToDo 작업 완료

빠르게 작업을 끝냈다.

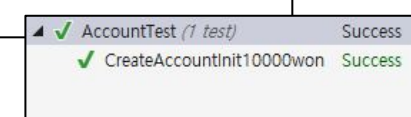
Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

```
class Account {  
public:  
  
    Balance() {  
        return balance;  
    }  
};
```



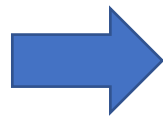
```
class Account {  
public:  
    explicit Account(int balance)  
        : balance(balance) {  
    }  
  
    int getBalance() {  
        return balance;  
    }  
private:  
    int balance = 10000;  
};
```



리팩토링 단계

필요 없는 초기화를 제거했음

```
class Account {  
public:  
    explicit Account(int balance)  
        : balance(balance) {  
    }  
  
    int getBalance() {  
        return balance;  
    }  
  
private:  
    int balance = 10000;  
};
```



```
class Account {  
public:  
    explicit Account(int balance)  
        : balance(balance) {  
    }  
  
    int getBalance() {  
        return balance;  
    }  
  
private:  
    int balance;  
};
```

Step 3. 할일 정하기

입금 구현하기

- deposit(금액) : 현재 계좌에 금액을 추가한다.

```
TEST(AccountTest, Deposit) {  
    Account account(balance: 10000);  
    account.deposit(500);  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10500);  
}
```

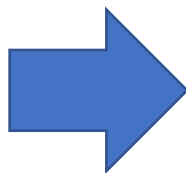
ToDo 작업 완료

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

```
class Account {  
public:  
    explicit Account(int balance)  
        : balance(balance) {  
    }  
  
    int getBalance() {  
        return balance;  
    }  
  
    void deposit(int i) {  
        balance += i;  
    }  
  
private:  
    int balance;  
};
```



▲ ✓ AccountTest (2 tests)	Success
✓ CreateAccountInit10000won	Success
✓ Deposit	Success

리팩토링

money로 네이밍 변경

리팩토링 수행 후,
Test를 한번 한다.

```
class Account {  
public:  
    explicit Account(int money)  
        : balance(money) {  
    }  
  
    int getBalance() {  
        return balance;  
    }  
  
    void deposit(int money) {  
        balance += money;  
    }  
  
private:  
    int balance;  
};
```

▲ ✓ AccountTest (2 tests)	Success
✓ CreateAccountInit10000won	Success
✓ Deposit	Success

Step 4. 할일 정하기

출금하기 구현하기

- withdraw(금액) : 현재 계좌에 금액을 뺀다.

```
TEST(AccountTest, Withdraw) {  
    Account account(money: 10000);  
    account.withdraw(600);  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 9400);  
}
```

Green

ToDo 작업 완료

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

✓ AccountTest (3 tests)	Success
✓ CreateAccountInit10000won	Success
✓ Deposit	Success
✓ Withdraw	Success

```
class Account {
public:
    explicit Account(int money)
        : balance(money) {
    }

    int getBalance() {
        return balance;
    }

    void deposit(int money) {
        balance += money;
    }

    void withdraw(int i) {
        balance -= i;
    }

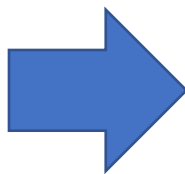
private:
    int balance;
};
```

리팩토링

변수 이름을 변경한다.

- i 를 money 로 변경

```
void withdraw(int i) {  
    balance -= i;  
}
```



```
void withdraw(int money) {  
    balance -= money;  
}
```

리팩토링

ret 임시변수를

모두 inline variable 시키자. (수동으로 작업한다.)

✓ AccountTest (3 tests)	Success
✓ CreateAccountInit10000won	Success
✓ Deposit	Success
✓ Withdraw	Success

```
TEST(AccountTest, CreateAccountInit10000won) {  
    Account account(money: 10000);  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10000);  
}  
  
TEST(AccountTest, Deposit) {  
    Account account(money: 10000);  
    account.deposit(money: 500);  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 10500);  
}  
  
TEST(AccountTest, Withdraw) {  
    Account account(money: 10000);  
    account.withdraw(i: 600);  
    int ret = account.getBalance();  
    EXPECT_EQ(ret, 9400);  
}
```



```
TEST(AccountTest, CreateAccountInit10000won) {  
    Account account(money: 10000);  
    EXPECT_EQ(account.getBalance(), 10000);  
}  
  
TEST(AccountTest, Deposit) {  
    Account account(money: 10000);  
    account.deposit(money: 500);  
    EXPECT_EQ(account.getBalance(), 10500);  
}  
  
TEST(AccountTest, Withdraw) {  
    Account account(money: 10000);  
    account.withdraw(i: 600);  
    EXPECT_EQ(account.getBalance(), 9400);  
}
```

리팩토링

중복코드를 제거하기 위해,
Test Fixture를 준비하자.

```
TEST(AccountTest, CreateAccountInit10000won) {  
    Account account(money: 10000);  
    EXPECT_EQ(account.getBalance(), 10000);  
}  
  
TEST(AccountTest, Deposit) {  
    Account account(money: 10000);  
    account.deposit(money: 500);  
    EXPECT_EQ(account.getBalance(), 10500);  
}  
  
TEST(AccountTest, Withdraw) {  
    Account account(money: 10000);  
    account.withdraw(i: 600);  
    EXPECT_EQ(account.getBalance(), 9400);  
}
```

```
class AccountFixture : public testing::Test {  
public:  
    Account account{ money: 10000 };  
};
```

Test 코드 상단에 위 코드를 추가한다.

리팩토링

Test Fixture를 적용한다.

```
class AccountFixture : public testing::Test {
public:
    Account account{ money: 10000 };
};

TEST_F(AccountFixture, CreateAccountInit10000won) {
    EXPECT_EQ(account.getBalance(), 10000);
}

TEST_F(AccountFixture, Deposit) {
    account.deposit(money: 500);
    EXPECT_EQ(account.getBalance(), 10500);
}

TEST_F(AccountFixture, Withdraw) {
    account.withdraw(i: 600);
    EXPECT_EQ(account.getBalance(), 9400);
}
```

▲ ✓ AccountFixture (3 tests)	Success
✓ CreateAccountInit10000won	Success
✓ Deposit	Success
✓ Withdraw	Success

[참고] 로버트 C 마틴의 TDD 세 가지 Rule.

켄트백과 차이가 있음

로버트 C 마틴의 세 가지 규칙

1. 실패하는 UnitTest를 작성할 때까지, Production Code를 작성하지 않는다.
2. **Compile은 실패하지 않으면서**, 실행이 실패하는 정도로만 Unit Test를 작성한다.
3. 현재 실패하는 UnitTest에 통과될 정도로만 실제 코드를 작성한다.

세 가지 규칙을 따르면 개발과 테스트가 대략 30초 주기로 묶인다.

이 방식대로 라면, 수천개에 달하는 Test Case가 나오지만, 관리 문제를 유발하기도 한다.

[도전] TDD로 다음 기능을 개발해본다.

1. 5% 복리 적용하기

- 5% 복리가 적용되어 금액이 올라가는 메서드

2. 은행 이자 Setter로 만들기

- 원하는 은행 이자로 지정하는 기능 추가.

3. 은행 이자율로 복리 적용하기

- 1번에서 구현했던 기능을 수정하여 개발

5. n년 이후, 예상 복리 금액 알려주기 기능 추가.

- n년 동안 복리 이자를 적용하면 얼마가 되는지 알려주는 기능 추가하기