

프로그래밍 역량 강화 전문기관, 민코딩

클린코드 - 2부



목차

1. Comment
2. Formatting
3. Unit Test

주석

Comment

주석은 유지보수 되지 않는다.

주석은 유지보수 되지 않는다.

- 동작에 영향을 끼치지 않기 때문
- 코드 작성자가 아닌 사람이,
다른사람이 작성한 주석을 건드리지 않음
- 주석을 작성한 이후,
코드는 추가되었지만, 주석에 설명이 빠진다.

유지보수를 하다보면
부정확해지는 주석이 있는 것 보다는,
없는 것이 더 좋다.

두 코드를 볼 때, 독자를 위한 코드는?

```
// 직원이 복지 혜택을 받을 조건이 되는지 검사한다.  
if ((employee.flags & HOURLY_FLAG) &&  
    (employee.age > 65))
```

```
if (employee.isPossible복지혜택( ))
```

메서드명으로 표현될 수 있는 경우

주석대신 메서드로 설명이 가능하면, 메서드로 표현을 권장

```
// 직원이 복지 혜택을 받을 조건이 되는지 검사한다.  
if ((employee.flags & HOURLY_FLAG) &&  
    (employee.age > 65))
```

```
if (employee.isPossible복지혜택( ))
```

피하면 좋을 주석 1

팀에서 정한 의무적으로 다는 주석

- 모든 함수에 Javadocs를 달거나,
모든 변수에 주석을 달아야 한다는 규칙은 피하는 것이 좋다.

이력 남기기

- 주석으로 날짜별 이력 남기는 경우, 유지보수 안됨

주저리 주저리 주석

- 같은말 반복하고,
이해하는데 큰 도움 안되는 의미 없는 주석

피하면 좋을 주석 2

저자를 표시하는 주석

- `//이런 기능 내가 추가함 by inho.choi ^^v`
- 버전관리도구를 믿고, 남기지말자.

주석으로 처리한 코드

- 타인이 보기에는 무언가 사연이 있는 듯하여, 지울 수 없다.
- 버전관리도구를 믿고, 남기지말자.

괄호에 다는 주석

- `} //while`
- `} //for`

[도전] 주석 정리하기

필요하다고 생각하는 주석은 남기고,
불필요한 주석은 삭제하자.

- <https://gist.github.com/mincoding-jh/a56cec963f8f8a4489a3bdc166202785>

```
1 // Copyright (C) emilybache, Inc. All right reserved.
2 // https://github.com/emilybache/GildedRose-Refactoring-Kata
3
4 // History
5 //-----
6 // 230910 : 판매기한 조건 추가
7 // 230101 : 새로운 아이템 추가
8 // 220104 : 신규 코드 작성
9
10
11
12 #include <string>
13 #include <vector>
14
15 using namespace std;
16
17 class Item
18 {
19 public:
20     string name; // 이름
21     int sellIn; // 판매기한
22     int quality; // 가격
23
24     // 생성자 (이름, 판매기한, 가격) 입력된 파라미터로 초기화를 한다.
25     Item(string name, int sellIn, int quality) : name(name), sellIn(sellIn), quality(quality)
26     {}
27 };
28
29 class GildedRose
30 {
31 public:
32     vector<Item> items;
33
34     // 생성자
35     GildedRose(vector<Item> &items) : items(items)
36     {}
37
38     void updateQuality()
39     {
40         for (int i = 0; i < items.size(); i++)
41         {
42             // Aged Brie, Backstage Pass 등은 아닌 아이템이라면
43             if (items[i].name != "Aged Brie" && items[i].name != "Backstage passes to a TAFKAL80ETC concert")
44             {
```

Formatting

뉴스 기사처럼

뉴스 기사

질서 정돈된 뉴스 기사는
독자들이 필요한 내용을 빠르게 이해할 수 있다.



뉴스 기사의 장점을 소스코드에 적용

Title 은 메서드 명으로 비유할 수 있다.

- 어떤 내용인지 파악할 수 있도록한다.
- 반전이 없도록한다.

타이틀로 설명이 부족하다면,
주석으로 부제를 적절히 넣어주자.



질서 정돈

코드를 읽는 사람들을 위해
질서와 정돈속에서 소스코드를 구현한다.

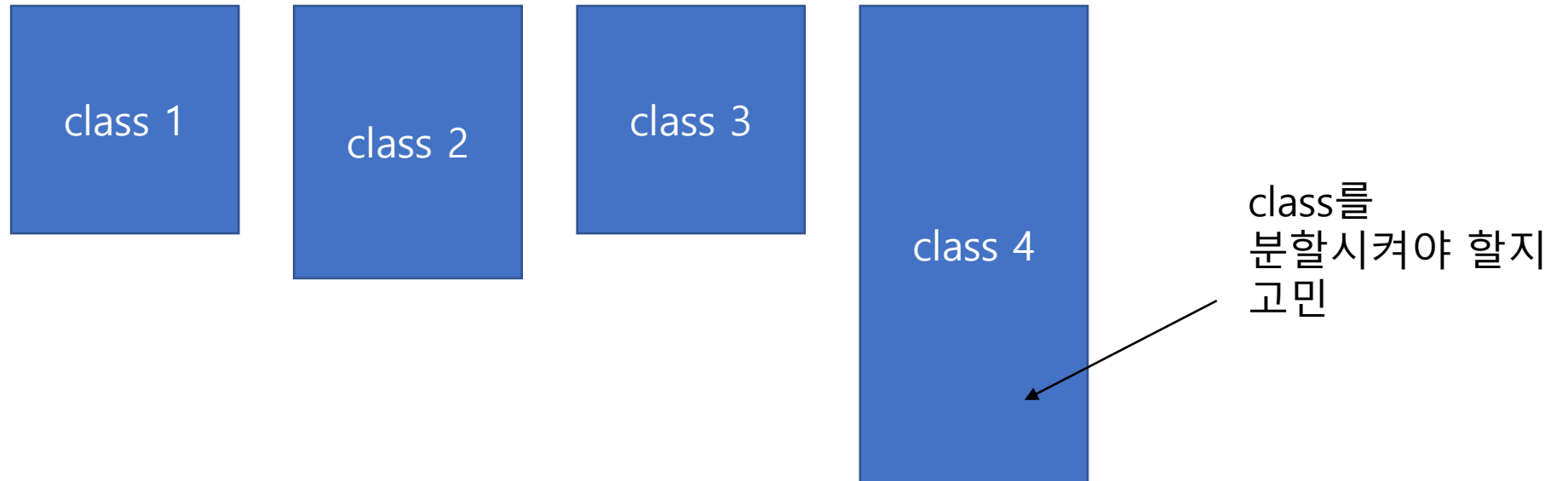
- ✓ 코드 Format에 대한 팀 규칙을 정하고,
- ✓ 그 규칙을 착실히 따르고,
- ✓ IDE에 그 규칙을 지정하는 것을 권장한다.



이번 챕터는 코드 내용 보다는,
코드의 질서 & 정돈에 관한 권장 규칙

Class 의 크기가 무리하게 크지 않도록 한다.

클래스 파일의 크기가 유별나게 크지 않도록 한다.



소스코드, 가로 스크롤

모니터 사이즈는 다들 다르지만,
일반적인 화면 크기에서 스크롤을 넘기지 않도록 한다.

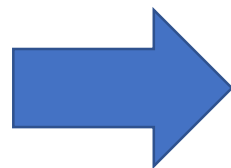
ex) 120 글자 수준

소스코드 소스코드 소스코드 소스코드 소스코드 소스코드 소스코드...
소스코드

정렬

소스코드를 세로 라인으로 정렬하려고 노력하지 않는다.
끊임없이 IDE 자동 기능과 싸워야한다.

```
int      sock;  
Socket   socketInstance;  
OutputStream outStream;  
InputStream inputStream;
```



```
int sock;  
Socket socketInstance;  
OutputStream outStream;  
InputStream inputStream;
```


글작성시 단락의 중요성

단락 : 내용상 끝은 하나의 토막

- 각 단락별로 엔터로 구분해야, 독자들이 빠르게 읽기 좋다.

어쩌면 산문이 그 사람의 글쓰기 실력을 더 적나라하게 잘 보여주는 것 같다.

어떻게 할까.

나는 고민한다. 일단 내 방식대로 편집하고 나서, 교정본을 보낼까, 아니면 일단 작가의 초집해서 그냥 보내버릴까.

엔터 친 곳을 모두 붙이면서, 붙어 있는 곳을 모두 엔터 치면서 수제비 반죽 뜨듯 적당한 크이를 만드는 일, 글을 몇 번이나 다시 읽어야 한다.

1안은 나의 수고를 전제로 해야하는 것이고, 2안은 눈감아버리는 것이지만, 오랜 경험상 전해서 보내든, 후자를 편집해서 보내든 작가는 생각보다 관심이 없다.

왜냐고? 작가 눈에는 모든 글이 예뻐보이니까!

책을 기다리는 작가는, 글을 보는 것이 아니라, (안성필) 책을 본다. 책을 준비하는 편집자는 기 전에 (완성시켜야 하는)글을 본다.

글이 완성되어야 다음 페이지로 넘어갈 수 있다. 원고가 출판사로 넘어가는 순간부터, 이제 편집자가 더 글을 많이 읽고 더 고민하게 된다.

시나 소설을 쓰는 작가들이 출판사에 오래 재직하지 못하고 방향하는 이유가 바로 여기에 있다. 글쓰기할 에너지를 편집에 다 쏟기 때문이다.

물론 어느 정도 이골(?)이 난 베테랑들은 본인의 글쓰기에 전혀 영향을 받지 않는다. 남의

이면서, 붙어 있는 곳을 모두 엔터 치면서 수제비 반죽 뜨듯 적당한 크기의 덩어리를 만드는 일, 글을 몇 번이나 다시 읽어야 한다. 1안은 나의 수고를 전제로 해야하는 것이고, 2안은 눈감아버리는 것이지만, 오랜 경험상 전자를 편집해서 보내든, 후자를 편집해서 보내든 작가는 생각보다 관심이 없다. 왜냐고? 작가 눈에는 모든 글이 예뻐보이니까! 책을 기다리는 작가는, 글을 보는 것이 아니라, (안성필) 책을 본다. 책을 준비하는 편집자는 책을 보기 전에 (완성시켜야 하는)글을 본다. 글이 완성되어야 다음 페이지로 넘어갈 수 있다. 원고가 출판사로 넘어가는 순간부터, 이제 작가보다 편집자가 더 글을 많이 읽고 더 고민하게 된다. 시나 소설을 쓰는 작가들이 출판사에 오래 재직하지 못하고 방향하는 이유가 바로 여기에 있다. 본인 글쓰기할 에너지

작가의 글을 편집하다가, 이마를 손으로 짚는 경우가 종종 있다. 비문이나 맞춤법, 띄어쓰기 등의 문제가 서둘러 수정하면 그만이지만, 이상한 곳에서 행과 연을 나누거나 단락이 나뉘져 있으면, 마우스 포인터가 부들부들 떠다. 어찌지,하고 말이다. 이상한 곳을 어떻게 아냐고? 간단하다. 직접 소리 내어 읽어보면 누구나 쉽게 안다. 한 덩어리로 읽어야 할 곳이 띄어져 있거나, 숨차도록 뻑뻑하게 글로 이어져 있다면, 고개를 갸웃할 수밖에 없다. 이유는 아마 다음의 두 가지 중 하나일 것이다. 글의 흐름을 작가가 컨트롤하고 있지 못하거나, 작가가 전혀 의식하지 못하거나. 시의 경우, 의도적인 행갈이와 연결이가 있으니 '나름' 존중해야겠지만, 산문의 경우는 다르다. 어쩌면 산문이 그 사람의 글쓰기 실력을 더 적나라하게 잘 보여주는 것 같다.

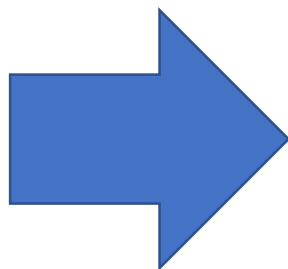
어떻게 할까. 나는 고민한다. 일단 내 방식대로 편집하고 나서, 교정본을 보낼까, 아니면 일단 작가의 초고대로 편집해서 그냥 보내버릴까. 엔터 친 곳을 모두 붙이면서, 붙어 있는 곳을 모두 엔터 치면서 수제비 반죽 뜨듯 적당한 크기의 덩어리를 만드는 일, 글을 몇 번이나 다시 읽어야 한다. 1안은 나의 수고를 전제로 해야하는 것이고, 2안은 눈감아버리는 것이지만, 오랜 경험상 전자를 편집해서 보내든, 후자를 편집해서 보내든 작가는 생각보다 관심이 없다. 왜냐고? 작가 눈에는 모든 글이 예뻐보이니까!

책을 기다리는 작가는, 글을 보는 것이 아니라, (안성필) 책을 본다. 책을 준비하는 편집자는 책을 보기 전에 (완성시켜야 하는)글을 본다. 글이 완성되어야 다음 페이지로 넘어갈 수 있다. 원고가 출판사로 넘어가는 순간부터, 이제 작가보다 편집자가 더 글을 많이 읽고 더 고민하게 된다. 시나 소설을 쓰는 작가들이 출판사에 오래 재직하지 못하고 방향하는 이유가 바로 여기에 있다. 본인 글쓰기할 에너지

개행

여러줄의 묶음은 완결된 하나의 생각을 나타낸다.

```
class BoldWidget : public ParentWidget {
public:
    static const string REGEXP;
    static const regex pattern;
    BoldWidget(ParentWidget* parent, const string& text) : ParentWidget(parent) {
        smatch match;
        regex_search(_Str: text, &_Matches: match, _Re: pattern);
        addChildWidgets(text: match[1]);
    }
    string render() {
        string html = "<b>";
        html.append(_Right: childHtml()).append(_Ptr: "</b>");
        return html;
    }
};
```



```
class BoldWidget : public ParentWidget {
public:
    static const string REGEXP;
    static const regex pattern;

    BoldWidget(ParentWidget* parent, const string& text) : ParentWidget(parent) {
        smatch match;
        regex_search(_Str: text, &_Matches: match, _Re: pattern);
        addChildWidgets(text: match[1]);
    }

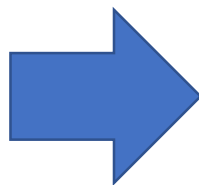
    string render() {
        string html = "<b>";
        html.append(_Right: childHtml()).append(_Ptr: "</b>");
        return html;
    }
};
```

세로 밀집도

연관성이 있는 내용은 붙여주는 것이 좋다.

- 의미없는 주석으로 인해, 단락이 끊어졌다.

```
class ReporterConfig {  
    /**  
     *The class name of the reporter listener  
     */  
private :  
    std::string m_className;  
  
    /**  
     * The properties of the reporter listener  
     */  
    std::vector<Property> m_properties = std::vector<Property>();  
public :  
    void addProperty(Property property) {  
        m_properties.push_back(property);  
    }  
};
```



```
class ReporterConfig {  
private :  
    std::string m_className;  
    std::vector<Property> m_properties = std::vector<Property>();  
  
public :  
    void addProperty(Property property) {  
        m_properties.push_back(property);  
    }  
};
```

개념의 유사도

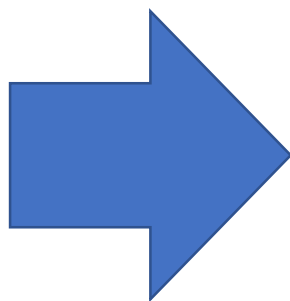
비슷한 개념의 메서드들은
함께 모여 배치하자.

```
class StringManipulator {  
private:  
    string str;  
public:  
    StringManipulator(string str) {  
        this->str = str;  
    }  
    int getLength() {  
        return str.length();  
    }  
  
    string toUpperCase() {  
        string result;  
        for (char c : str) {  
            result += toupper(_C: c);  
        }  
        return result;  
    }  
    string toLowerCase() {  
        string result;  
        for (char c : str) {  
            result += tolower(_C: c);  
        }  
        return result;  
    }  
};
```

들여쓰기

Line 수를 줄이기 위한 최고의 노력을 하지말자.

```
class CommentWidget :public TextWidget {  
public:  
    const string REGEXP = "^$[^\r\n]*(?:(:\r\n)|\n|\r)?";  
  
    CommentWidget(ParentWidget parent, string text) : TextWidget(parent, text) { }  
    string render() { return ""; }  
};
```



```
class CommentWidget :public TextWidget {  
public:  
    const string REGEXP = "^$[^\r\n]*(?:(:\r\n)|\n|\r)?";  
  
    CommentWidget(ParentWidget parent, string text)  
        : TextWidget(parent, text)  
    {  
    }  
    string render()  
    {  
        return "";  
    }  
};
```

[도전] Formatting 실습

읽기 좋은 형태로 맞추어보자.

- <https://gist.github.com/mincoding-jh/35a197cac161545b17bf0dc4ceb1e082>



Unit Test 개요

모듈 단위의 Testing

배경지식, Unit Test 란?

✓모듈 단위로, Testing을 하는 것

입력값을 넣어 보았을 때,

결괏값이 기대값과 동일하게 나오는지 확인한다.

✓주로 검증팀 보다는, 개발팀에서 Test를 수행



Unit Test 종류

xUnit

- 스몰토크의 SUnit 에서 파생된 테스트 프레임워크 이름
- 켄트백이 설계함

언어별 가장 인기있는 Unit Test Framework

- C++ : Google Test (xUnit 기반)
- java : junit (xUnit 기반)

Unit Test 도구 목록

- https://en.Wikipedia.org/wiki/List_of_unit_testing_frameworks

Google Test 란

Google의 Testing Technology 팀에서 만든 C++ Test Framework

- xUnit 아키텍처 기반
- Windows / Linux / Mac 등 다양한 플랫폼 사용 가능

Github & Document

- <https://github.com/google/googletest>
- <https://google.github.io/googletest/>

Visual Studio에서 Google Test 사용

네 가지 설치 방법



1. Test Adapter for Google Test

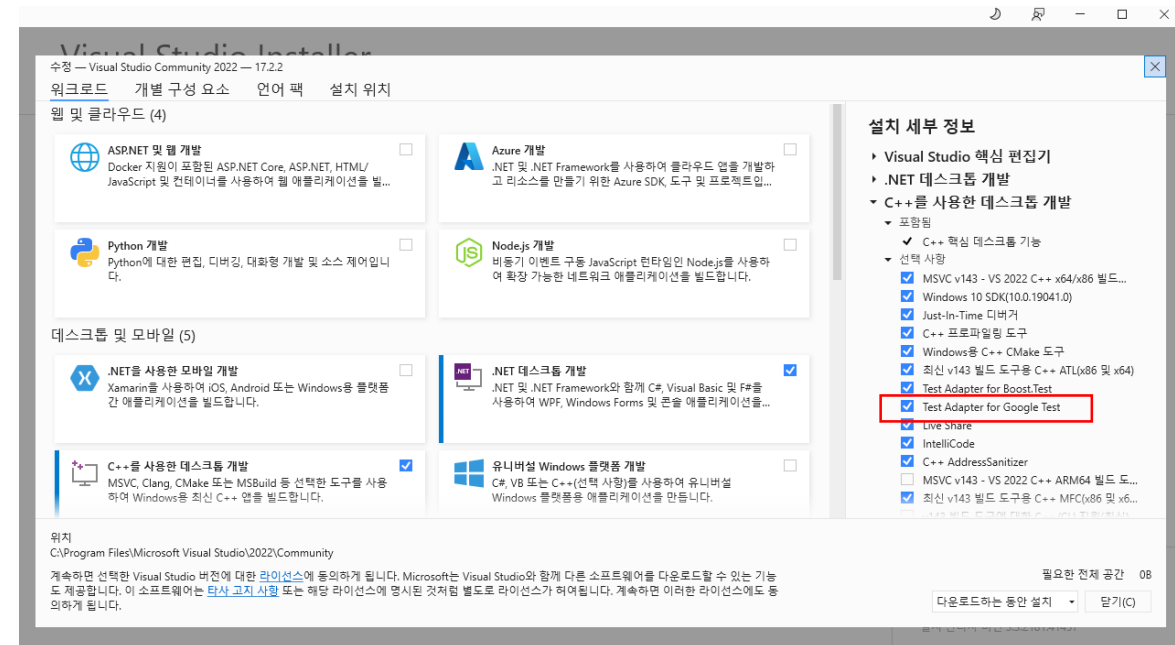
- Visual Studio 2017 15.5 이상 버전부터 공식 지원

2. Google Test Adapter

- 비공식, Visual Studio Marketplace 사용

3. Nuget 사용 (2014년 이후 Update 없음)

4. 수동 설치



Visual Studio 기본으로 내장

Test Adapter 주요기능

1. Test Discovery

- 작성한 Unit Test들을 자동으로 검색

2. Test Execution

- 쉽게 실행시켜줌

3. Test Debugging

- Unit Test Fail 시 Trace 기능 제공

4. GUI 기반 Test 환경

- Toolbar 제공 / 편리한 설정 / 빠른 결과 리포트 확인

main.cpp 작성

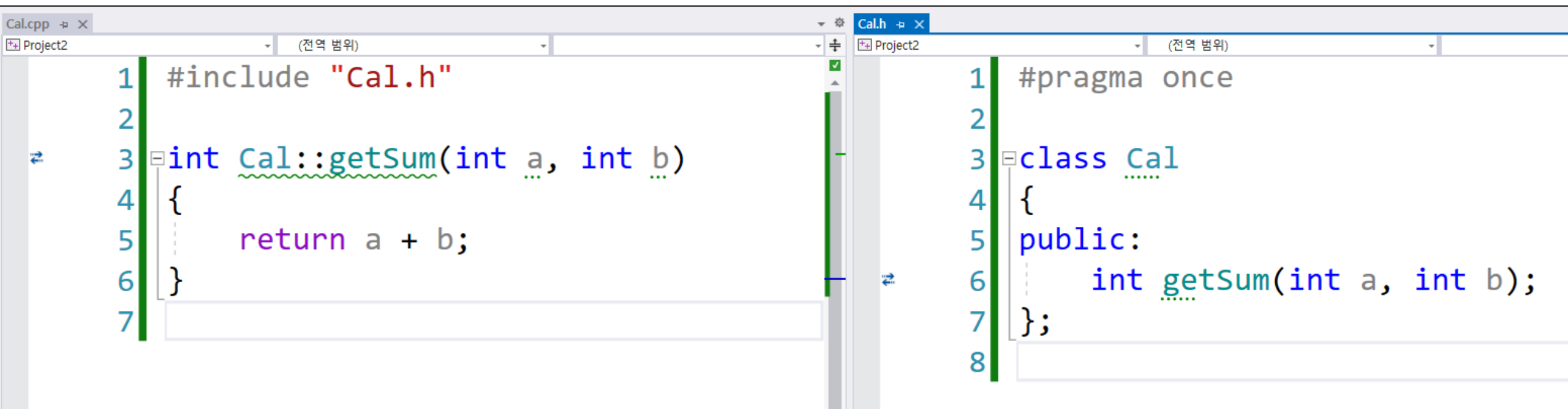
main.cpp 파일을 작성한다.

```
#include <iostream>

int main()
{
    return 0;
}
```

파일 추가

- ✓ sum.cpp / sum.h 작성
- ✓ 유의사항 : main.cpp 에 Cal Class를 만들지 말고, class 파일을 따로 만들자.
 - Google Test는 main함수가 포함된 파일을 include 하면 동작되지 않는다.



The image shows two side-by-side code editors. The left editor, titled 'Cal.cpp', contains the following code:

```
1 #include "Cal.h"
2
3 int Cal::getSum(int a, int b)
4 {
5     return a + b;
6 }
7
```

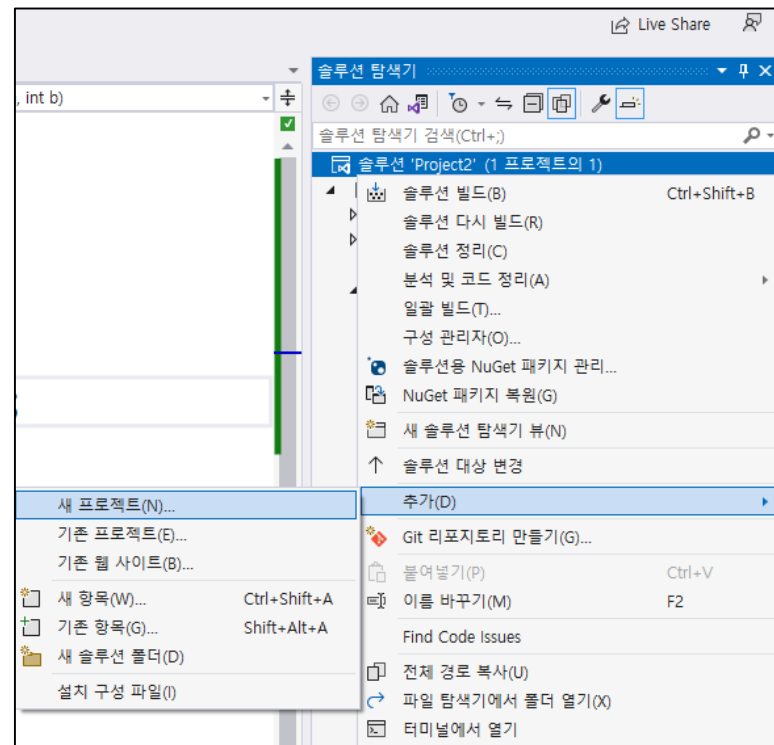
The right editor, titled 'Cal.h', contains the following code:

```
1 #pragma once
2
3 class Cal
4 {
5 public:
6     int getSum(int a, int b);
7 };
8
```

솔루션 내, Test Project 추가

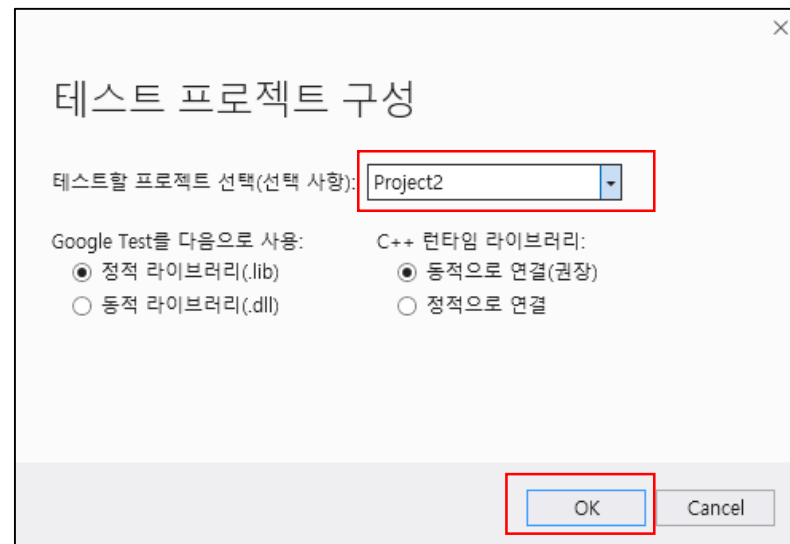
솔루션 내부, 새 프로젝트 추가하기

- ## ■ 새 프로젝트



Google Test 프로젝트 추가

프로젝트 생성시 검색어 : **test** 라고 검색하자.



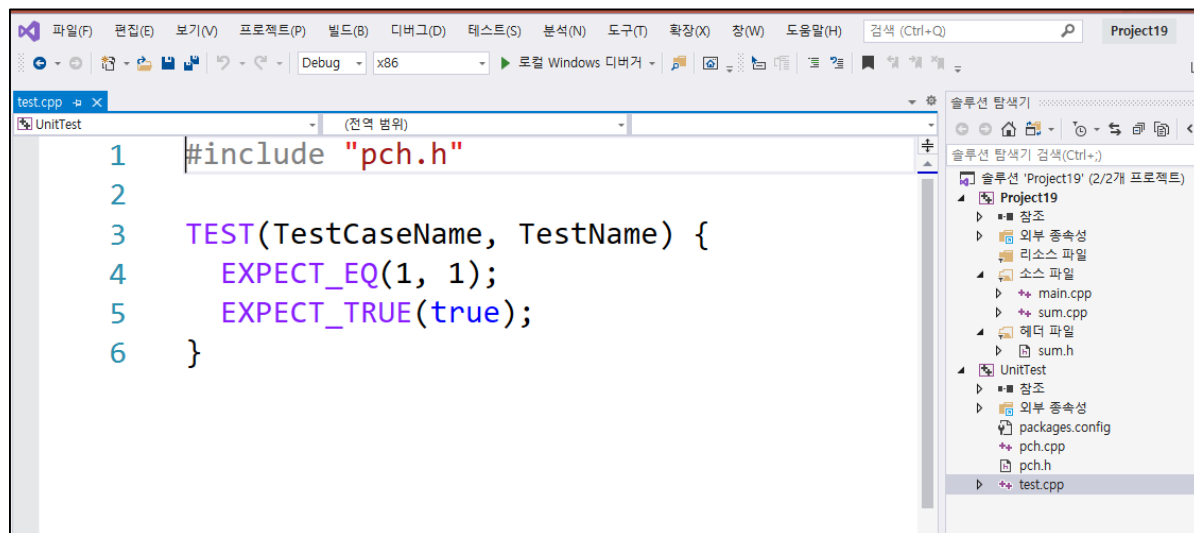
프로젝트 생성 완료

pch.cpp / pch.h

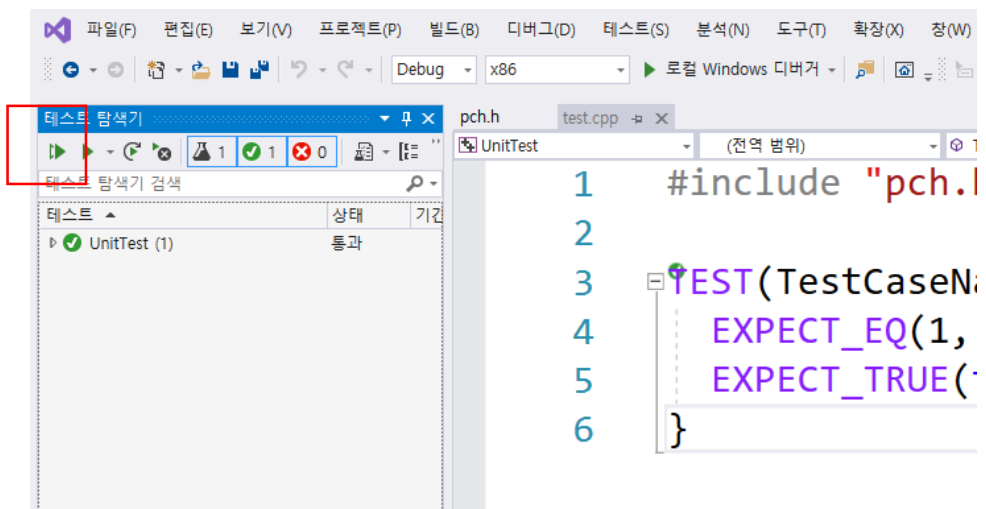
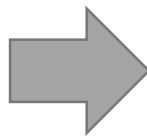
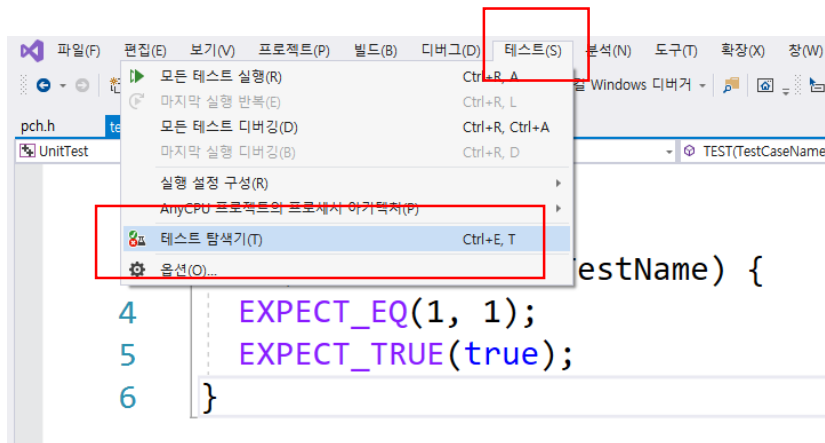
- Precompiled Header
- 변경사항이 없을 때는 빌드하지 않는 파일

test.cpp

- Unit Test 샘플 파일



테스트 탐색기 실행



Unit Test Code 작성하기

간단한 Unit Test Code 작성 후 Test 실행해보기

Unit Test Code 1

테스트할 cpp 파일 Include

```
#include "pch.h"
#include "../Project19/sum.cpp"

TEST(SimpleSum, Positive) {
    EXPECT_EQ(3, getSum(1, 2));
    EXPECT_EQ(10, getSum(5, 5));
}

TEST(TestCaseName, TestName) {
    EXPECT_EQ(1, 1);
    EXPECT_TRUE(true);
}
```

Unit Test Code 2

TEST(TestSuite , TestCase)

- 한 Test Suite에 여러개의 Test를 포함할 수 있음

```
#include "pch.h"
#include "../Project19/sum.cpp"

TEST(SimpleSum, Positive) {
    EXPECT_EQ(3, getSum(1, 2));
    EXPECT_EQ(10, getSum(5, 5));
}

TEST(TestCaseName, TestName) {
    EXPECT_EQ(1, 1);
    EXPECT_TRUE(true);
}
```

Visual Studio

1. Test Case Name
2. Test Name

Google Test은 이름 변경

1. Test Case Name → Test Suite
2. Test Name → Test Case

[참고] 이름 변경

이전 Google Test (Visual Studio 에서 사용중인 이름)

1. Test Case Name
2. Test Name

Test 표준에 맞추어 Google Test API 이름 변경

1. Test Suite
2. Test Case

Beware of the nomenclature

Note: There might be some confusion arising from different definitions of the terms *Test*, *Test Case* and *Test Suite*, so beware of misunderstanding these.

Historically, googletest started to use the term *Test Case* for grouping related tests, whereas current publications, including International Software Testing Qualifications Board (ISTQB) materials and various textbooks on software quality, use the term *Test Suite* for this.

The related term *Test*, as it is used in googletest, corresponds to the term *Test Case* of ISTQB and others.

The term *Test* is commonly of broad enough sense, including ISTQB's definition of *Test Case*, so it's not much of a problem here. But the term *Test Case* as was used in Google Test is of contradictory sense and thus confusing.

googletest recently started replacing the term *Test Case* with *Test Suite*. The preferred API is *TestSuite*. The older *TestCase* API is being slowly deprecated and refactored away.

출처 : <http://google.github.io/googletest/primer.html>

Unit Test Code 3

EXPECT_EQ(테스트 결과 값, 예상결과)

- EXPECT_EQ(Actual, Expected)
- 두 결과값이 같아야 PASS

```
#include "pch.h"
#include "../Project19/sum.cpp"

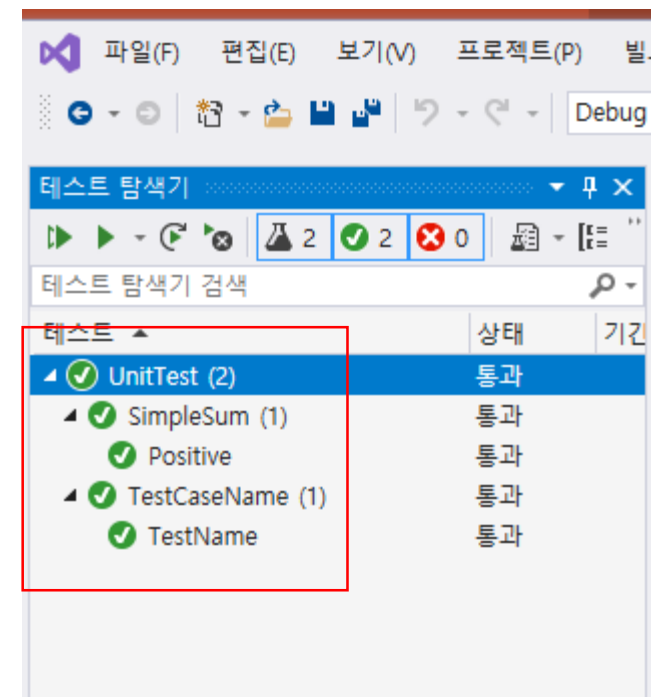
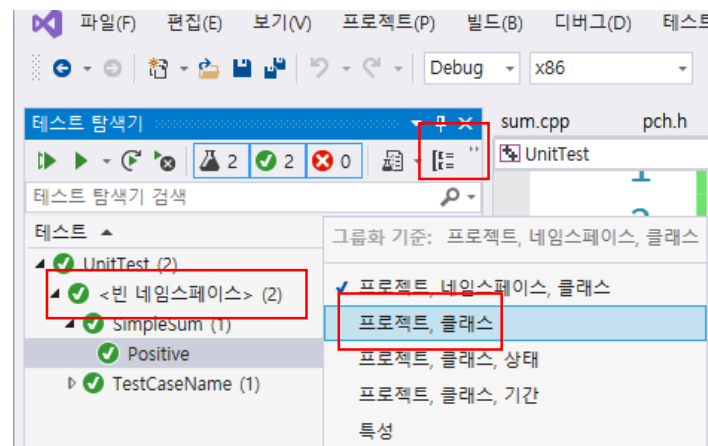
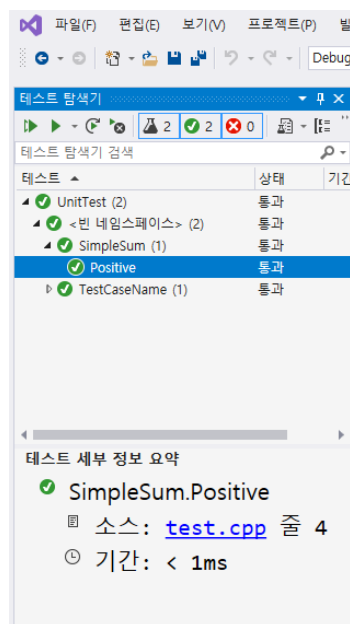
TEST(SimpleSum, Positive) {
    EXPECT_EQ(3, getSum(1, 2));
    EXPECT_EQ(10, getSum(5, 5));
}

TEST(TestCaseName, TestName) {
    EXPECT_EQ(1, 1);
    EXPECT_TRUE(true);
}
```

테스트 결과

PASS

- 결과를 보다 깔끔하게 보기 위해, 사용하지 않는 단계는 감춘다.



테스트 코드 추가

FAIL 발생하는 Unit Test 코드

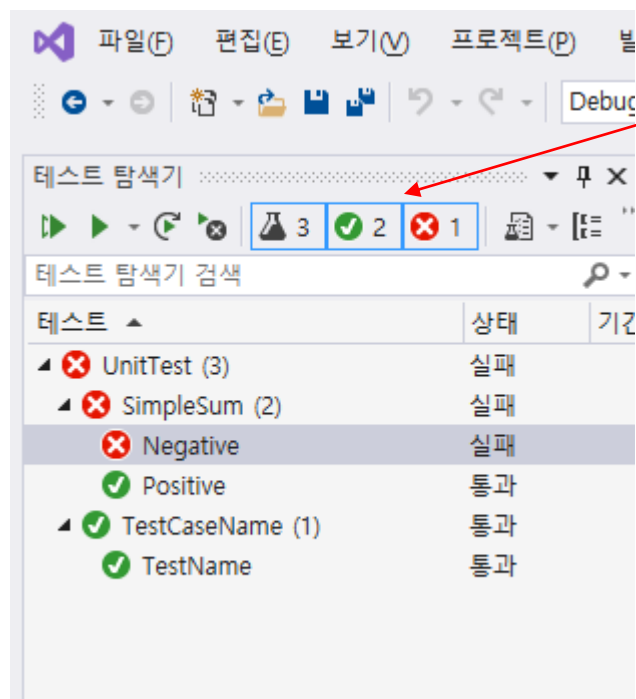
- TestName : Negative
- Test Code 디버깅을 위한 코드 작성

```
#include "pch.h"  
#include "../Project19/sum.cpp"
```

```
✓  
TEST(SimpleSum, Positive) {  
    EXPECT_EQ(3, getSum(1, 2));  
    EXPECT_EQ(10, getSum(5, 5));  
}
```

```
✗  
TEST(SimpleSum, Negative) {  
    int ret = getSum(1, 2);  
    EXPECT_EQ(15, ret);  
}
```

테스트 결과

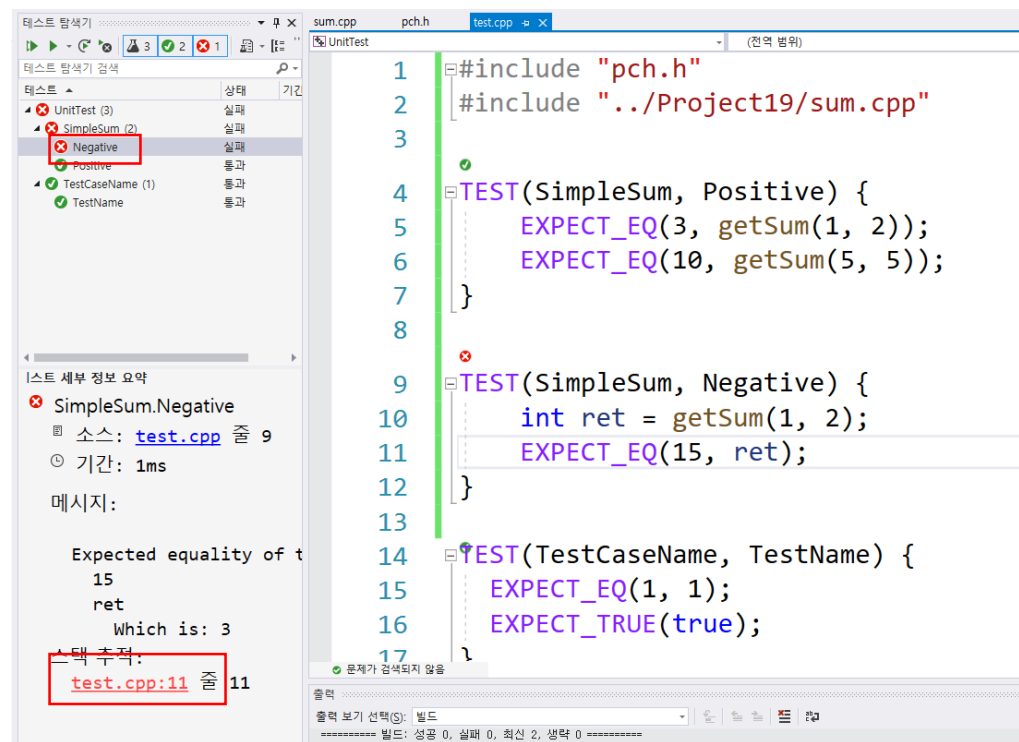


총 3개의 Unit Test 중
2개는 PASS
1개는 FAIL

Unit Test 디버깅 1

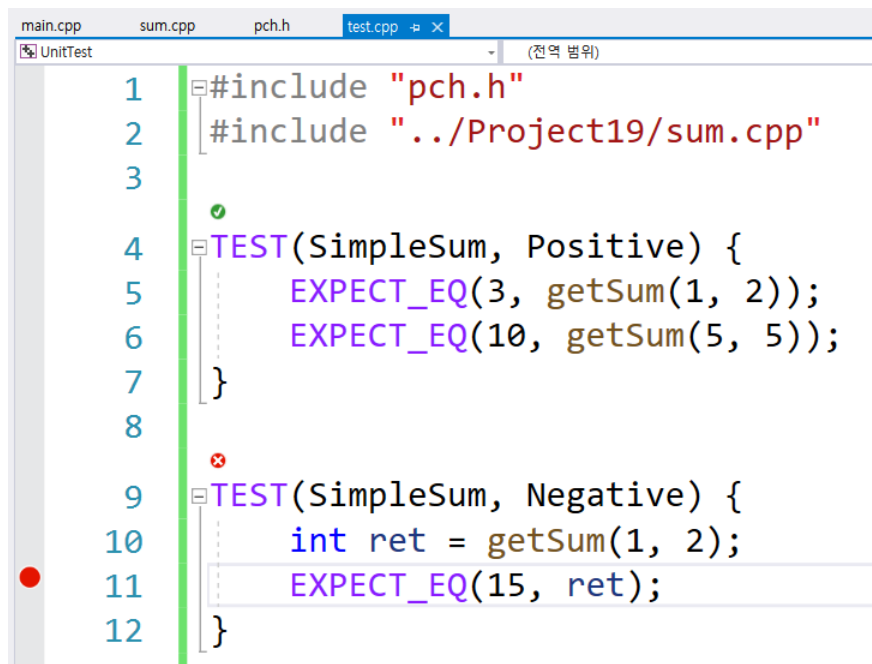
테스트 탐색기

- Fail 된 Test Name 클릭
- Fail이 발생한 Line 확인



Unit Test 디버깅 2

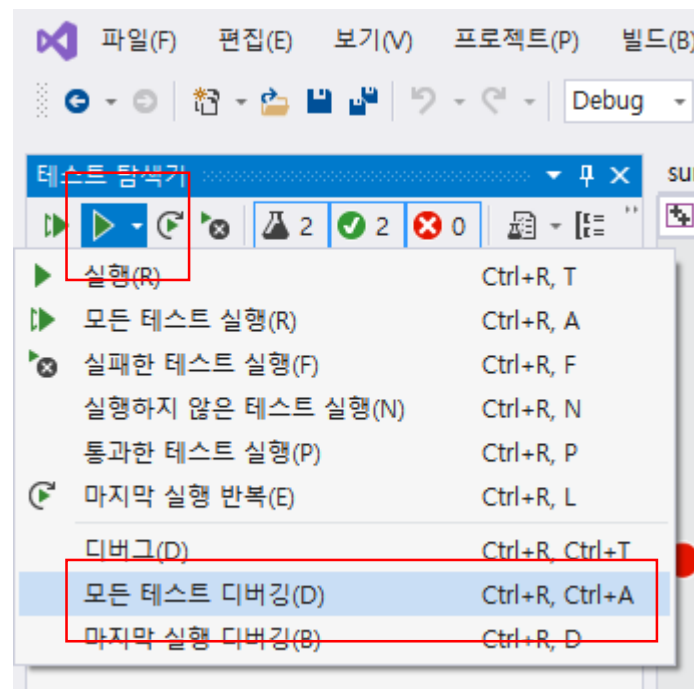
1. Test Code에 BreakPoint 를 건다. (F9)
2. 모든 테스트 디버깅



The screenshot shows a code editor with a file explorer at the top displaying 'main.cpp', 'sum.cpp', 'pch.h', and 'test.cpp'. The 'test.cpp' file is open, showing the following code:

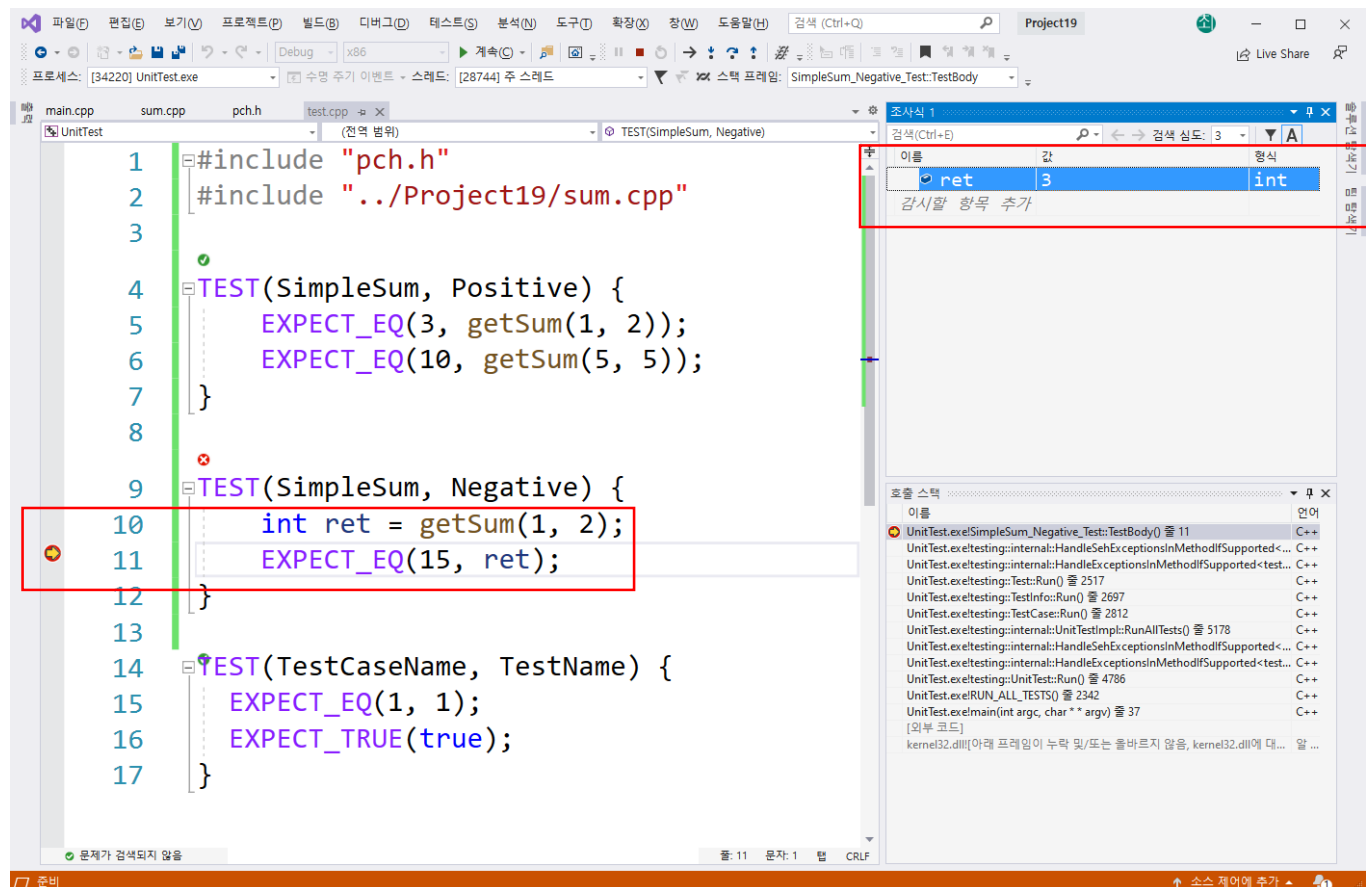
```
1 #include "pch.h"
2 #include "../Project19/sum.cpp"
3
4 TEST(SimpleSum, Positive) {
5     EXPECT_EQ(3, getSum(1, 2));
6     EXPECT_EQ(10, getSum(5, 5));
7 }
8
9 TEST(SimpleSum, Negative) {
10    int ret = getSum(1, 2);
11    EXPECT_EQ(15, ret);
12 }
```

A red dot indicating a breakpoint is placed on the left margin next to line 11. A green vertical line is positioned at the start of line 4.



Unit Test 디버깅 3

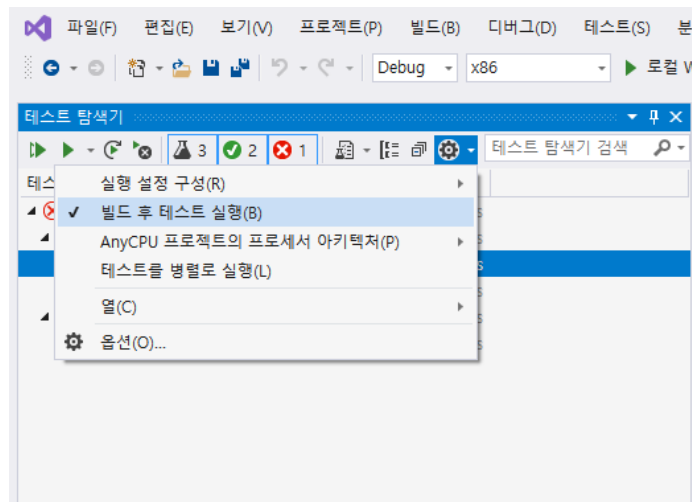
조사식 창을 사용하여
Unit Test Fail 분석 가능



자동 Unit Test 수행

테스트 탐색기 톱니바퀴 클릭

- 빌드 후 테스트 실행 선택



```
sum.cpp* x
(전역 범위)

1  #include "sum.h"
2  int getSum(int a, int b) {
3      return a + b + 9999;
4  }
```

sum.cpp 소스코드 수정 후
솔루션 전체 빌드 (Ctrl + Shift + B)



The screenshot shows the Visual Studio Test Explorer window after the build. The test results are as follows:

테스트	상태	기간
UnitTest (3)	실패	2ms
SimpleSum (2)	실패	2ms
Negative	실패	1ms
Positive	실패	1ms
TestCaseName (1)	통과	< 1ms
TestName	통과	< 1ms

자동 UnitTest 완료

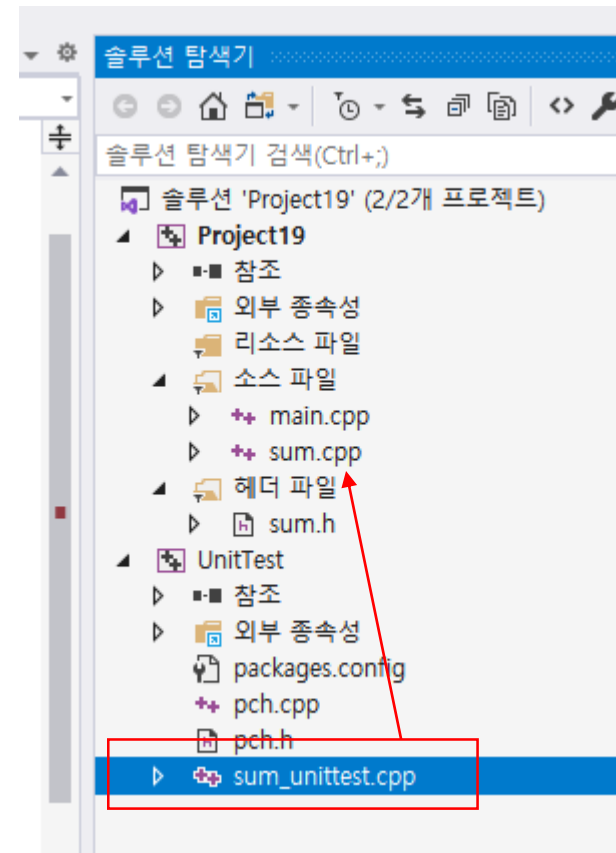
[권장] 파일명 규칙

sum.cpp 을 테스트하는 unittest 파일

- sum_unittest.cpp

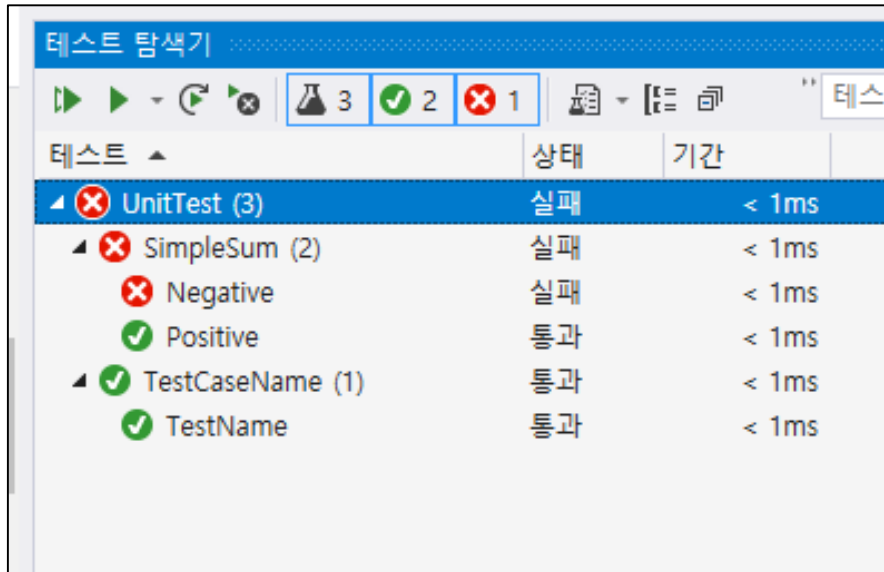
cpp 파일별 Unit Test code 구현

- Unit Test 프로젝트를 독립적 두고,
이곳에 Unit Test Code를 모두 넣어둬



GoogleTest 그룹핑

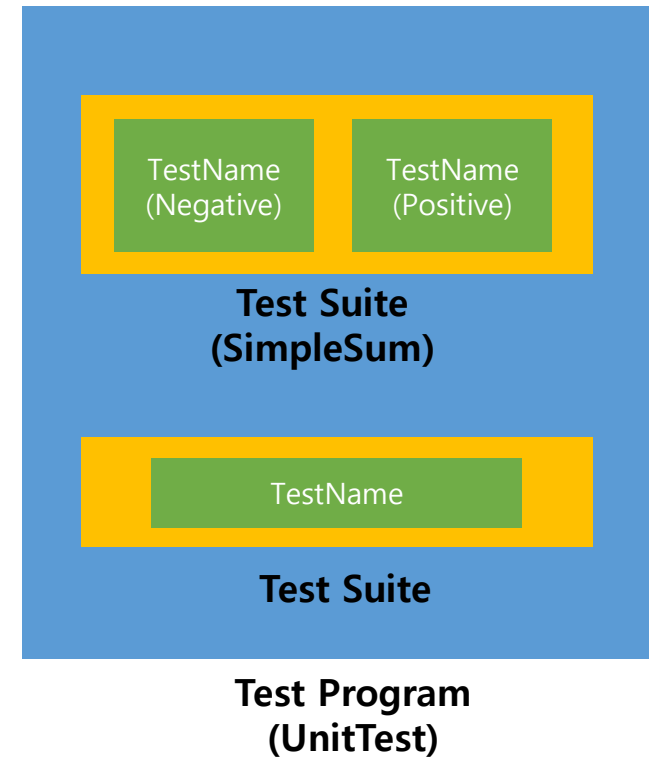
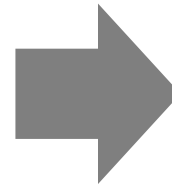
Test Program > Test Suite > Test



테스트 탐색기

▶ 실패 3 ▶ 성공 2 ▶ 실패 1

테스트	상태	기간
▶ ❌ UnitTest (3)	실패	< 1ms
▶ ❌ SimpleSum (2)	실패	< 1ms
❌ Negative	실패	< 1ms
✅ Positive	통과	< 1ms
▶ ✅ TestCaseName (1)	통과	< 1ms
✅ TestName	통과	< 1ms



Unit Test

깨끗한 유닛테스트 만들기

테스트코드도 깨끗하게 유지하자.

- ✓테스트 코드가 지저분하면, 관리 되고 있는 것 대비, 유지보수 능력이 떨어진다. 결국 실제 코드도 망가질 수 있다.

BUILD - OPERATE - CHECK 패턴

테스트코드를 보기 편하게 구역을 나눈다.

1. BUILD
2. OPERATE
3. CHECK

.

**AAA (Arrange, Act, Assert) 로 구역을 나누어
Unit Test를 작성한다.**

유의사항 1. 테스트 코드도 유지보수해야한다.

```
public void testGetPageHierarchyAsXml() throws Exception {
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(new FitNesseContext(root), request);
    String xml = response.getContent();

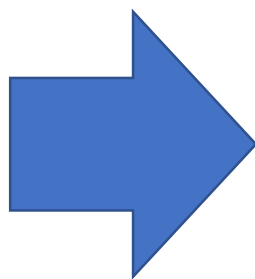
    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHierarchyAsXmlDoesntContainSymbolicLinks() throws Exception {
    WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
    WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
    symLinks.set("SymPage", "PageTwo");
    pageOne.commit(data);

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
    assertNotSubString("SymPage", xml);
}
```



```
public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>");
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>");
    assertResponseDoesNotContain("SymPage");
}
```

유의사항 2. 하나의 Test Code에는 한가지 의미만.

여러가지 테스트 코드들을
하나의 테스트 함수에 넣지 말자.

```
public void testAddMonths() {  
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);  
  
    SerialDate d2 = SerialDate.addMonths(1, d1);  
    assertEquals(30, d2.getDayOfMonth());  
    assertEquals(6, d2.getMonth());  
    assertEquals(2004, d2.getYYYY());  
  
    SerialDate d3 = SerialDate.addMonths(2, d1);  
    assertEquals(31, d3.getDayOfMonth());  
    assertEquals(7, d3.getMonth());  
    assertEquals(2004, d3.getYYYY());  
  
    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));  
    assertEquals(30, d4.getDayOfMonth());  
    assertEquals(7, d4.getMonth());  
    assertEquals(2004, d4.getYYYY());  
}
```

위 코드는 3개의 테스트를 한 함수에 몰아둔 코드이다.
위 코드 보다 3개의 테스트 함수를 만드는 것이 더 좋다.

[도전] AbsoluteSum

- ✓getAbsoluteSum(ArrayList<Integer>) 제작
 - 수를 저장한 동적배열을 넣으면,
절대값으로 변경된 동적 배열을 리턴하는 메서드
 - 양수, 음수, Zero, Mix 일 때 테스트 코드 만들기

가독성 있는 Test Code 예시 1

- ✓아래 Test Code는 AAA 구조로 잘 작성했지만,
조금 더 assert문을 간결하게 작성할 수 도 있다.

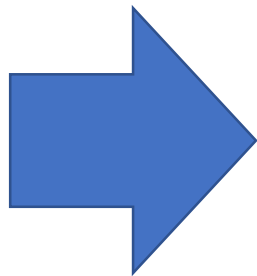
```
@Test
public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}
```

가독성 있는 Test Code 예시 2

✓ 조금 더 가독성 있게 표현하기

- 왼쪽 / 오른쪽 모두 가독성이 좋은 편이지만, 가독성 있게 표현을 하고자 하는 노력의 예시이다.

```
@Test
public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}
```



```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```


가독성 있는 Test Code 예시 3

✓ Test 함수가 많아질 경우
가독성이 좋다는 것이 잘 느껴진다.

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBCh1", hw.getState());
}

@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBch1", hw.getState());
}

@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCh1", hw.getState());
}

@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

깨끗한 테스트는 FIRST를 따른다. - 1

Fast

- 테스트가 느리면 자주 돌리지 못한다. 자주 돌리기 위해 빨라야한다.
- 코드를 마음껏 리팩토링 할 수 있게 빨라야한다.

Independent

- 각 테스트는 의존하면 안된다. 어떤 테스트가 먼저 수행될지 모른다.
- Fail 발생시 테스트 순서가 안맞아 Fail이 발생되었는지 의심이 없어야한다.

Repeatable

- 어떤 시간이던지, 네트워크가 안되는 등, 어떤 환경이라던지 항상 반복 가능해야한다.
- Fail 발생시, 테스트 시간에 테스트를 안돌려서 등 의심이 없어야한다.

깨끗한 테스트는 FIRST를 따른다. - 2

Self-Validating

- 테스트는 Bool 값으로 결과가 나와야한다.
- Pass 이지만, 불량이 발생할 확률이 있습니다. 가 아니라, Pass 이다.

Timely

- 테스트는 적시에 작성해야한다.
- (TDD 개발이라면) 모듈을 개발하기 전 작성한다.