

프로그래밍 역량 강화 전문기관, 민코딩

---

# Unit Test의 이해



# 목차

1. Unit Test 개요
2. Unit Test 준비
3. Assertion
4. Assertion 응용
5. Test Fixture
6. Test Fixture 활용 예시
7. Global set-up과 tear-down
8. Unit Test 의미와 작성 원칙

# Unit Test 개요

# OOP 에서 말하는 모듈 이란?

## 유례

- 큰 시스템을 작은 조각의 문제로 나누어 생각할 때, 작은 조각을 '모듈' 이라고 한다.

## 모듈의 정의

- 독립적으로 기능을 수행할 수 있는 작은 단위
- 한가지 일을 맡으며, 독립적이고 단순해야 한다.
- 일반적으로 함수 or 클래스 하나를 의미

# Unit Test

모듈 / Unit 단위로, Testing을 하는 것

입력값을 넣어 보았을 때,

결과값이 기대값과 동일하게 나오는지 확인한다.

주로 검증팀보다는, 개발팀에서 Test를 수행



# Unit Test 프레임워크

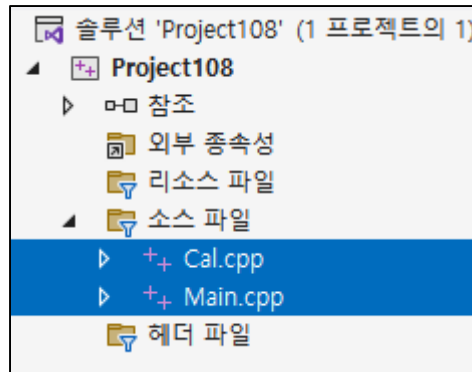
가장 많이 사용되는 Unit Test 프레임워크

1. Java : **JUnit**
2. C++ : **Google Test**
3. Python : **unittest** 내장

Unit Test 준비

# Google Test 동작 준비

## Cal.cpp / Main.cpp 준비



```
class Cal
{
public:
    int getSum(int a, int b)
    {
        return a + b;
    }
};
```

테스트 연습을 진행할 클래스 생성

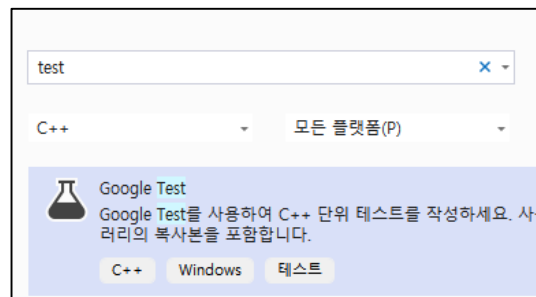
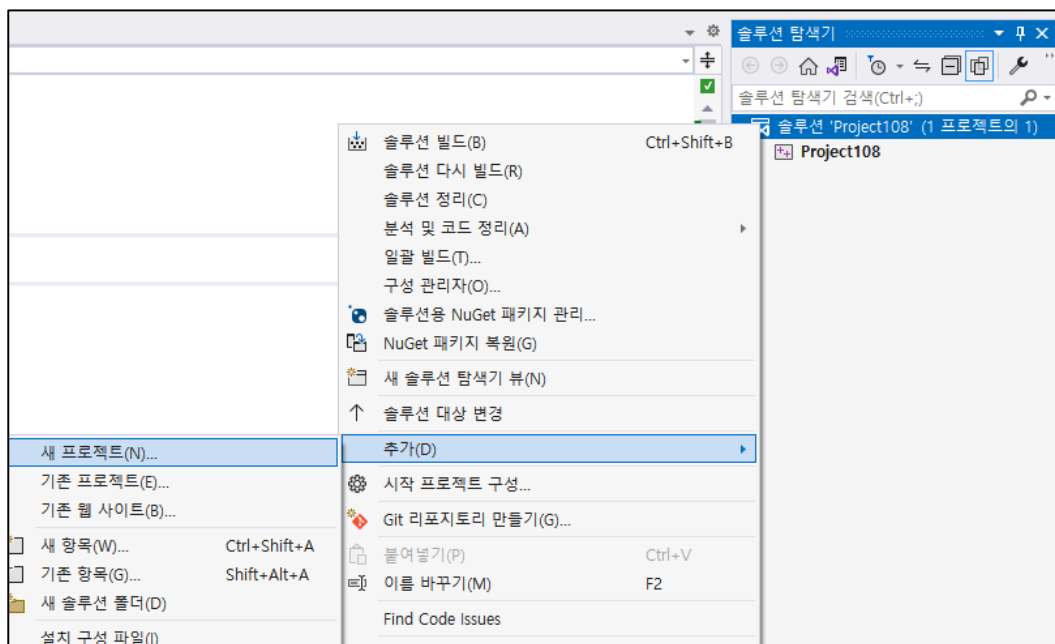
```
int main(int argc, char* argv[])
{
}
```

한 프로젝트에 main 함수는 반드시 존재해야하기 때문에 만들어 줌.



# 구글 테스트 준비

같은 솔루션에 프로젝트 하나 준비



## 테스트 프로젝트 구성

테스트할 프로젝트 선택(선택 사항):

Project108

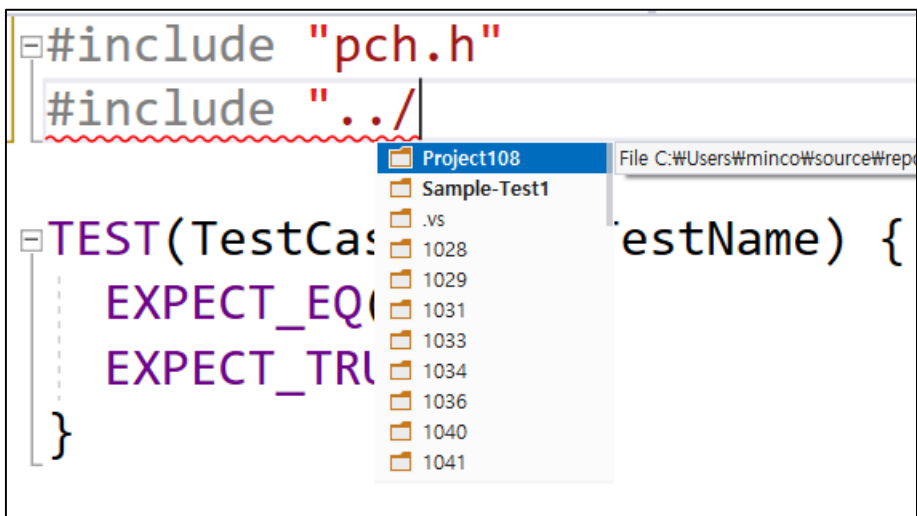
Google Test를 다음으로 사용:

- ☒ 정적 라이브러리(.lib)
- ☐ 동적 라이브러리(.dll)

C++ 런타임 라이브러리:

- ☒ 동적으로 연결(권장)
- ☐ 정적으로 연결

# 테스트할 대상 추가하기



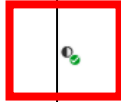
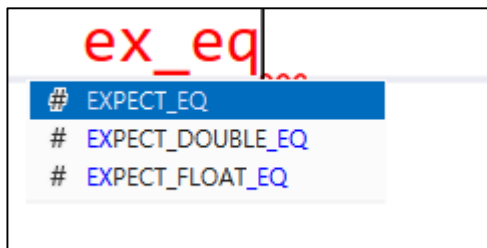
"../" 까지 쓰고 Ctrl + Space 누르면 된다.

```
#include "pch.h"
#include "../Project108/Cal.cpp"
```

Cal.h 가 아니라, Cal.cpp 임을 유의하자.  
본인의 프로젝트 이름에 맞게 적자.

# 자동완성

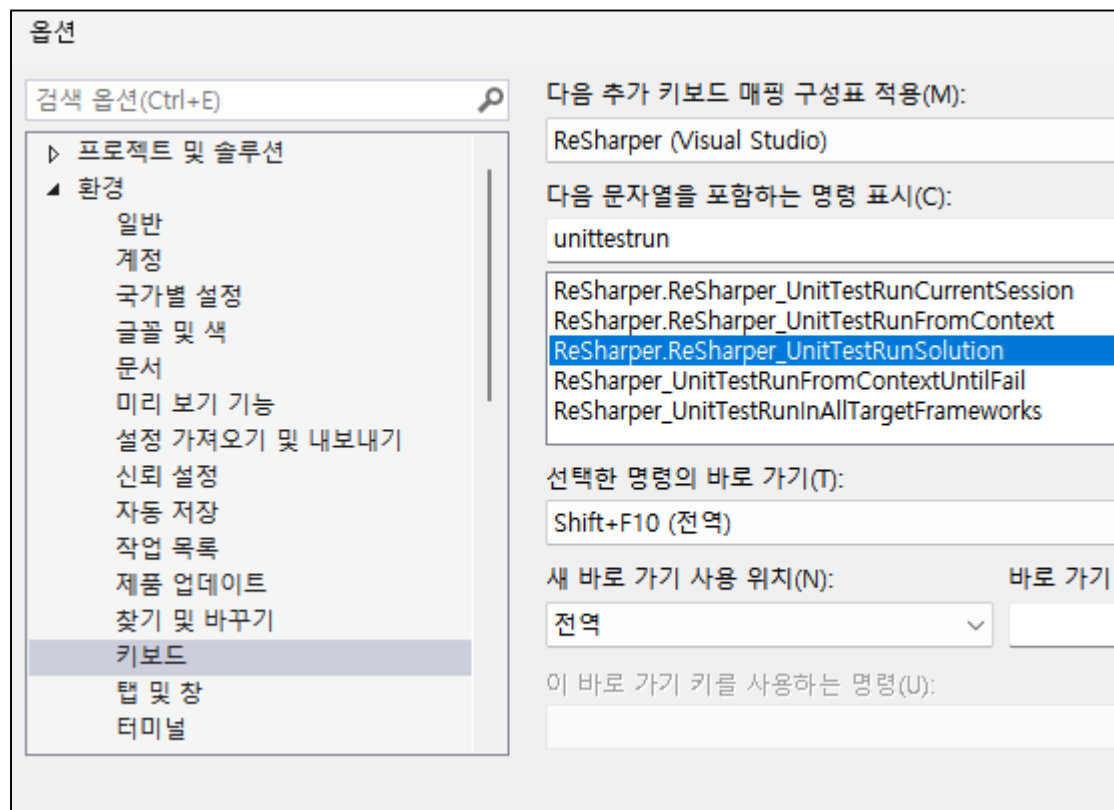
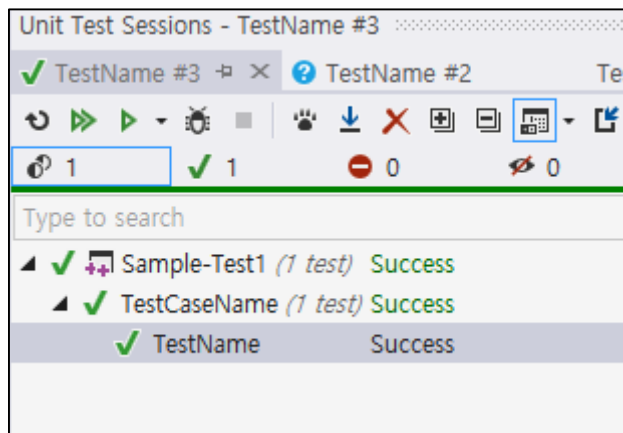
ex\_eq 누르고 탭키 누르면 된다.



```
1 #include "pch.h"
2 #include "../Project108/Cal.cpp"
3
4 TEST(TestCaseName, TestName) {
5     Cal cal;
6     int ret = cal.getSum(a: 1, b: 2);
7     EXPECT_EQ(ret, 3);
8 }
```

# 테스트 결과 확인

단축키 설정시  
쉽게 테스트를 실행 가능



단축키 설정 (Shift + F10)

# Assertion

EXPECT 와 ASSERT

# Assertion

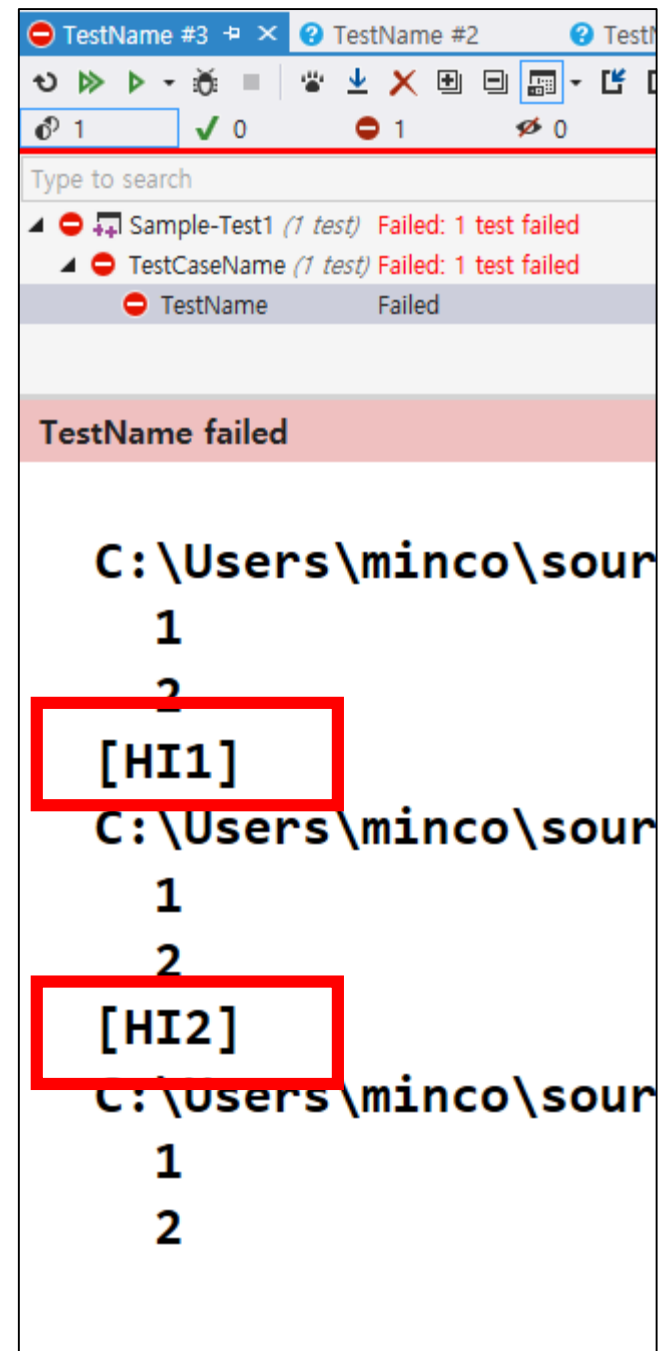
## ✓Assertion의 두 가지

- EXPECT\_EQ : 실패 하더라도, 실행중인 TestCase는 끝까지 진행함
- ASSERT\_EQ : 실패시 실행중인 TestCase를 중단 후, 다른 TestCase 수행

# Assert문과 Expect문 테스트

Assert문을 만나면  
바로 TC가 꺼지고  
다음 TC가 진행된다.

```
TEST(TestCaseName, TestName) {  
    EXPECT_EQ(1, 2); //fail  
    std::cout << "[HI1]\n";  
  
    EXPECT_EQ(1, 2); //fail  
    std::cout << "[HI2]\n";  
  
    ASSERT_EQ(1, 2); //fail  
    std::cout << "[HI3]\n";  
  
    EXPECT_EQ(1, 2); //fail  
    std::cout << "[HI4]\n";  
}
```



# 여러가지 EXPECT와 ASSERT 종류

✓ <https://google.github.io/googletest/reference/assertions.html>

Assert 어설션	Expect 어설션	검증하는 것
ASSERT_TRUE(상태)	EXPECT_TRUE(상태)	상태가 참인지
ASSERT_FALSE(상태)	EXPECT_FALSE(상태)	상태가 거짓인지

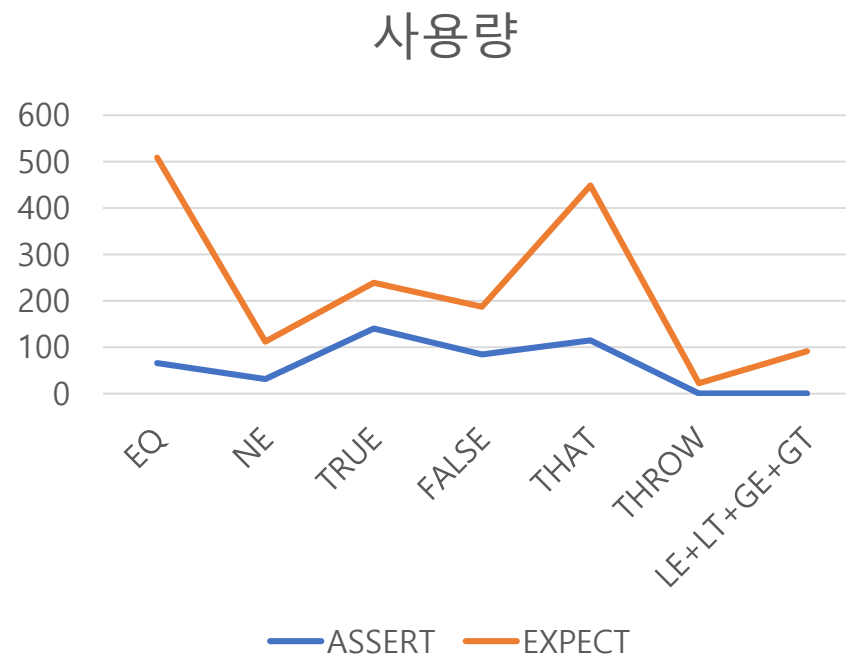
Assert 어설션	Expect 어설션	검증하는 것
ASSERT_EQ(값1, 값2) ASSERT_EQ(Actual, Expected)	EXPECT_EQ(값1, 값2) EXPECTED_EQ(Actual, Expected)	값1 == 값2
ASSERT_NE(값1, 값2)	EXPECT_NE(값1, 값2)	값1 != 값2
ASSERT_LT(값1, 값2)	EXPECT_LT(값1, 값2)	값1 < 값2
ASSERT_LE(값1, 값2)	EXPECT_LE(값1, 값2)	값1 <= 값2
ASSERT_GT(값1, 값2)	EXPECT_GT(값1, 값2)	값1 > 값2
ASSERT_GE(값1, 값2)	EXPECT_GE(값1, 값2)	값1 >= 값2



# 어떤 것을 많이 쓸까?

- ✓ googleapis - 구글 Cloud C++ 프로젝트 통계 (<https://github.com/googleapis/google-cloud-cpp>)
  - 여기에 언급되지 않은 Assertion 문은, 10회 미만이며 자주 쓰지 않으므로 생략한다.

	ASSERT	EXPECT
EQ	66	509
NE	31	112
TRUE	140	239
FALSE	84	187
THAT	115	449
THROW	-	22
LE+LT+GE+GT	-	91



# 기본 Assertion 문 연습하기

	ASSERT	EXPECT
EQ	66	509
NE	31	112
TRUE	140	239
FALSE	84	187
THAT	115	449
THROW	-	22
LE+LT+GE+GT	-	91

각자 연습해보자.

- LT : Less than, (**값1** < **값2**)
- LE : Less than or Equal (**값1** <= **값2**)
- GT : Greater than
- GE : Greater than or Equal

```
TEST(TestCaseName, TestName) {  
    EXPECT_LT(1, 2);  
    EXPECT_LE(1, 2);  
    EXPECT_GT(2, 1);  
    EXPECT_GE(2, 1);  
  
    EXPECT_NE(1, 10);  
}
```

# THAT

	ASSERT	EXPECT
EQ	66	509
NE	31	112
TRUE	140	239
FALSE	84	187
THAT	115	449
THROW	-	22
LE+LT+GE+GT	-	91

## EQUAL\_THAT 목적

- EXPECT\_EQ 보다 영어처럼 읽기 편하게 사용하기 위함
- Matcher라는 추가 검사기능을 사용할 수 있음(<http://google.github.io/googletest/reference/matchers.html>)

## EXPECT\_EQ과 THAT의 비교

- EXPECT\_EQ(값1, 값2)
  - 값1 이 기댓값인지, 값2가 기댓값인지 구분이 없음
- EXPECT\_THAT(Actual, Expected)
  - 해석 : Actual이 Expected이어야만 한다.

```
TEST(TestCaseName, TestName) {  
  
    EXPECT_THAT("ABC", "CD");  
  
    EXPECT_LT(1, 2);  
    EXPECT_LE(1, 2);  
    EXPECT_GT(2, 1);  
    EXPECT_GE(2, 1);  
  
    EXPECT_NE(1, 10);  
}
```

gMock을 설치해야 사용가능하다.  
지금은 EXPECT\_THAT을 사용할 수 없다.

# THROW

Exception 이 발생해야 PASS

	ASSERT	EXPECT
EQ	66	509
NE	31	112
TRUE	140	239
FALSE	84	187
THAT	115	449
THROW	-	22
LE+LT+GE+GT	-	91

```
void testFunc()
{
    std::string s;
    s.resize(_New_size: -1); //Exception!
}

TEST(TestCaseName, TestName) {
    EXPECT_THROW(testFunc(), std::exception);
}
```

std::exception 이 발생해야 Pass 이다.

# [도전] 코드리뷰 – Sample 1, 2, 4번

✓구글링 : googletest github

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

✓googletest 폴더 > sample 폴더

1. sample1\_unittest.cc 코드 리뷰 (sample1.cc / .h도 확인해야함)
2. sample2번, 4번 코드리뷰 (3번은 예외)

[참고] EXPECT\_STREQ은 C언어 스타일 문자열 비교이며,  
String Class는 EXPECT\_EQ로 비교할 수 있다.

Assertion 응용

## [주의] 테스트 스위트, 이름에 Underbar 사용금지

내부적으로 Underbar를 사용하여 매크로를 처리한다.

**아래 두 테스트 모두 같은 매크로를 생성하는 문제 발생!**

- TEST(Time, Flies\_Like\_An\_Arrow) { ... }
- TEST(Time\_Flies, Like\_An\_Arrow) { ... }

→ Time\_Flies\_Like\_An\_Arrow\_Test

# 실패 메시지 출력

✓EXPECT\_EQ(7, 1 + 1) << "1 + 1 은 2 이여요.";

```
TEST(TestCaseName, TestName) {  
    EXPECT_EQ(7, 1 + 1) << "2이어야한다.";  
}
```

TestName failed

```
C:\Users\minco\sou  
7  
1 + 1  
    Which is: 2  
2이어야한다.
```



# 명시적 Fail

FAIL() 은 무조건 FAIL이 발생한다.

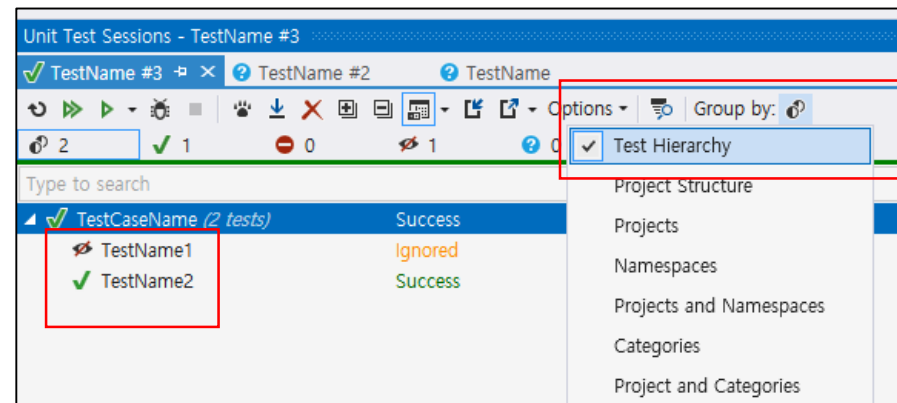
```
TEST(TestCaseName, TestName) {  
    FAIL();  
}
```

# DISABLED\_ 키워드

특정 테스트 생략

```
TEST(TestCaseName, DISABLED_TestName1) {  
}  
TEST(TestCaseName, TestName2) {  
}
```

“DISABLED\_” 키워드를 붙이면  
해당 Test는 Ignored 처리 된다.



# Test Fixture

SetUp / TearDown

# Test Fixture

특정 대상을 테스트할 수 있는 장치

- 테스트 장치를 한번 만들어두면,  
여러 대상을 지속적으로 테스트 할 수 있다.



# Test Fixture

## SW의 Test Fixture 의 구성

1. 테스트 하기 전 초기 세팅 코드 삽입 가능 (SetUp 코드)
2. 테스트 종료 후, 정리작업 세팅 코드 삽입 가능 (TearDown 코드)



Test Fixture 객체  
“기계장치”

**Fixture Class 생성**  
(testing::Test 상속)



테스트할 대상  
“의자”

**테스트 하고픈 대상**



# Fixture 예시

다음 내용을 숙지한다.

1. 반드시 `testing::Test`를 상속받는다.
2. `KFCFixture`에 Alt + Enter후 overriding Method 자동 추가 가능
3. 테스트 함수에서 접근이 가능하도록 public에 모두 구현하자.
4. `SetUp`과 `TearDown`은 protected로 해주자. (쉬운 구분을 위함)



Test Fixture 객체  
“기계장치”

```
class KFCFixture : public testing::Test
{
public:
    int chair; //테스트 대상

    void push()
    {
        chair -= 1;
    }

protected:
    void SetUp() override
    {
        std::cout << "테스트 준비중\n";
        chair = 10;
    }

    void TearDown() override
    {
        chair = 0;
        std::cout << "테스트 정리 완료\n";
    }
};
```

# Fixture를 사용한 테스트

TEST\_F( ) 매크로를 사용한다.

- Test Fixture는 객체생성없이, **public, protected** 멤버들 모두 사용가능

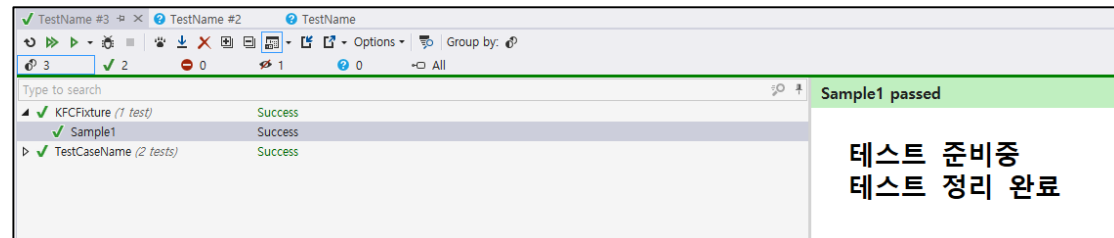
```
class KFCFixture : public testing::Test
{
public:
    int chair; //테스트 대상

    void push()
    {
        chair -= 1;
    }

protected:
    void SetUp() override
    {
        std::cout << "테스트 준비중\n";
        chair = 10;
    }

    void TearDown() override
    {
        chair = 0;
        std::cout << "테스트 정리 완료\n";
    }
};
```

```
TEST_F(KFCFixture, Sample1)
{
    push();
    EXPECT_EQ(chair, 9);
}
```



# Test Fixture 활용 예시

중복 Test 코드 제거



# Test Fixture 활용 예시

테스트 대상이 되는 객체 (의자 역할)

- Zegop Class
- getZegop( )을 수행할 때 마다 내부적으로 가지고 있는 값을 제공승을 올리고, 리턴한다.

```
class Zegop
{
private:
    int num;

public:
    Zegop(int num) : num(num) {}
    int getZegop()
    {
        num = num * num;
        return num;
    }
};
```

# Zegop 테스트코드

중복코드가 많다.

테스트 장치를 하나 만들어두고,  
반복되는 작업은  
테스트장치에 넣어두자.

```
TEST(ZegopTest, Zegop1)
{
    std::cout << "테스트준비!\n";
    Zegop z(num: 10);
    EXPECT_EQ(z.getZegop(), 100);
    std::cout << "테스트끝!\n";
}

TEST(ZegopTest, Zegop2)
{
    std::cout << "테스트준비!\n";
    Zegop z(num: 10);
    EXPECT_EQ(z.getZegop(), 100);
    EXPECT_EQ(z.getZegop(), 10000);
    std::cout << "테스트끝!\n";
}

TEST(ZegopTest, Zegop3)
{
    std::cout << "테스트준비!\n";
    Zegop z(num: 10);
    EXPECT_EQ(z.getZegop(), 100);
    EXPECT_EQ(z.getZegop(), 10000);
    EXPECT_EQ(z.getZegop(), 1000000000);
    std::cout << "테스트끝!\n";
}
```

# Test Fixture 준비하기

자주 쓰는 기능들을  
Fixture 객체로 만듦.

```
class ZegopFixture : public testing::Test
{
protected:
    void SetUp() override
    {
        std::cout << "테스트준비!\n";
    }
    void TearDown() override
    {
        std::cout << "테스트끝!\n";
    }

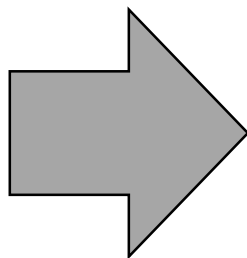
public:
    Zegop z{ num: 10 };
};
```

# 테스트코드 리팩토링

```
TEST(ZegopTest, Zegop1)
{
    std::cout << "테스트준비!\n";
    Zegop z(num: 10);
    EXPECT_EQ(z.getZegop(), 100);
    std::cout << "테스트끝!\n";
}

TEST(ZegopTest, Zegop2)
{
    std::cout << "테스트준비!\n";
    Zegop z(num: 10);
    EXPECT_EQ(z.getZegop(), 100);
    EXPECT_EQ(z.getZegop(), 10000);
    std::cout << "테스트끝!\n";
}

TEST(ZegopTest, Zegop3)
{
    std::cout << "테스트준비!\n";
    Zegop z(num: 10);
    EXPECT_EQ(z.getZegop(), 100);
    EXPECT_EQ(z.getZegop(), 10000);
    EXPECT_EQ(z.getZegop(), 100000000);
    std::cout << "테스트끝!\n";
}
```



```
TEST_F(ZegopFixture, Cal_CC1)
{
    std::cout << z.getZegop();
}

TEST_F(ZegopFixture, Cal_CC2)
{
    std::cout << z.getZegop();
    std::cout << z.getZegop();
}

TEST_F(ZegopFixture, Cal_CC3)
{
    std::cout << z.getZegop();
    std::cout << z.getZegop();
    std::cout << z.getZegop();
}
```

# Test 결과화면 (Google Test Console)

```
Microsoft Visual Studio 디버그 콘솔
Running main() from D:\w\work\1\src\ThirdParty\googletest\googletest\src\gtest_main.cc
[=====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 3 tests from ZegopFixture
[ RUN      ] ZegopFixture.Cal_CC1
테스트 준비!
100테스트 끝!
[ OK      ] ZegopFixture.Cal_CC1 (1 ms)
[ RUN      ] ZegopFixture.Cal_CC2
테스트 준비!
10010000테스트 끝!
[ OK      ] ZegopFixture.Cal_CC2 (1 ms)
[ RUN      ] ZegopFixture.Cal_CC3
테스트 준비!
10010000100000000테스트 끝!
[ OK      ] ZegopFixture.Cal_CC3 (1 ms)
[-----] 3 tests from ZegopFixture (5 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test case ran. (9 ms total)
[ PASSED  ] 3 tests.
```

# Global set-up과 tear-down

중복 Test 코드 제거

# Global 설정 set-up과 tear-down

전체 테스트 시작시

한번 호출되는 곳

- Global set-up
- Global tear-down

```
Microsoft Visual Studio 디버그 콘솔

[=====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 3 tests from ZegopFixture
[ RUN      ] ZegopFixture.Cal_CC1
테스트준비!
100테스트끝!
[ OK      ] ZegopFixture.Cal_CC1 (0 ms)
[ RUN      ] ZegopFixture.Cal_CC2
테스트준비!
10010000테스트끝!
[ OK      ] ZegopFixture.Cal_CC2 (0 ms)
[ RUN      ] ZegopFixture.Cal_CC3
테스트준비!
100100001000000000테스트끝!
[ OK      ] ZegopFixture.Cal_CC3 (0 ms)
[-----] 3 tests from ZegopFixture (4 ms total)
[-----] Global test environment tear-down
[=====] 3 tests from 1 test case ran. (5 ms total)
[ PASSED  ] 3 tests.

C:\Users\minco\source\repos\Project108\x64\Debug\Sample-Test
) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요..._
```

# Global SetUp / TearDown

**unit\_test 파일에 main code를 추가해준다.**

- AddGlobalTestEnvironment : 전역 설정
- new로 만든 환경설정 class, delete 금지!
  - Google Test가 알아서 delete를 진행함.

```
class GlobalEnv : public testing::Environment {
public:
    void SetUp() {
        std::cout << "TEST READY!!\n";
    }

    void TearDown() {
        std::cout << "CLEAN UP!!\n";
    }
};

int main(int argc, char* argv[]) {
    testing::InitGoogleTest(&argc, argv);
    testing::AddGlobalTestEnvironment(new GlobalEnv);
    return RUN_ALL_TESTS();
}
```



# Global SetUp / TearDown 결과

```
Microsoft Visual Studio 디버그 콘솔

[=====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
TEST READY!!
[-----] 3 tests from ZegopFixture
[ RUN      ] ZegopFixture.Cal_CC1
테스트준비!
100테스트끝!
[ OK       ] ZegopFixture.Cal_CC1 (1 ms)
[ RUN      ] ZegopFixture.Cal_CC2
테스트준비!
10010000테스트끝!
[ OK       ] ZegopFixture.Cal_CC2 (0 ms)
[ RUN      ] ZegopFixture.Cal_CC3
테스트준비!
100100001000000000테스트끝!
[ OK       ] ZegopFixture.Cal_CC3 (0 ms)
[-----] 3 tests from ZegopFixture (3 ms total)

[-----] Global test environment tear-down
CLEAN UP!!
[=====] 3 tests from 1 test case ran. (5 ms total)
[ PASSED  ] 3 tests.

C:\Users\minco\source\repos\Project108\x64\Debug\Sample-Test
1.exe(프로세스 32360개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...■
```

# [도전] 코드리뷰 – Sample 3, 5번

- ✓구글링 : googletest github
  - <https://github.com/google/googletest>
  - googletest 폴더 > sample 폴더

# [도전] 홀짝 Unit Test

vector 에 숫자 값을 넣으면,

짝수면 "O" / 홀수면 "X"를 구분해주는 vector를 리턴하는 모듈

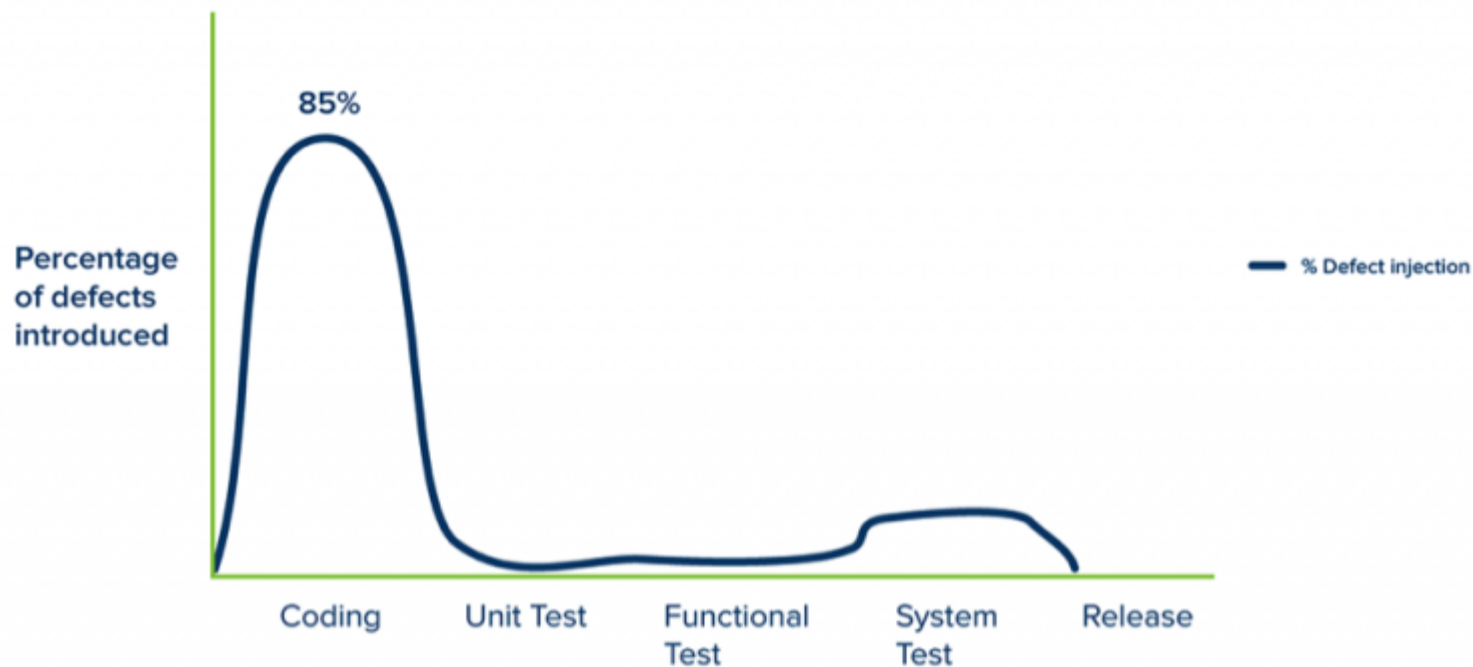
- 입력예시 : {1, 2, 3, 0}
- 출력예시 : {"X", "O", "X", "O"}
- 모두 짝수이거나 홀수면 null 리턴

# Unit Test 의미와 작성 원칙

# Unit Test 의 의미 : 결함의 비용 최소화

## ✓ 결함이 만들어지는 타이밍

- 대부분의 버그는 코딩 단계에서 만들어진다.

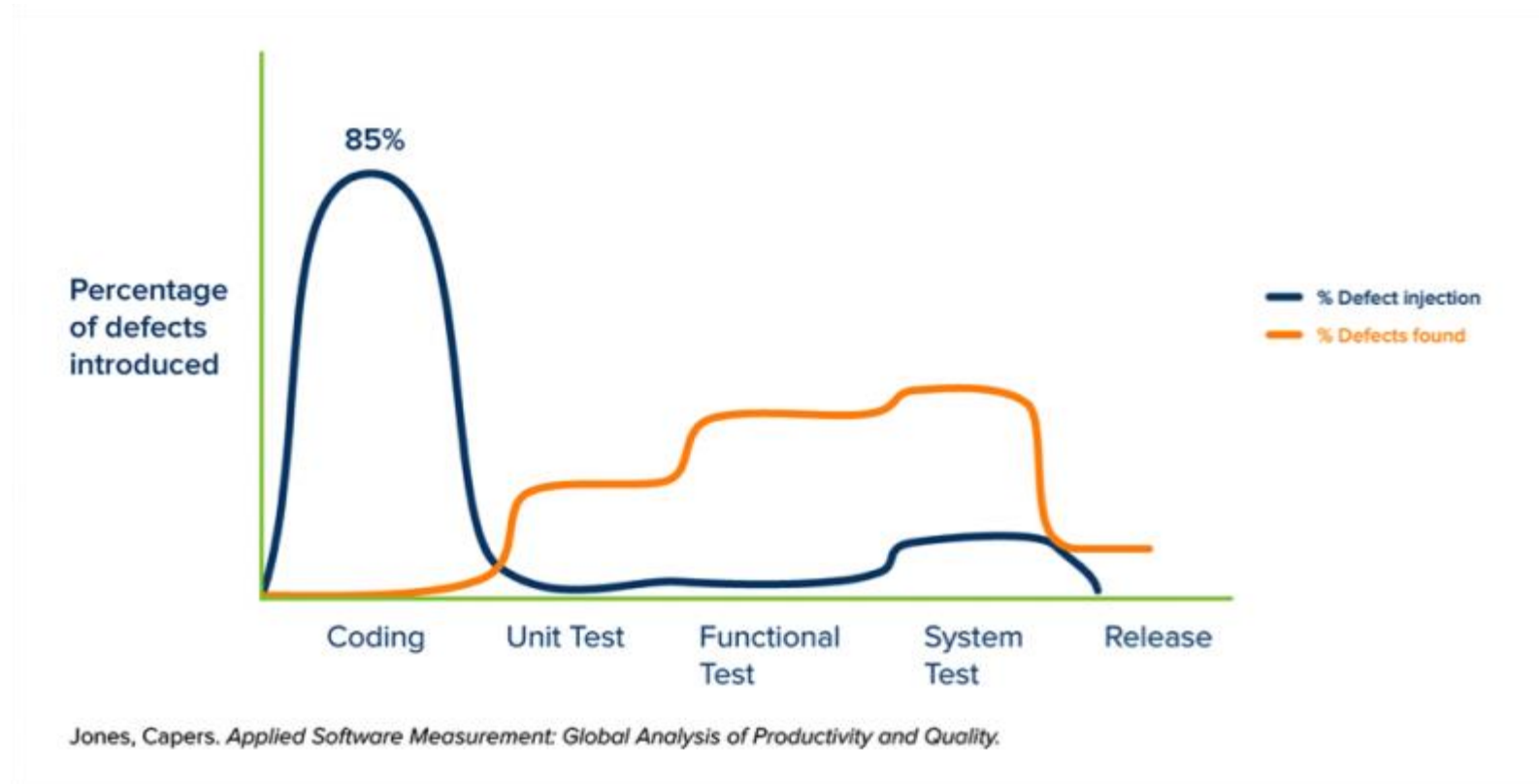


Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.

# 결함 처리 비용

## ✓결함의 발견

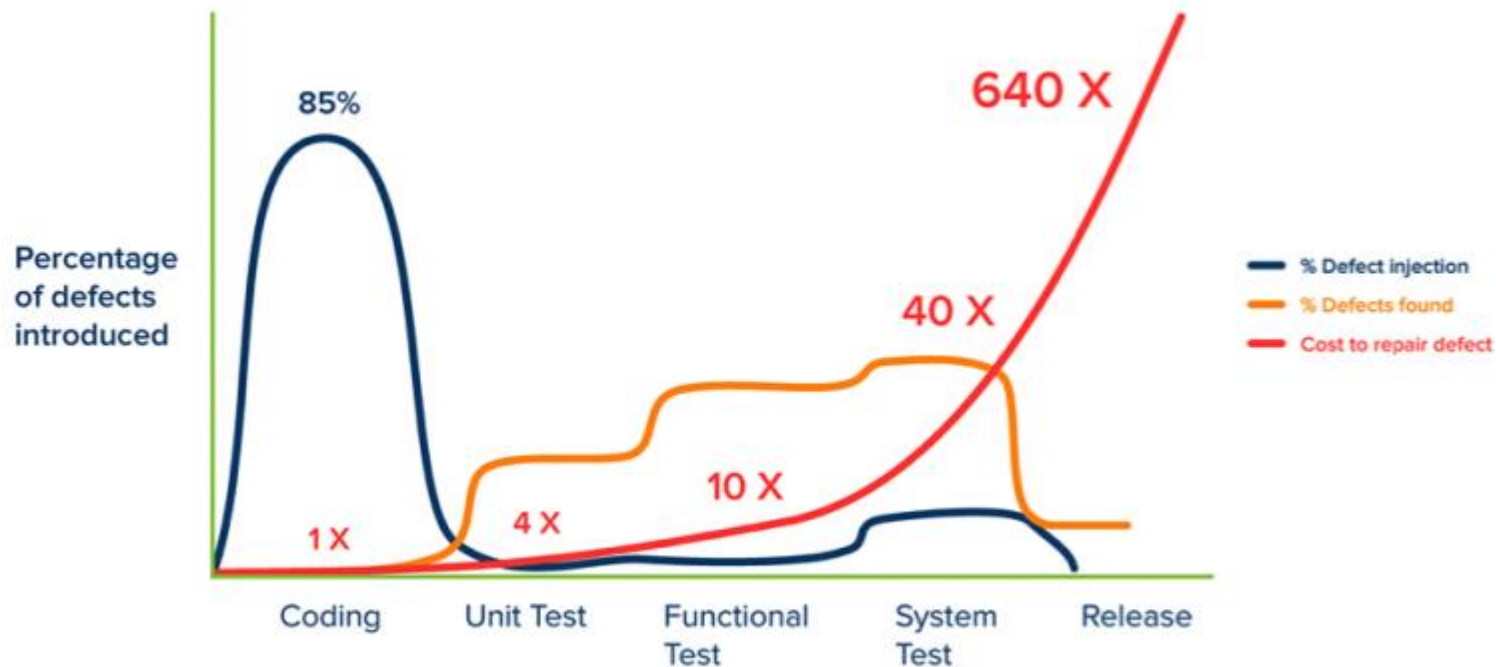
- 테스트를 시작할 때부터, 많은 결함이 발견된다.



# 결함 처리 비용

## ✓개발의 각 단계에서 결함을 수정하는데 드는 비용의 차이

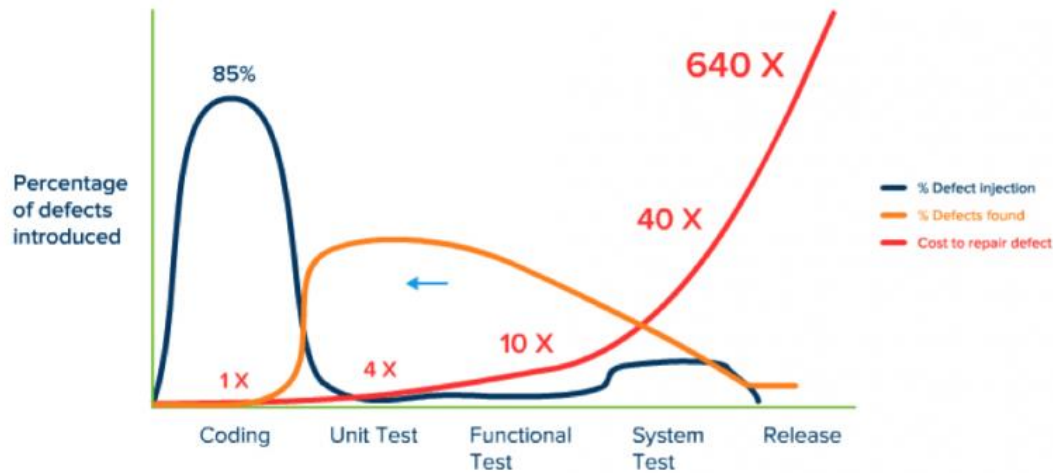
- 개발이 진행될수록, 결함이 발견될수록 비용이 급격히 증가



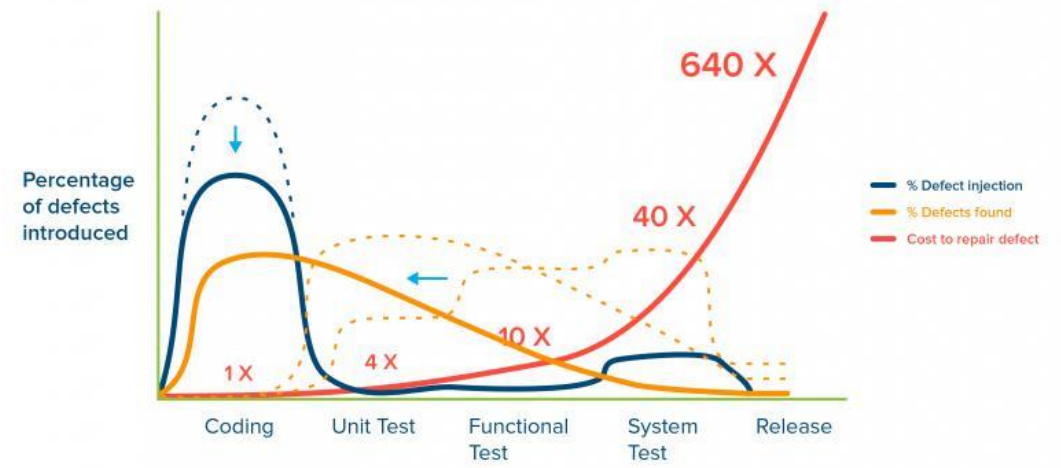
# 결함 처리 비용

## ✓테스트를 초기에 수행하면 일어나는 변화

- 코드의 문제를 일찍 발견할수록 그로 인한 영향이 줄어들고 문제를 해결하는 비용이 줄어든다



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.



# Unit Test 작성 원칙 (F. I. R. S. T.) by 클린코드

## Fast

- 테스트는 빨라야 한다.
- 테스트는 빨리 자주 돌릴 수 있어야 한다. 그래야 초반에 문제를 찾아낼 수 있다

## Independent

- 각 테스트는 서로 의존하면 안 된다.
- 테스트가 실패할 때 원인 분석에는, 테스트 순서에 따른 Fail 의심이 없어야 한다.

## Repeatable

- 테스트는 어떤 환경에서도 반복 가능해야 한다.
- 테스트가 실패할 때 원인 분석에는, 특정 시간 / 특정 환경에 따른 Fail 의심이 없어야 한다.

## Self-Validating

- 테스트는 bool 값으로 결과를 내야 한다.
- Fail일 확률이 존재한다는 결과는, 도움이 되지 않는다.

## Timely

- 테스트는 적시에 작성해야 한다
- 적시 = 개발하기 전
- 초반에 버그를 잡을 수 있도록 적시에 작성한다.