프로그래밍 역량 강화 전문기관, 민코딩

Matcher & Test Double
(with Google Test)

목차

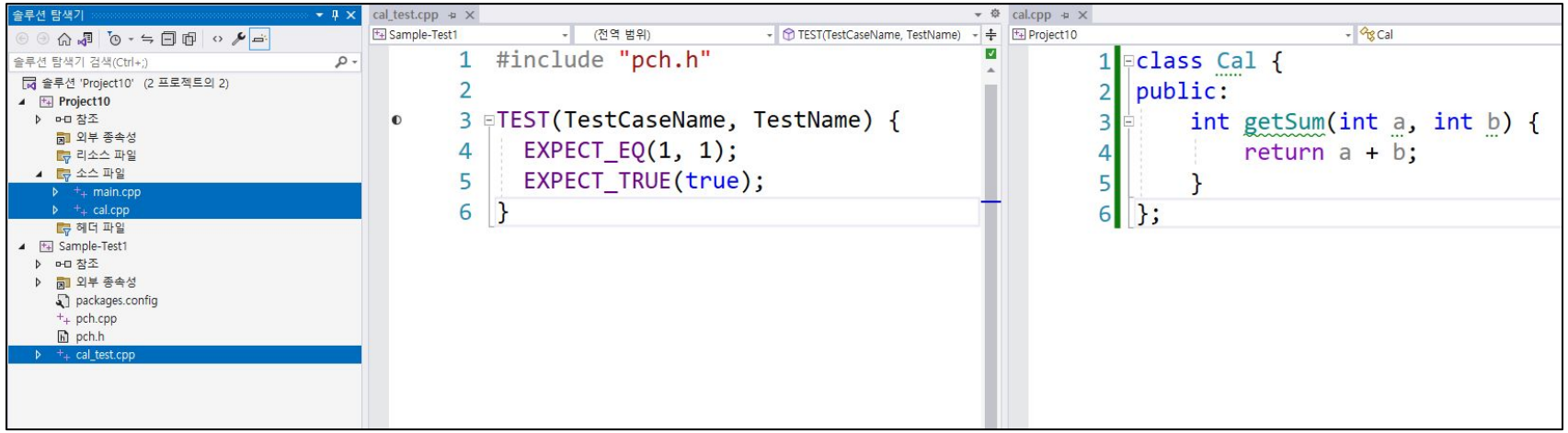
Matcher와 Mocking을 위한 프로젝트 준비

프로젝트 준비하기

Google Test 테스트를 위한 프로젝트 준비

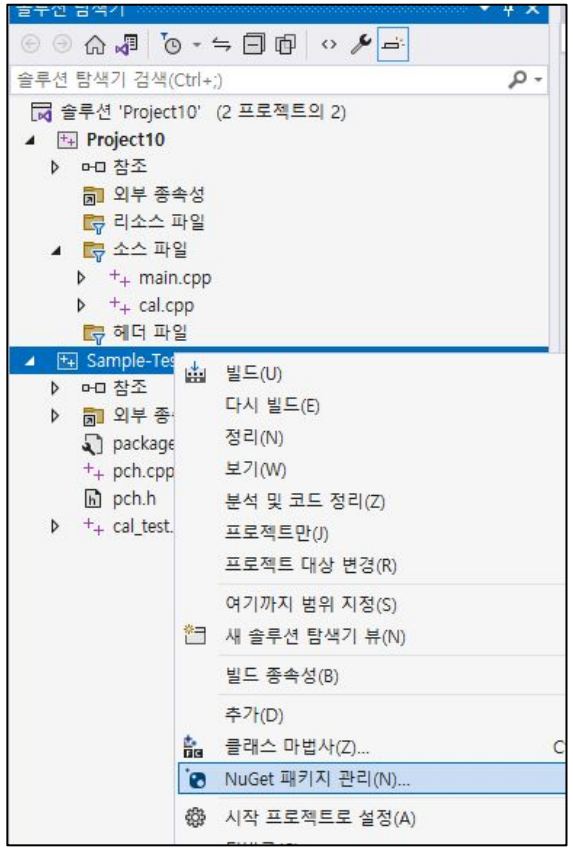
✔ 왼쪽 창 : cal_test.cpp (google test)

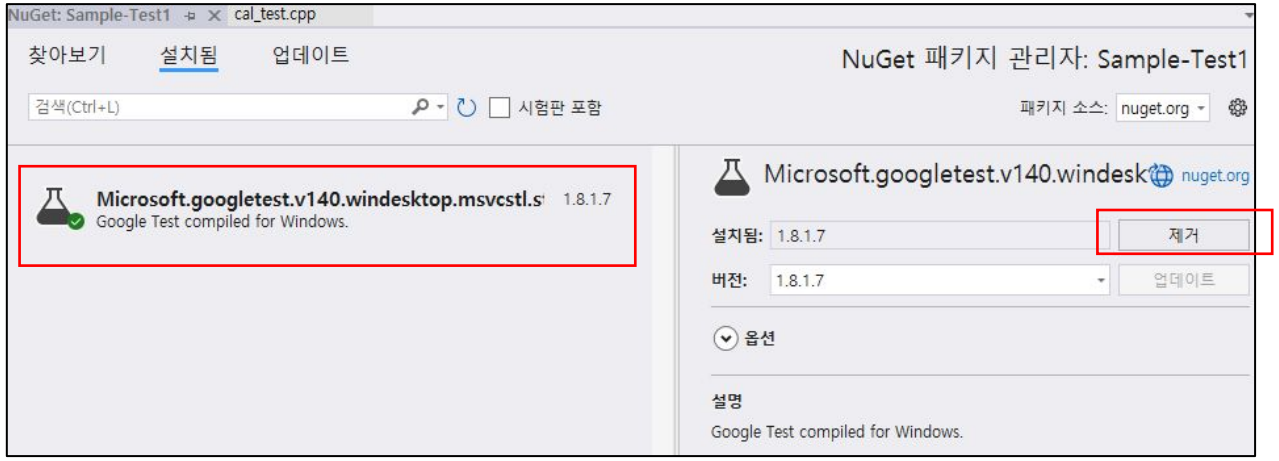
✔ 오른쪽 창 : cal.cpp



기존 설치된 Microsoft google test 제거

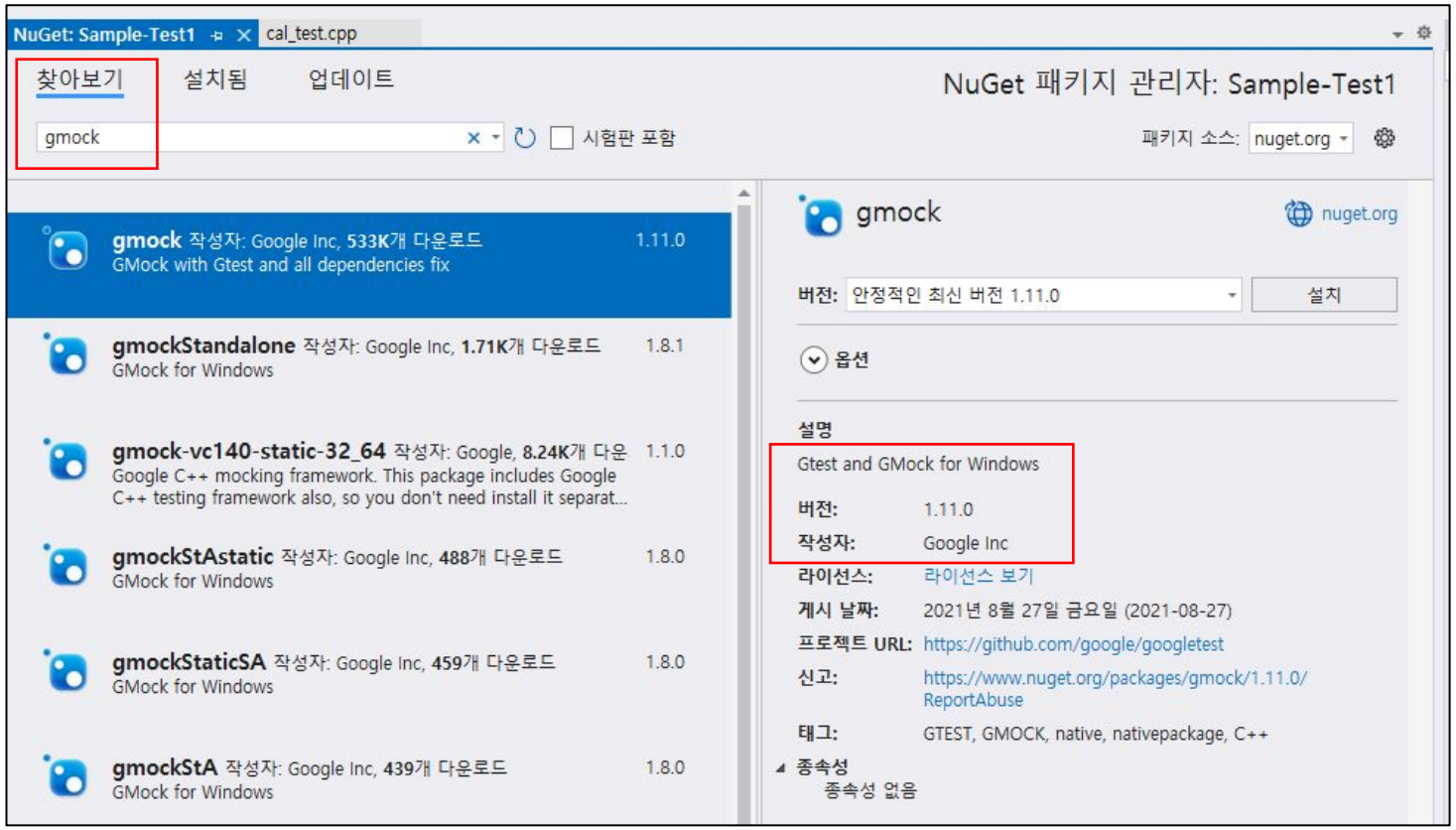
✔ NuGet에서 제거하기 (다른 패키지로 설치할 예정)





NuGet : gmock 설치

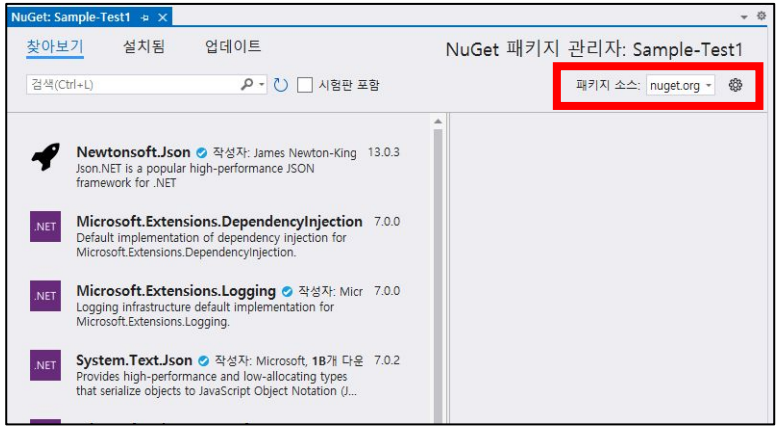
✔ Google에서 올린, **GTest + GMock Library**

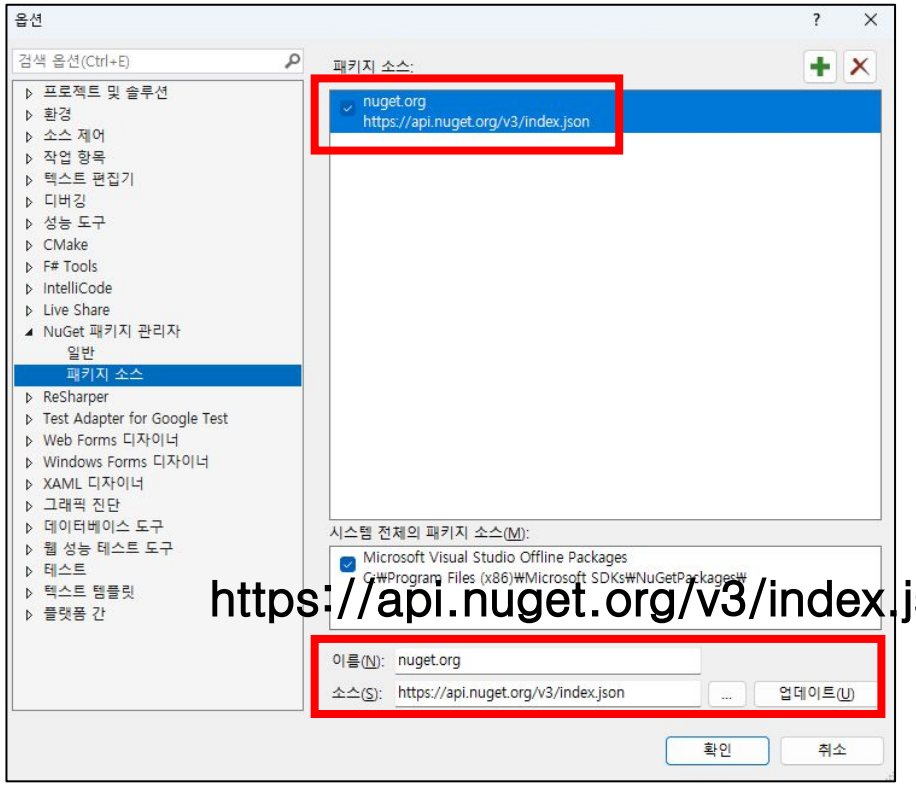


[Trouble Shooting] NuGet 저장소 확인

✔ 현재 다음과 같이 패키지 소스 경로가 맞는지 확인한다.

- 다르다면 아래와 같이 수정한다.

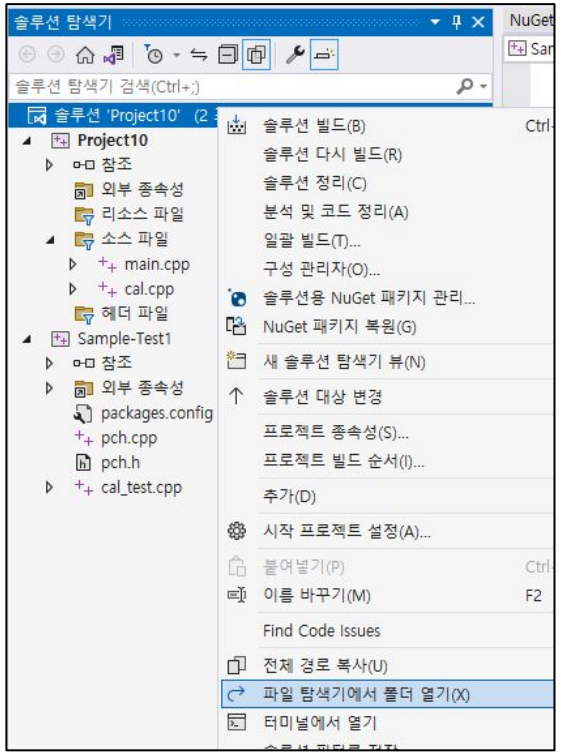


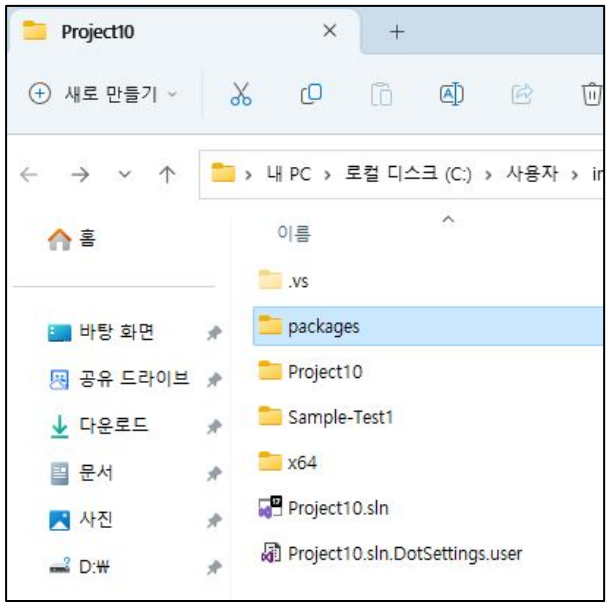


https://api.nuget.org/v3/index.json

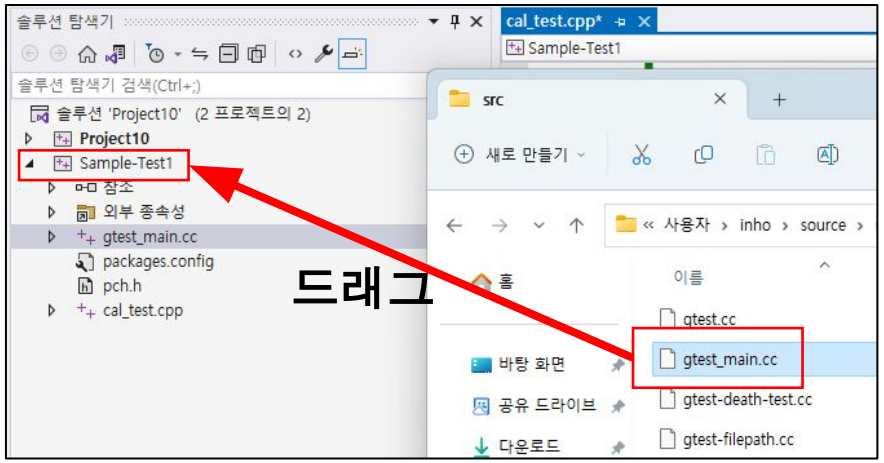
main 파일 옮기기

✔ gtest_main.cc 파일을 프로젝트에 추가해준다.





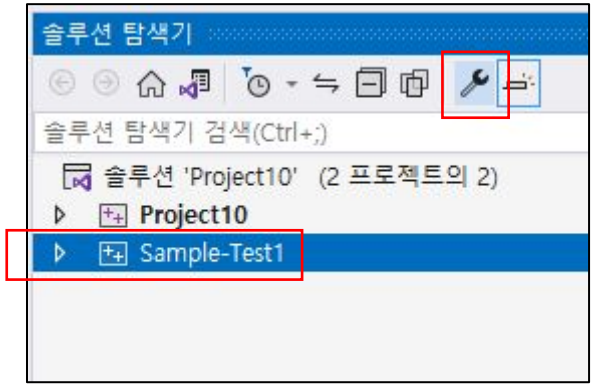
package 폴더에 진입하자.



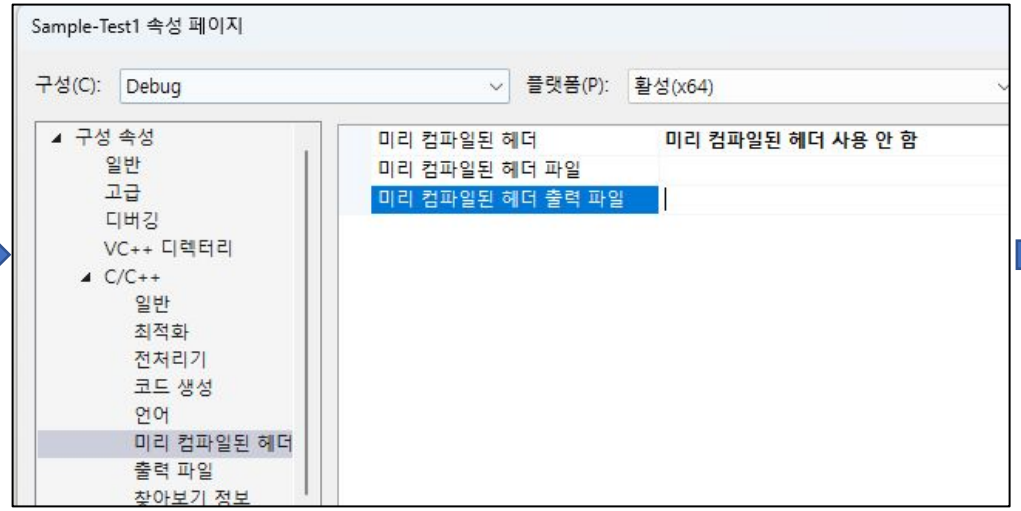
~~₩packages~~₩gmock.1.11.0₩lib₩native₩src₩gtest₩src₩gtest_main.cc

pch.h 제거

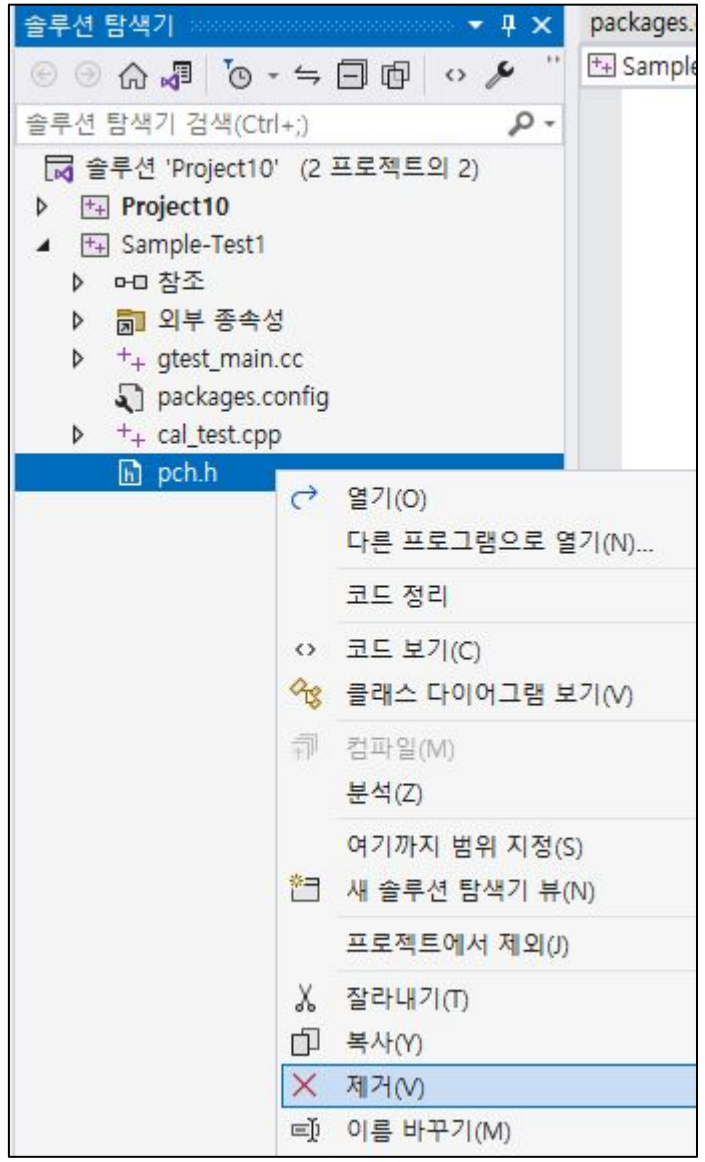
✔미리 컴파일된 헤더 삭제하기



프로젝트 > 설정



C/C++
> 미리 컴파일된 헤더
> 사용안함, 옵션 모두 제거

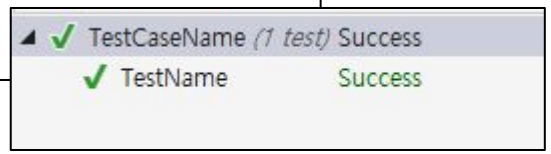


더이상 필요없는 pch.h 파일 제거

Test 해보기

- ✔ 필수 include 추가하기
- ✔ EXPECT_EQUAL 대신
  EXPECT_THAT 문법 사용하기

```
#pragma once
#include "gtest/gtest.h"
#include "gmock/gmock.h"

using namespace testing;
TEST(TestCaseName, TestName) {
    EXPECT_THAT(1, Eq(1));
}
```



Matcher

Matcher란?

✔ Assertion 문을 더 가독성 있게 만드는 기능

✔ 복잡한 Assertion 문을 더 간략하게 구현할 수 있다.

Matcher 예시

매크로
EXPECT_THAT(actual_value, matcher)
ASSERT_THAT(actual_value, matcher)

✔ 사람이 읽기 편하게 Matcher가 지원된다.

- EXPECT_THAT은 Matcher를 쓸 수 있는 Assertion 이다.

```
TEST(TestCaseName, TestName) {
    int x = 10;
    EXPECT_THAT(x, Eq(10));
}
```

x is Equal to 10

```
TEST(TestCaseName, TestName) {
    int x = 10;
    EXPECT_THAT(x, Ne(100));
}
```

x is Not Equal 100

```
TEST(TestCaseName, TestName) {
    EXPECT_THAT(5, AnyOf(1, 3, 5, 8, 9));
}
```

이중에 5가 있다면, Pass

Matcher를 써야하는 이유

1. 조금 더 가독성 있는 Assertion을 만들고 싶을 때
2. 테스트 의도를 더 명확하게 나타내고 싶을 때
3. 중복코드를 줄이고, 복잡한 코드를 간결하게 만들 고 싶을 때

```
TEST(TestCaseName, TestName) {
    int arr[] = {1, 3, 5, 8, 9};

    bool found = false;
    for (int i = 0; i < 5; i++) {
        if (arr[i] == 5) {
            found = true;
            break;
        }
    }

    if (!found) FAIL();
}
```

```
TEST(TestCaseName, TestName) {

    EXPECT_THAT(5, AnyOf(1, 3, 5, 8, 9));
}
```

포인터 / 레퍼런스 Matcher

- ✔ IsNull( )
- ✔ NotNull( )
- ✔ Ref( )

```
TEST(TestCaseName, TestName) {

    int* p1 = nullptr;
    int* p2 = new int();

    EXPECT_THAT(p1, IsNull());
    EXPECT_THAT(p2, NotNull());
}
```

p1 is Null
p2 is Not Null

```
TEST(TestCaseName, TestName) {

    int x = 10;
    int& p = x;

    EXPECT_THAT(p, Ref(x));
}
```

p is Reference for x

대소비교 Matcher

- ✔ Gt( ) / Lt( )
- ✔ Ge( ) / Le( )

```
TEST(TestCaseName, TestName) {

    int x = 10;

    EXPECT_THAT(x, Gt(2));
}
```

x is Greater than 2

```
TEST(TestCaseName, TestName) {

    int x = 10;

    EXPECT_THAT(x, Lt(14));
}
```

x is Less than 14

[도전] 수 검증

어떤 값이

25보다 크고
35보다 작고
31이 아니면 Pass

소수점 비교

✔ 오차를 허용한 근접치 비교

```
TEST(TestCaseName, TestName) {

    double pi = 3.14;

    EXPECT_THAT(pi, DoubleNear(3.1415, 0.1));
}
```

3.14와 3.1415 값이
절대오차값 0.1 범위 내에 있는지 검사

Custom Message 출력

✔ EXPECT_EQ과 사용법이 같다.

```
TEST(TestCaseName, TestName) {
    EXPECT_THAT(1, Eq(2)) << "KFC MESSAGE";
}
```

```
TestName failed

C:\Users\inho\source\repos\Project10\Sample-Test1
  \cal_test.cpp(10): error: Value of: 1
Expected: is equal to 2
  Actual: 1 (of type int)
KFC MESSAGE
```

컨테이너 Matcher

- ✔ Contains( ) : 포함 여부 판단
- ✔ ContainerEq( ) : 동일한 컨테이너 값을 가졌는지 확인
- ✔ IsEmpty( )

```
TEST(TestCaseName, TestName) {

    vector<int> arr = { 1, 3, 5, 7, 9 };

    EXPECT_THAT(arr, Contains(3));
}
```

컨테이너 내부에
3이 존재한지 확인

```
TEST(TestCaseName, TestName) {

    vector<int> arr1 = { 1, 3, 5, 7, 9 };
    vector<int> arr2 = { 1, 3, 5, 7, 9 };

    //vector, map 모두 가능
    EXPECT_EQ(arr1, arr2);
    EXPECT_THAT(arr1, ContainerEq(arr2));
}
```

모든 값이 동일한지 확인
(EXPECT_EQ과 동일)

```
TEST(TestCaseName, TestName) {

    vector<int> arr;

    EXPECT_THAT(arr, IsEmpty());
}
```

값이 비어있는지 확인

문자열 Matchers

- ✔ StrEq( ) : 같은 문자열 검사
- ✔ StartsWith() : 시작하는 문자열
- ✔ EndsWith() : 끝나는 문자열
- ✔ MatchesRegex() : 정규식

```
TEST(TestCaseName, TestName) {
    string str1 = "ABCD";
    string str2 = "ABCD";
    EXPECT_THAT(str1, StrEq(str2));
}
```

같은 문자열인지 확인

```
TEST(TestCaseName, TestName) {
    string str = "LORD OF KFC";
    EXPECT_THAT(str, StartsWith("LORD"));
    EXPECT_THAT(str, EndsWith("KFC"));
}
```

LORD로 시작되며, 동시에 KFC로 끝나야함

```
TEST(TestCaseName, TestName) {
    string str = "aB13z";

    EXPECT_THAT(str, MatchesRegex("a.*z"));
}
```

a로 시작하고 z로 끝나야함

[도전] Matcher를 이용한 TDD

배열을 넣으면, 변경된 배열이 Return 되는 모듈

기능

1. 모든 원소 값은 1 씩 더해진다.
2. 원소 값이 10이 되면, 0으로 변경된다.
3. 원소 값이 4가 되면, 즉시 5로 변경된다.
4. 10이상 or 4가 입력되면 exception이 발생된다.

예시

- {1, 2, 3, 2} □ {2, 3, 5, 3}
- {9, 9, 3, 3} □ {0, 0, 5, 5}

Matcher를 이용하여 TDD 개발!

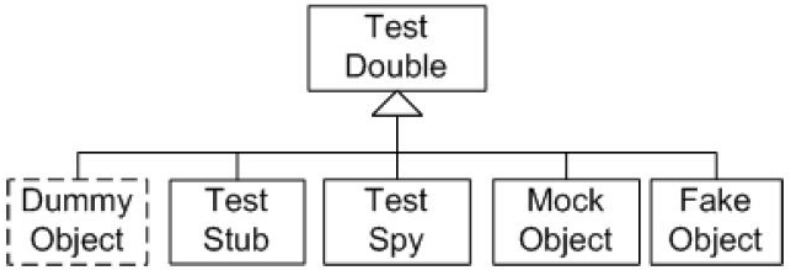
구어로 Mocking이라고 한다.

Test Double (Mocking)

테스트 더블

객체 행동 흉내내기 위함 (스턴트맨과 같은 대역)

일반적으로 다른 객체를 테스트하기 위해 만든다.



테스트 더블 쓰는 이유?

유닛 테스트하기 어려울 때는 많이 발생 한다.

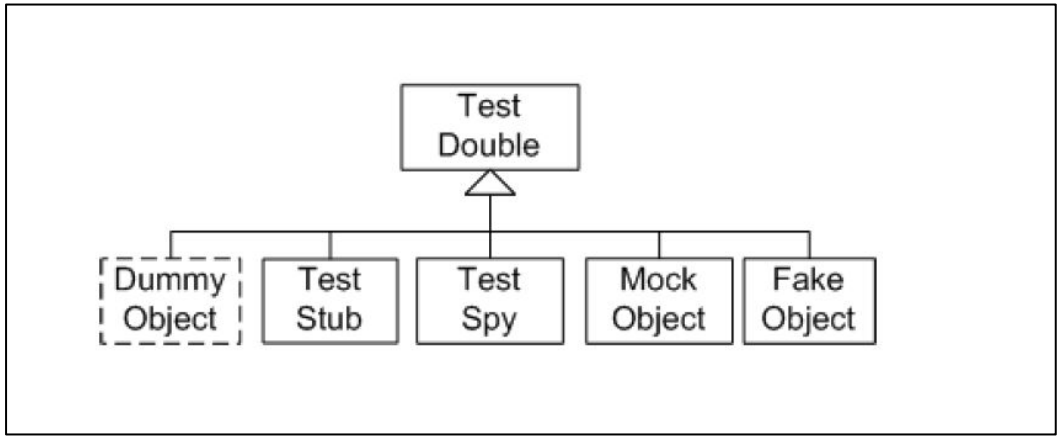
- 서비스를 중단 시키고, 유닛테스트를 해야한다면? ☐ 대역필요
- 오픈시간 등, 제약사항 있는 외부 API를 사용해야한다면? ☐ 대역 필요
- 고객 DB 접근 모듈을 테스트 해야한다면? ☐ 대역 필요
- 미완성된 모듈을 포함해서 테스트해야한다면? ☐ 대역필요

또는 유닛테스트가 너무 느리다면?

☐ 대역 필요

하나씩 의미를 살펴보자.

1. Dummy
2. Stub
3. Spy
4. Mock Object
5. Fake Object



더미 (Dummy)

동작만을 위해 아무런 의미 없이 만드는 객체

- issueTo( )
  - Argument에 누가 발행했는지 넣어줘야 한다.
  - 넣는 대상은, 테스트에 관심사가 아니다.

- numberOfTimesIssued( )
  - 몇 번 Issue (발행)했는지 확인

```
TEST(TestCaseName, TestName) {
    Book book( str: "Cpp Book", text: "123456");

    Member dummyMember;

    book.issueTo(dummyMember);

    EXPECT_EQ(book.numberOfTImesIssued(), 1);
}
```

스텁(Stub)

테스트로 작성된 호출에
미리 준비된 응답을
해주는 기능을 뜻하는 용어이다.

Stub Object

- 더미인데,
  실제로 동작되는 것 처럼 만든 객체

스파이 (Spy)

실제 객체와 완전 동일하다.

스텁 기능도 있다.

그리고 Behavior(행동) 검증도 가능하다.

스텁 예시

- 배열에다가 add("안녕") 를 호출하면 false를 리턴하도록 세팅
- 배열에다가 add("뭐해") 를 호출하면 true를 리턴하도록 세팅

Behavior 검증 예시

- 총 몇 번 호출되었는지 검사
- 순차적으로 이런 메서드가 호출되었는지 검사

스파이 예시

실제와 같은 SPY 객체를 만들어두고,

몰래 역할을 부여한다.

- 실제와 똑같이 행동하도록 해.
- 그런데 만약 이렇게 요청이 들어오면 이렇게 대답하도록 해

서비스와 같은 행동이 다 끝나고,

정보를 몰래 가서 물어본다.

- 서비스 도중에, 이 함수 총 몇 번 호출되었어?

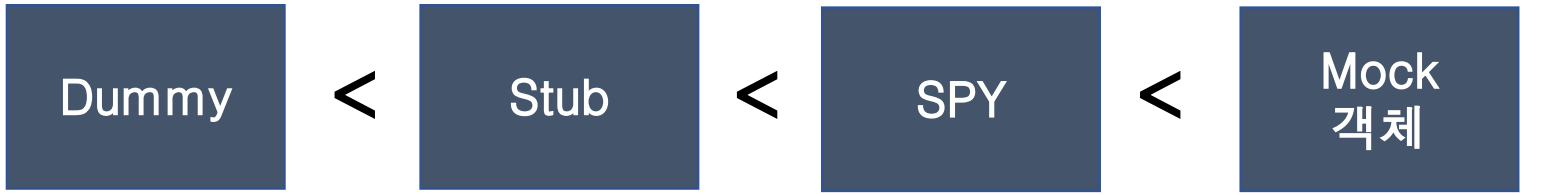
목 (Mock)

SPY와 같은 기능을 가진다. (Stub / Behavior 검증 가능)

그런데 SPY와 달리, 실제 객체가 아니라 가짜 객체이다.

정리

네 가지 Test Double 정리



빌드만 되어라

Dummy +
심심이

Stub 기능을 갖고,
Spy 처럼
Behavior 검증 가능한
진짜객체

SPY 기능을 갖는
가짜객체

가짜라 빠른
테스트가능

Q&A) 전부 SPY 만 쓰면 되지않나?

Mock 은 가짜고,

SPY 는 진짜에 감시 기능을 심는 것이면,

진짜로 테스트하는게 무조건 좋은게 아닌가?

1. API 외부 접속 등.. 상황이 어려울때 Mocking을 쓰는 것인데,
   진짜 객체가 필요하면 아예 Mocking 을 안했을 것이다.

2. 해당 객체 검증이 중요하지 않다면,
   가짜로 만드는 것이 테스트가 더 빨라질 것이다.

Fake object

실제로 동작하는 객체인데,

테스트용으로 만든 가짜 객체

- 가짜 데이터 들을 모두 만들어 두어야한다.

예시

- DB를 제어하는 Class를 유닛테스트를 하고자 한다.

- MySQL 의 고객 DB Data 를 그대로 사용할 수 없으니까,
  테스트를 위한, 테스트용 DB 구축 및 DB 객체를 만들어두고,
  이것으로 UnitTest를 한다.

실제로 잘 동작되는,
그리고 테스트를 위해서 만든 객체를 Fake Object 라고 한다.

Mocking Library

gMock

테스트 대상이 되는 클래스를 준비한다.

✔ 가짜를 만들기 전,
원본 객체를 하나 준비한다.

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

Mock 기본코드 해석 - 1

✔ Mock 객체 생성
- 상속을 통해, 가짜 객체 생성
- Mocking에 사용할 메서드를 지정한다.

```
#pragma once
#include "gtest/gtest.h"
#include "gmock/gmock.h"
#include "../Project10/cal.cpp"

using namespace testing;
using namespace std;

class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
};


TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .Times(2);

    int ret1 = mock_cal.getSum(1, 2);
    int ret2 = mock_cal.getSum(1, 2);
}
```

Mock 기본코드 해석 - 2

✔ Expectation 설정

- getSum(1, 2) 가
  2회 호출 되어야만 한다.

- gMock은
  Test를 수행하기 전,
  미리 설정을 해야한다.

```
#pragma once
#include "gtest/gtest.h"
#include "gmock/gmock.h"
#include "../Project10/cal.cpp"

using namespace testing;
using namespace std;

class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
};


TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .Times(2);

    int ret1 = mock_cal.getSum(1, 2);
    int ret2 = mock_cal.getSum(1, 2);
}
```

Mock 기본코드 해석 - 3

✔ Actual Test 수행

- 실제 테스트를 수행한다.

- 테스트 함수가 끝나면,
  Expectation 대로 수행했는지
  자동 검사된다.

```
#pragma once
#include "gtest/gtest.h"
#include "gmock/gmock.h"
#include "../Project10/cal.cpp"

using namespace testing;
using namespace std;

class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
};


TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .Times(2);

    int ret1 = mock_cal.getSum(1, 2);
    int ret2 = mock_cal.getSum(1, 2);
}
```

[참고] 함수 자체가 호출되었는지 판별 방법

✔ '_' 는 wild card를 뜻한다.

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(_, _))
        .Times(2);

    int ret1 = mock_cal.getSum(1, 2);
    int ret2 = mock_cal.getSum(3, 5);
}
```

Mocking시, 가장 많이 사용되는 두 가지 기능

Stubbing & Behavior Verification

Mock 객체의 메서드 출력 결과 기본 값은 0

mock 객체는 가짜로 동작한다.

- 기본 return 은 0 이다.

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    cout << mock_cal.getSum(3, 5);
}
```

```
CalMock passed

GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: getSum(3, 5)
          Returns: 0
```

Stubbing

✔ A 값이 Parameter로 들어오면,
B 값을 Return 해야 함을 세팅해 두는 것.

✔ **공식용어**

- Stub, Stubbing

✔ Stub을 구현하기 위한 gMock 기능 이름

- Actions

Stubbing 이해 - 1

getSum(1, 2) 호출하면, 100을 리턴.

getSum(5, 6)을 호출하면
처음에는 1을 리턴하고
두번째는 2를 리턴한다.
그 이후로는 모두 50을 리턴한다.

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 100));

    EXPECT_CALL(mock_cal, getSum(5, 6))
        .WillOnce( Return( value: 1))
        .WillOnce( Return( value: 2))
        .WillRepeatedly( Return( value: 50));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
}
```

```
CalMock passed

100
1
2
50
50
```

특정 메서드를 호출했을 때,
어떤 Return 값을 반환할 지 지정 가능

Stubbing 이해 - 2

getSum(1, 2) 호출하면, 100을 리턴.

getSum(5, 6)을 호출하면
처음에는 1을 리턴하고
두번째는 2를 리턴한다.
그 이후로는 모두 50을 리턴한다.

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 100));

    EXPECT_CALL(mock_cal, getSum(5, 6))
        .WillOnce( Return( value: 1))
        .WillOnce( Return( value: 2))
        .WillRepeatedly( Return( value: 50));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
    cout << mock_cal.getSum(5, 6) << "\n";
}
```

```
CalMock passed

100
1
2
50
50
```

특정 메서드를 호출했을 때,
어떤 Return 값을 반환할 지 지정 가능

string 객체 Mocking 하기

✔ length(), Stubbing 하기

```
class MockString : public string {
public:
    MOCK_METHOD(int, length, (), ());
};

TEST(CalTest, CalMock) {
    MockString mock_str;

    EXPECT_CALL(mock_str, length())
        .WillRepeatedly( Return( value: 1));

    cout << mock_str.length() << "\n";
    cout << mock_str.length() << "\n";
}
```

```
CalMock passed

1
1
```

Behavior 검증

✔ 테스트가 끝난 후에,
특정 동작을 수행했는지 확인해 보는 “동작(Behavior)“ 검증

✔ 공식용어
- Behavior Verification

✔ Behavior를 구현한 gMock의 기능 이름
- Cardinality

Behavior 검증 : 호출 되어야 한다.

✔ getSum( ) 메서드가 호출되기를 기대하고 있음
- Times를 생략하면, 정확히 1 회, 호출되어야 Pass

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;
    EXPECT_CALL(mock_cal, getSum(_, _))
        .Times(2);

    int ret1 = mock_cal.getSum(1, 2);
    int ret2 = mock_cal.getSum(3, 5);
}
```

Times(2) : 2회 호출 되어야만 Pass

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum(_, _));

    cout << mock_cal.getSum(1, 2) << "\n";
}
```

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum);

    cout << mock_cal.getSum(10, 1) << "\n";
}
```

모든 인자가 '_' 인 경우는
함수 이름만 적어도 된다.

정상적인 사용과, 잘못된 사용 예시 1

✔ Behavior 검증과 Stubbing을 동시에 하고 싶다면, 메서드 체이닝을 한다.

```
TEST(TestCaseName, TestName) {

    MockCal mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .Times(2)
        .WillRepeatedly(Return(100));

    cout << mock_cal.getSum(1, 2);
    cout << mock_cal.getSum(1, 2);
}
```

Behavior 검증이 먼저 온 후,
Stubbing을 걸어준다.

```
TEST(TestCaseName, TestName) {
    MockCal mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly(Return(100))
        .Times(2);

    cout << mock_cal.getSum(1, 2);
    cout << mock_cal.getSum(1, 2);
}
```

컴파일 에러 발생
(구글테스트 제약사항)

잘못된 사용 예시 2

✔ 메서드 체이닝 없이, 따로 작성하면 정상 동작 되지 않는다.

```
TEST(TestCaseName, TestName) {
    MockCal mock_cal;
    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly(Return(100));

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .Times(2);

    cout << mock_cal.getSum(1, 2);
    cout << mock_cal.getSum(1, 2);
}
```

Test Pass 이지만,
Times 는 적용되지만,
Stubbing 건 것은 사라진다.

```
TEST(TestCaseName, TestName) {
    MockCal mock_cal;

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .Times(2);

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly(Return(100));

    cout << mock_cal.getSum(1, 2);
    cout << mock_cal.getSum(1, 2);
}
```

Test Fail
Times(2) 에서
두번 호출되지 않아 에러 발생

Matcher 사용

✔ 기대하는 호출
- getSum(4가 아닌 수, 1) 을 1회만 호출.

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum(Ne(4), Eq(1)));

    cout << mock_cal.getSum(10, 1) << "\n";
}
```

Behavior 검증 : Cardinality

✔ **카더널리티**

- Times(0) : 호출되면 안됨
- Times(n) : 정확히 n 번 호출
- Times(AtLeast(n)) : n 번 이상
- Times(AtMost(n)) : n 번 이하
- Times(Between(n1, n2)) : n1 ~ n2 회

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum)
        .Times(AtLeast(n: 2));

    cout << mock_cal.getSum(10, 1) << "\n";
    cout << mock_cal.getSum(10, 1) << "\n";
    cout << mock_cal.getSum(10, 1) << "\n";
}
```

2번 이상 호출 되어야만 한다.

[참고] 호출 순서 검증 가능

✔ 반드시
해당 순서에 맞게
동작해야만 Pass

```
class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
    MOCK_METHOD(int, getGop, (int a, int b), ());
};

TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    {
        InSequence seq;
        EXPECT_CALL(mock_cal, getSum(1, 1));
        EXPECT_CALL(mock_cal, getGop(1, 2));
        EXPECT_CALL(mock_cal, getSum(1, 3));
        EXPECT_CALL(mock_cal, getGop(1, 4));
    }

    cout << mock_cal.getSum(1, 1) << "\n";
    cout << mock_cal.getGop(1, 2) << "\n";
    cout << mock_cal.getSum(1, 3) << "\n";
    cout << mock_cal.getGop(1, 4) << "\n";
}
```

[도전] 처음부터 구현해보기

mock 객체 만들고 Cal 클래스에
0을 리턴하는 getValue( ) 메서드를 제작한다.

1. getValue() 호출 시 10000을 리턴하는 Mock 객체를 만든다.
   (Stub)

2. EXPECT_THAT 으로 getValue() 호출 시, 10000 이 맞는지 검사하기
   (State 검증)

3. getValue( )가 2회 호출되어야 Pass 설정하기
   (Behavior 검증)

[정리] Stubbing vs Behavior Verification

Stubbing

- A 값을 Parameter로 넣으면, B 값이 return 되도록 세팅

Behavior Verification

- Test 종료 후, 특정 메서드가 동작 되었는지 검증 가능
- 호출 횟수 / 순서대로 호출 여부 등 확인 가능

Stubbing 응용

여러 번 Stub 한 경우는?

✔ 마지막으로 Stubbing한 것으로 적용된다.

```
TEST(CalTest, CalMock) {
    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 10));

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 20));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getSum(1, 2) << "\n";
}
```

```
CalMock passed

20
20
```

Cal Class에 메서드를 추가 후, 테스트 해보기

✔ getValue 메서드 추가

```
class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
    MOCK_METHOD(int, getValue, (), ());
};

TEST(CalTest, CalMock) {

    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 10));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getValue() << "\n";
}
```

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

    int getValue() {
        return x;
    }

private:
    int x = 100;
};
```

테스트 결과

✔ 10과 0이 출력이 된다.

- 단, Stub이 안된 메서드를 출력했다는 Warning이 뜬다.

```
class MockCalculator : public Cal {
public:
    MOCK_METHOD(int, getSum, (int a, int b), ());
    MOCK_METHOD(int, getValue, (), ());
};

TEST(CalTest, CalMock) {

    MockCalculator mock_cal;

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 10));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getValue() << "\n";
}
```

```
CalMock passed

10
GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: getValue()
          Returns: 0
NOTE: You can safely ignore the above warning unless this call shou
  not suppress it by blindly adding an EXPECT_CALL() if you don't m
  call.  See https://github.com/google/googletest/blob/master/docs/
  gmock_cook_book.md#knowing-when-to-expect for details.
0
```

NiceMock / StrictMock - 1

✔ Stub 이 안된 Mock Method를 호출할 때,
Pass + Warning / Pass / Fail 을 선택할 수 있다.

```
TEST(CalTest, CalMock) {

    NiceMock<MockCalculator> mock_cal;

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly( Return( value: 10));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getValue() << "\n";
}
```

```
CalMock passed

10
0
```

NiceMock은
경고 메시지가 뜨지 않음

```
TEST(CalTest, CalMock) {

    StrictMock<MockCalculator> mock_cal;

    EXPECT_CALL(mock_cal, getSum(1, 2))
        .WillRepeatedly(Return(10));

    cout << mock_cal.getSum(1, 2) << "\n";
    cout << mock_cal.getValue() << "\n";
}
```

```
CalMock failed

10
unknown file: e
  mock functior
  default value
    Function ca
          Retur
0
```

StrictMock은
에러처리가 된다.

NiceMock / StrictMock - 2

✔ EXPECT_CALL 을 호출하지 않고 Mock Method 호출시
Pass + Warning / Pass / Fail 을 선택할 수 있다.

```
TEST(CalTest, CalMock) {

    NiceMock<MockCalculator> mock_cal;
    cout << mock_cal.getSum(1, 2) << "\n";
}
```

```
CalMock passed

0
```

NiceMock은
경고 메시지가 뜨지 않음

```
TEST(CalTest, CalMock) {

    StrictMock<MockCalculator> mock_cal;
    cout << mock_cal.getSum(1, 2) << "\n";
}
```

```
CalMock failed

unknown file: error
  mock function cal
  default value.
    Function call:
          Returns:
0
```

StrictMock은
에러처리가 된다.

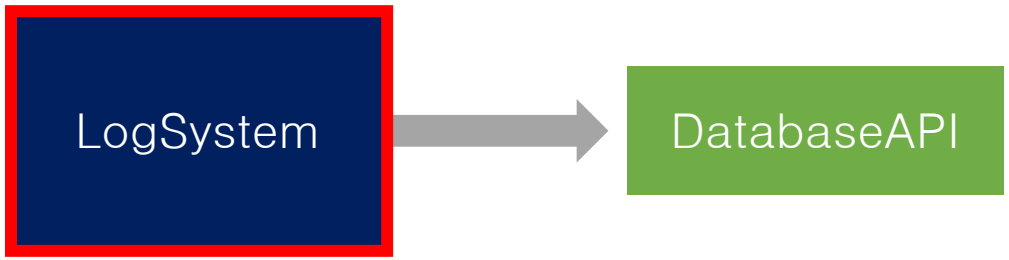
목 주입하기

Mock Injection

다음과 같이 구현하자.

✔ **테스트 하고자 하는 모듈**
- LogSystem

✔ **LogSystem의 의존성 모듈**
- DatabaseAPI

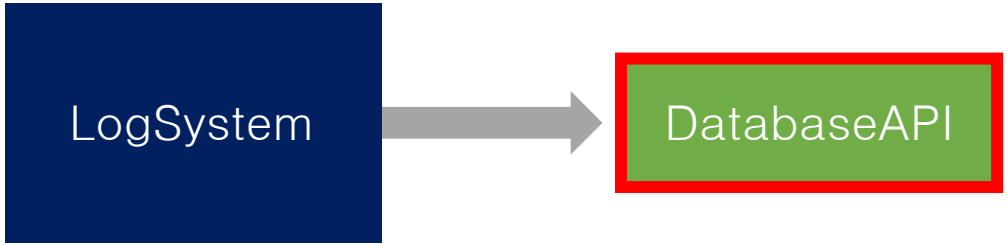



```
class LogSystem {
public:
    string getLogMessage(string content) {
        string msg = "";
        msg += string("[") + DB.getDBName() + string("] ");
        msg += content + string("\n");
        return msg;
    }

private:
    DatabaseAPI DB;
};
```

```
class DatabaseAPI {
public:
    string getDBName() {
        return name;
    }

private:
    string name = "MySon_DB";
};
```

테스트 할 때, 문제점

✔ **테스트 하고자 하는 모듈**
- LogSystem

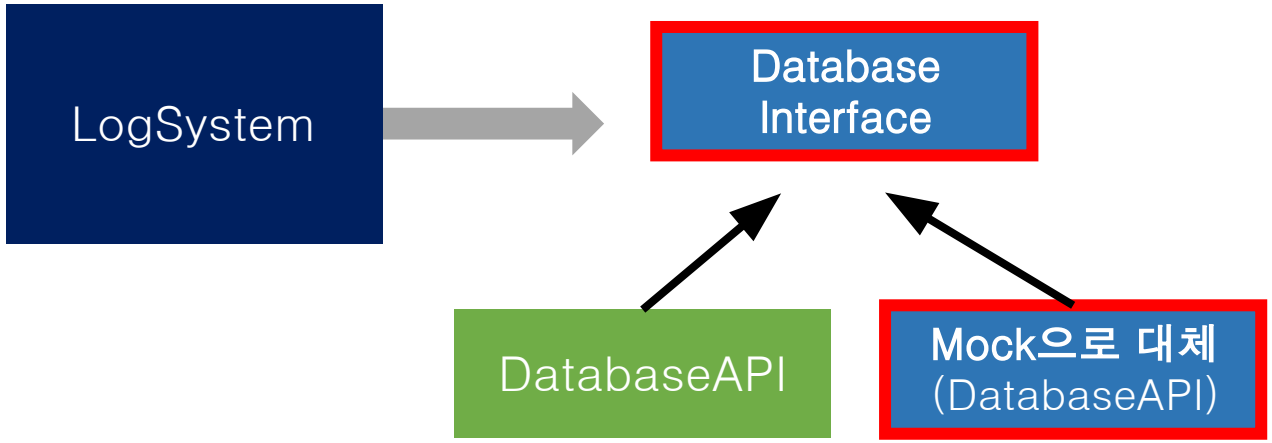


✔ **DatabaseAPI 동작**
- Unit Test를 할 때 마다,
  Database의 getDBName API를 수행하고,
  서버 트래픽을 발생시킨다고 가정한다.

- 이로 인해, 테스트 할 때마다 서버 성능이 낮아진다.

```
class DatabaseAPI {
public:
    string getDBName() {
        return name;
    }

private:
    string name = "MySon_DB";
};
```

Mock으로 대체



✔ **테스트 하고자 하는 모듈**
- LogSystem

✔ Mocking을 하면 트래픽 문제 없이
테스트 하고자 하는 모듈을 테스트가 할 수 있다.

```
class LogSystem {
public:
    string getLogMessage(string content) {
        string msg = "";
        msg += string("[") + DB.getDBName() + string("] ");
        msg += content + string("\n");
        return msg;
    }

private:
    DatabaseAPI DB;
};
```

이 DB 객체를 Mock객체로 변경필요

Mock으로 대체를 위한 준비작업

✔ Database 객체를
내부에서 생성하는 것이 아닌,
밖에서 주입해주는, DI 로 리팩토링

```
using namespace std;
interface DBAPI {
    virtual string getDBName() = 0;
};

class DatabaseAPI : public DBAPI{
public:
    string getDBName() {
        return name;
    }
private:
    string name = "MySon_DB";
};
```

```
class LogSystem {
public:
    explicit LogSystem(DBAPI* db)
        : DB(db) {
    }

    string getLogMessage(string content) {
        string msg = "";
        msg += string("[") + DB->getDBName() + string("] ");
        msg += content + string("\n");
        return msg;
    }

private:
    DBAPI *DB;
};
```

유닛테스트가 가능하도록 리팩토링 하는 것이다.

Mock 주입하기

✔ Mock 객체를 준비하여, DI 해준다.

```
class DBMock : public DBAPI {
public:
    MOCK_METHOD(string, getDBName, (), (override));
};

TEST(TestCaseName, TestName) {

    DBMock db_mock;
    EXPECT_CALL(db_mock, getDBName)
        .WillRepeatedly(Return(string("TEST DOUBLE")));

    LogSystem system(&db_mock);
    cout << system.getLogMessage("KFC");
}
```

```
TestName passed

[TEST DOUBLE] KFC
```

Mock 주입하여 Unit Test 하기

✔ LogSystem 의 DB 트래픽이 없다.

✔ 지금까지 수행한 일

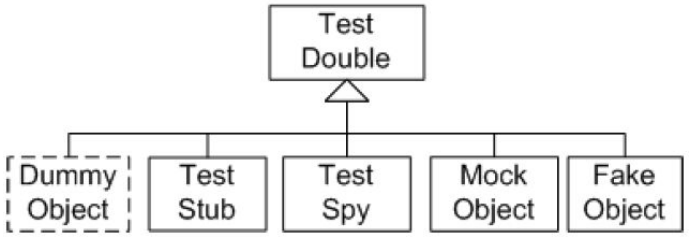
1. Unit Test가 가능하도록 리팩토링
2. Mock 객체 생성
3. Mock 객체 주입

```
class DBMock : public DBAPI {
public:
    MOCK_METHOD(string, getDBName, (), (override));
};

TEST(TestCaseName, TestName) {

    DBMock db_mock;
    EXPECT_CALL(db_mock, getDBName)
        .WillRepeatedly(Return(string("TEST DOUBLE")));

    LogSystem system(&db_mock);
    //cout << system.getLogMessage("KFC");
    string actual = system.getLogMessage("KFC");

    //state 검증 추가
    EXPECT_THAT(actual, StrEq(string("[TEST DOUBLE] KFC\n")));
}
```

[중요] Mock Library를 써야 하는 이유

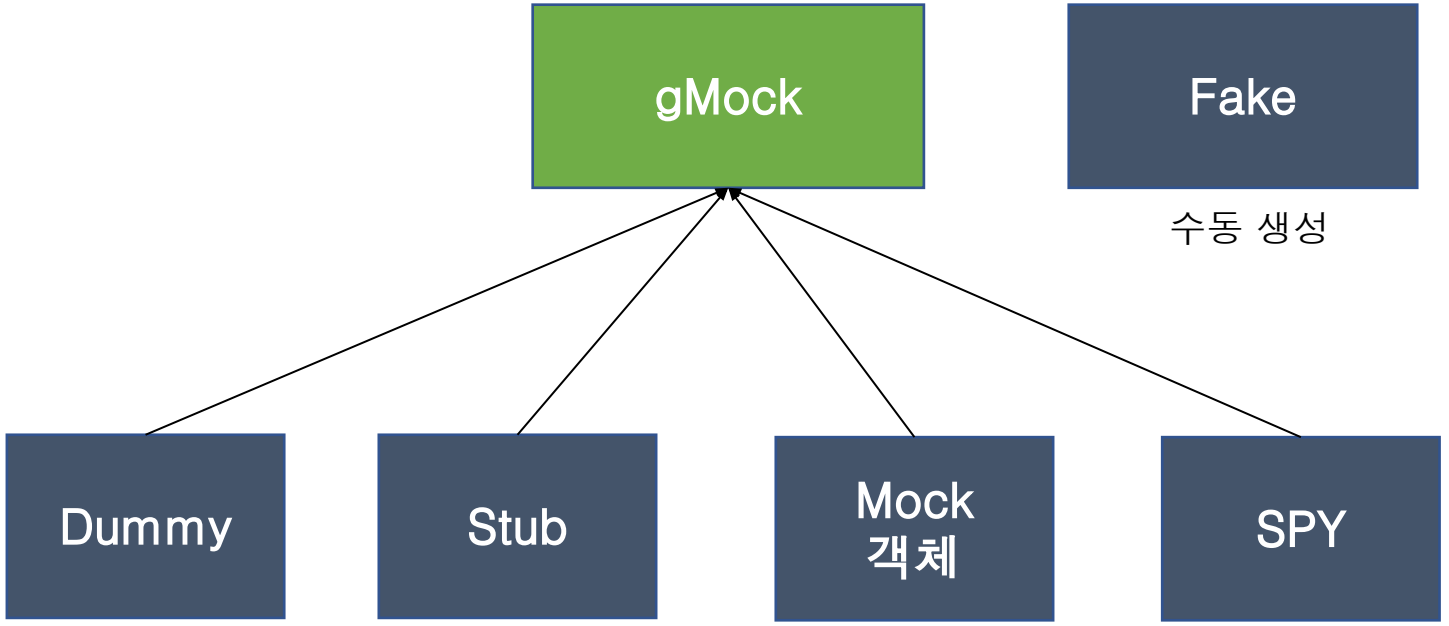
Mock Library를 쓰는 이유

- 직접 만들면, 개발자 마다 구현 방법이 모두 다르다. (유지보수의 어려움)
- 가장 유명한 Library를 사용하면, 같은 방법으로 구현한다.



Dummy>Stub>Spy>Mock || Fake

이론적 Test Double 구성



Library 사용