

# Test Double (with pytest)

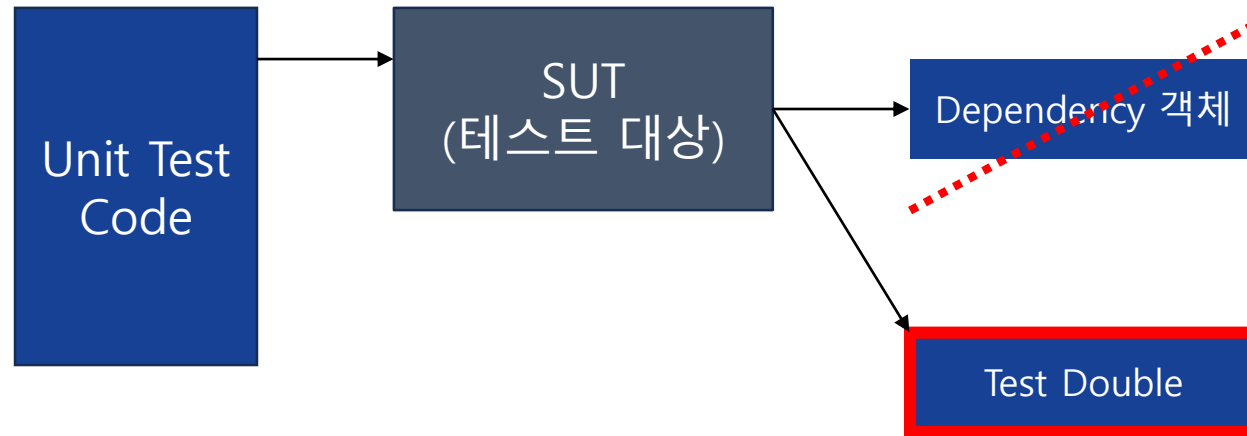


# Test Double

# 테스트 더블

실제 객체 대신 사용하는 가짜 객체이다.

일반적으로 테스트하고자 하는 객체를 대체하는 것이 아니라,  
테스트하고자하는 객체의 의존객체(dependency)를 대체한다.



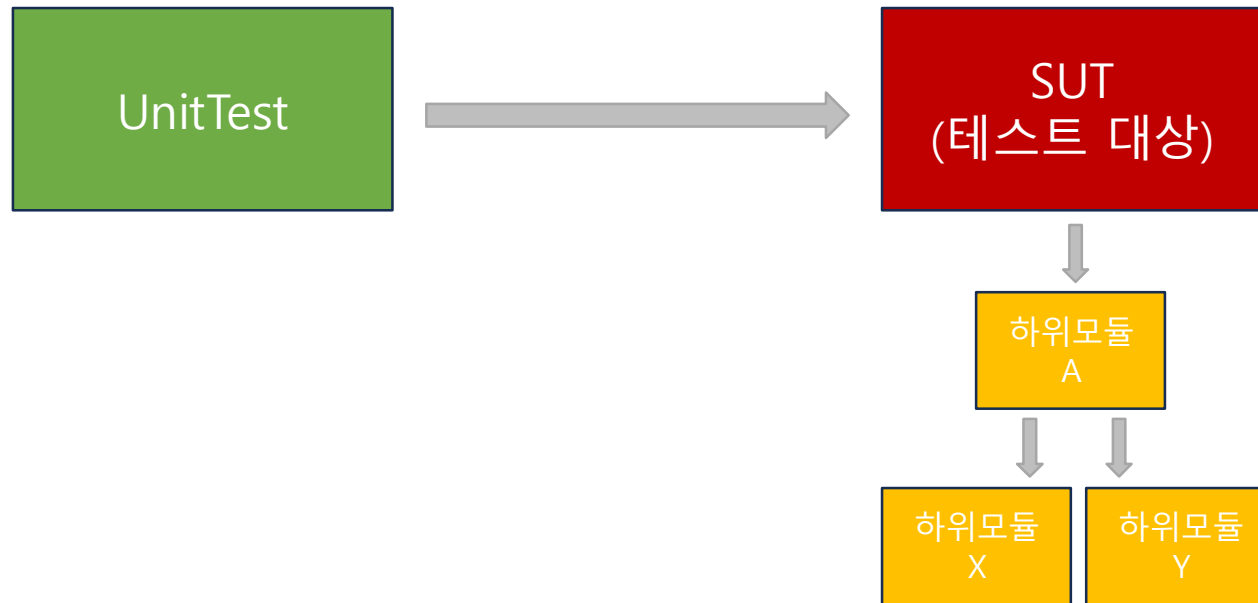
SUT 의 의존객체를 대체한다

# 테스트 더블을 쓸 수 있는 상황 -1

현재 설계에서 SUT를 생성하기 위해서는?

SUT를 생성하려면, 하위 모듈을 만들어 생성자에 넣어주어야 한다.

- ex) `sut = SUT(A(X, Y))`

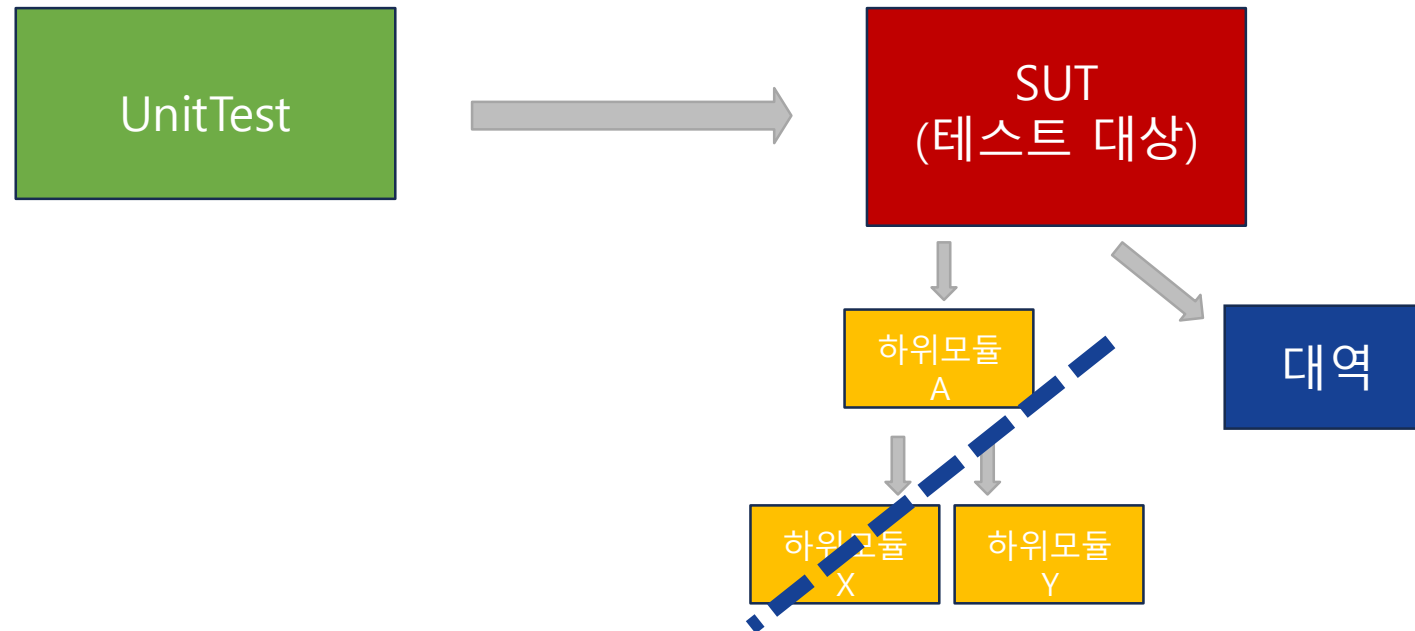


SUT 는 하위 모듈 A 에 의존하고 있다

## 테스트 더블을 쓸 수 있는 상황 -2

Unit Test에서 테스트하고 싶은 대상은 SUT의 코드다.

SUT 모듈만 테스트하면 되는데, 의존성 때문에 테스트가 어려워진다.  
하위 모듈 A는 대역을 사용하여 테스트하기 쉽게 만들어 준다.



하위 모듈 A를 준비하기 어려운 상황에서,  
SUT 테스트하기 위해 대역을 사용한다.

## 테스트 더블 쓰는 이유?

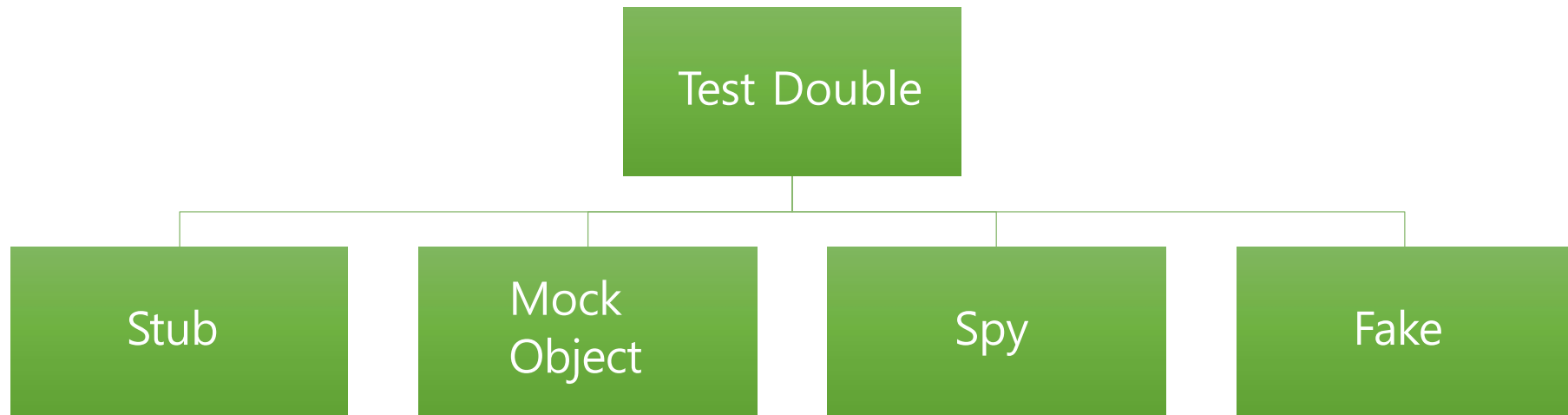
SUT가 다음과 같은 하위 모듈을 사용할 때, 대역을 검토한다.

- 오픈시간 등, 제약사항 있는 외부 API를 사용하는 하위 모듈
- 같은 파라미터인데, 상황에 따라 매번 다른 값이 리턴될 수 있는 하위 모듈
- 미완성된 모듈
- 너무 느린 모듈

등등....

# Test Double 유형

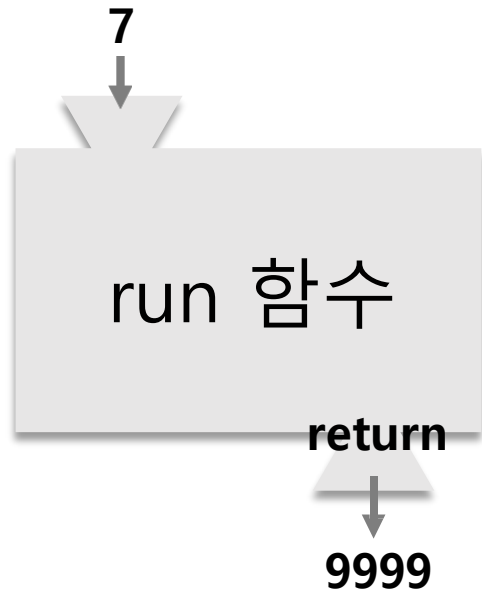
1. Dummy (컴파일만 되도록)
2. Stub
3. Mock Object
4. Spy
5. Fake



# 스텝 (Stub)

특정 값을 리턴하도록 미리 설정된 객체

e.g. run 함수에 7이 들어오면 무조건 9999 를 리턴 하도록 세팅



“Stubbing을 한다” 라고 표현한다.

```
class Monster:
    def move(self):
        ret = run(7)
        ...
```

ret 에는 9999 가 입력된다.



## 위이크 (Fake)

- 테스트용으로 사용하기 위해 만든 가짜 객체로 보통 가짜 데이터를 미리 만들어 두고 사용한다.

예시 : 테스트 목적의 DB , 테스트 전용 API Server

```
class Monster :  
    def __init__( ):  
        self.db = RealDB()  
  
    def attack( p ) :  
        self.db.record_attack(p)
```

**실제 객체**

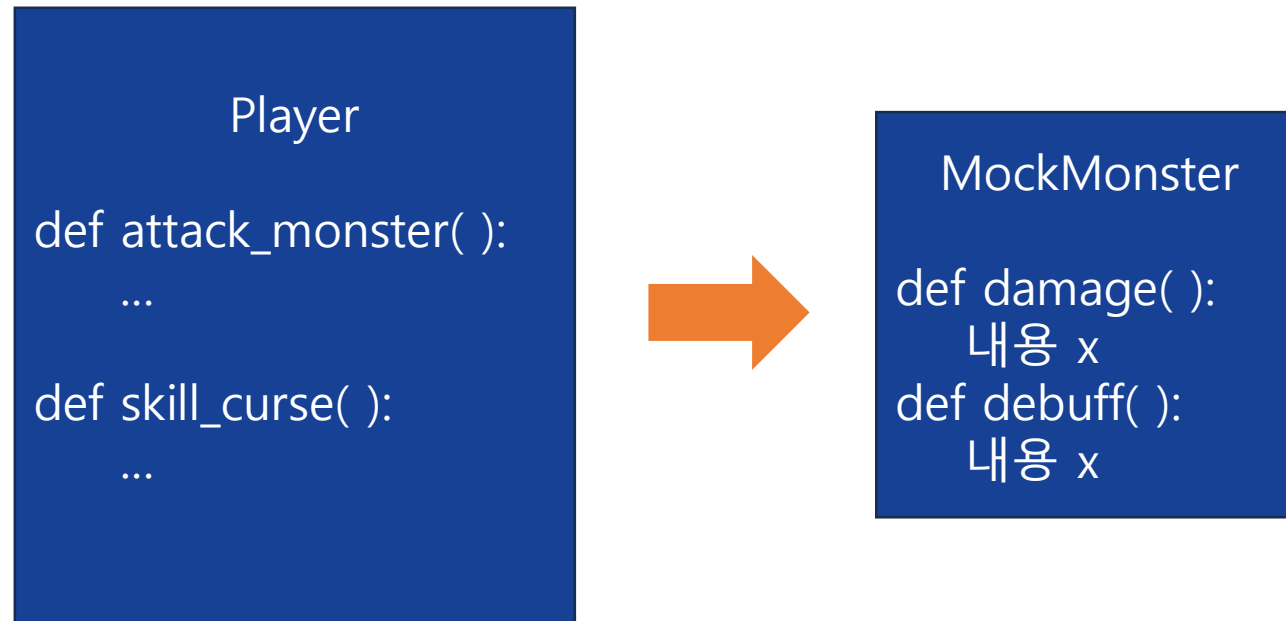
```
class Monster :  
    def __init__( ):  
        self.db = FakeDB()  
  
    def attack( p ) :  
        self.db.record_attack(p)
```

**Fake (대역)**

```
class FakeDB :  
    def __init__( ):  
        self.records = [ ]  
  
    def record_attack( power ):  
        self.records.append(power)
```

## 목 객체 (Mock Object)

호출 여부, 호출 횟수, 호출 순서를 검증하는 데 사용되는 객체로 주로 Mock 라이브러리가 생성해준다.



Test 코드에서 Mock 을 이용하면 다음과 같은 내용을 검증할 수 있다.

- Player 가 `attack_monster` 를 사용했을 때, `damage(10)` 을 2회 호출한다.
- Player 가 `skill_curse` 를 사용했을 때, `damage` 1회 , `debuff` 2회를 순서대로 호출한다.

## 스파이 (Spy)

Mock 처럼 호출 기록을 하며, Mock 과 달리 실제 객체와 동일하게 동작을 한다

Mock 은 라이브러리가 생성해주는 반면, Spy는 직접 만들어서 사용하는 편이다.

로깅을 해서 모니터링용으로 사용할 수 있다

### MockMonster

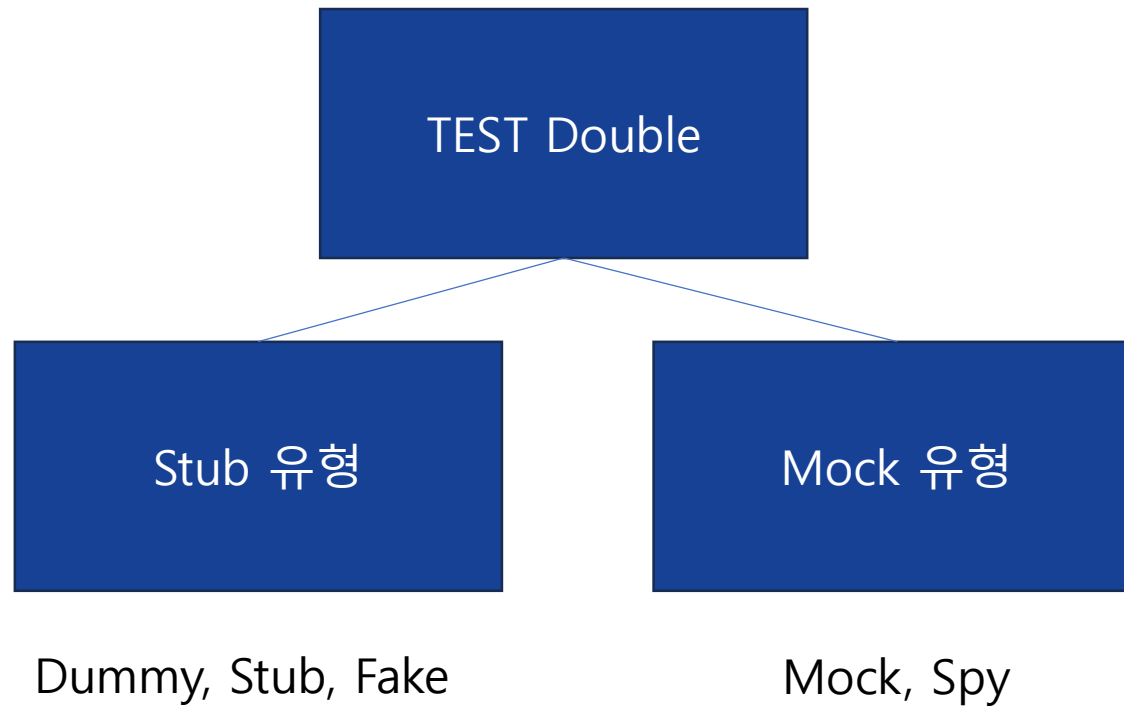
```
def damage( ):
    내용 x
def debuff( ):
    내용 x
```

### SpyMonster

```
def damage( ):
    실제 내용 + 기록
def debuff( ):
    실제 내용 + 기록
```

# Test Double 정리

Test Double 의 4가지 분류를 Stub 과 Mock 으로 구분 지을 수 있다.



## 목 객체 (Mock Object) 의 단점

### 실제 동작의 결과보다는 상호작용을 검증

Mock 을 사용하면 "이 메서드를 몇 번 호출했는가" 같은 호출 여부만 확인하게 된다.  
이는 테스트하고자 하는 실제 결과의 본질과 거리가 있다.

### Mock 으로 작성시 테스트가 잘 깨지기 쉽다

Mock 은 함수 호출 횟수, 함수 호출 순서 등과 같은 구현 사항을 검증하므로  
해당 구현사항이 변경되면 테스트가 쉽게 깨진다.

# Stub 유형, Mock 유형

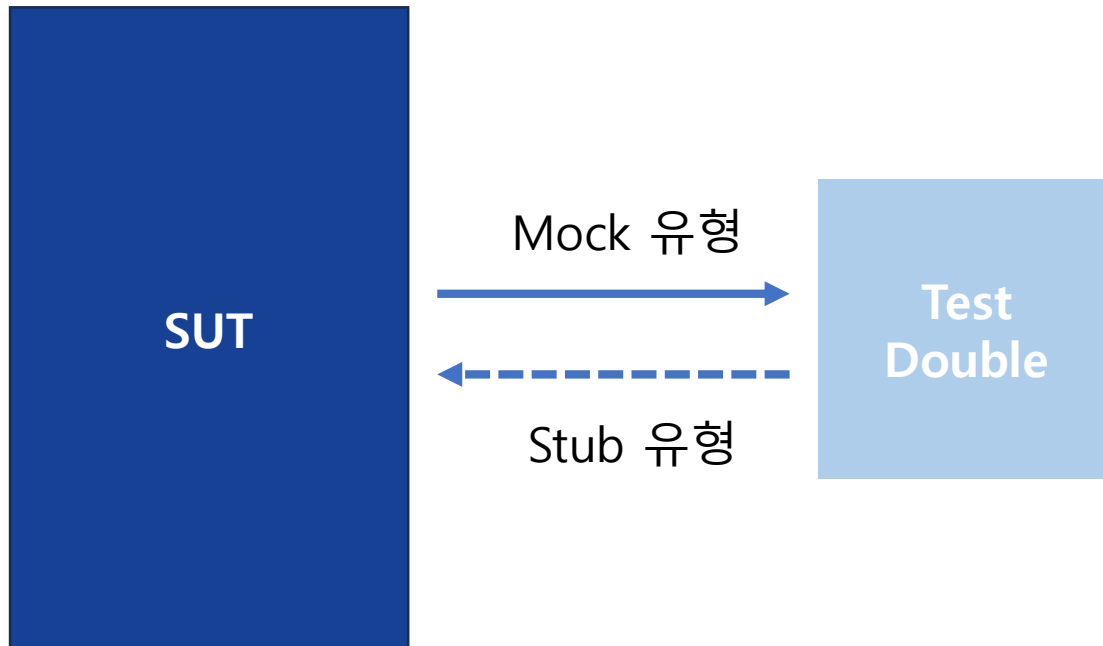
## Mock 유형 ( Mock, Spy )

SUT에서부터 **외부로 나가는 상호작용을** 검사하는 데 사용한다.

## Stub 유형 ( Stub, Fake )

Dependency로부터 SUT **내부로 들어오는 상호작용을** 모방하는 데 사용한다.

Dependency 에서 SUT로 입력되는 정보를 얻을 수 있다.



```
class SUT: 1 usage new *
    def __init__(self, test_double):
        self.dep = test_double

    def run(self): 1 usage new *
        ret = self.dep.method()
        ...
```



# 기본 Mock 사용방법

stubbing 과 behavior 검증

# mock 학습을 위한 기본 코드 작성

- cal.py 생성 : 객체
- test\_cal.py 생성 : 테스트 코드
- pytest-mock 설치

```
cal.py x
1 class Cal:
2     def get_sum(self, a, b):
3         return a + b
4
5     def say_hi(self):
6         return 'HI'
7
8     def say_hello(self):
9         return 'HELLO'
10
```

```
from calc import Cal

def test_calc(): new *
    assert 1 == 1
```

```
✓ Test Results
  ✓ test_cal
    ✓ TestCal
      ✓ test_cal
```



# Mock object 생성

## pytest\_mock 에서 제공하는 mocker fixture 를 이용한다

mock.Mock() 을 사용하면, Mock Object가 생성된다.

```
from pytest_mock import MockerFixture

def test_calc(mocker: MockerFixture):
    mk = mocker.Mock()
```

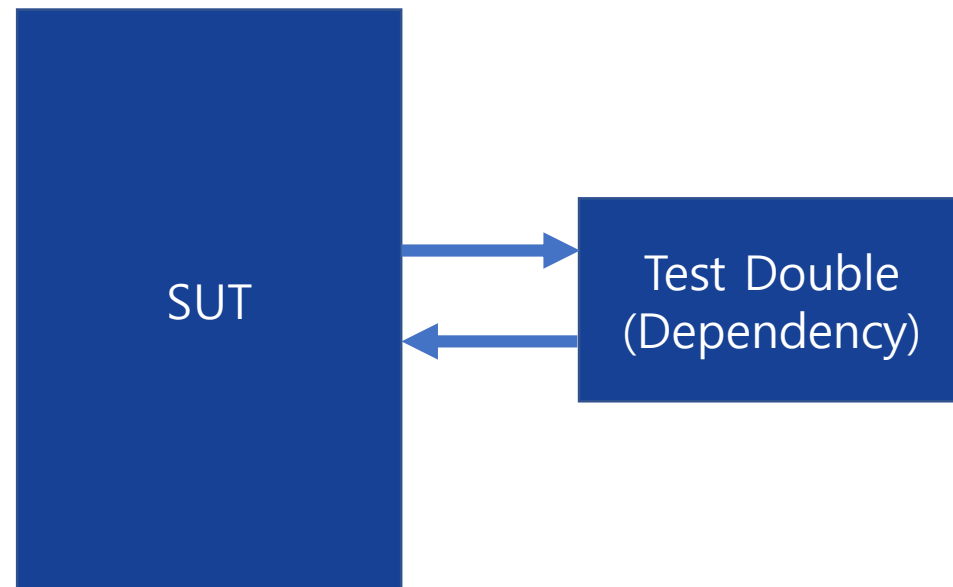
Library에서는 앞선 분류와 다르게  
Mock Object 를 가지고 Mock / Stub 둘 다 다룬다

# Mock object 로 할 수 있는 것

## Mock object 를 이용해서

1. stub(stubbing) : mock 객체가 SUT 코드 쪽에 특정값을 제공하도록 한다.
2. mock(상호작용 검증) : SUT 코드에서 mock 객체를 어떻게 호출했는지 검증한다.  
(횟수,인자 등 검증)

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock()  
  
    sut = Calc(mk)  
    sut.run() # 내부에서 mk와 상호작용
```



SUT 와 상호작용하는 부분은 뒤에서 다루고 우선은 라이브러리 사용법부터 익혀보도록 한다

# Stubbing

Mock 을 통해서 stubbing 을 할 수 있다.

```
from pytest_mock import MockFixture

from calc import Cal

def test_calc(mock: MockFixture):
    mk: Cal
    mk = mock.Mock(spec=Cal)
    mk.get_sum.return_value = 3

    assert mk.get_sum(5, 5) == 3
```

특정 메서드가 호출되었는지 확인 가능

```
def test_calc(mock: MockFixture):  
    mk = mock.Mock(spec=Calc)  
    mk.get_sum(1, 2)  
  
    mk.get_sum.assert_called()
```

## [도전] Mock 객체 사용해보기

Real Class 제작 후 , Mock Object 를 이용해서 stub 과 행동검증을 한다.

- stub 1 : abc() 호출시 10 리턴
- stub 2 : bts() 호출시 20 리턴
- 검증 1 : abc와 bts의 합이 30인지 테스트
- 검증 2 : abc와 bts가 한번씩 호출 되었는지 확인



# Stubbing

stubbing 과 behavior 검증

## Stubbing – return\_value

간단한 stub 일때 return\_value 를 사용한다.

```
from pytest_mock import MockFixture

from calc import Cal

def test_calc(mock: MockFixture):
    mk: Cal
    mk = mock.Mock(spec=Cal)
    mk.get_sum.return_value = 3

    assert mk.get_sum(5, 5) == 3
```

## Stubbing – side\_effect

stubbing 으로 side\_effect 를 이용하면,

1. Sequence 를 할당해 여러값들을 차례대로 리턴할 수 있다.
2. 다른 함수를 지정할 수 있다.
3. 예외를 stubbing 할 수 있다.

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
    mk.get_sum.side_effect =
```

1,2,3 내용



## side\_effect sequence

Sequence 를 할당해 여러값들을 차례대로 리턴할 수 있다.

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
    mk.get_sum.side_effect = [3,1,4,1,5,"hi"]  
    print()  
    print(mk.get_sum())  
    print(mk.get_sum())  
    print(mk.get_sum())  
    print(mk.get_sum())  
    print(mk.get_sum())
```

3  
1  
4  
1  
5  
hi

## side\_effect 다른 함수 지정

다른 함수를 지정해서 사용할 수 있다

```
def func(x):  
    return x * 2  
  
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Calc)  
    mk.get_sum.side_effect = func  
  
    assert mk.get_sum(30) == 60
```

## side\_effect 예외 stubbing

예외발생을 stubbing 할 수 있다.

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Calc)  
    mk.get_sum.side_effect = ValueError("잘못된 인자")  
  
    with pytest.raises(ValueError, match="잘못된 인자"):  
        mk.get_sum(5, 5)
```

## [도전] Argument 값에 따라 다른 값 return

- Stub 해보기

mock 객체의 `get_sum( 1, 2 )` 호출시 : 1000 리턴

mock 객체의 `get_sum( 2, 3 )` 호출시 : 2000 리턴

mock 객체의 `get_sum( 9, 9 )` 호출시 : 5000 리턴

나머지 값은 Exception 발생

- 힌트 : `side_effect`와 `custom function`을 사용한다.



# Behavior 검증

stubbing 과 behavior 검증

## assert\_called()

메서드가 호출 된 적이 있는지 검사하는 Assertion

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
  
    mk.get_sum(2,3)  
    mk.get_sum(2,3)  
    mk.get_sum(2,3)  
  
    mk.get_sum.assert_called()
```

## assert\_called\_once( )

단 1회 호출 되어야만 PASS

```
def test_calc(mock: MockerFixture):  
    mk = mock.Mock(spec=Cal)  
  
    mk.get_sum(2, 3)  
  
    mk.get_sum.assert_called_once()
```

## assert\_called\_with(아규먼트)

마지막으로 호출된 호출인자를 확인한다

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
  
    mk.get_sum(2, 3)  
    mk.get_sum(2, 3)  
    mk.get_sum(2, 3)  
    mk.get_sum(1, 2)  
  
    mk.get_sum.assert_called_with(1, 2)
```



## 호출 인자 확인

호출인자 값을 읽어서 검증에 활용할 수 있다

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
  
    mk.say_hi("hi", 123)  
    args, kwargs = mk.say_hi.call_args  
  
    print()  
    print(args, kwargs)
```

## [참고] 메서드 호출 순서 , 인자 검사

- import call 필요
- 반드시 해당 순서대로 호출되어야만 PASS

```
from unittest.mock import call

def test_calc(mock: MockerFixture):
    mk = mocker.Mock(spec=Cal)

    mk.get_sum(2, 3)
    mk.get_sum(5, 3)
    mk.get_sum(1)

    mk.get_sum.assert_has_calls([call(2, 3), call(5, 3), call(1)])
```

## 총 몇 회 호출되었는지 확인

- call\_count 로 확인 가능

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
  
    mk.get_sum(2, 3)  
    mk.get_sum(5, 3)  
    mk.say_hi(1)  
  
    assert mk.get_sum.call_count == 2  
    assert mk.say_hi.call_count == 1
```

## [참고] 호출 기록 확인

mock\_calls 를 이용하면 호출된 기록을 확인할 수 있다.

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock(spec=Cal)  
  
    mk.get_sum(2, 3)  
    mk.get_sum(5, 3)  
    mk.say_hi(1)  
  
    print()  
    print(mk.mock_calls)
```

## [도전] 호출 횟수 Behavior 검사

size( ) 호출 횟수가

2번 이상, 5 번 이하일때 PASS 이고

그렇지 않으면 FAIL 이 발생하도록 만들기

## [도전] 1,2,3,ValueError

side\_effect를 이용해서 4번째 호출에 ValueError가 나오도록 stub 을 걸어본다

1. 4번째에 ValueError 가 나오는지 검증한다
2. 호출이 4번 되었는지 검증하는 코드를 작성한다



# Patching

실제 객체의 일부 메서드를 가짜로 변경

# 기존 객체의 일부를 Mock으로 변경

## 인스턴스 생성 후, 메서드를 Mock( ) 으로 변경 가능

- Python 에서는 외부에서 Class 내부 Attribute를 변경할 수 있지만, 캡슐화 원칙을 어기는 것으로 이러한 방법은 권장되지 않는다.
- @property 속성의 메서드는 외부에서 변경 불가.
- 원래대로 되돌려야 할 경우 직접 복원해야 함

```
def test_calc(mock: MockerFixture):  
    c = Cal()  
    c.say_hi = mock.Mock()  
    c.say_hi.return_value = 3  
  
    print(c.say_hi())
```



# mock.patch 사용하기 1


mock.patch를 사용해서 patch(임시로 덮어쓰우기) 를 할 수 있다.

- Mocker 는 pytest\_mock.MockerFixture 타입이다.  
타입 힌트를 사용하면 메서드 검색하기 쉽다.

- patch하는 메서드명을 문자열로 작성한다  
mock.patch('파일명.클래스명.메서드명')

```
import pytest_mock

from calc import Cal

def test_cal(mock: pytest_mock.MockerFixture): new *
    mock_method1 = mock.patch('calc.Cal.method1')
     mock_method2 = mock.patch('calc.Cal.method2')
```

# mock.patch 사용하기 2

## return 값 변경하기

Patch 가 된 메서드의 리턴값을 변경할 수 있다.

```
def test_cal(mock: pytest_mock.MockFixture): new *  
    mock_method1 = mock.patch('calc.Cal.method1')  
    mock_method2 = mock.patch('calc.Cal.method2')  
  
    mock_method1.return_value = 'KFC!'  
    mock_method2.return_value = 'BBQ!'  
  
    sut = Cal()  
    assert sut.method1() == 'KFC!'  
    assert sut.method2() == 'BBQ!'
```

```
class Cal: 2 usages new *  
    def method1(self): 1 usage new *  
        return None  
  
    def method2(self): 1 usage new *  
        return None
```

Cal.method1, Cal.method2 가 교체 되었다

## [도전] patch 코드 이해하기

- method1 을 Stubbing 하는 소스코드 이해해보자.

```
def mock_get_sum(a, b): 1 usage new *  
    if (a, b) == (1, 2): return 10  
    if (a, b) == (1, 5): return 20  
    if (a, b) == (2, 4): return 50  
    return a + b  
  
def test_cal(mock: pytest_mock.MockFixture): new *  
    mock_method1 = mock.patch('calc.Cal.method1')  
    mock_method1.side_effect = mock_get_sum  
  
    sut = Cal()  
    assert sut.method1(1, 5) == 20
```

# Side Effect 적용하기

- 호출 순서대로 return 값 결정하기

```
def test_cal(mock: pytest_mock.MockFixture):  
    mock_method1 = mocker.patch('calc.Cal.method1')  
    mock_method1.side_effect = ['AA', 'BB', 'CC']  
  
    sut = Cal()  
  
    assert sut.method1() == 'AA'  
    assert sut.method1() == 'BB'  
    assert sut.method1() == 'CC'
```

AA

BB

CC

## [도전] get\_gop\_three

- 다음 Test의 출력 결과를 예상해보고 실험해보자.

```
def test_cal(mock: pytest_mock.MockFixture):  
    mock_sum = mock.patch('calc.Cal.get_sum')  
    mock_sum.return_value = 100  
  
    sut = Cal()  
    print(sut.get_gop_three(1, 2))  
    print(mock_sum.call_count)
```

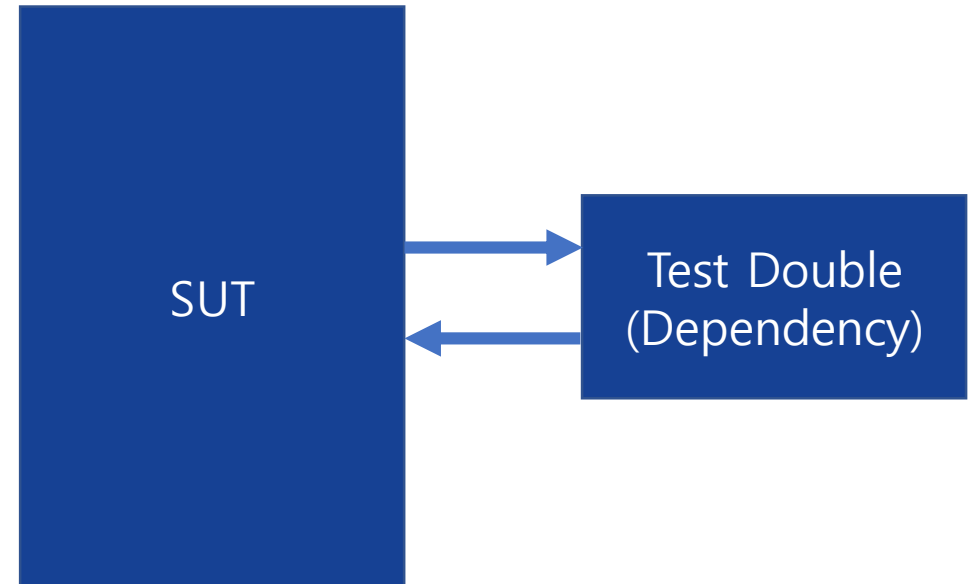
```
class Cal:  
    def get_sum(self, a, b):  
        return a + b  
  
    def get_gop_three(self, a, b):  
        t1 = self.get_sum(a, b)  
        t2 = self.get_sum(a, b)  
        t3 = self.get_sum(a, b)  
        return t1 + t2 + t3  
  
    def say_hi(self):  
        return 'HI'  
  
    def say_hello(self):  
        return 'HELLO'
```

# [정리] Mock object 로 할 수 있는 것

## Mock object 를 이용해서

1. stub(stubbing) : mock 객체가 SUT 코드 쪽에 특정값을 제공하도록 한다.
2. mock(상호작용 검증) : SUT 코드에서 mock 객체를 어떻게 호출했는지 검증한다.  
(횟수,인자 등 검증)

```
def test_calc(mock: MockerFixture):  
    mk = mocker.Mock()  
    Stubbing  
    sut = Calc(mk)  
    sut.run() # 내부에서 mk와 상호작용  
    Behavior Verification
```





# Mock Injection

목 주입하기

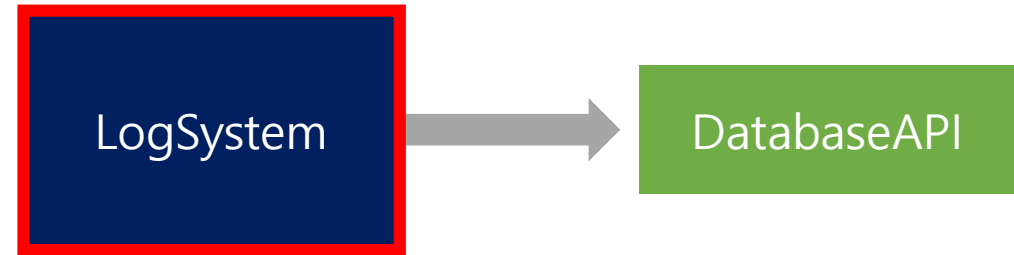
## 다음과 같이 구현하자.

- 테스트 하고자 하는 모듈

LogSystem

- LogSystem의 의존성 모듈

DatabaseAPI



```
class LogSystem:
    def __init__(self):
        self._db = DatabaseAPI()

    def log_message(self, content):
        return f'[{self._db.name}] {content}'
```

```
class DatabaseAPI:
    def __init__(self, name='MySon_DB'):
        self._name = name

    @property
    def name(self):
        return self._name
```



# 테스트 할 때, 문제점

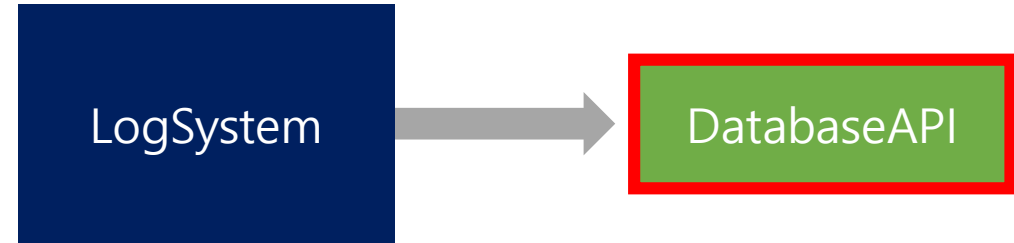
- 테스트 하고자 하는 모듈

LogSystem

- **DatabaseAPI 동작**

Unit Test를 할 때 마다,  
Database의 **getDBName API**를 수행하고,  
서버 트래픽을 발생시킨다고 가정한다.

이로 인해, 테스트 할 때마다 서버 성능이 낮아진다.



```
class DatabaseAPI:
    def __init__(self, name='MySon_DB'):
        self._name = name

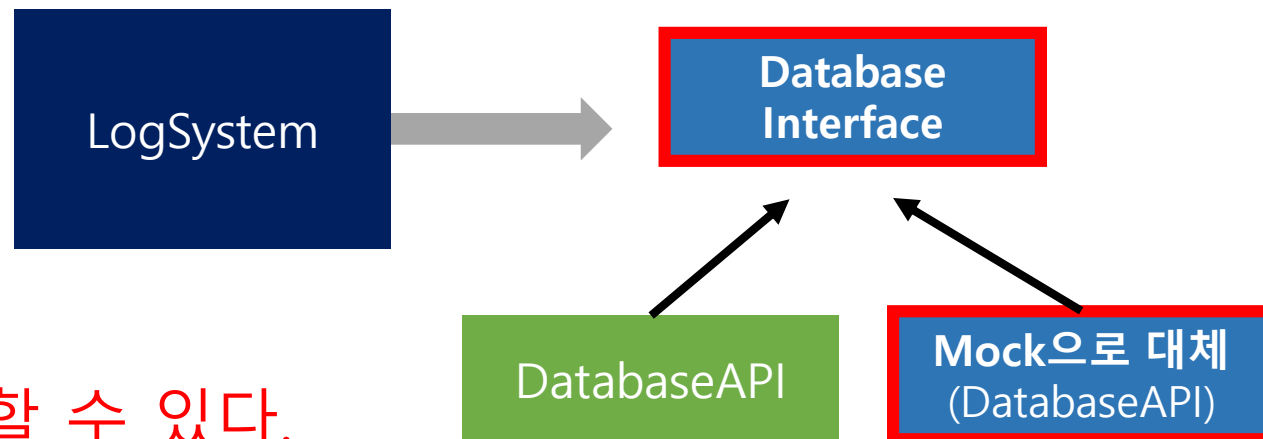
    @property
    def name(self):
        return self._name
```

# Mock으로 대체

- 테스트 하고자 하는 모듈

LogSystem

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



```
class LogSystem:
    def __init__(self):
        self._db = DatabaseAPI()

    def log_message(self, content):
        return f'[{self._db.name}] {content}'
```

이 DB 객체를 Mock객체로 변경필요

# Mock으로 대체를 위한 준비작업

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

```
from abc import ABC, abstractmethod
```

```
class DBAPI(ABC):  
    @abstractmethod  
    def name(self):  
        pass
```

```
class DatabaseAPI(DBAPI):  
    def __init__(self, name='MySon_DB'):  
        self._name = name  
  
    @property  
    def name(self):  
        return self._name
```

```
class LogSystem:  
    def __init__(self, db):  
        self._db = db  
  
    def log_message(self, content):  
        return f'[{self._db.name}] {content}'
```

유닛테스트가 가능하도록 리팩토링 하는 것이다.

# Mock 주입하기

- Mock 객체를 준비하여, DI 해준다.

```
from unittest.mock import PropertyMock
from calc import Cal
```

```
def test_real():
    c = Cal.DatabaseAPI()
    app = Cal.LogSystem(c)
    print(app.log_message('hi'))
```

```
def test_mock(mock):
    # PropertyMock 을 이용해 Cal.DatabaseAPI.name 을 patch (new_callable 기본값 Mock)
    mock.patch.object(Cal.DatabaseAPI, 'name', new_callable=PropertyMock, return_value='DOUBLE')

    c = Cal.DatabaseAPI()
    app = Cal.LogSystem(c)
    print(app.log_message('hi'))
```

```
[DOUBLE] hi
[MySon_DB] hi
```

# Mock 주입하여 Unit Test 하기

- Mocking을 사용한 UnitTest는 LogSystem 의 **DB 트래픽이 없다.**

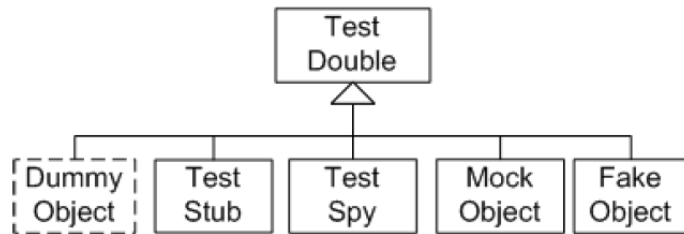
- 지금까지 수행한 일

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

# [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

