PYCON KOREA 2018

우아하게 준비하는 테스트와 리팩토링

PYCON KR 2018

한성민

INDEX

Today we will discuss about

- 우리가 개발을 올바르게 하고 있을까?
- 현업에서도 말하지 않는 올바른 개발
- 코드를 더 능숙하게, 일을 더 재미있게
- 가끔은 재치를

Clean Code
클린코드

깨진 유리창 이론

"만약 한 건물의 유리창이 깨진채로 방치되어 있다면
머지않아 그 건물의 다른 유리창도 깨질 것이다."

만약 문제가 그대로 방치되어 있다면
머지 않아 돌이킬 수 없는 문제를 야기할 수 있다.

일상 생활의 깨진 유리창

User A

```
def buzz(num):
  message = 'buzz!' if num % 3 == 0\
    else num

  print(message)


def main():
  for num in range(1, 10):
    buzz(num)


if __name__ == '__main__':
  main()
```

User B

```
def buzz(num):
  message = 'buzz!' if num % 3 == 0\
    else num

  print(message)


def main():
  for num in range(1, 10):
    buzz(num)


if __name__ == '__main__':
  main()
```

두명의 프로그래머가 369 게임 (buzz)를 만든다고 가정해봅시다.

User A

```
def buzz(num):
    message = ('clap' if num > 5\
        else 'buzz!') if num % 3 == 0\
            else num

    print(message)

def main():
    for num in range(1, 10):
        buzz(num)


if __name__ == '__main__':
    main()
```

깨진 유리창 발생!

User B

```
def buzz(num):
    message = 'buzz!' if num % 3 == 0\
        else num

    print(message)


def main():
    for num in range(1, 10):
        buzz(num)


if __name__ == '__main__':
    main()
```

User A

```
def buzz(num):
    message = ('clap' if num > 5\
        else 'buzz!') if num % 3 == 0\
            else num

    print(message)
```

```
def main():
    for num in range(1, 10):
        buzz(num)


if __name__ == '__main__':
    main()
```

User B

```
def buzz(num, rule):
    message = ('clap' if num > 5\
        else 'buzz!') if num % rule == 0\
            else num

    print(message)
```

```
def main():
    for num in range(1, 10):
        buzz(num)


if __name__ == '__main__':
    main()
```

깨진 유리창 발생!

User A

```
def buzz(num, rule, type):
    message = ('clap' if num > 5\
        else 'buzz!') if num % rule == 0\
            else num

    if type == 'MESSAGE':
        print(message)
    elif type == 'PIPE':
        import piper
        pipe = None
        try:
            pipe = piper.getNewPiper()
        except:
            pipe = piper.getGlobal()
        pipe.send()
    elif type == 'LOG':
        import logger
        print(message)
        logger.debug('TYPE: {}'.format(('CLAP' if num > 5\
        else 'BUZZ') if num % rule == 0 else 'NORMAL'))
    else: print(message)


  print(message)
...
```

User B

```
def buzz(num, rule):
    message = ('clap' if num > 5\
        else 'buzz!') if num % rule == 0\
            else num

    print(message)

def main():
    for num in range(1, 10):
        buzz(num)


if __name__ == '__main__':
    main()
```

...

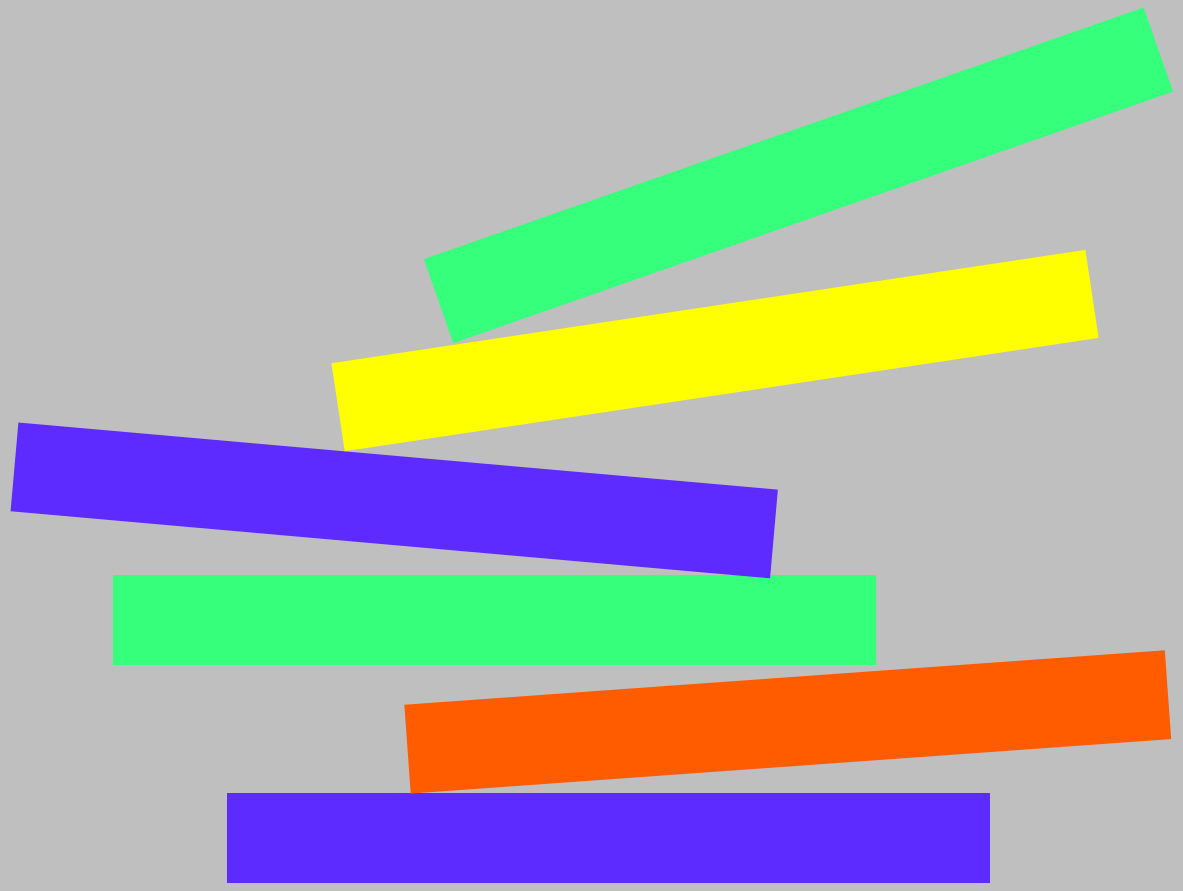
우리는 이런 코드의 잠재적 문제들을 코드 악취(Code Smell)라고 부릅니다.

보이스카우트 규칙

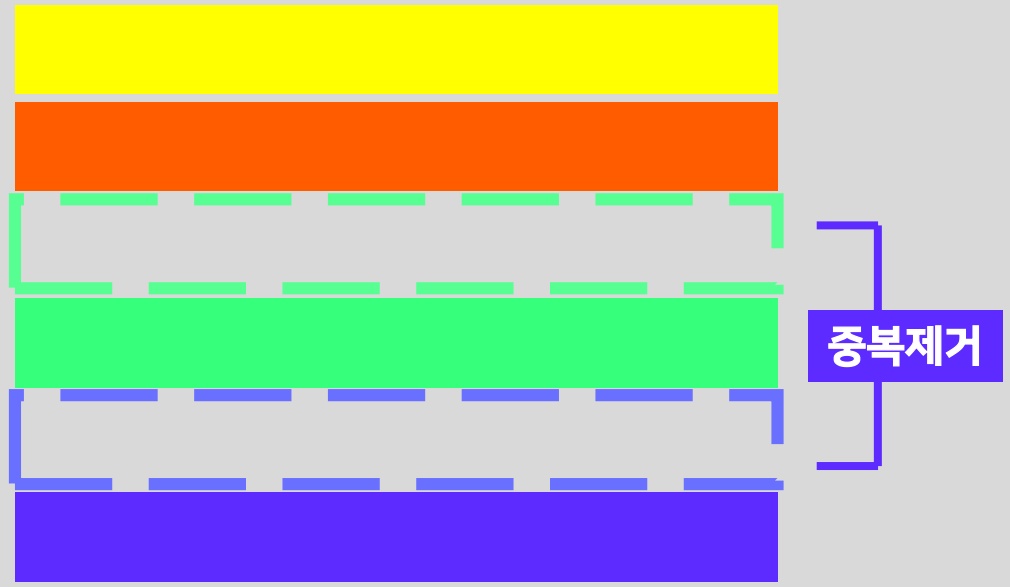
Boy Scout Rule

언제나 처음 왔을 때보다 깨끗하게 해놓고 캠프장을 떠날 것

클린 코드는 코드의 잠재적 문제를 해결하여 생산성을 높입니다.



레거시(LEGACY)



클린 코드(CLEAN CODE)

클린 코드 핵심 가이드

가드클러즈(GuardClause)는 여러분의 인덴트(Indent)를 줄여줍니다

clean_code_1.py

```python
def example():
    user = get_user()
    if user != None and user.type == 'anonymous':
        post = get_post(user)
        if post != None and post.title != None:
            comment = get_comment(post)
            status = do_with_comment(comment)
            if status == False:
                return Status.FAIL
            else:
                return Status.SUCCESS
        return Status.FAIL
    return Status.FAIL
```

```python
def example():
    user = get_user()
    if user == None or user.type == 'anonymous':
        return Status.FAIL

    post = get_post(user)
    if post == None or post.title == None:
        return Status.FAIL

    comment = get_comment(post)
    status = do_with_comment(comment)

    if status == False:
        return Status.FAIL

    return Status.SUCCESS
```

클린 코드 핵심 가이드

NoneObject는 코드의 복잡한 조건을 간단하게 만들어줍니다

clean_code_2.py

```python
def example():
    user = get_user()
    if user == None or user.type == 'anonymous':
        return Status.FAIL

    post = get_post(user)
    if post == None or post.title == None:
        return Status.FAIL

    comment = get_comment(post)
    status = do_with_comment(comment)

    if status == False:
        return Status.FAIL

    return Status.SUCCESS
```

```python
def example():
    user = get_user()
    if user.get('type') == None:
        return Status.FAIL

    post = get_post(user)
    if post.get('title') == None:
        return Status.FAIL

    comment = get_comment(post)
    status = do_with_comment(comment)

    if status == False:
        return Status.FAIL

    return Status.SUCCESS
```

클린 코드 핵심 가이드

Bool, Object에서 None, False을 체크할 경우 `not` Syntax Sugar를 사용하세요

clean_code_3.py

```python
def example():
    user = get_user()
    if user.get('type') == None:
        return Status.FAIL

    post = get_post(user)
    if post.get('title') == None:
        return Status.FAIL

    comment = get_comment(post)
    status = do_with_comment(comment)

    if status == False:
        return Status.FAIL

    return Status.SUCCESS
```

```python
def example():
    user = get_user()
    if not user.get('type'):
        return Status.FAIL

    post = get_post(user)
    if not post.get('title'):
        return Status.FAIL

    comment = get_comment(post)
    status = do_with_comment(comment)

    if not status:
        return Status.FAIL

    return Status.SUCCESS
```

클린 코드 핵심 가이드

각 함수에서 예외처리를 진행하거나 유효성 오류의 경우 NoneObject를 반환하세요

clean_code_4.py

```python
def example():
    user = get_user()
    if not user.get('type'):
        return Status.FAIL

    post = get_post(user)
    if not post.get('title'):
        return Status.FAIL

    comment = get_comment(post)
    status = do_with_comment(comment)

    if not status:
        return Status.FAIL

    return Status.SUCCESS
```

```python
def example():
    user = get_user()
    post = get_post(user)
    comment = get_comment(post)
    status = do_with_comment(comment)

    if not status:
        return Status.FAIL

    return Status.SUCCESS
```

클린 코드 핵심 가이드

짧은 조건은 삼항 연산자를 사용하세요

clean_code_5.py

```python
def example():
    user = get_user()
    post = get_post(user)
    comment = get_comment(post)
    status = do_with_comment(comment)

    if not status:
        return Status.FAIL

    return Status.SUCCESS
```

```python
def example():
    user = get_user()
    post = get_post(user)
    comment = get_comment(post)
    status = do_with_comment(comment)

    return Status.FAIL if not status\
        else Status.SUCCESS
```

클린 코드 핵심 가이드

단순한 형태의 IF, SWITCH는 Dict Acessing 형식으로 변경해주세요

clean_code_6.py

```python
def get_user_permit(code):
    if code == 'A':
        return Permission.ADMIN
    elif code == 'U':
        return Permission.USER
    else:
        return Permission.ANONYMOUSE
```

```python
def get_user_permit(code):
    permissions = {
        'A': Permission.ADMIN,
        'U': Permission.USER,
        '_': Permission.ANONYMOUSE
    }

    return permissions.get(code, permissions['_'])
```

클린 코드 핵심 가이드

함수 이름은 snake_case로 지정하고 행동(Action)을 이름의 가장 앞에 명명하세요

clean_code_7.py

```python
def user_name_find(name):
    # do something
    pass


def userPermissionChecking(permit):
    # do something
    pass


def CommentMessage(id, message):
    # do something
    pass
```

```python
def find_user_name(name):
    # do something
    pass


def check_user_permit(permit):
    # do something
    pass


def do_comment(id, message):
    # do something
    pass
```

클린 코드 핵심 가이드

주석이 필요한 복잡한 로직은 함수로 분리하고, 함수 명을 주석 대신 사용하세요

clean_code_8.py

```python
def example():
    user = getUser()
    post = getPost(user)

    if not post:
        return Status.FAIL

    # remove post with admin
    if is_user_admin(user):
        remove_post(post.id)
        send_log(user.id)
        send_log(post.owner)
    else:
        send_error(user.id)

    return Status.SUCCESS
```

```python
def example():
    user = getUser()
    post = getPost(user)

    if not post: return Status.FAIL

    remove_post_with_admin(post, user)
    return Status.SUCCESS


def remove_post_with_admin(post, user):
    if is_user_admin(user):
        remove_post(post.id)
        send_log(user.id)
        send_log(post.owner)
    else: send_error(user.id)
```

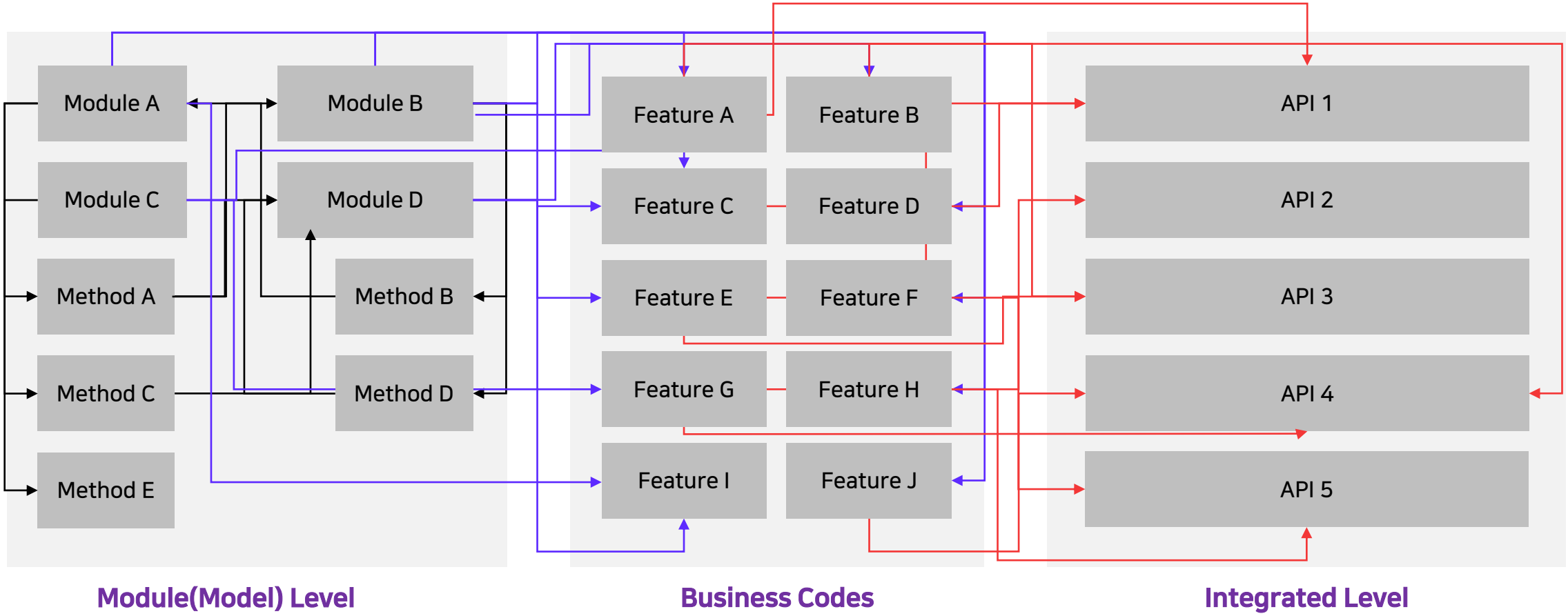
클린 코드 핵심 가이드

그 밖에도 많은 클린 코드 방법들이 존재합니다.

- 3줄 이상의 라인이 중복된 코드는 함수로 분리하세요 (그 이하는 인라인 함수 혹은 인라인 유지)
- 주석이 없는 코드가 제일 좋은 코드임을 명심하세요
- 클래스와 함수에 너무 많은 기능을 주지 마세요, 많은 기능이 묶인 함수면 코드 기능 별로 함수를 분리하세요
- 함수에 부여되는 인수는 4개를 넘지 않도록 하세요
- 복잡한 조건은 캡슐화를 하여 직관적인 이름을 명시하여 쉽게 만들어주세요
- .py 파일에 두개 이상의 클래스를 정의하지 마세요 (클래스 별로 파일을 분리하세요)

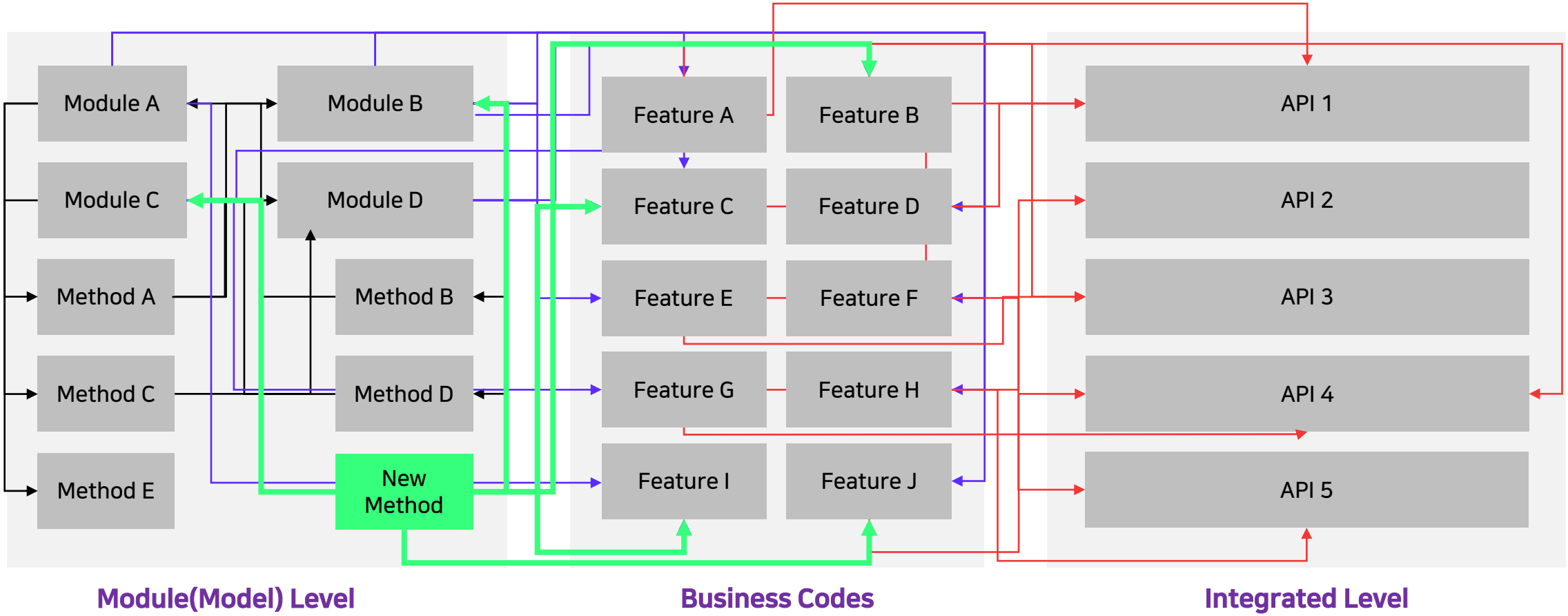
Test and Refactoring
테스트 그리고 리팩토링

잠깐, 테스트 코드 작성이 정말 필요한가요?



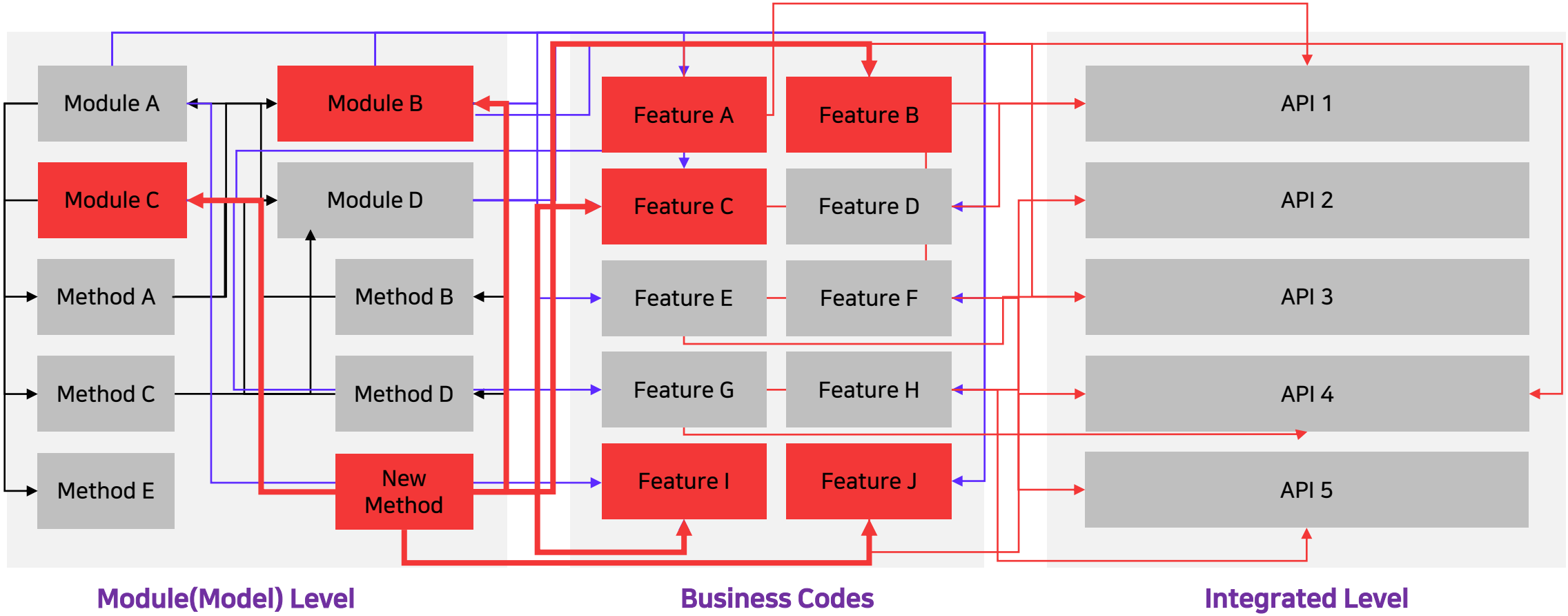
생각보다 실제 환경에서 동작하는 코드는 복잡합니다.

잠깐, 테스트 코드 작성이 정말 필요한가요?



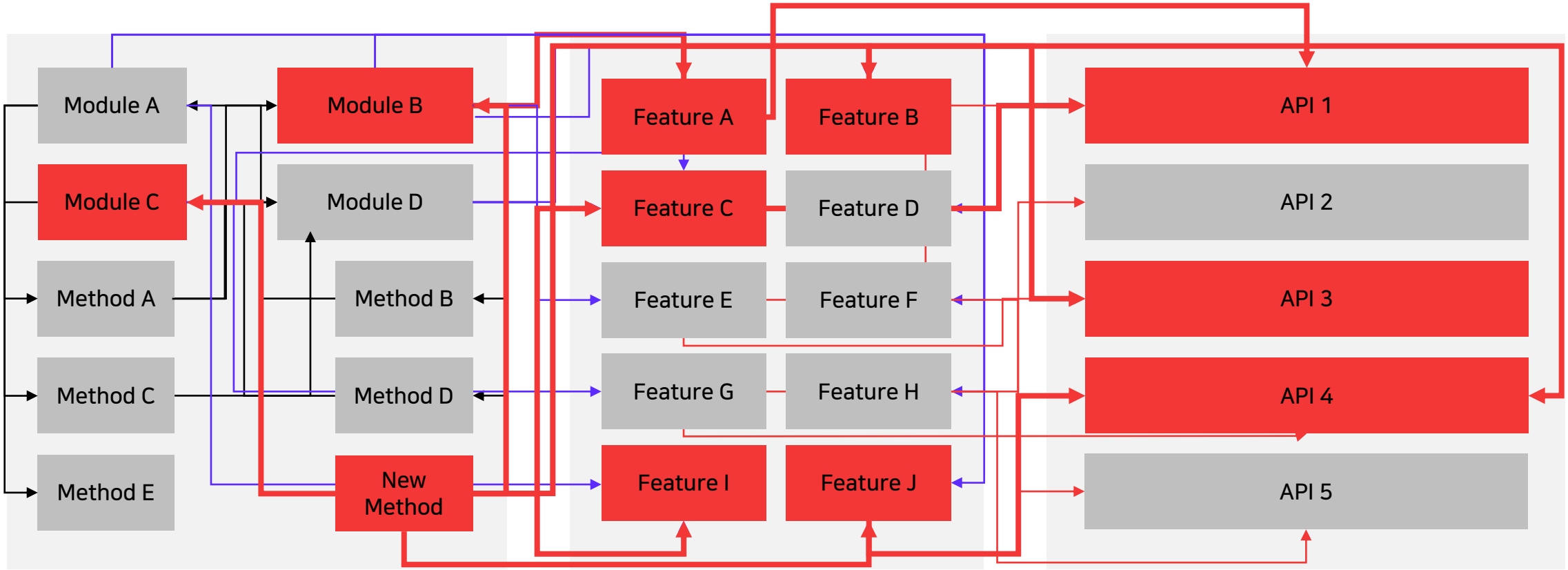
여러분이 하나의 가장 적은 단위의 코드를 추가하거나 수정했다고 가정합시다.

잠깐, 테스트 코드 작성이 정말 필요한가요?



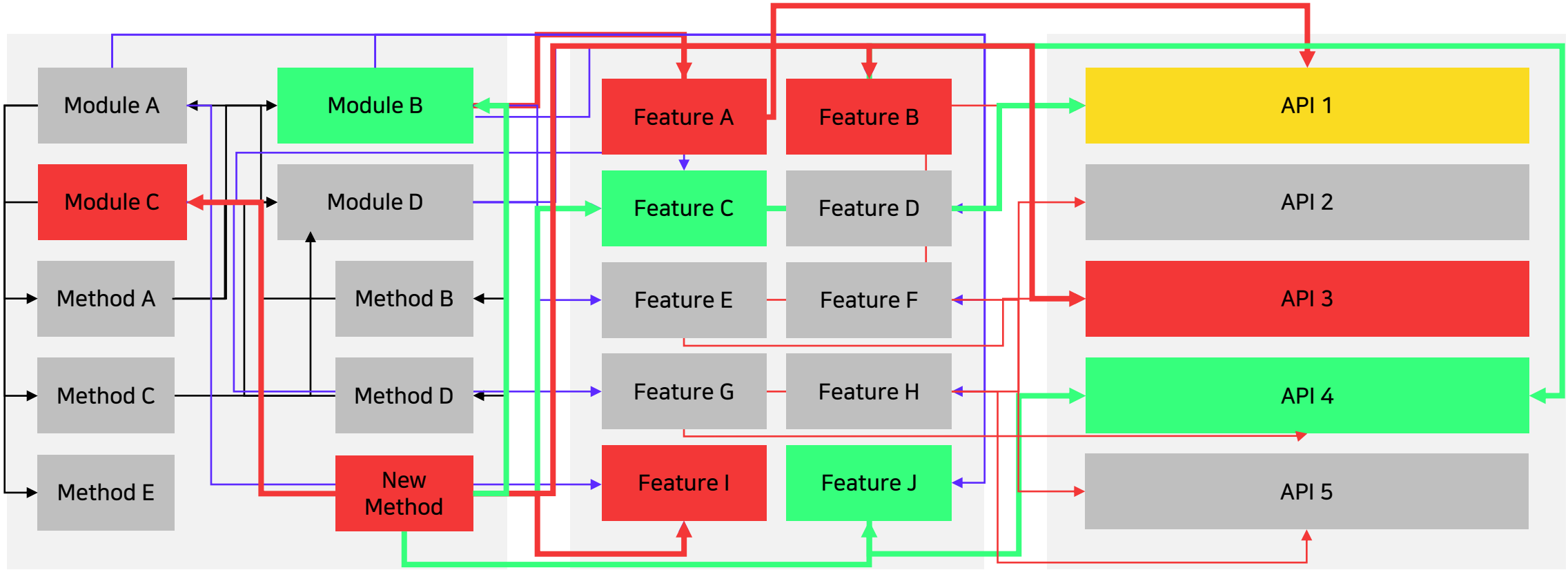
만약 그 코드가 문제가 있으면 그건 정말로 큰 부작용(Side-Effect)를 발생시킵니다.

잠깐, 테스트 코드 작성이 정말 필요한가요?



만약 그 코드가 문제가 있으면 그건 정말로 큰 부작용(Side-Effect)를 발생시킵니다.

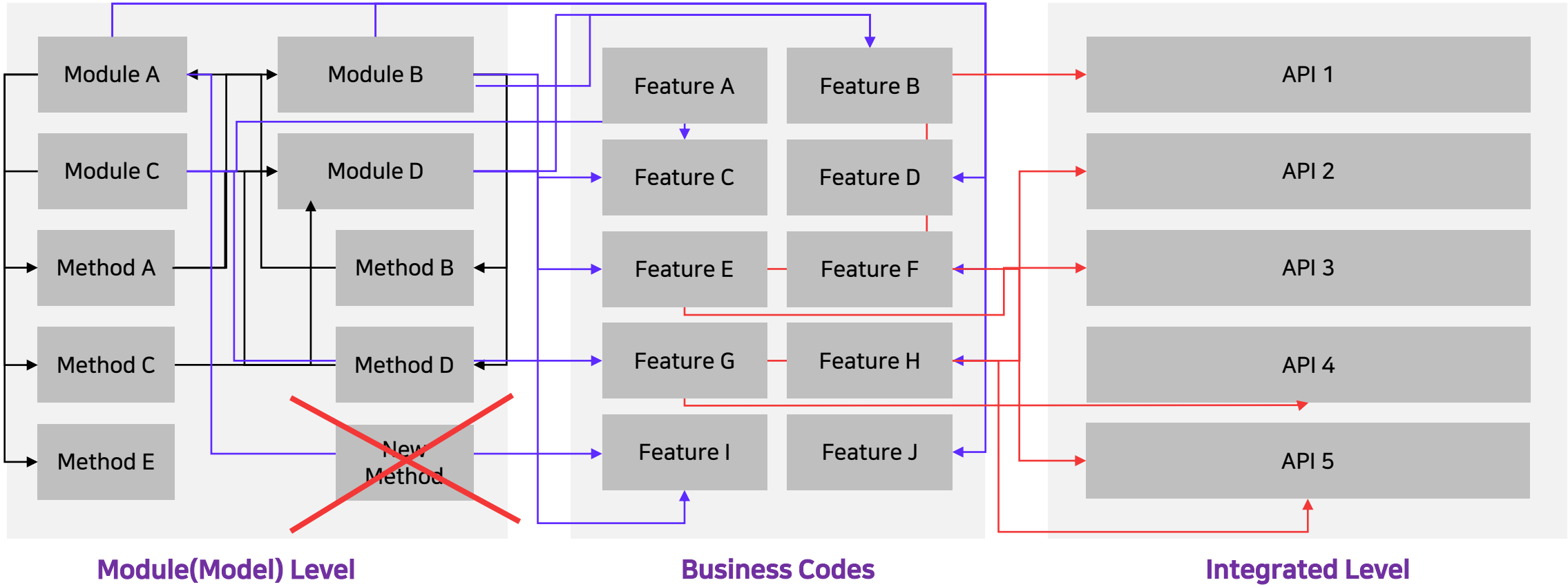
잠깐, 테스트 코드 작성이 정말 필요한가요?



특정 조건에서만 발생하는 버그로 인해, 일부는 동작하고 일부는 동작하지 않는다면

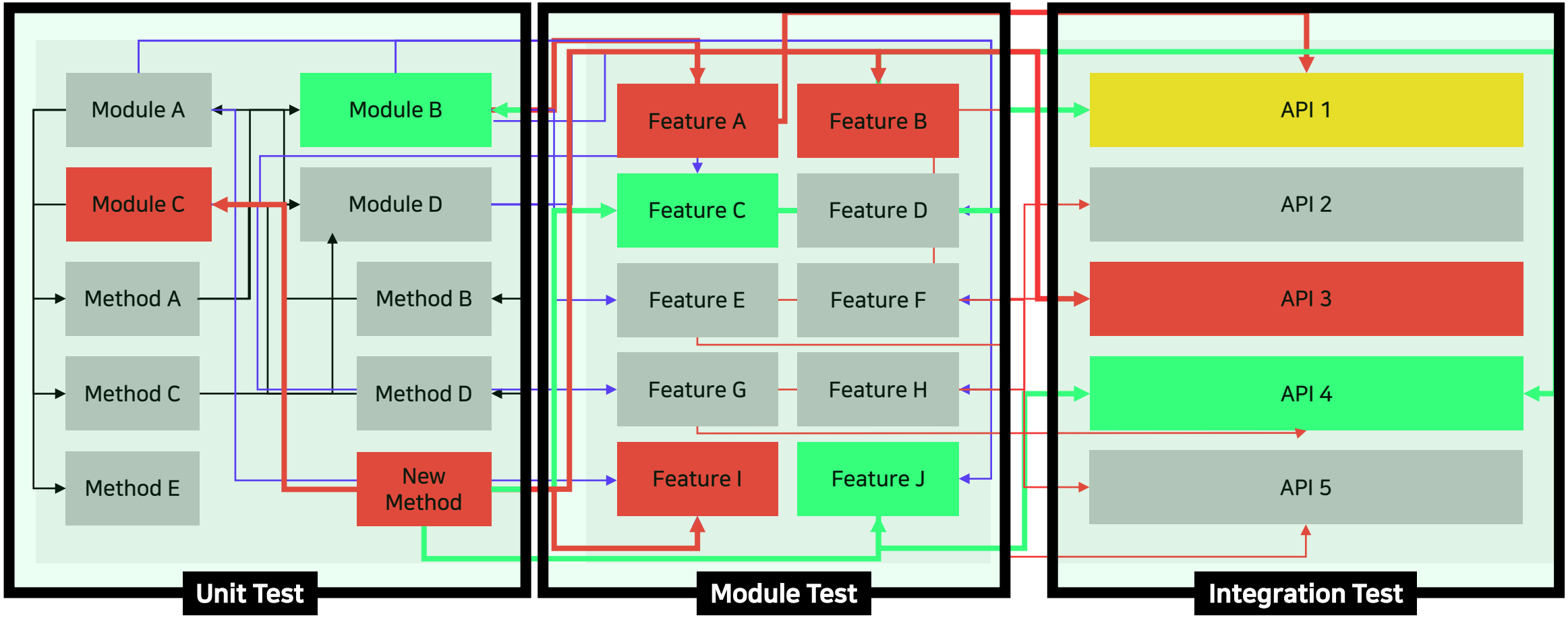
사람이 검수로 버그를 발견하기도 힘들어집니다.

잠깐, 테스트 코드 작성이 정말 필요한가요?



이렇게 되면 코드 하나를 추가하는 것이 두려워지고 신뢰할 수 없게 됩니다.

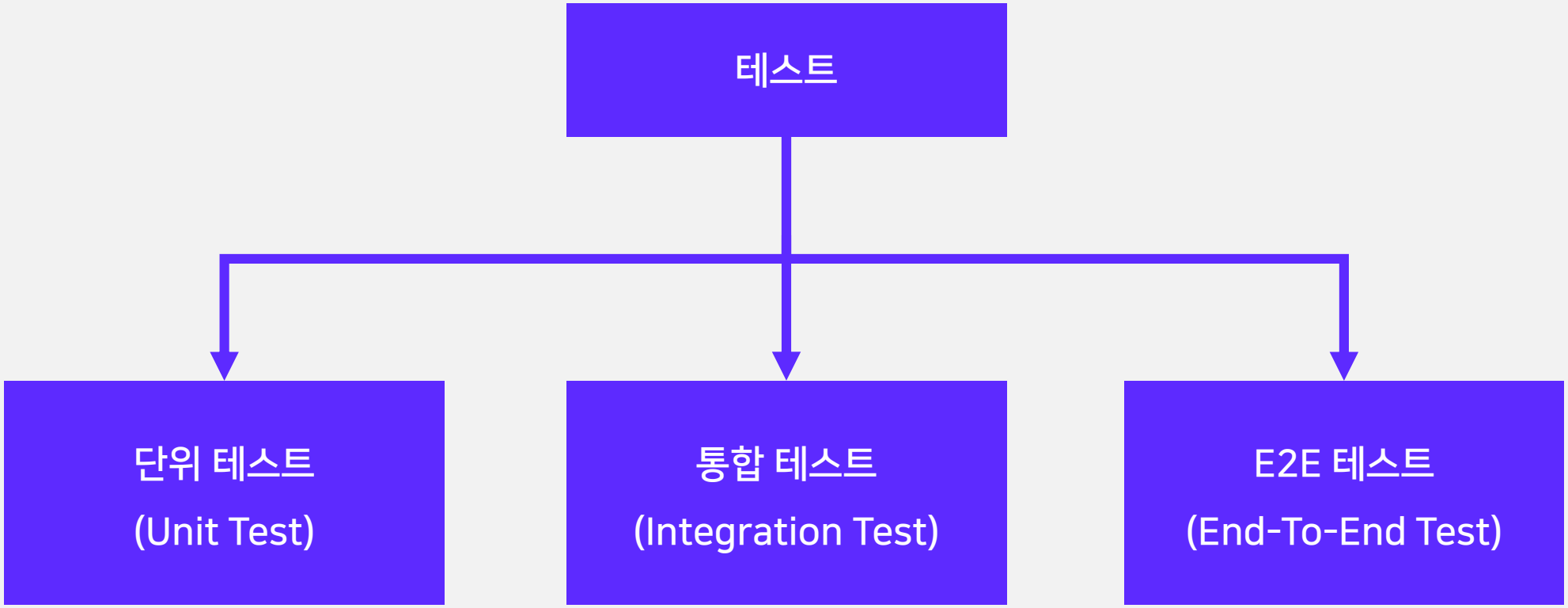
잠깐, 테스트 코드 작성이 정말 필요한가요?



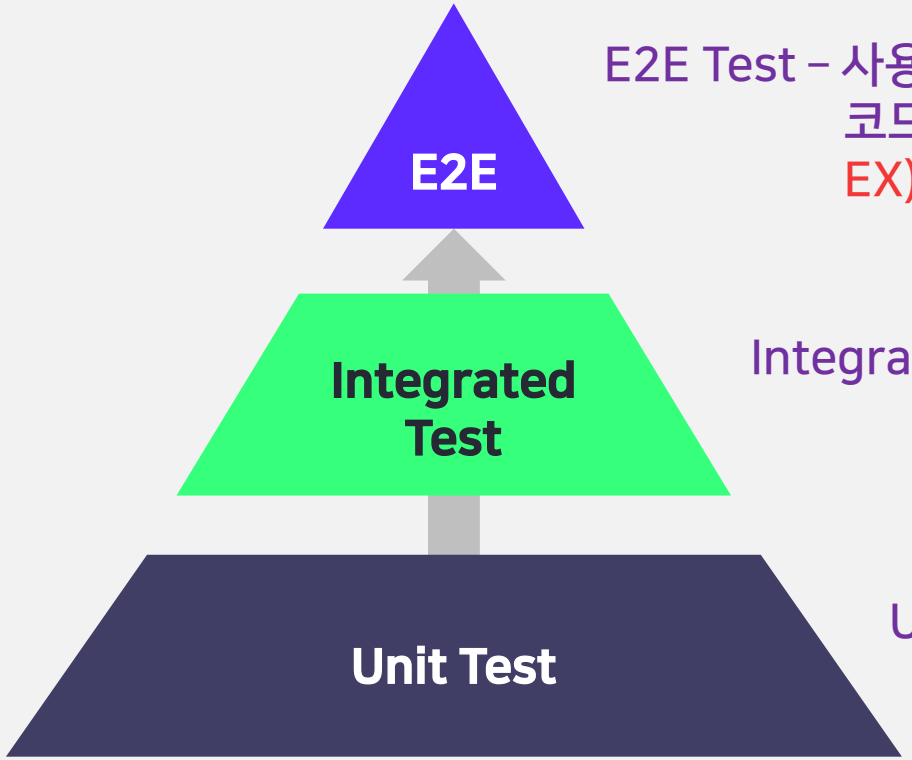
테스트 코드는 이런 기능 추가와 수정이 발생할 때

발생할 수 있는 사이드 이펙트 탐지를 돕고 코드를 신뢰할 수 있게 만들어줍니다.

일반적으로 고려하는 테스트



테스트 코드 별 특징

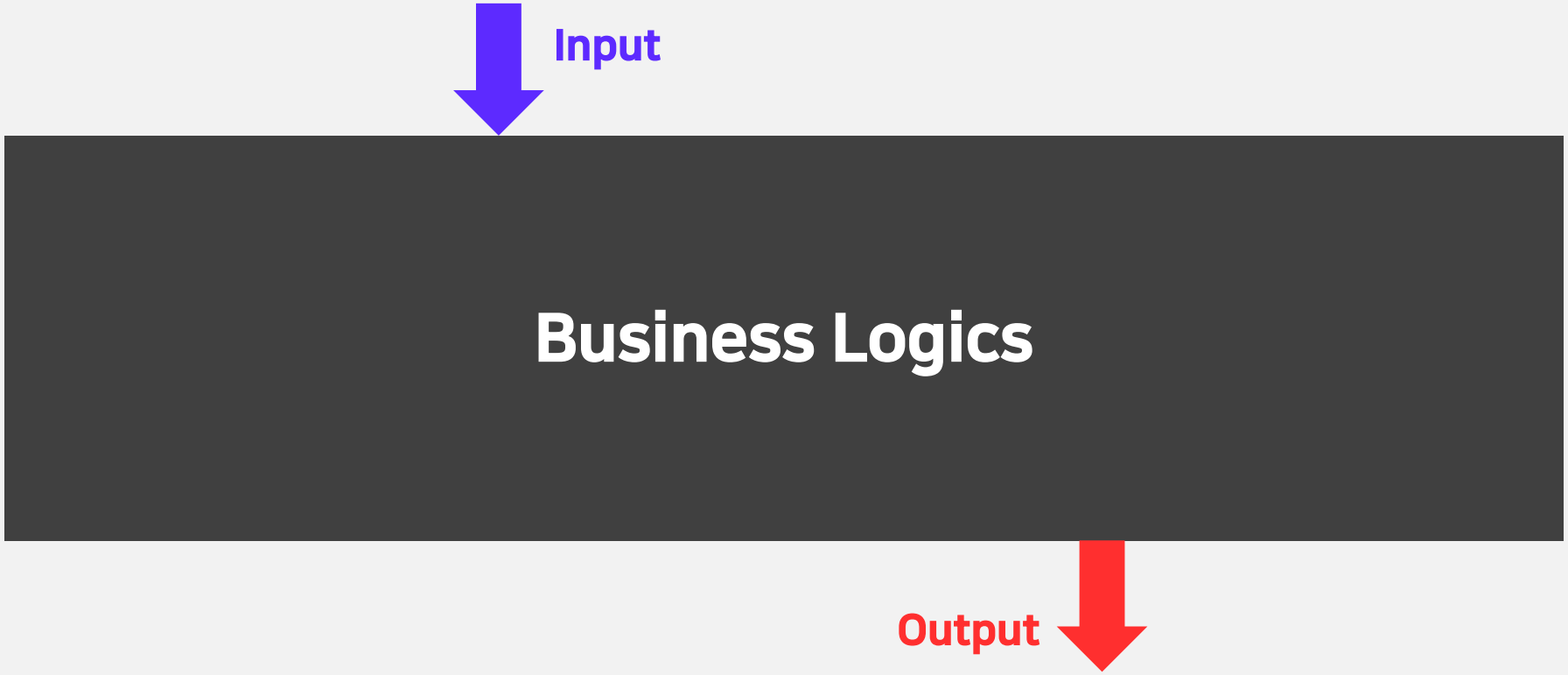


E2E Test – 사용자가 실제로 사용하는 UI와 같은 복잡한 환경에서 테스트
코드와 관계없이 사용자의 OS와 레지스트리, 메모리 등에 의한 모든 결함 탐지
EX) Web UI Test, GUI Program Test, Android App E2E Test

Integrated Test – 각종 함수와 메서드가 모여진 코드의 집합을 전체적으로 테스트
실제 사용하는 사용자 기능 위주의 테스트 가능

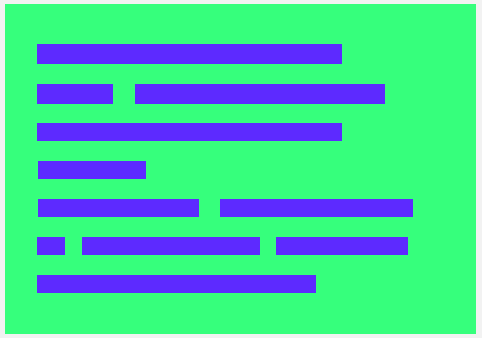
Unit Test – 함수와 메서드의 최소 단위와 각 비즈니스 코드의 테스트 등
일반적인 테스트 코드들에 해당

테스트 기본 원리



테스트 대상은 어떤 입력 값(Input)에 따라 출력 값(Output)이 결정됩니다.

테스트 코드 별 특징



White Box Test

코드의 내부를 들여다보고
코드를 테스트하는 방식

Black Box Test

코드의 내부를 모르는 상태에서
입력 값과 출력 값으로 코드를 테스트하는 방식

테스트 대상의 환경에 따라 크게 두가지 방식의 테스트 방법이 있습니다.

테스트 코드 별 특징

White Box Test

- 구문 커버리지
- 조건/분기 커버리지
- MC/DC 커버리지

커버리지(Coverage)

- 제어 흐름 그래프

Black Box Test

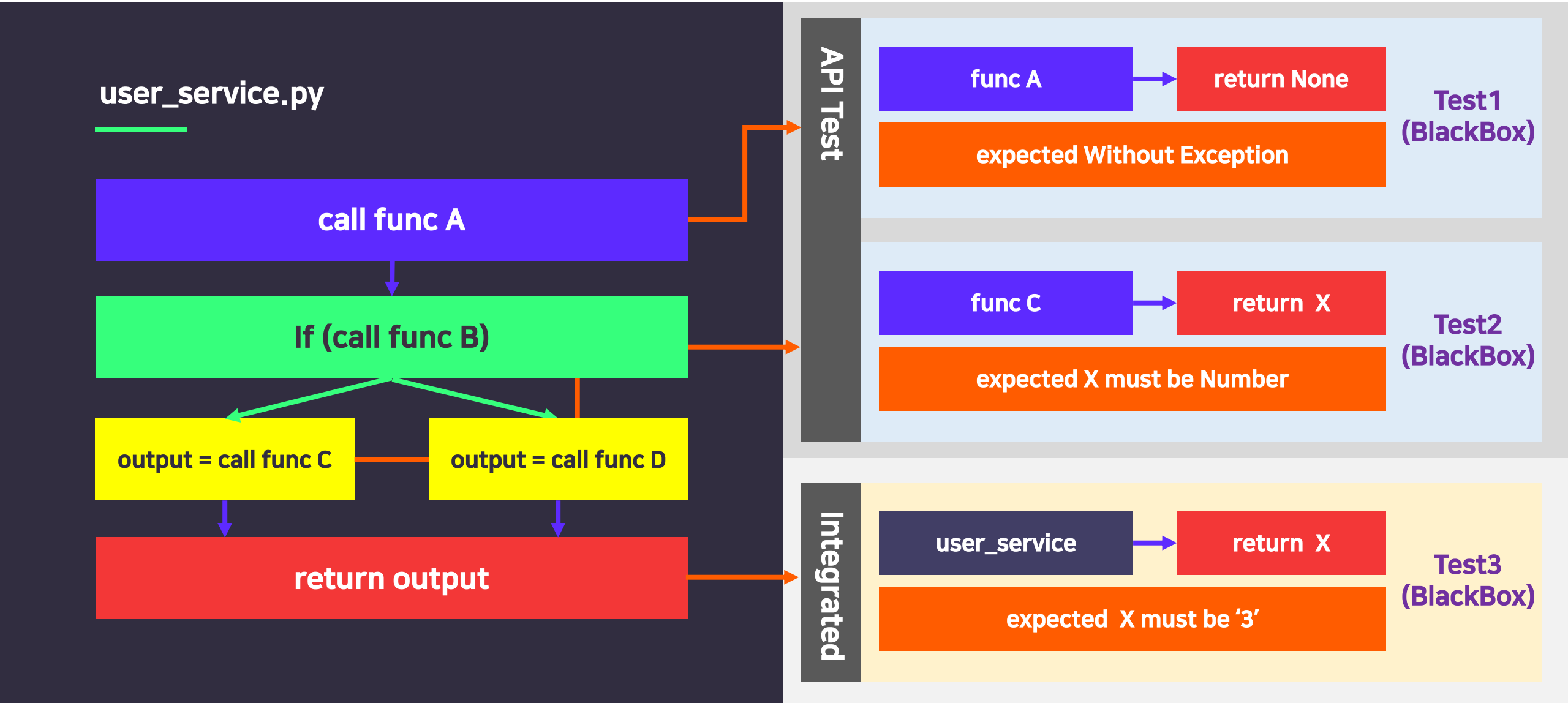
- 동등 클래스 분할
- 경계 값 분석

테스트 케이스(TC)

- 페어와이즈 분석

테스트 코드의 큰 두가지 분류에 따라 접근하는 방법도 서로 달라집니다.

올바른 테스트 디자인


user_service.py
call func A
If (call func B)
output = call func C
output = call func D
return output
API Test
func A
return None
expected Without Exception
Test1
(BlackBox)
func C
return X
expected X must be Number
Test2
(BlackBox)
Integrated
user_service
return X
expected X must be '3'
Test3
(BlackBox)

올바른 테스트 디자인

Test Code

Refactoring

테스트 코드는 리팩토링과는 다르게 최소 단위부터 상위 단위로 순서대로 작업해야 합니다.

test/basic/main.py

```python
__VERSION__ = '0.0.1'


def sum(a, b):
    return a + b


def get_version():
    return __VERSION__


if __name__ == '__main__':
    sum_num = sum(1, 2) # 1 + 2 = 3
    version = get_version() # 0.0.1

    print('sum', sum_num)
    print('version', version)
```

테스트를 위한 간단한 함수를 작성해봤습니다.

test/basic/main.py

```python
__VERSION__ = '0.0.1'

def sum(a, b):
    return a + b

def get_version():
    return __VERSION__

if __name__ == '__main__':
    sum_num = sum(1, 2) # 1 + 2 = 3
    version = get_version() # 0.0.1

    print('sum', sum_num)
    print('version', version)
```

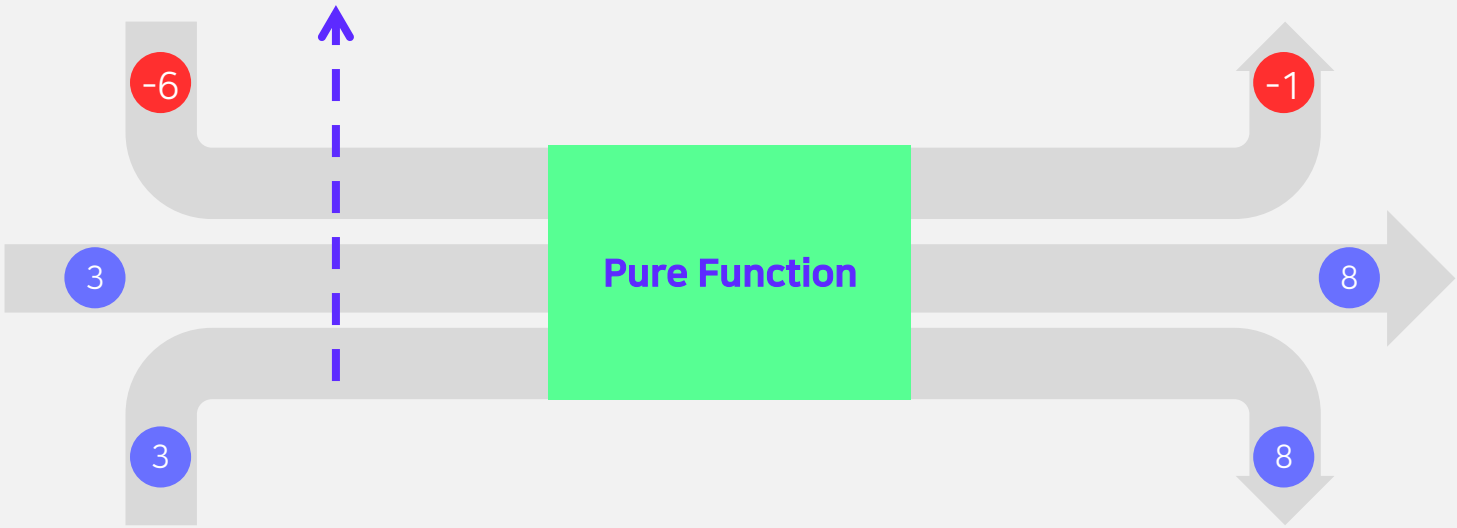
출력은 입력에 따라 결정됩니다.
만약 같은 입력이 들어온다면 출력도 같습니다.

입력이 존재하지 않습니다.
따라서 출력은 언제나 같습니다.

위 두 함수를 우리는 순수 함수라 부릅니다.

순수 함수 (Pure Function)

순수 함수의 이런 성질을 참조 투명성 (Referential Transparency, RT)라고 합니다.



함수에 제공된 입력 값에 의해 출력 값이 결정되는 함수

입력 값 외에 어떤 상태나, 환경에 의해서 출력 값이 결정되지 않는 함수이기 때문에

테스트를 진행하기에 적절합니다.

test/basic/main.py

```python
__VERSION__ = '0.0.1'

def sum(a, b):
    return a + b


def get_version():
    return __VERSION__


if __name__ == '__main__':
    sum_num = sum(1, 2) # 1 + 2 = 3
    version = get_version() # 0.0.1

    print('sum', sum_num)
    print('version', version)
```

간단한 테스트 케이스(TC)를 설계합니다.

Test Cases (TCs)

TC1 `func sum(a, b)`

Given (a=3, b=4)
Expected 7

TC2 `func sum(a, b)`

Given (a=-1, b=5)
Expected 4

TC3 `func sum(a, b)`

Given (a=None, b=2)
Expected NaN

TC4 `func get_version()`

Given ()
Expected 3 columns dot separated numbers

test/basic/test_main.py

```python
import unittest
import math
import numpy as np
from main import sum, get_version


def is_array_numeric(array):
    return np.asarray(array).dtype.kind.lower() in ('b', 'u', 'i', 'f', 'c')


class TestBasicFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(-1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        target = sum(None, 2)
        expected = math.nan(target)
        self.assertTrue(target, expected)

    def test_get_version(self):
        target = get_version()
        items = target.split('.')
        expected_number_item = 3

        self.assertEqual(len(items), expected_number_item)
        self.assertTrue(is_array_numeric(items))
```

Test Cases (TCs)

TC1

`func sum(a, b)`

Given (a=3, b=4)

Expected 7

TC2

`func sum(a, b)`

Given (a=-1, b=5)

Expected 4

TC3

`func sum(a, b)`

Given (a=None, b=2)

Expected NaN

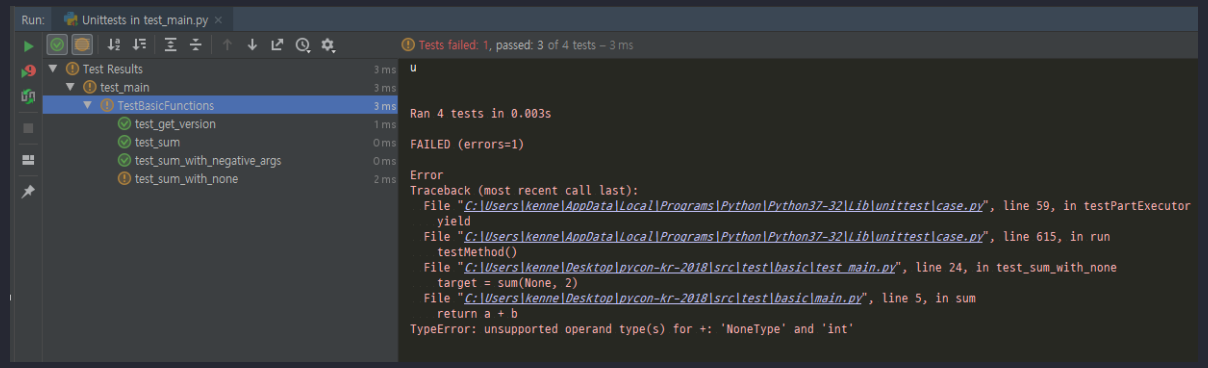
TC4

`func get_version()`

Given ()

Expected 3 columns dot separated numbers

test/basic/test_main.py




```python
class TestBasicFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(-1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        target = sum(None, 2)
        expected = math.nan(target)
        self.assertTrue(target, expected)

    def test_get_version(self):
        target = get_version()
        items = target.split('.')
        expected_number_item = 3

        self.assertEqual(len(items), expected_number_item)
        self.assertTrue(is_array_numeric(items))
```

Test Cases (TCs)

TC1

`func sum(a, b)`

Given (a=3, b=4)
Expected 7
Actual 7 (Passed)

TC2

`func sum(a, b)`

Given (a=-1, b=5)
Expected 4
Actual 4 (Passed)

TC3

`func sum(a, b)`

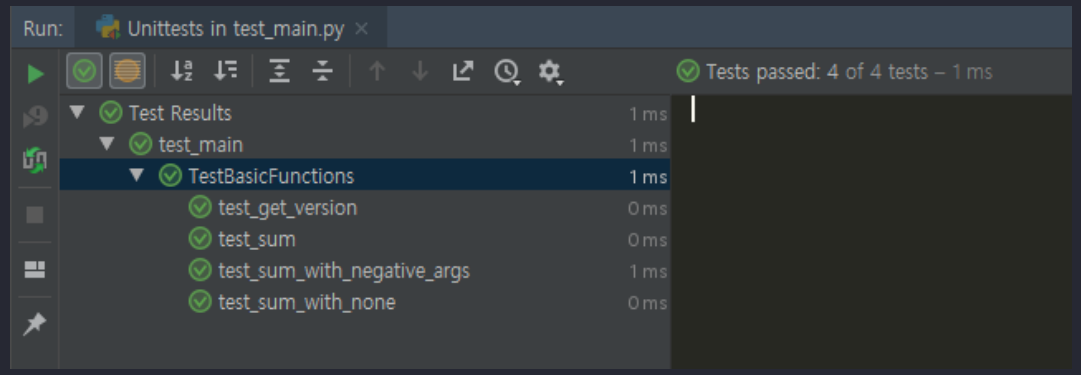
Given (a=None, b=2)
Expected NaN
Actual raise Exception (Failed)

TC4

`func get_version()`

Given ()
Expected 3 columns dot separated numbers
Actual 3 columns dot separated numbers (Passed)

test/basic/test_main.py




```python
class TestBasicFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(-1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        with self.assertRaises(TypeError):
            sum(None, 2)

    def test_get_version(self):
        target = get_version()
        items = target.split('.')
        expected_number_item = 3

        self.assertEqual(len(items), expected_number_item)
        self.assertTrue(is_array_numeric(items))
```

Test Cases (TCs)

TC1

`func sum(a, b)`

Given (a=3, b=4)

Expected 7

Actual 7 (Passed)

TC2

`func sum(a, b)`

Given (a=-1, b=5)

Expected 4

Actual 4 (Passed)

TC3

`func sum(a, b)`

Given (a=None, b=2)

Expected raise Exception

Actual raise Exception (Passed)

TC4

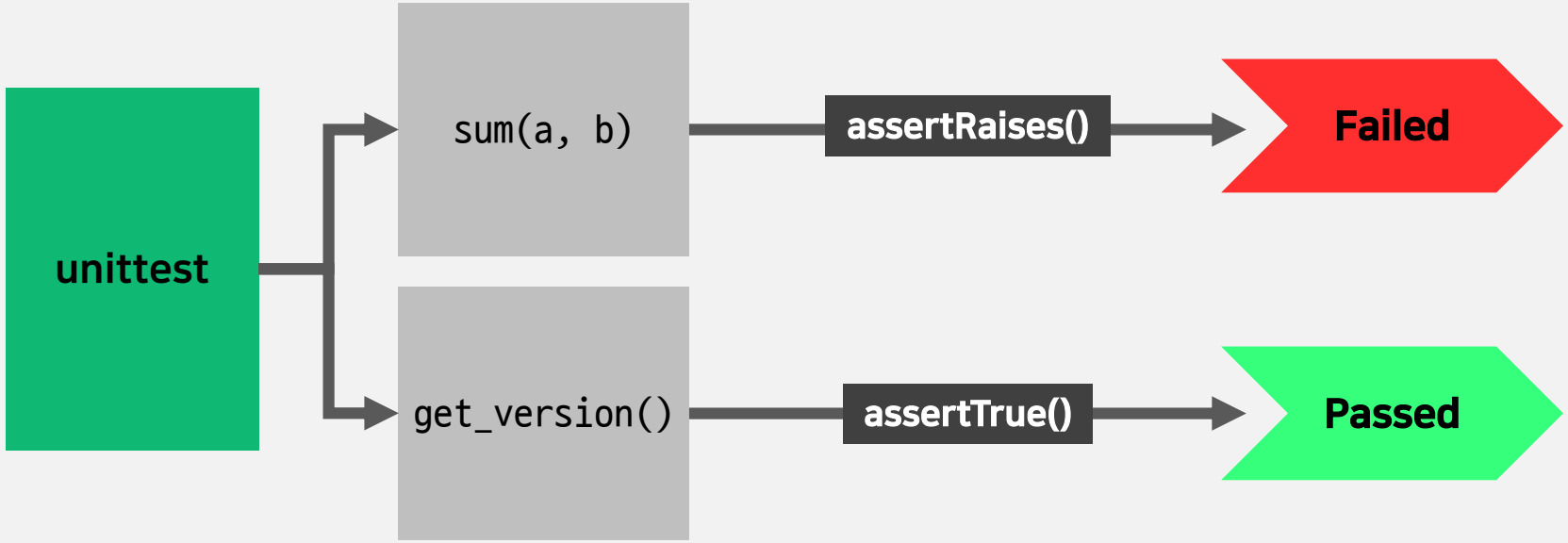
`func get_version()`

Given ()

Expected 3 columns dot separated numbers

Actual 3 columns dot separated numbers (Passed)

단위 테스트 도구 (unittest)



단위 테스트(혹은 유닛테스트)는 단일 객체, 함수, 메소드 단위의 코드에 대해서 검증을 하는 테스트 방법입니다.
가장 작은 단위의 테스트로 수행하는 것이 목적이기 때문에 테스트 대상 함수는 순수 함수로 제공되는 것이 가장 좋습니다.
파이썬에서는 unittest 모듈을 이용하여 단위 테스트 기능을 사용할 수 있습니다.

단위 테스트 assert 함수들

- assertEqual — 두 값의 일치 여부를 검증
- assertNotEqual — 두 값의 불일치 여부를 검증
- assertTrue — 값이 참임을 검증
- assertFalse — 값이 거짓임을 검증
- assertIs — 두 값의 참조 일치 여부를 검증
- assertIsNot — 두 값의 참조 불일치 여부를 검증
- assertIsNotNone — 값이 None이 아님을 검증
- assertIn — 값의 포함 관계를 검증
- assertNotIn — 값의 비포함 관계를 검증
- assertIsInstance — 값의 인스턴스 타입 일치 여부를 검증
- assertNotIsInstance — 값의 인스턴스 타입 불일치 여부를 검증

unittest 모듈에서 제공하는 주요 메소드들을 정리해봤습니다.

단위 테스트를 작성할 때 위와 같은 검증 메소드를 사용할 수 있습니다.

test/complex/main.py

```python
import requests
import re

_URL_ = 'https://pycon.kr'


def sum(a, b):
    res = requests.get(_URL_)
    year = int(re.findall(r'\d+', res.url)[0])
    if year == 2018:
        return a + b
    return -1


if __name__ == '__main__':
    print(sum(1, 2))
```

이번에는 조금 더 복잡한 분기를 가진 함수를 작성합니다.

test/complex/main.py

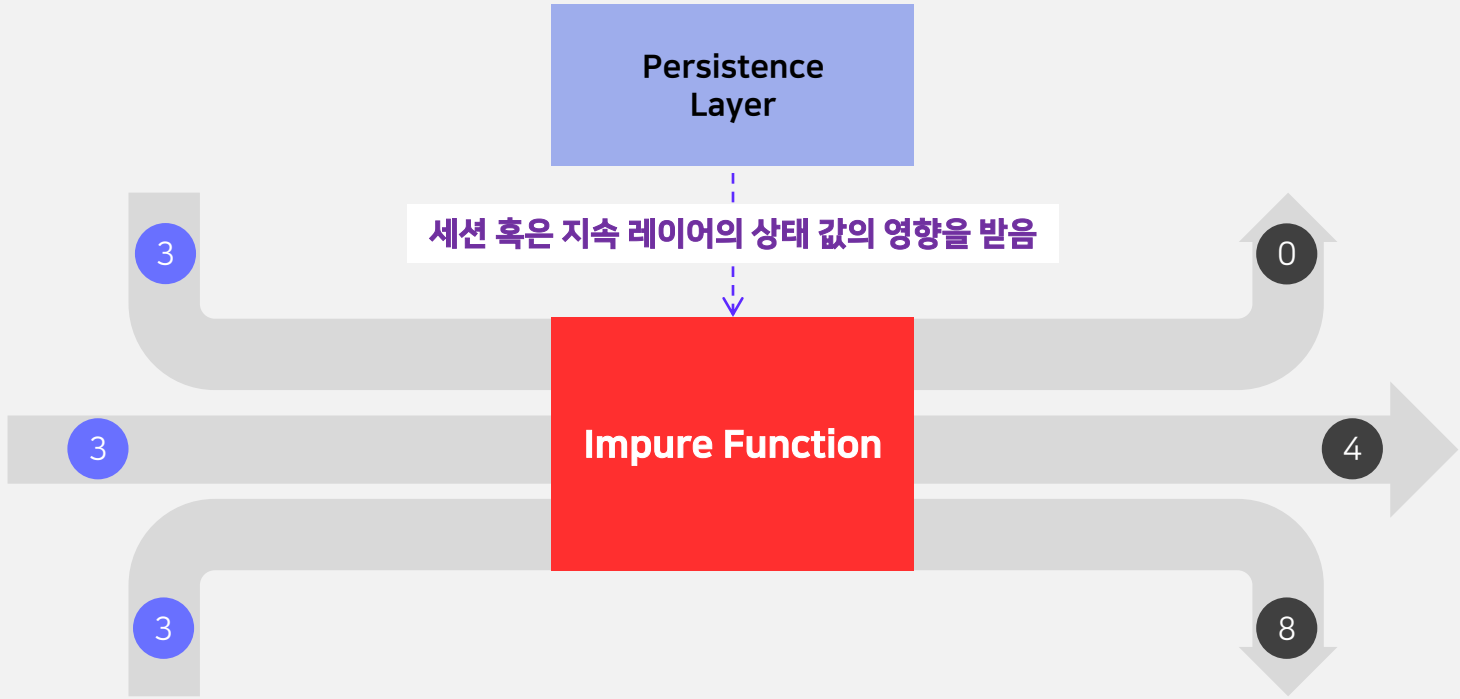
```python
import requests
import re

_URL_ = 'https://pycon.kr'


def sum(a, b):
    res = requests.get(_URL_)
    year = int(re.findall('\d+', res.url)[0])
    if year == 2018:
        return a + b
    return -1


if __name__ == '__main__':
    print(sum(1, 2))
```

원격에서 정보를 가져옵니다.

원격에서 가져온 정보에 따라 출력 값을 달리 제공합니다.

위와 같이 입력 값 외에 상태가 출력 값에 영향을 끼친다면
우리는 이것을 비 순수 함수라 부릅니다.

비 순수 함수 (Impure Function)



함수에 같은 입력 값이 제공되더라도

함수 내부에서 참조하는 데이터베이스, 세션, 원격 통신 등의 지속 레이어(Persistence Layer)에 의해

출력 값이 변경되는 함수를 의미합니다.

test/complex/main.py

```python
import requests
import re

_URL_ = 'https://pycon.kr'


def sum(a, b):
    res = requests.get(_URL_)
    year = int(re.findall(r'\d+', res.url)[0])
    if year == 2018:
        return a + b
    return -1


if __name__ == '__main__':
    print(sum(1, 2))
```

아까와 동일하게 TC를 구성해봅시다.

Test Cases (TCs)

TC1 `func sum(a, b)`

Given (a=3, b=4)

Expected 7

TC2 `func sum(a, b)`

Given (a=-1, b=5)

Expected 4

TC3 `func sum(a, b)`

Given (a=None, b=2)

Expected NaN

test/complex/test_main.py

```python
import unittest
from main import sum

class TestComplexFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(-1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        with self.assertRaises(TypeError):
            sum(None, 2)
```

Test Cases (TCs)

TC1 `func sum(a, b)`

Given (a=3, b=4)

Expected 7

TC2 `func sum(a, b)`

Given (a=-1, b=5)

Expected 4

TC3 `func sum(a, b)`

Given (a=None, b=2)

Expected raise Exception

test/complex/test_main.py

```python
import unittest
from main import sum

class TestComplexFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(-1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        with self.assertRaises(TypeError):
            sum(None, 2)
```

Test Cases (TCs)

TC1

`func sum(a, b)`

Given (a=3, b=4)
Expected 7
Actual 7 (Passed)

TC2

`func sum(a, b)`

Given (a=-1, b=5)
Expected 4
Actual 4 (Passed)

TC3

`func sum(a, b)`

Given (a=None, b=2)
Expected raise Exception
Actual raise Exception (Passed)

모든 테스트 코드는 통과합니다.

test/complex/test_main.py

```python
import unittest
from main import sum

class TestComplexFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(-1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        with self.assertRaises(TypeError):
            sum(None, 2)
```

Test Cases (TCs)

TC1

`func sum(a, b)`

Given	(a=3, b=4)
Expected	7
Actual	**-1 (Failed)**

TC2

`func sum(a, b)`

Given	(a=-1, b=5)
Expected	4
Actual	**-1 (Failed)**

TC3

`func sum(a, b)`

Given	(a=None, b=2)
Expected	raise Exception
Actual	**-1 (Failed)**

하지만 아마도 내년 PYCON에서는 모두 에러가 발생할 것입니다.

test/complex/main.py

```python
import requests
import re

_URL_ = 'https://pycon.kr'


def sum(a, b):
    res = requests.get(_URL_)
    year = int(re.findall(r'\d+', res.url)[0])
    if year == 2018:
        return a + b
    return -1


if __name__ == '__main__':
    print(sum(1, 2))
```

여기서 문제는 우리의 테스트 코드에서
원격 정보를 제어할 수 없어서 발생하고 있습니다.

test/complex_di/main.py

```python
import requests
import re

_URL_ = 'https://pycon.kr'


def _get_redirect(url):
    return requests.get(url)


def _get_year(get_redirect):
    return int(re.findall(r'\d+', get_redirect(_URL_).url)[0])


def sum(get_year, get_redirect, a, b):
    year = get_year(get_redirect)
    return a + b if year == 2018 else -1


if __name__ == '__main__':
    print(sum(_get_year, _get_redirect, 1, 2))
```

완전한 의존성 관리를 위해서는
가장 원격 통신과 관계된 단말 모듈이 대체 되어야 하지만
이번에는 코드 직관성을 위해 DAO 역할을 하는 _get_year를
대체하는 형식으로 진행 해보겠습니다.

기존 함수에 의존성을 받기 위해
매개변수 get_year, get_redirect가 추가되었습니다.

test/complex_di/test_main.py

```python
import unittest
from main import sum


def _get_year(_):
    return 2018


class TestComplexFunctions(unittest.TestCase):
    def test_sum(self):
        target = sum(_get_year, None, 3, 4)
        expected = 7
        self.assertEqual(target, expected)

    def test_sum_with_negative_args(self):
        target = sum(_get_year, None, -1, 5)
        expected = 4
        self.assertEqual(target, expected)

    def test_sum_with_none(self):
        with self.assertRaises(TypeError):
            sum(_get_year, None, None, 2)
```

Test Cases (TCs)

TC1

`func sum(a, b)`

Given	(a=3, b=4)
Expected	7
Actual	7 (Passed)

TC2

`func sum(a, b)`

Given	(a=-1, b=5)
Expected	4
Actual	4 (Passed)

TC3

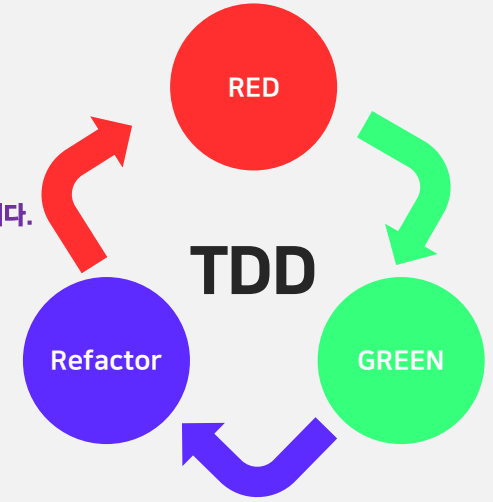
`func sum(a, b)`

Given	(a=None, b=2)
Expected	raise Exception
Actual	raise Exception (Passed)

TDD and The Dilemma
TDD와 함정

TDD (Test Driven Development)

1. 테스트 코드를 작성합니다.
물론 새로 작성한 테스트 코드는 Failed가 됩니다.



2. 테스트 코드를 통과할 정도의
코드를 채웁니다.
테스트 결과는 Passed가 되어야 합니다.
이 과정에서 테스트 코드를 수정하면 안됩니다.

3. 코드의 동작에는 차이가 없게 하면서
코드를 최적화합니다.
이 과정에서 테스트는 Passed를 유지해야합니다.

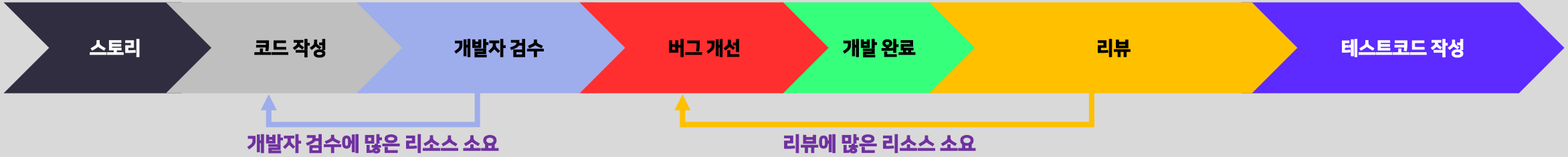
TDD는 테스트 주도 개발을 의미합니다.

TDD는 어떤 프레임워크나 도구가 아닌 하나의 개발 방법론으로 존재하고 있습니다.

일반적인 테스트 작성 업무와 TDD의 차이

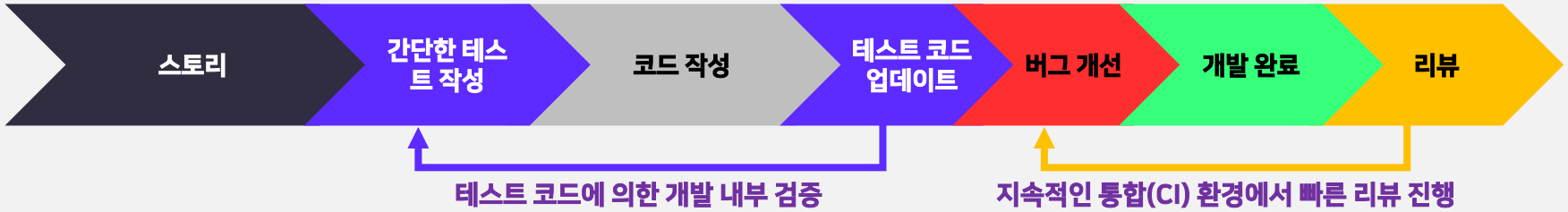
일반적인 테스트 코드 작성

개발 코드를 작성 후 개발자가 검수를 하며 버그를 테스트
리뷰 이후 혹은 리뷰 전에 테스트 코드를 작성



TDD

오류가 발생하지 않을 정도로 테스트 코드를 먼저 작성 후
코드의 기능이 추가 되면서 테스트 코드도 같이 구체화, 개발자 검수에 대한 테스크가 아에 없거나 적어짐



TDD 실제 업무 예시

"임의의 수를 리스트로 입력하면 짝수 인 것의 개수를 세어 그 개수의 5를 곱해 반환하는 기능 요구"

사용자의 스토리로부터 먼저 개발에 필요한 Task 요구사항을 구체화 합니다.

혹은 앞으로 개발할 내용의 기능을 구체화 합니다.

우리는 이것을 피쳐(feature) 혹은 스팩(spec)이라 부릅니다.

tdd/test_main.py

```python
import unittest

from main import calc


class TestTdd(unittest.TestCase):
    def test_calc(self):
        given = (1, 2, 3, 4, 5)
        target = calc(given)
        expected = 15
        self.assertEqual(target, expected)

    def test_calc_with_various_odd(self):
        given = (1, 3, 5, 6, 7)
        target = calc(given)
        expected = 20
        self.assertEqual(target, expected)

    def test_calc_with_empty(self):
        given = ()
        target = calc(given)
        expected = 0
        self.assertEqual(target, expected)

    def test_calc_with_none(self):
        given = None
        with self.assertRaises(TypeError):
            calc(given)
```

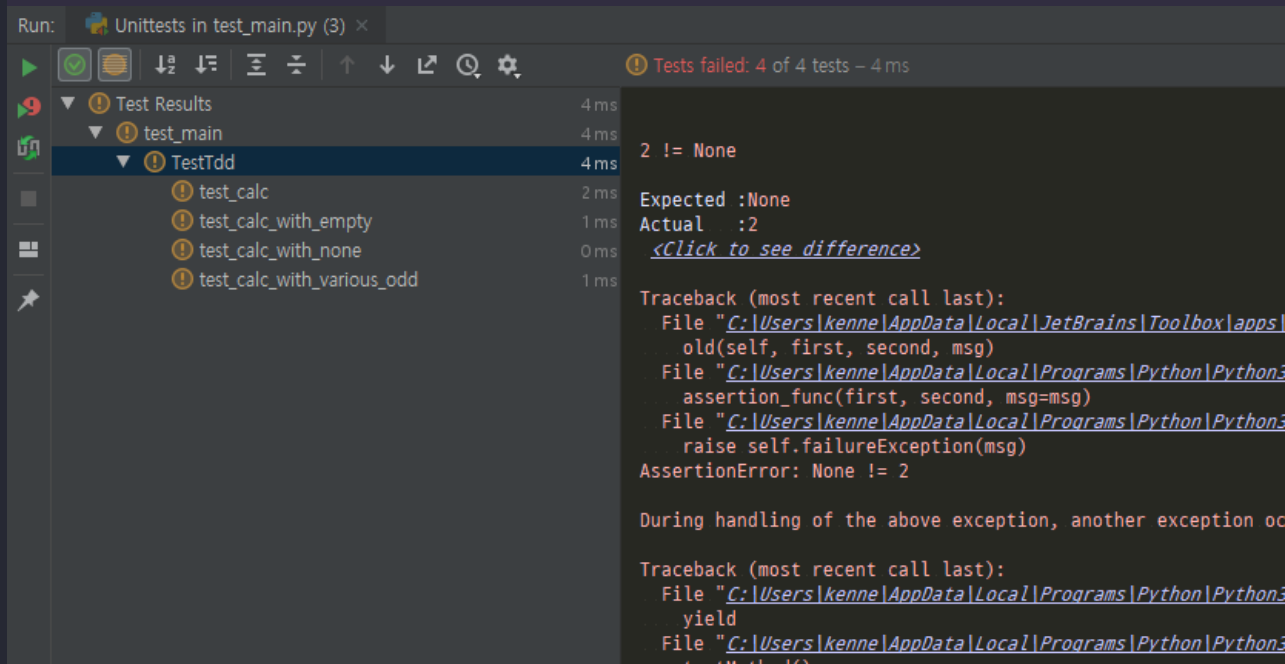
tdd/main.py

```python
def calc(num_list):
    pass
```

빌드 에러가 발생하지 않을 정도의 최소 구현체 코드만을 작성하고

테스트 코드를 먼저 작성합니다.

테스트 코드는 당연히 에러가 발생하게 됩니다.



tdd/test_main.py

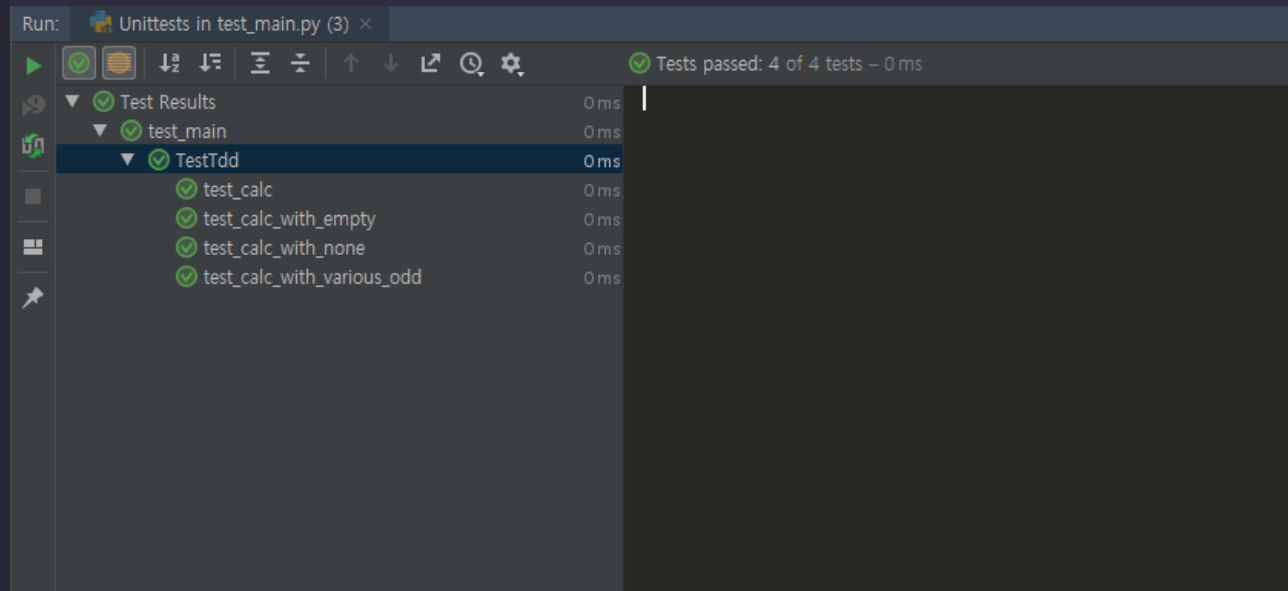
```python
import unittest

from main import calc


class TestTdd(unittest.TestCase):
    def test_calc(self):
        given = (1, 2, 3, 4, 5)
        target = calc(given)
        expected = 15
        self.assertEqual(target, expected)

    def test_calc_with_various_odd(self):
        given = (1, 3, 5, 6, 7)
        target = calc(given)
        expected = 20
        self.assertEqual(target, expected)

    def test_calc_with_empty(self):
        given = ()
        target = calc(given)
        expected = 0
        self.assertEqual(target, expected)

    def test_calc_with_none(self):
        given = None
        with self.assertRaises(TypeError):
            calc(given)
```

tdd/main.py

```python
def calc(num_list):
    return len(list(filter(lambda num: num % 2 == 1, num_list)))
```

이번에는 반대로 테스트가 통과 될 정도로

최소한의 코드를 작성합니다.



tdd/test_main.py

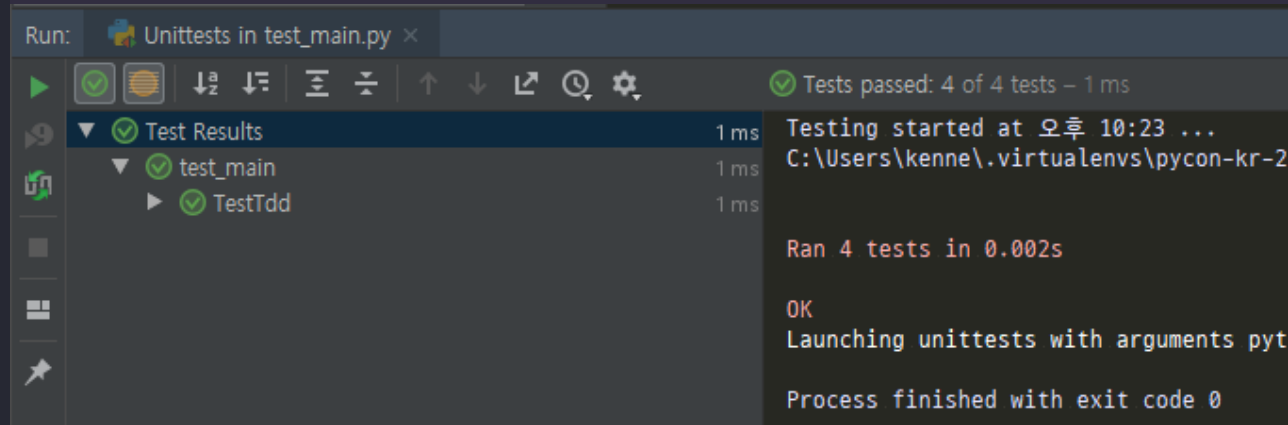
```python
import unittest

from main import calc


class TestTdd(unittest.TestCase):
    def test_calc(self):
        given = (1, 2, 3, 4, 5)
        target = calc(given)
        expected = 15
        self.assertEqual(target, expected)

    def test_calc_with_various_odd(self):
        given = (1, 3, 5, 6, 7)
        target = calc(given)
        expected = 20
        self.assertEqual(target, expected)

    def test_calc_with_empty(self):
        given = ()
        target = calc(given)
        expected = 0
        self.assertEqual(target, expected)

    def test_calc_with_none(self):
        given = None
        with self.assertRaises(TypeError):
            calc(given)
```

tdd/main.py

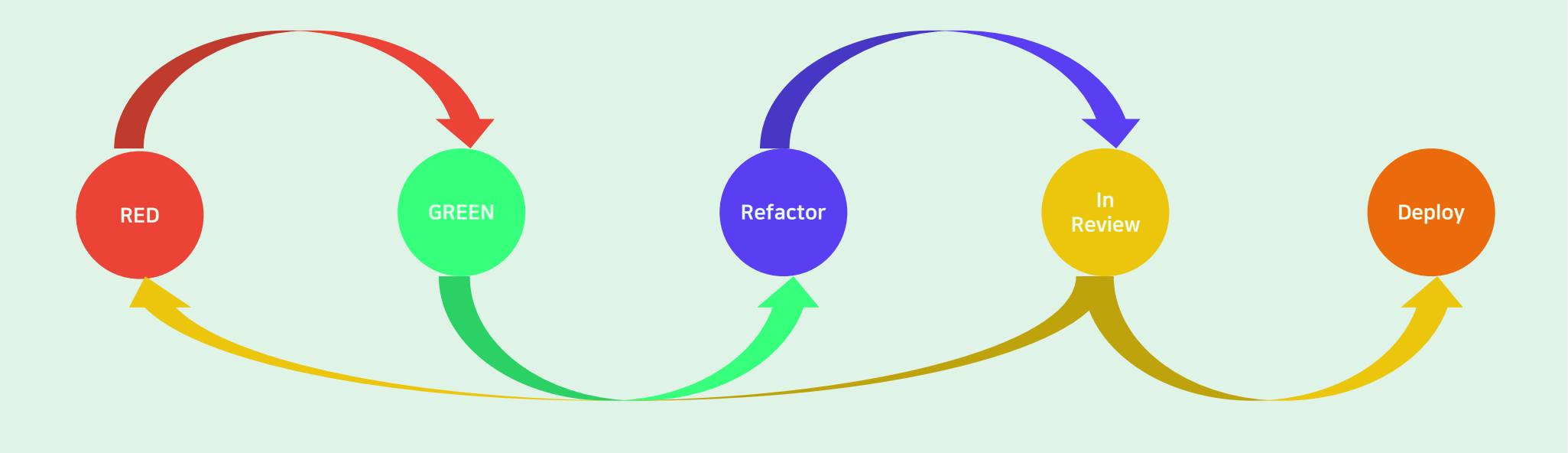
```python
def _filter_odd(num):
    return num % 2 == 1


def _get_odd_len(num_list):
    return len(list(filter(_filter_odd, num_list)))


def calc(num_list):
    return _get_odd_len(num_list) * 5
```

마찬가지로 테스트가 통과되는 범위에서

리팩토링을 진행합니다, 이후 다시 첫번째 과정을 반복합니다.



은 총알은 없다. (No silver bullet)

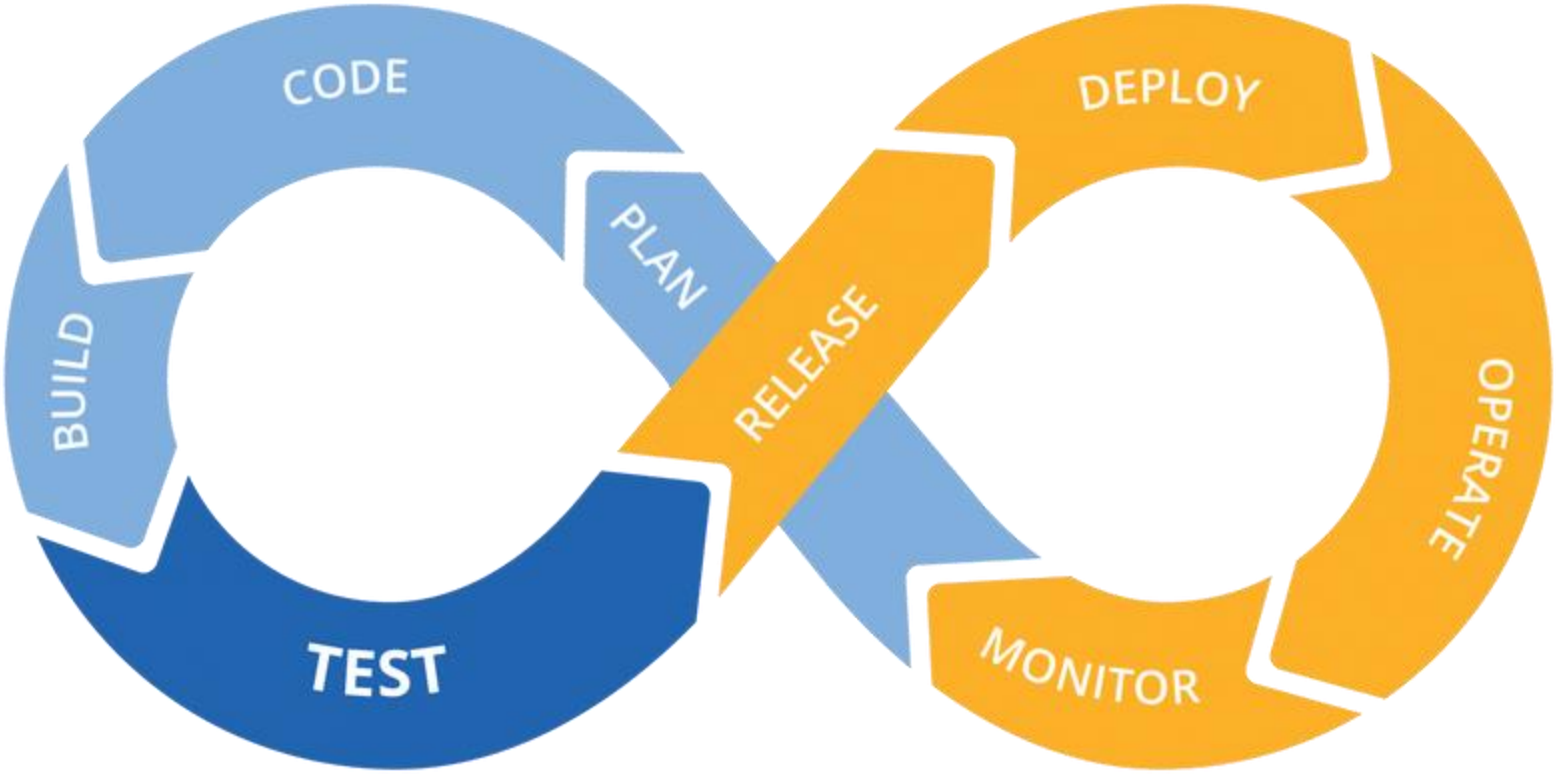


Workflow Automation with CI

TDD는 여러분의 버그를 미연에 없애주고 생산성을 높여주는 도구가 아닙니다.
개발 스팩의 변경이 빈번하지 않아야 하고 무엇보다 CI(Continuous Integration) 환경이 구성되어 있어야
비로소 TDD의 장점을 극대화 할 수 있습니다.

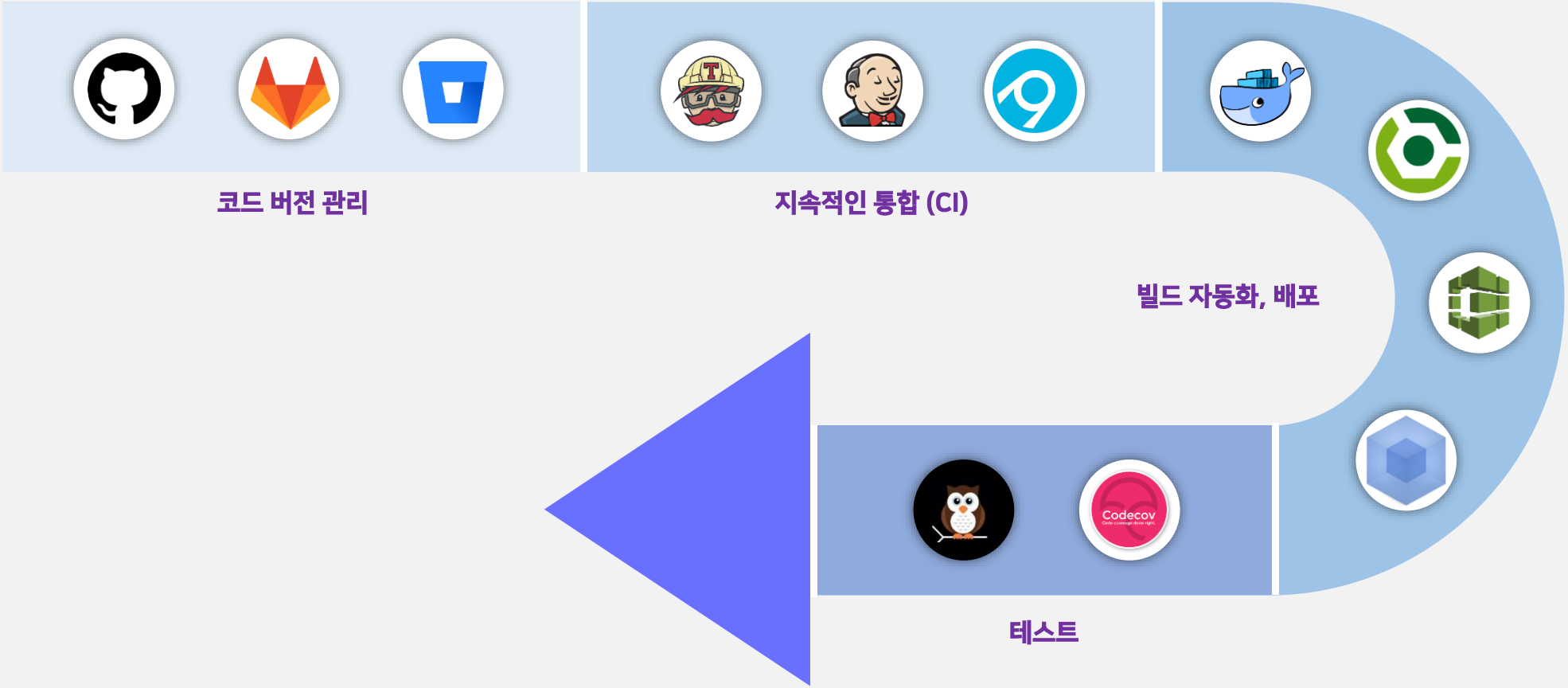
지속적인 통합이란?

https://instabug.com/blog/continuous-integration-tools



일감 관리, 테스트, 빌드, 배포, 모니터링 등의 개발의 관리에 필요한 일련의 작업을 자동화할 수 있는 환경을 말합니다.

지속적인 통합 설계



PYCON KOREA 2018

Question?

PYCON KOREA 2018

Thank you!

https://github.com/KennethanCeyer/pycon-kr-2018

kennethan@nhpcw.com