

Refactoring 기법과 Code Smell

리팩토링 학습 순서

[STEP 1] 리팩토링 기법의 기본기를 살펴본다.

[STEP 2] 코드에서 유지보수성 / 확장성에 문제가 될 수 있는 잠재적 결함 (code smell)에 대해 알아보며, 각 증상을 해결하는 리팩토링 기법에 대해 자세히 다룬다.

[STEP 1]



리팩토링시 기본기

[STEP 2]

Code Smell

불필요한 복잡성 관련
(Long Method, Large Class 등)

데이터 구성 관련 Smell
(Long Parameter List 등)

객체 관계 및 의존성 문제
(Message Chains 등)

책임 분리와 응집도 저하
(Divergent Change, Shotgun Surgery 등)



[STEP 1] 리팩토링 기본 기법

리팩토링에서 자주 활용되는 기본 작업



리팩토링시
자주 사용되는 기법

리팩토링 작업의 원칙

리팩토링 작업을 할 때는 작은 단위로 작업하고, 테스트가 항상 가능해야 한다.

1. 작은 단위로 리팩토링

작은 단위로 리팩토링을 해야 만약 실수를 했을 때 쉽게 버그를 해결할 수 있다.

2. 코드가 깨지지 않도록 리팩토링 하자

리팩토링 작업을 할 때는 테스트 코드를 갖춘 상태에서 리팩토링을 해야 한다.

또한, 한 번에 많은 작업을 하면 필시 코드가 깨지게 되므로 작은 단위로 리팩토링 한다.

변수 쪼개기 (Split Temporary Variable)

여러 목적을 갖는 변수를 쪼갬다.

- ✓ 대입이 2번 이상 이뤄진다면 변수가 여러 목적으로 사용되고 있을 수 있다.
- ✓ 하나의 변수가 다양한 목적으로 재사용 되는 경우, 읽는 이에게 혼란을 준다.

```
temp = 2 * ( height + width );  
print(temp)
```

```
# ...
```

```
temp = height * width;  
print(temp)
```



```
hap = 2 * (height + width)  
print(hap)
```

```
# ...
```

```
area = height * width  
print(area)
```

[도전] 변수 쪼개기

다중적으로 사용되는 임시 변수를 분리 시켜본다

```
temp = 24 # 섭씨
temp = (temp * 9 / 5) + 32 # 화씨로 변환
print("섭씨 온도를 화씨로 변환:", temp)

temp = 10.0 # 길이 (미터)
area = temp * temp # 면적 계산
print("정사각형 면적:", area)
```

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/refactoring/split_variable.py

문장 이동 (Move Statements)


statements를 적절한 위치에 배치 해줘야 가독성, 유지보수성을 높여줄 수 있다.

- ✓ statements 예) 변수 선언문, 할당문, 함수 호출문, 반복문, 방어절 등등

(예시) 변수 선언문과 사용 코드를 가깝도록 변경

- ✓ 덜 중요한 변수일수록 변수의 범위(스코프)를 작게 가지면 좋다.
- ✓ 특히, 함수 추출이 쉽게 되려면 같이 추출될 변수와 동작들이 같이 모여 있어야 한다.

```
result = 0 ;  
  
// ...  
// ...  
  
for i in range(100):  
    result += do_something()
```



사용되는 곳과 선언하는 코드간 거리가 멀다

```
// ...  
// ...  
result = 0 ;  
for i in range(100):  
    result += do_something()
```

함수 추출도 쉽고 변수 파악도 쉽다

Guard Clauses

Guard Clauses(방어절)는 특정 조건을 확인하여 early return 하는 방법이다

- ✓ 중첩된 조건을 줄이고, 코드의 주요 동작을 더 잘 띄게 한다.

Kent Beck 의 가드절 python 코드 사례

- ✓ 가독성 측면에서 비교해본다

```
Parameters:
    queue_name(str): The name of the new queue.
    """

    if queue_name not in self.queues:
        self.emit_before("declare_queue", queue_name)
        self.queues[queue_name] = Queue()
        self.emit_after("declare_queue", queue_name)

        delayed_name = dq_name(queue_name)
        self.queues[delayed_name] = Queue()
        self.delay_queues.add(delayed_name)
        self.emit_after("declare_delay_queue", delayed_name)

def enqueue(self, message, *, delay=None):
    """Enqueue a message.
```

```
Parameters:
    queue_name(str): The name of the new queue.
    """

    if queue_name in self.queues:
        return

    self.emit_before("declare_queue", queue_name)
    self.queues[queue_name] = Queue()
    self.emit_after("declare_queue", queue_name)

    delayed_name = dq_name(queue_name)
    self.queues[delayed_name] = Queue()
    self.delay_queues.add(delayed_name)
    self.emit_after("declare_delay_queue", delayed_name)

def enqueue(self, message, *, delay=None):
    """Enqueue a message.
```


loop 쪼개기 (Split Loop)

loop 에 여러 기능이 섞여 있는 경우, 각 기능별 loop 로 분리할 수 있다.
분리된 loop 들은 각각 한가지 기능을 하는 함수들로 추출 및 표준 알고리즘 이용 가능

```
numbers = [1, 2, 3, 4, 5]

# 1. 배열의 각 원소를 2배
# 2. 배열의 각 원소들의 누적합
sum_ = 0
for num in numbers:
    num *= 2
    sum_ += num
```



```
numbers = [1, 2, 3, 4, 5]

# 데이터를 변환
doubled = []
for num in numbers:
    doubled.append(num * 2)

# 합산 작업
sum_ = 0
for num in doubled:
    sum_ += num
```



```
numbers = [1, 2, 3, 4, 5]

doubled = [num * 2 for num in numbers]
sum_ = sum(doubled)
```

변수를 점진적으로 클래스로 옮기기

다음 변수 `a` 를 `Something` 이라는 클래스로 옮기려고 한다.

```
a = 10      # 선언  
a += 5      # 변경  
print(a)    # 읽기
```



```
s = Something(10)  
s.add_a(5)  
print(s.get_a())
```

의존성이 낮은 코드부터 옮겨주는 작업을 해준다.

1. 선언
2. 변경
3. 읽기

[도전] 변수를 클래스로 옮기기

level, hp, mp, name 변수들을 Player 클래스로 옮기는 작업을 한다.

- ✓ 코드들 간의 의존도를 고려해서 옮겨 주도록 한다.
- ✓ 옮기는 과정에서 같이 캡슐화 될 로직들을 판단한다.

```
class Player:
    ...

def test_game_player_effect():
    level = 10
    hp = 180
    mp = 200
    name = "hwan"

    # level up effect
    hp += 10
    mp -= 20
    level += 1

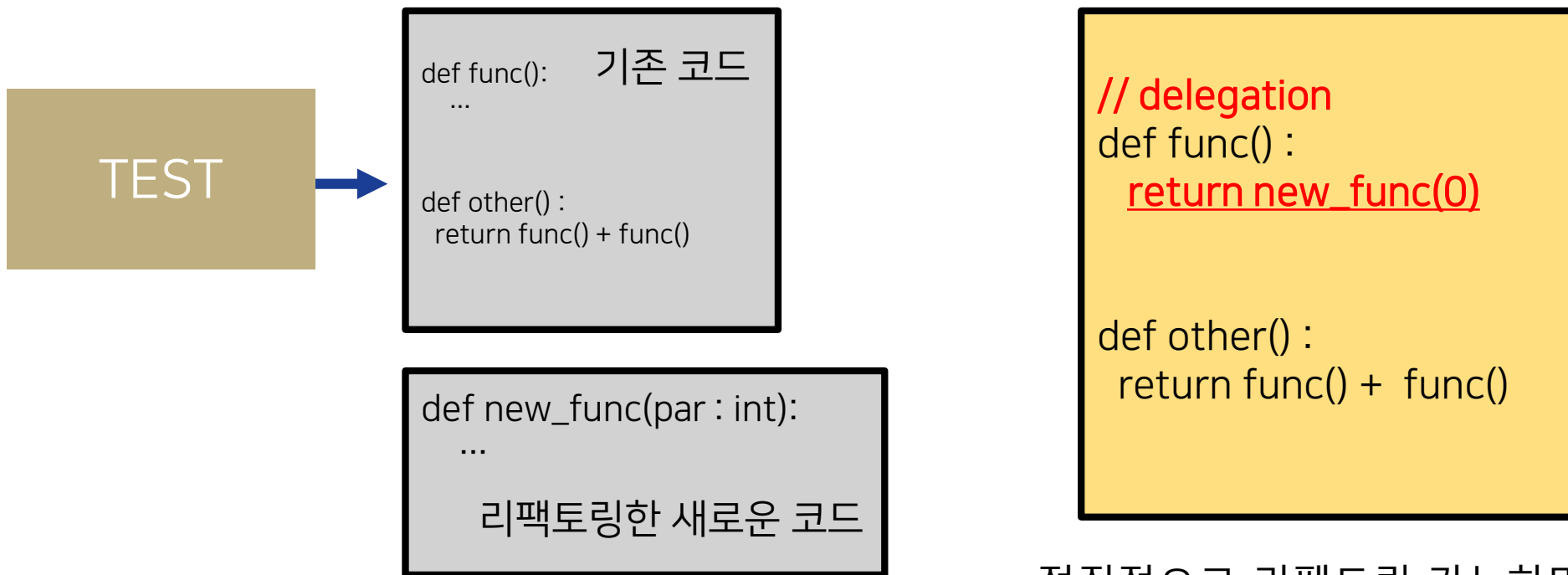
    assert hp == 190
    assert mp == 180
    assert level == 11
```

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/refactoring/extract_class.py

위임 사용하기

위임을 이용하면 기존 테스트 코드와의 호환성을 유지하면서 리팩토링을 할 수 있다.

모든 테스트가 마무리 된 후에 기존 코드를 새로운 코드로 대체한다



점진적으로 리팩토링 가능하며,
기존 테스트 케이스들을 이용할 수 있다

[도전] 새로운 함수 도입

테스트 코드를 새로 짜지 않고, 새로 만든 함수를 테스트 할 수 있어야 한다.

함수 정의

```
def sum_from_1_to_100():  
    result = 0  
    for num in range(1, 101):  
        result += num  
    return result
```

테스트 코드

```
def test_sum_from_1_to_100():  
    ret = sum_from_1_to_100()  
    assert ret == 5050
```

새로 만들 함수는 좀 더 일반화된 함수이다.

```
def newRangedSum(int a, int b):  
    return (b)*(b+1)/2 - (a-1)*a / 2;
```

[Python-CRA-Example/refactoring/delegate.py](https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/Python-CRA-Example/refactoring/delegate.py) at main ·
[jeonghwan-seo/Python-CRA-Example](https://github.com/jeonghwan-seo/Python-CRA-Example)



[STEP 2] Code Smell과 리팩토링

나쁜 증상을 식별하고 리팩토링을 해본다

코드 스멜의 패턴을 파악해보고, 이를 해결하는 리팩토링 기법들을 알아본다.



불필요한 복잡성을 띄는 Code Smell

SW 의 복잡도는 개발자들의 작업시간에 영향을 끼친다.

따라서 불필요한 복잡성을 제거하여 가독성이 좋고, 이해하기 좋은 구조로 변경해야한다.

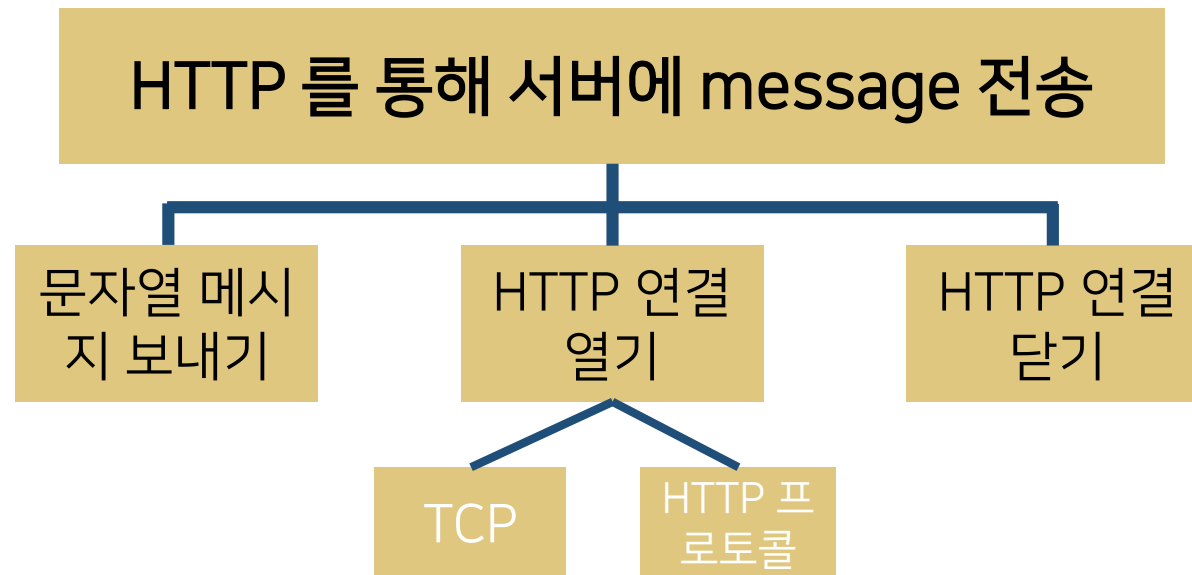
불필요한 복잡성으로 인한 Code Smell 패턴

- ✓ 긴 함수 (Long Method)
- ✓ 거대한 클래스 (Large Class)
- ✓ 추측성 일반화 (Speculative Generality)

[Bad Smell] 긴 함수

과도하게 길고 복잡한 함수는 이해, 유지보수를 어렵게 만든다.

- ✓ 새로운 기능들을 기존 함수에 계속 추가하고 분리하지 않게 되면 함수는 점점 거대해진다.
- ✓ 긴 함수들은 한 가지 이상의 일을 하는 경우가 많으며, 이는 분리의 신호가 된다.

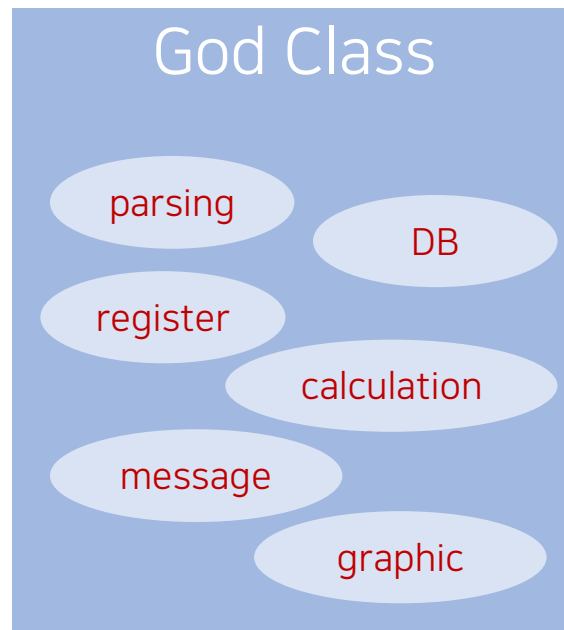


긴 함수를 서브 함수들로 분리하면 코드의 가독성, 유지보수성을 높일 수 있다

[Bad Smell] 거대한 클래스

클래스가 작게 시작했지만 기능이 계속 추가되면서 점점 거대 해진다.

- ✓ 클래스가 많은 멤버 함수, 멤버 변수, 코드 라인들을 포함하게 된다.
- ✓ 하나의 클래스가 여러 기능들을 다루다 보면 서로 관련 없는 기능들로 인해 응집도가 떨어지게 될 수 있다.



클래스는 응집도가 높아야 좋지만,
거대해질수록 응집도가 낮아질 가능성이 있다.

과도하게 짧은 함수 or 클래스

너무 지나친 함수 or 클래스 쪼개기는 오히려 안 좋을 수 있다.

- ✓코드의 흐름을 파악할 때 왔다 갔다 해야 해서 파악이 힘들 수 있다.
- ✓짧은 함수, 클래스들을 많이 만들다 보면 **얕은 추상화**가 많이 생기게 될 수 있다.
얕은 함수들이 많아지면 오히려 전체 시스템의 복잡도는 높아진다.
- ✓인터페이스만 늘고 복잡도를 낮춰야 하는 실질적인 추상화가 이뤄지지 않게 된다.

추상화 체크리스트

다음 질문들에 대해 YES 일 경우, 좋은 추상화/안 좋은 추상화를 구분해보자

질문	좋은 추상화 / 안 좋은 추상화 구분
이 추상화가 무언가 복잡한 일을 대신 해주는가?	
이 추상화를 사용하면 내가 알아야 할 정보가 줄어드는가?	
이 추상화를 쓰려면 오히려 내부를 더 많이 알아야 하는가?	
함수/클래스의 name 이 하는 일만 반복적으로 설명하고 있는가?	
이름과 인터페이스만 봐도 무엇을 하는지 명확히 추론 가능한가?	
사용하기 위해 많은 매개변수, 사전 지식, 복잡한 순서를 요구하는가?	
함수 이름은 고수준인데 실제 내용은 아주 저수준의 구현을 하는가?	
현재 맥락 외에도 유사한 상황에서 쉽게 재사용 가능한가?	

함수 인라인하기 (Inline Function)

오히려 함수 추출된 경우가 복잡하다면 본문 내용을 그대로 사용하는 것을 고려한다.

인라인 고려해볼 수 있는 상황

- ✓ 불필요한 간접 호출로 인해 코드 가독성이 떨어지거나 흐름 파악이 힘든 경우
- ✓ 함수 본문 내용이 호출자 입장에서 오히려 명확한 경우
- ✓ 얇은 추상화로 인해 오히려 호출자가 내부 구현을 알아야 정확히 사용 가능한 경우

```
def add_empty_value_for_attribute(self, attribute: str) :  
    self.data[attribute] = "";
```

- 호출자가 내부 구현을 알아야 잘 사용할 수 있다.
- 또한 `data[attribute] = ""` 이 한 줄이 차라리 이해가 더 잘 간다.

[참고] IDE 에서 코드로 점프하는 단축키

IDE 에서 정의, 선언 등으로 점프하는 단축키를 알아두자

- ✓ 대부분 IDE 에서 “**ctrl + 클릭**” 은 코드 탐색 기능

pycharm 에서 단축키

- ✓ 정의로 이동 : **ctrl + B** 또는 **ctrl + 클릭**
- ✓ 정의 미리보기 : **ctrl + shift + i**
- ✓ 구조 보기 : **ctrl + F12** (현재 파일의 함수, 필드 목록을 보여준다)
- ✓ 참조 찾기 : **alt + F7** (참조하고 있는 코드를 검색해준다)


Replace Temp with Query

임시 변수로 코드가 복잡한 경우에 임시 변수를 Query 함수(함수 호출)로 변경해보는 것을 고려할 수 있다.

query 함수
: 정보를 요청하는 함수
상태를 변경 X

```
class Rectangle:
    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height

    def print_area(self) -> float:
        area = self.width * self.height # 임시 변수
        print(f"Area: {area}")
        return area
```



```
class Rectangle:
    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height

    def area(self) -> float:
        return self.width * self.height

    def print_area(self) -> float:
        print(f"Area: {self.area()}")
        return self.area()
```

query 함수로 교체하여 불필요한
임시변수 사용을 없앴다.

[Bad Smell] 추측성 일반화

“나중에 필요할 거야” 라는 생각으로 지금 당장 쓰지 않을 코드들을 만들어 두곤 한다.

- ✓ 당장 필요 없는 추상 클래스, 인터페이스와 같은 확장 포인트(hooking)를 만들어 두는 것
- ✓ 당장 필요 없는 매개변수
- ✓ 사용한 곳이 없는 함수, 클래스
- ✓ 당장 불필요한 과도한 일반화

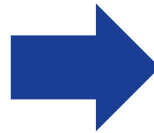
➔ 과도한 예측은 복잡성을 늘리므로 경계해야 한다.

➔ 확장 가능하게 설계하라 ≠ 확장하지도 않았는데 미리 복잡한 추상화를 만들라

함수의 매개변수화 (parameterize function)

여러 함수들이 거의 동일한 기능을 수행하지만
내부적으로 다른 값들을 사용하는 경우 코드 중복일 가능성이 있다.
사용하는 값에 대한 매개변수를 이용해 좀 더 일반화된 함수로 변경한다.

```
def increaseFivePercentPower(self):  
    self.power = self.power * 1.05  
  
def increaseTenPercentPower(self):  
    self.power = self.power * 1.10
```



```
def increasePower(self, ratio):  
    self.power = self.power *  
        (double) (100 + ratio) / 100
```

[도전] 함수의 매개변수화

소득 분위 별 세금을 계산하는 코드이다. 여기서 중복된 코드를 매개변수를 이용해 일반화 한다.

```
class TaxCalculator:
    def calculate_tax(self, income: float) -> float:
        tax = 0
        tax += self._lower_bracket(income) * 0.1
        tax += self._middle_bracket(income) * 0.2
        tax += self._upper_bracket(income) * 0.3
        return tax

    def _lower_bracket(self, income: float) -> float:
        return min(income, 30000.0)

    def _middle_bracket(self, income: float) -> float:
        return min(income, 100000.0) - 30000 if income > 30000 else 0

    def _upper_bracket(self, income: float) -> float:
        return income - 100000 if income > 100000 else 0
```

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/refactoring/function_paramterized.py

데이터 구성 관련 Code Smell

연관된 데이터가 흩어져 다닌다면 적절한 관리가 필요하다.

적절한 관리를 하게 된다면 유지보수성, 가독성, 확장성 등을 향상시킬 수 있다.

흩어져 있는 데이터들로 나타나는 Code Smell 패턴

- ✓데이터 뭉치 (Data Clumps)
- ✓긴 매개변수 리스트 (Long Parameter List)
- ✓원시 타입 집착 (Primitive Obsession)

[Bad Smell] 데이터 뭉치

항상 같이 뭉쳐 다니는 데이터들은 묶어 줄 수(캡슐화) 있다.

```
class Drawer {  
    public void drawCircle (double x, double y, double radius){}  
    public void drawTriangle (double x, double y, double height, double width){}  
    public void drawRectangle (double x, double y, double height, double width){}  
    // ...  
}
```



Point



Dimension

함수의 매개변수 리스트에서 항상 같이 전달되는 값들이 존재
➔ 의미있는 클래스로 묶어준다.

[Bad Smell] 데이터 뭉치

항상 같이 뭉쳐 다니는 데이터들은 묶어 줄 수 있다.

```
class Drawer:
    def draw_circle(self, x: float, y: float, radius: float) -> None:
        ...

    def draw_triangle(self, x: float, y: float, height: float, width: float) -> None:
        ...

    def draw_rectangle(self, x: float, y: float, height: float, width: float) -> None:
        ...
```

Point

Dimension

함수의 매개변수 리스트에서 항상 같이 전달되는 값들이 존재
→ 의미있는 클래스로 묶어준다.

[Bad Smell] 긴 파라미터 리스트

긴 파라미터 리스트의 생성자, 함수들은

- ✓ 가독성이 떨어지고 사용하기 어렵다.
- ✓ 파라미터 순서 실수 등의 버그 유발 가능성이 있다.

```
class Product:
```

```
    def __init__(self, name: str, price: float, category: str, stock_quantity: int,  
                  manufacturer: str, supplier: str, sku: str, color: str,  
                  weight: float, height: float, width: float, depth: float):
```

너무 과도한 파라미터 개수의 생성자

-> 다른 설계 방안을 생각해보는 것이 좋다.

[Bad Smell] 원시 타입 집착

int, char, double, string 등 원시 타입을 과도하게 사용하면

- ✓ string, int 등은 어떤 역할인지 명확하지 않기 때문에 표현력이 떨어진다.
- ✓ 여러 함수에서 데이터를 다루는 로직을 구현하기 때문에 로직들이 흩어진다.
- ✓ 정확한 타입의 데이터가 아닐 수 있기에 타입 안전성이 떨어진다.

```
def create_user(name: str, phone_num: str, money: float, date: str):
```

```
...
```

↓

```
class Phone :  
    string 지역번호;  
    string 앞자리;  
    string 뒷자리;
```

```
    관련 로직 1 ()  
    관련 로직 2 ()  
}
```

↓

```
class Money:  
    double amount;  
  
    관련 로직 1 ()  
    관련 로직 2 ()
```

↓

```
class Date:  
    int year;  
    int month;  
    int day;  
  
    관련 로직 1 ()  
    관련 로직 2 ()
```

객체 통째로 넘기기 (Preserve Whole Object)

객체로부터 값들을 얻은 후, 따로따로 넘기고 있는 경우에는 차라리 객체를 통째로 넘기도록 변경할 수 있다.

```
a: Point = Point(1, 2)
b: Point = Point(3, 4)

dist = calc_distance(a.x, a.y, b.x, b.y)
```



```
a: Point = Point(1, 2)
b: Point = Point(3, 4)

dist = calc_distance(a, b)
```


변경을 할 때 확인 가능한 응집도 Smell

응집도가 떨어지는 코드를 변경하는 경우, 흩어져 있는 코드들로 인해서 변경 작업이 어려워진다.

- ✓ 이런 증상을 확인하면 응집도가 떨어졌다는 신호!
- ✓ 변경하기 쉬운 코드로 만들기 위해서는 응집도를 높여야 하며, 이를 위해서 변경을 할 때 응집도가 저하된 증상을 알아 본다.

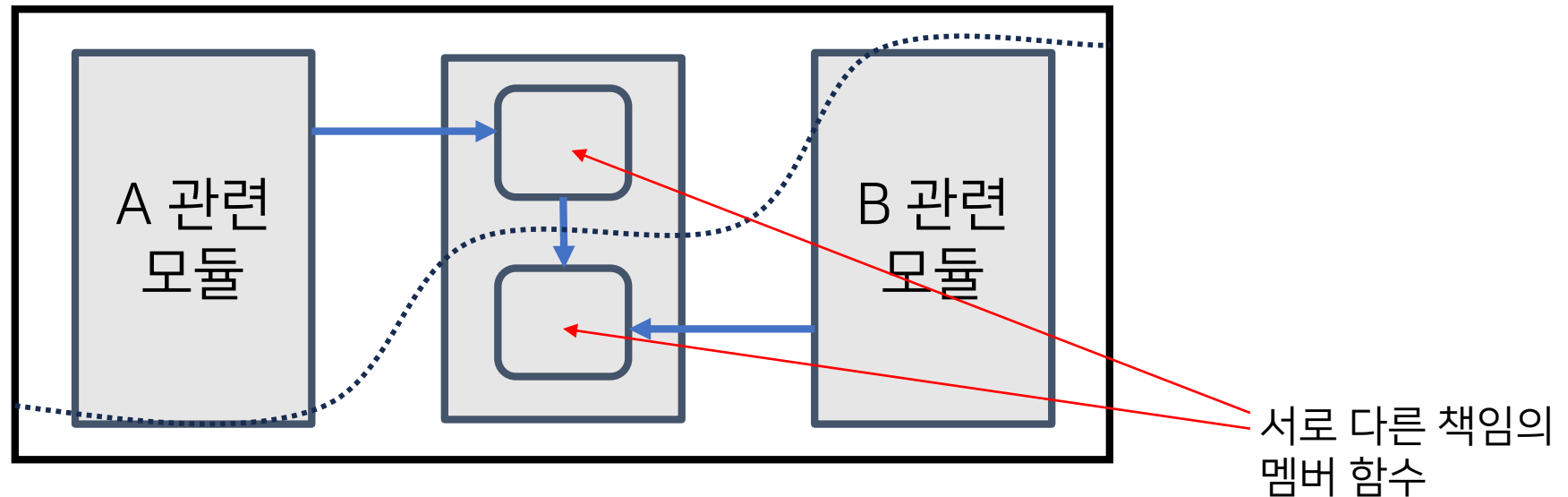
응집도가 떨어지는 증상으로 나타나는 Code Smell 패턴 종류

- ✓ 다양한 변경 이유 (Divergent Change)
- ✓ 산탄총 수술 (Shotgun Surgery)

[Bad Smell] 다양한 변경 이유 (Divergent Change)

한 클래스가 다른 이유들로 자주 변경되는 경우 = 여러 책임을 갖는 경우

예를 들어, 하나의 클래스가
"새로운 데이터베이스를 추가할 때, 새로운 금융상품을 추가할 때 등등"수정된다면,
이 클래스는 서로 다른 변경 이유를 한 데 모으고 있다는 것이다.



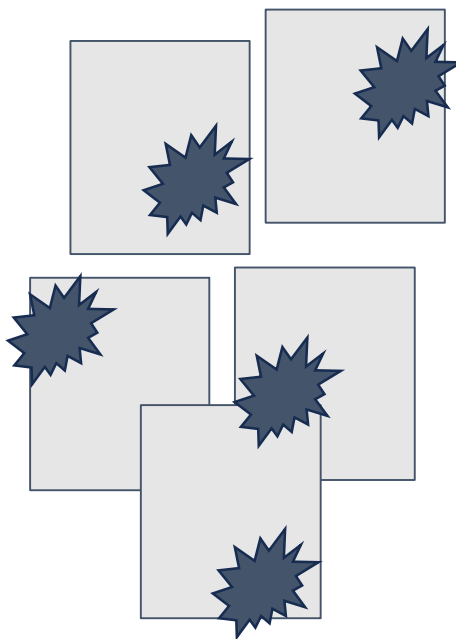
이렇게 다른 이유로 하나의 클래스가 변경 되는 경우 **분리하는 작업**이 필요하다.

[Bad Smell] 산탄총 수술 (Shotgun Surgery)

특정 변경을 할 때마다 여러 클래스들을 수정해야 하는 경우.
즉, 변경 해야하는 지점들이 여러 군데 퍼져 있는 경우이다.



변경 발생



예를 들어, **할인 정책**이 바뀌었을 때 **여러 클래스**(Order, PaymentProcessor, Invoice 등)에서 관련된 코드들을 수정해야 한다면 Shotgun Surgery 증상이다.

한 객체가 다른 객체에 의존할 때 불필요한 간접 참조를 하거나 과도한 의존성을 갖는 경우가 있다.

이런 구조는 객체 간 결합도를 높이고, 변경 시 영향을 받는 범위가 늘어 유지보수성이 떨어지게 되므로 의존성을 재조정 해야 한다.

- 메시지 체인 (Message Chains)
- 중개자 (Middle Man)

[Bad Smell] 메시지 체인

Client 가 원하는 Server 객체를 이용하려 하는데 여러 단계 걸쳐서 이용할 때 체이닝 구조가 나타나는 경우가 있다.

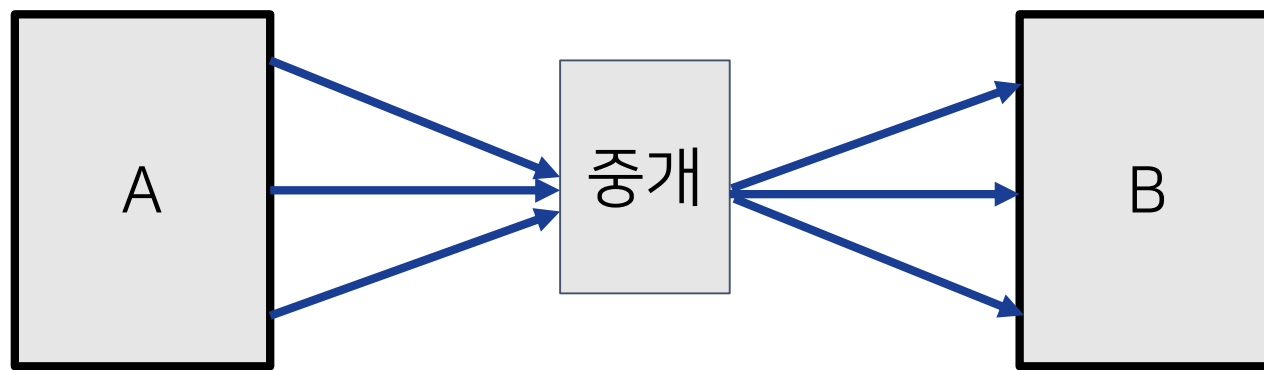
Client.getOrder().getProduct().getProductName()

하나의 객체가 다른 객체를 참조하고, 그 객체가 다시 또 다른 객체를 참조하는 식의 코드

이런 구조는 클라이언트 코드가 객체 간 관계를 너무 많이 알아야 한다.
캡슐화가 깨지고 결합도가 증가하는 구조

[Bad Smell] 중개자

한 클래스가 **하는 일 없이 다른 클래스에 구현을 위임만** 하고 있다면, 지나친 간접 참조를 하고 있는 상태로 중개자 역할을 해당 클래스를 제거를 하자.



중개자 클래스를 통해 B는 A로부터 숨겨져(캡슐화 되어) 있다.
이 과정에서 위임이 사용되는데, 중개자 클래스가 단순 위임만 한다면 나쁜 증상이 될 수 있다.

감사합니다.