Law of Demeter

2022년 3월 13일 일요일 오전 2:48

Object-Oriented Programming: An Objective Sense of Style 에서 처음 소개
어떤 객체가 다른 객체에 대해 지나치게 많이 알다보니 결합도가 높아지고 좋지 못한 설계를 야기한다는 것을 발견하고 이를 개선하고자 객체에게 자료를 숨기는 대신 함수를 공개하도록 하였는데 이것이 디미터의 법칙이다

다른 객체가 어떠한 자료를 갖고 있는지 속사정을 몰라야 한다는 것을 의미하여 이러한 이유로 Don't talk to strangers(낯선 이에게 말하지 말라) 또는 Principle of least knowledge(최소 지식 원칙) 으로도 알려져 있다

직관적으로 이해하는 예시로는 여러개의 .를 사용하지 말라는 법칙으로 많이 알려져 있으며 이 법칙을 준수함으로 캡슐화를 높혀 결합도를 낮추고 응집도를 높일 수 있다

- 객체들의 협력 경로를 제한하는 것으로 .을 통한 결합관계를 없애라는 의미로서 단순 생각하면 된다

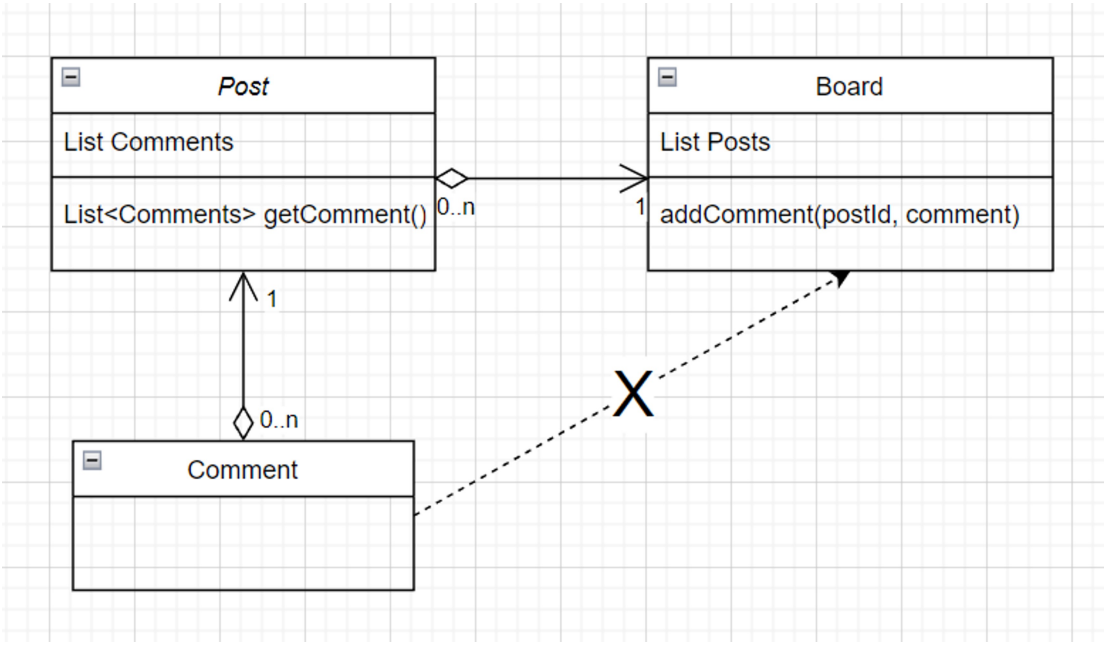
객체 지향 프로그래밍에서 중요한 것은 객체가 어떤 데이터를 가지고 있는가? 가 아니라 객체가 어떤 메시지를 주고 받는가? 이기 때문에 객체간 대화를 제대로 하지 못하면 객체 지향 프로그래밍을 제대로 하지 못하고 있다는 것

대화라면 무엇일까?
객체 한개로 프로그래밍을 모두 다 하려고 하지 않는다 즉 객체는 다른 객체와 협력이 필요하고 이를 연결 하기 위해 분명히 틀림없이 다른 객체와 연결될 부분이 생긴다 여기서 결합도를 낮추려는 노력이 필요하다. 즉 사람처럼 대화에도 요령이 필요하다는 것이다

Don't Talk to Starangers

- 하나의 객체에서 다른 낯선 객체에 메시지를 보내는 설계를 피하라는 것이다
- 객체는 내부적으로 보유하고 있거나 메시지를 통해 확보한 정보만 가지고 의사결정을 내려야 하고 다른 객체를 탐색해 뭔가를 일어나게 해서는 안된다
- 즉 최소 지식의 원칙으로 객체가 알아야 하는 다른 객체를 최소한으로 유지하는 코딩을 하라는 것이다

아래와 같은 구조가 되면 안된다



```
public class Post {
    private final List<Comment> comments;

    public Post(List<Comment> comments) {
        this.comments = comments;
    }

    public List<Comment> getComments() {
        return comments;
    }
}
```

```
public class Board {
    private final List<Post> posts;

    public Board(List<Post> posts) {
        this.posts = posts;
    }

    public void addComment(int postId, String content) {
        posts.get(postId).getComments().add(new Comment(content));
    }
    ...
}
```

posts.get(postId).getComments().add(new Comment(content));

왜 Board 에서 멀리 떨어져 있는 Comment를 추가해야 하는가
왜 낮선 객체에 메시지를 보내는 설계를 피해야 할까?
왜 디미터의 법칙을 위반하면 생기는 문제가 뭐길래 코드로 기능을 만들 줄 알면 되는데 왜 이런걸 고려해야 하는가

일단 Post 객체의 인스턴스 변수인 List<Comment> comments 에서 Comment 로 수정이 되면 getComment() 의 반환이 달리지 때문에 Board 의 addComment() 도 깨지게 된다

```
public class Post {
    private final Comments comments;

    public Post(Comments comments) {
        this.comments = comments;
    }

    public Comments getComments() {
        return comments;
    }
}
```

에러의 파도타기와 같다 즉 Board 의 addComment 매서드에서는 Post 객체도 알고 Comment 객체도 알고 알아도 너무 많이 알고 있어서 Board 객체는 Post 객체의 변화에도 영향을 받고 Comment 객체의 변화에도 영향을 받게 된다

이것 하나로 전체를 얘기할 순 없다 다만 경험상 저런 코드와 개발자들이 프로젝트 내에 즐비하다면 결국 나중에 하나의 변화에 모든 부분에 악영향을 끼

칠 수 있다 즉 벼룩잡으려나 초가 삼간 태운다는 말이 코드의 변화에서 나오게 된다
객체간 결합도가 높아지고 객체 구조의 변화에 쉽게 무너진다면 변화에 유연하게 대응하지 못하게 된다는 것이다

프로그래머 수학적으로 생각하라(즉 개념을 공식화 규칙화를 해보자)

- 디미터의 법칙은 노출 범위를 제한하기 위해 객체의 모든 매서드는 다음에 해당 되는 매서드만 호출해야 한다
  - 객체 자신의 매서드들
  - 메서드의 파라미터로 넘어온 객체들의 매서드들
  - 메서드 내부에서 생성, 초기화 된 객체의 매서드들
  - 인스턴스 변수로 가지고 있는 객체가 소유한 매서드들

```
class Demeter {
    private Member member;

    public myMethod(OtherObject other) {
        // ...
    }

    public okLawOfDemeter(Paramemter param) {
        myMethod();     // 1. 객체 자신의 메서드
        param.paramMethod();    // 2. 메서드의 파라미터로 넘어온 객체들의 메서드
        Local local = new Local();
        local.localMethod();    // 3. 메서드 내부에서 생성, 초기화된 객체의 메서드
        member.memberMethod();   // 4. 인스턴스 변수로 가지고 있는 객체가 소유한 메서드
    }
}
```

주의 사항

1. 자료구조라면 디미터 법칙을 거론할 필요가 없다 (하지만 이렇게 생각하는 이분법이 어려움)
   a. 객채라면 내부 구조를 숨겨야 하므로 디미터의 법칙을 지켜야 하나 자료구조는 내부 구조를 노출해야 하므로 디미터의 법칙이 적용되지 않는다
   b. 자료구조와 객체의 차이점
      i. 자료구조를 사용하는 절차적인 코드는 기존 자료구조를 변경하지 않으면서 새 함수를 추가하기 쉽다
      ii. 객체지향 코드는 기존 함수를 변경하지 않으면서 새 클래스를 추가하기 쉽다
   c. 이런 차이라면
      i. 새로운 함수가 아니라 새로운 자료 타입이 필요한 경우는 클래스와 객체지향 기법으로 결합도가 낮은 상황에 추가가 쉽다
      ii. 새로운 자료타입이 아니라 새로운 함수 즉 행위(매서드) 추가가 되는 경우 모든 객체에 변경이 필요할때 적절하게(??) 의존성을 끊어주거나 추상 매소드로서 뼈대와 본체를 구분 할 수 있는 식견이 필요해 보인다
      iii. 일단 상속을 쓰려면 이런 함수의 변경에 대해 객체지향이 취약하다는 부분을 꼭 인지하고 사용해라 인지는 경험과 스스로 고민하는 부분에서 통찰력을 얻으려는 노력이 필요하다

- 자료구조 형태의 예시
  - 여기에 둘레 길이를 구하는 함수를 추가한다면 어디만 수정하면 되는가?
    - 도형 클래스에는 영향없이 Geometry 쪽만 수정 하면 됨
    - 여기에 새로운 자료구조(도형) 이 추가되면 Geometry 쪽 변경이 나중에 매우 부담 스러울 수 있다
    - 그런데 이렇게 할 바에는 객체지향적인 디자인 패턴(전략, 추상클래스/매서드, 골격 구조를 좀 더 고민하는 게 나을 듯)
    - 다시한번 말하는데 경우에 따라 애매한 경우가 생기므로 이분법적으로 생각하면 안된다
      - 참고 : https://johngrib.github.io/wiki/law-of-demeter/
      - 데이터 중심으로 생각하면 아래가 맞아보이나 우리는 기능중심으로 생각하고 기능이 복잡한 경우가 참 많다

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
```

```
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuchShapeException {
        if (shape instanceof Square) {
            Square s = (Square) shape;
            return s.side * s.side;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.height * r.width;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        } else {
            throw new NoSuchShapeException();
        }
    }
}
```

- 객체지향 형태의 예시(단 상속 구조)
    - 새로운 도형이 추가되어도 기존 것에 영향없이 내가 원하는 것만 구현해서 사용하면 된다. 즉 결합도가 상당히 낮다
    - 다만 새로운 함수를 추가하게되면 Shape 에 어떤 함수가 추가될 때 다른 모든 도형에 영향을 주게 된다
    - 매소드 추가에 대해 의존성이 커지는 경우가 생기는데 이를 해결할 방법이 없는 것은 아니다
        - 디자인 패턴
        - Shape 의 결합도를 끊고 다른 방식으로 도형을 생성
            - Factory 에 넣고 내가 원하는 형태의 도형을 입력받아서 생성할 수도 있다
                - Generate / Run 등등 이런 기법을 이용해 본다
            - 전략적인 부분으로 보고 새로운 매소드의 추가시에는 해당 클래스에만 적용이 되는지 아니면 모두에 적용이 되야 하는지를 구분

```
public class Square implements Shape {
    private Point topLeft;
    private double side;

    public override double area() {
        return side * side;
    }
}

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

    public override double area() {
```

```
        return height * width;
    }

}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public override double area() {
        return PI * radius * radius;
    }
}
```

2. 하나의 .(도트)를 강제하는 규칙이 아니다

```
IntStream.of(1,15,3,20).filter(x  ->x  >10).count();
```

이 코드는 디미터의 법칙을 위한반 코드가 아니다
즉 디미터의 법칙은 결합도와 관련되어 있는 법칙이고 객체지향에서 매소드 노출에 대한 캡슐화를 제대로 하지 못한 경우에 문제가 된다
결합도가 문제되는 경우가 객체의 내부 구조가 외부에 공개되어 사용될 때 의존성이라는 것이 강제로 생기기 때문이다

위 코드가 문제되지 않는 이유는 IntStream 의 내부 구조가 노출되지 않았고 IntStream을 다른 형태로 변환할 뿐 객체를 둘러싸고 있는 캡슐은 유지한다

한 줄에 하나 이상의 점을 찍는 모든 케이스가 아니라는 것을 꼭 명심하고 (자료구조인 경우도 고려) 객체 내부 구현에 대한 정보가 외부로 노출 될 때 이걸 사용하는 녀석이 생겼을 때는 미리 예측하고 코딩해라
코딩이 어려운건 코드를 짜는게 어려운게 아니라 생각없이 짜는게 본인 스스로를 어렵게 하는 것이다
다른 사람이 내 코드를 사용한다는 것을 생각하고 다른 사람을 위해 배려하자
다른 사람이 사용할 일이 없는 완전 1인 66개발자여도 유지보수와 확장성을 위해 고민하자

From Clean Code

모듈은 자신이 조작하는 객체의 속사성을 몰라야 한다는 법칙
객체는 자료를 숨기고 함수를 공개하는 한편 (은닉, 캡슐) 객체가 조회 함수로 내부 구조를 마음껏 공개하면 안된다는 의미이다
생각해봐라 우리는 숨기는 것을 분명 배웠는데 왜 항상 Public 매소드로 내부를 항상 공개한다
 - 즉 롬복이 쓰레기 인가도 싶은데 생각없이 쓰는 경우가 잘못된걸로 생각하면 롬복에 책임을 100% 세우기는 어려울 수 있다

내부 구조를 숨기지 않고 노출하고 있었다면 한번 낯선 객체가 여기에 접근하게 가만히 두지 않았는지 반성해볼 만한 포인트이다
디미터의 법칙으로 코딩하려면 친구랑만 놀 수 있게 의존성이 결합되어야 한다

짧은 설명
- 일반적으로 한 모듈은 주변 모듈을 모를수록 좋다. A가 B를 사용하고 B가 C를 사용한다 하더라도 A가 C를 알아야 할 필요가 없다는 것이다

From Object

협력 경로를 제한하는 것이 포인트로 객체 서로 간 내부 구조에 강하게 결합되지 않게 멀리 있는 객체는 접근하지 말라는 것이다

객체의 내부 구조를 묻지 말고 무언가를 시키려는 관점이 필요
디미터의 법칙은 객체가 자기 자신을 책임지는 자율적인 존재야 한다는 사실을 강조한다
정보를 처리하는 데 필요한 책임을 정보를 알고 있는 객체에게 할당하기 때문에 응집도가 높아진다
하지만
무비판적인 수용으로 즉 생각없는 코딩에서는 퍼블릭 인터페이스에 관점으로는 반대로 응집도가 낮아진다

이론이 실전에 안맞는 경우가 생길 수 있는데 뭔가 정답을 내리려 하지 말고 의미로서 생각하고 적용해라 디미터의 법칙은 객체의 내부 구조를 묻는 매시지가 아니라 수신자에게 무언가를 시키는 매시지가 더 좋은 매시지라고 속삭인다

대가들도 설명의 주입에서 한계를 느끼게 하는 대목
정규화과 역정규화 간 차이와 비슷한 부분으로 봐도 될 것 같다 과하면 모자르니만 못한 부분으로 원칙에 사로잡혀서 예외가 필요한 경우를 놓치지 말자

속삭이는 코드를 짜는 것을 이해 못할 개발자가 100% 비중에서 상대적인 많은 수를 차지 한다면
결국 퍼블릭으로 Getter를 제공할 경우는 분명 생기는데 이것에 대해 반감을 가지기 보다는 꼭 필요할가? 비판적 사고를 해보라는 것으로 보이고
내부구조를 알거나 모르거나 그걸 가져오는데에 급급하게 조급하게 하지 말고 속삭이듯이(?)
수신자가 이걸 알고 하는데 무리가 없는지 수신자가 감당하기에 너무 많은 정보이거나 본인의 책임을 넘어서는 부분이 있는것은 아닌지? 코드레벨이 아닌 객체 스스로로서 이 일을 하는것이 문제가 되지 않는지 한번 물어보라는 것으로 들린다
다만 원천적인 설계 부터 참여해서 하는 개발자나 팀이 아닌 경우 유지보수성 업무가 정말 많은데 처음에 한 사람의 설계 철학이 느껴질 수 있으려면 사람들간 개발자간 이런 류의 대화화 협의가 꾸준히 되어야 하며 때론 다툼과 의견의 차이가 생길 수 있는데 적절한 대처나 문화가 필요할 것이다

Getter 로(롬복) 으로 디미터의 법칙이 위한되는 사례 (Setter는 없었다고 가정하자)
- Message Chain 이라는 악취

```
object.getChild().getContent().getItem().getTitle();
```

줄줄이 이어진 모습이 기차와 닮아서 열차 전복, 기차 충돌이라고도 한다 (접근자 매서드가 기차처럼 연이어 이어진다 - 롬복쓰면 당연할걸)
객체구조가 영원하지 않고 변경이 될 수 있다는 부분에서 위의 구조는 말이 안된다고 봐야 한다
차라리 4번 호출해라 한번에 끝내려 하지 말고

- Getter 라는 악취

```
/* 디미터의 법칙을 어긴 예 */
public violateLawOfDemeter(Parameмter param) {
    C c = param.getC();
    c.method();    // 인자로 받은 객체에서의 호출.
    param.getC().method();      // 위와 같음.
}
```

그래서 어쩌라고 XXX 야
정 어려우면 이걸 처음 봤을 때는 아래것만 생각하고 넘어간다

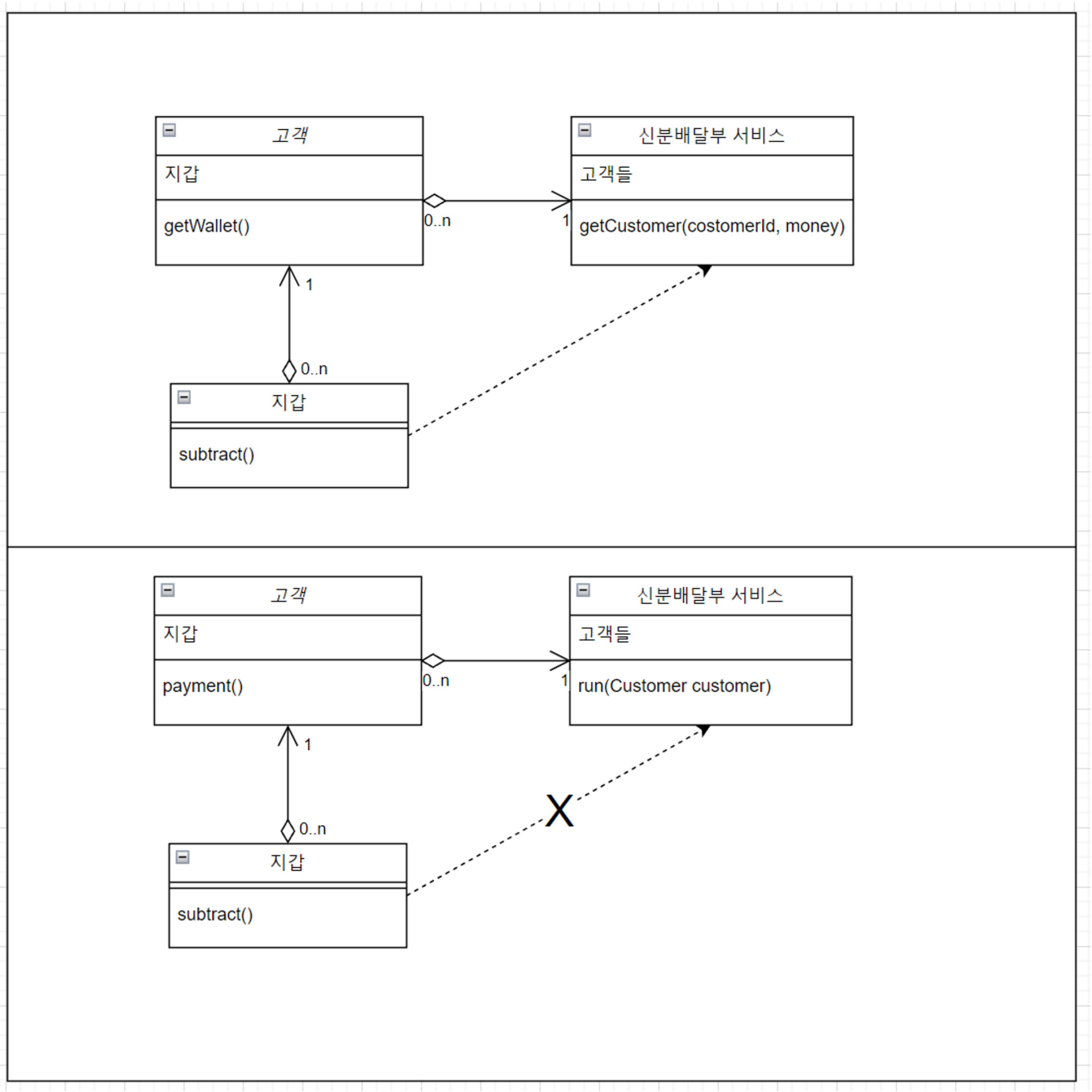
```
class A {
    private B b;
    public setA(B b) {
        b = b;
    }
    public myMethod(OtherObject other) {
        // ...
    }
    /* 디미터의 법칙을 잘 따른 예 */
    public okLawOfDemeter(Parameмter param) {
        myMethod();     // 자신의 메소드
        b.method();   // 자신의 멤버의 메소드
        Local local = new Local();
        local.method();    // 직접 생성한 객체의 메소드
        param.method();    // 메소드의 인자로 넘어온 메소드
    }
    /* 디미터의 법칙을 어긴 예 */
    public violateLawOfDemeter(Parameмter param) {
        C c = param.getC();
        c.method();    // 인자로 받은 객체에서의 호출.
        param.getC().method();      // 위와 같음.
    }
}
```

출처: https://coding-start.tistory.com/264 [코딩스타트]

출처: <https://coding-start.tistory.com/264>

- 코드 예시로 보기 괜찮은 내용

https://dev-monkey-dugi.tistory.com/86



내 결론

구현은 절차적으로 보면 한번에 끝내는게 가능하다 하지만 악취나는 코드가 아니려면

코드로 한번에 끝낸 다는게 불가능하다 결국 조각조각을 내야하고 그 조각을 잘 이어 붙여야 한다

https://tecoble.techcourse.co.kr/post/2020-06-02-law-of-demeter/

https://johngrib.github.io/wiki/law-of-demeter/#fn:andrew-227

https://bperhaps.tistory.com/entry/%EB%94%94%EB%AF%B8%ED%84%B0%EC%9D%98-%EB%B2%95%EC%B9%99Law-of-Demeter%EC%9D%B4%EB%9E%80-%EB%AD%98%EA%B9%8C-%EA%B7%B8%EB%A6%AC%EA%B3%A0-%EC%99%9C-%EC%A7%80%EC%BC%9C%EC%95%BC-%ED%95%A0%EA%B9%8C

https://coding-start.tistory.com/264

https://antilog.tistory.com/51

https://luran.me/428

https://dev-monkey-dugi.tistory.com/86

https://rlawls1991.tistory.com/entry/%EC%9D%BC%EA%B8%89-%EC%BB%AC%EB%A0%89%EC%85%98