객체지향 설계의 원칙과 책임

SOLID 원칙과 GRASP 패턴

2020. 06. 28

객체지향 프로그래밍이란?

- 프로그래밍 패러다임 중 하나로 프로그래밍에서 필요한 데이터를 추상화시켜 ‘상태’와 ‘행위’를 가진 객체를 만들고 그 객체들간의 유기적인 상호작용을 통해 로직을 구성하는 프로그래밍 방법
- 코드는 유연하고, 확장 할 수 있고, 유지보수가 용이하고, 재사용 할 수 있어야 한다
- 이러한 코드를 구현하고 적용하기 위해 OOP라는 방법론이 제안되었고, OOP 방식을 잘 준수하기 위한 원칙으로 SOLID 원칙이 제안되었다
- 현실 세계를 설계한다고 하지만 쉽지 않다
- 현실을 동일하게 설계 할 수 없다 (오히려 새로운 세계를 창조하는 것)

SOLID 원칙이란?

: 5가지 원칙을 의미, 과거의 많은 사람들이 고민과 변화를 거듭해 정리된 원칙

- SRP (Single Responsibility Principle): 단일 책임 원칙
- OCP (Open Closed Principle): 개방 폐쇄의 원칙
- LSP (Liskov Substitution Principle): 리스코프 치환 원칙
- ISP (Interface Segregation Principle): 인터페이스 분리 원칙
- DIP (Dependency Inversion Principle): 의존성 역전 원칙

Single Responsibility Principle (SRP, 단일 책임 원칙)

- 클래스나 함수가 한가지 책임만을 가져야 한다는 이야기
- 책임이란 변경의 근원, 변경의 원인이 같다면 같은 책임
- 만약 하나의 클래스가 여러가지 책임을 가지고 있다면, 그 클래스는 수시로 변경될 것이며, 그 클래스를 사용하고 있는 다른 클래스들도 영향을 받게 되기 때문에 좋은 설계라 할 수 없음
- 같은 책임을 가진 것끼리의 그룹화를 잘 해야한다

```java
class User {
    private String id;
    private String password;
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    // other get/set ...

    void save() {
        // 대충 사용자정보를 DB에 저장하는 로직
    }
}
```

- 속성 관리 책임

- 데이터 베이스 관리 책임

```java
class User {
    private String id;
    private String password;
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    // other get/set ...
}

class UserDB {

    User getUser(String name) {
        // 사용자정보를 DB에서 가져오는 로직...
        return user;
    }

    void save(User user) {
        // 대충 사용자정보를 DB에 저장하는 로직
    }
}
```

- User 클래스는 속성만을 관리

- UserDB 클래스는 데이터 베이스 관리

Single Responsibility Principle (SRP, 단일 책임 원칙)

- 클래스나 함수가 한가지 책임만을 가져야 한다는 이야기
- 책임이란 변경의 근원, 변경의 원인이 같다면 같은 책임
- 만약 하나의 클래스가 여러가지 책임을 가지고 있다면, 그 클래스는 수시로 변경될 것이며, 그 클래스를 사용하고 있는 다른 클래스들도 영향을 받게 되기 때문에 좋은 설계라 할 수 없음
- 같은 책임을 가진 것끼리의 그룹화를 잘 해야한다
- 하나의 수정사항으로 여러 클래스가 수정되고 있다면, 하나의 책임을 가진 클래스가 나누어져 있는건 아닌지 생각해볼 여지가 있다

Open Closed Principle (OCP, 개방 폐쇄의 원칙)

- 확장에는 열려 있고, 변경에는 닫혀 있다는 이야기
- 기존의 코드를 변경하지 않고(Closed), 기능을 수정하거나 추가할 수 있도록(Open) 해야 한다
- 디자인 패턴들의 기본 원칙이 되는 원칙
- 다형성과 인터페이스를 통해 변하는 부분과 변하지 않는 부분을 분리
- 코드를 유연하고 확장성 있게 만들어 주는 것
- 부모 클래스에서 자식 클래스를 만들 때, 자식 클래스에서 기능을 추가/수정 할 수 있지만, 자식 클래스를 위해 부모 클래스가 수정될 필요는 없다

OCP와 연관있는 디자인 패턴

- Template Method Pattern (템플릿 메소드 패턴)
  - 추상클래스 사용
- Strategy Pattern (전략 패턴)
  - 인터페이스 사용

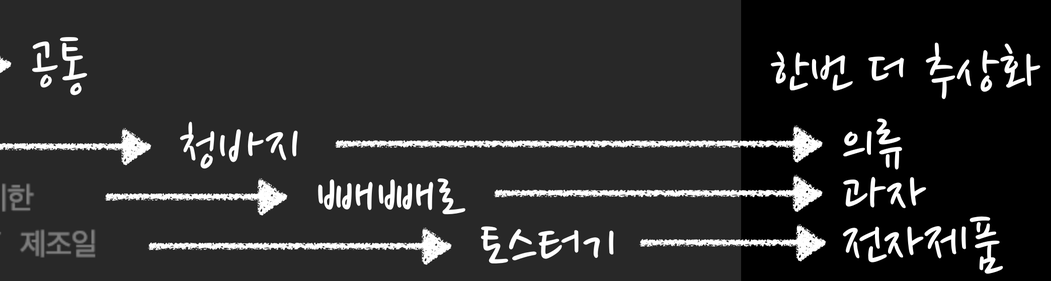
```java
// 상품 클래스 - 청바지, 빼빼로, 토스터기를 판다고 가정
class Product {
    String id;
    String name;
    int price;
    int size; // 청바지 사이즈
    String expireDate; // 유통기한
    String manufactureDate; // 제조일

    // 청바지 값을 넣기 위한 생성자
    public Product(String id, String name, int price, int size) {}

    // 빼빼로, 토스터기 값을 넣기 위한 생성자
    public Product(String id, String name, int price, String date) {
        this.id = id;
        this.name = name;
        this.price = price;

        if ("빼빼로".equals(name)) {
            this.expireDate = date;
        }

        if ("토스터기".equals(name)) {
            this.manufactureDate = date;
        }
    }
}
```




```java
List<Product> products = new ArrayList<>();

void readyForProduct() {
    Product pepero = new Product( id: "001",  name: "빼빼로",  price: 1300,  date: "2020-07-10");
    Product jean = new Product( id: "002",  name: "청바지",  price: 50000,  size: 30);
    Product toaster = new Product( id: "003",  name: "토스터기",  price: 100000,  date: "2019-12-20");

    products.add(pepero);
    products.add(jean);
    products.add(toaster);
}
```

```java
// 상품에서 공통부분을 뽑자
abstract class Product {
    String id;
    String name;
    int price;

    public Product(String id, String name, int price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
```

```java
abstract class Snack extends Product {
    String expireDate; // 유통기한

    public Snack(String id, String name, int price) {
        super(id, name, price);
    }

    public Snack(String id, String name, int price, String expireDate) {
        this(id, name, price);
        this.expireDate = expireDate;
    }
}
```

```java
abstract class Electronic extends Product {
    String manufactureDate; // 제조일

    public Electronic(String id, String name, int price) {
        super(id, name, price);
    }

    public Electronic(String id, String name, int price, String manufactureDate) {
        this(id, name, price);
        this.manufactureDate = manufactureDate;
    }
}

abstract class Clothes extends Product {
    int size;

    public Clothes(String id, String name, int price) {
        super(id, name, price);
    }

    public Clothes(String id, String name, int price, int size) {
        this(id, name, price);
        this.size = size;
    }
}
```

→ 클래스로 상품을 구체화하자

```java
class Pepero extends Snack {
    static final String NAME = "빼빼로";
    static final int PRICE = 1300;

    public Pepero(String id, String expireDate) {
        super(id, NAME, PRICE, expireDate);
    }
}

class Toaster extends Electronic {
    static final String NAME = "토스터기";
    static final int PRICE = 100000;

    public Toaster(String id, String manufactureDate) {
        super(id, NAME, PRICE, manufactureDate);
    }
}

class Jean extends Clothes {
    private static final String NAME = "청바지";

    public Jean(String id, int price, int size) {
        super(id, NAME, price, size);
    }
}
```

만약, 또 다른 과자가 추가된다면?

새로운 클래스로 구체화 시키면 되므로
기존의 코드에 영향없이 추가가 가능
-> OCP 만족

```java
// 상품을 준비합니다
void readyForProduct() {
    Pepero pepero1 = new Pepero( id: "S001",  expireDate: "2020-06-30");
    Pepero pepero2 = new Pepero( id: "S002",  expireDate: "2020-07-10");
    Pepero pepero3 = new Pepero( id: "S002",  expireDate: "2020-07-10");
    products.add(pepero1);
    products.add(pepero2);
    products.add(pepero3);

    Toaster toaster1 = new Toaster( id: "T001",  manufactureDate: "2015-06-21");
    Toaster toaster2 = new Toaster( id: "T002",  manufactureDate: "2020-03-01");
    products.add(toaster1);
    products.add(toaster2);

    Jean jean1 = new Jean( id: "J001",  price: 50000,  size: 27);
    Jean jean2 = new Jean( id: "J002",  price: 100000,  size: 27);
    products.add(jean1);
    products.add(jean2);
}
```

Liskov Substitution Principle (LSP, 리스코프 치환 원칙)

- 리스코프는 만든이의 이름
- 상속에 있어서 가장 중요한 기본 원칙 제시
- 자식 클래스는 부모 클래스로 치환될 수 있다는 원칙
- 업캐스팅을 해도 아무런 문제가 되지 않아야 한다는 것
- 부모 클래스와 자식 클래스의 ‘행위’가 일관성이 있어야 한다는 의미
- OCP를 가능하게 해주는 원칙 (OCP를 위반하지 않도록 인도하는 원칙)

잘 모르겠고 추가로 알아보자면.. - LSP

부모와 다른일을 하게 될 수 있다

- LSP를 만족시키는 가장 간단한 방법은 재정의(Override)하지 않는 것 (확장만 수행)
- 현실세계의 IS-A 관계가 항상 성립하진 않는다 (ex. Square IS-A Rectangle)
- instanceof/downcasting을 사용하는 것은 전형적인 LSP 위반의 징조

하위 클래스만의 고유한 기능이 있다는 뜻이므로 하위 클래스의 확장이 일어날 경우,
영향을 받는 부분이 될 수 있다

```java
// 사각형
class Rectangle {
    private int width;
    private int height;

    public Rectangle() {}

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() { return width; }
    public void setWidth(int width) { this.width = width; }

    public int getHeight() { return height; }
    public void setHeight(int height) { this.height = height; }

    // 면적 확인
    public int getArea() {
        return width * height;
    }
}
```

```java
// 정사각형
class Square extends Rectangle {
    public Square() {}

    // 정사각형은 네 변의 길이가 모두 같다
    public Square(int side) {
        super(side, side);
    }

    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

```
Rectangle 객체 테스트
 -> 사각형 값 테스트 성공. 10x2 = 20
Square 객체 테스트
 -> 사각형 값 테스트 실패. 10x2 = 10 이어야 한다. 실패한 결과값: 4
```

→ LSP 위반! 부모의 기능으로 치환 시 문제가 발생한다

```java
static void test_RectableArea(Rectangle rectangle) {
    rectangle.setWidth(10);
    rectangle.setHeight(2);
    int area = rectangle.getArea();
    if (area == 20) {
        System.out.println(" -> 사각형 값 테스트 성공. 10x2 = " + area);
    } else {
        System.out.println(" -> 사각형 값 테스트 실패. 10x2 = 10 이어야 한다. 실패한 결과값: " + area);
    }
}

public static void main(String[] args) {
    System.out.println("Rectangle 객체 테스트");
    Rectangle rectangle = new Rectangle( width: 2,  height: 5);
    test_RectableArea(rectangle);

    System.out.println("Square 객체 테스트");
    Rectangle square = new Square();
    square.setWidth(5);
    test_RectableArea(square);
}
```

Square가 Rectangle 의 자식이 맞을 지 고민필요
- 상속을 제거하던지
- 기능을 제대로 하지 못하는 getArea() 를 자식 클래스로 이동

따라서! 자식 클래스는 부모 클래스의 역할을 충실히 하면서 확장해나가야 한다는 것

Interface Segregation Principle (ISP, 인터페이스 분리 원칙)

- 자신이 사용하지 않는 인터페이스는 구현하지 말아야 한다는 원칙
- 하나의 일반적인 인터페이스 보다는, 여러개의 구체적인 인터페이스가 낫다
- SRP와 밀접하게 연관 → 한 기능에 변경이 발생하면, 기능을 사용하는 클라이언트들에도 영향을 미치므로
- ‘SRP를 잘 준수하기 위해서 ISP를 적용할 수 있다’라고 이야기 할 수 있다
- 사용하는 기능만 제공하도록 인터페이스를 분리함으로써 한 기능에 대한 변경의 여파를 최소화
- 클라이언트의 입장에서 인터페이스를 분리

ISP 를 위반하는 복합기 인터페이스

```
public interface multifunction {
    void copy();
    void fax(Address from, Address to);
    void print();
}
```

```
public class copyMachine implements multifunction {

    @Override
    public void copy() {
        System.out.println("###복사###")
    }

    @Override
    public void fax() {

    }

    @Override
    public void print() {

    }
}
```

→ 수정 필요! (사용하지 않는 인터페이스가 변경되어도 함께 수정이 일어난다)

ISP 를 만족하는 복합기 인터페이스

```java
public interface Print {
    void print();
}

public interface Copy {
    void copy();
}

public interface Fax {
    void fax();
}

public interface multifunction extends Print, Copy, Fax{
}
```

```java
public class copyMachine implements Copy {

    @Override
    public void copy() {
        System.out.println("###복사###")
    }
```

→ 다른 인터페이스의 수정에 영향 받지 않는다

Dependency Inversion Principle (DIP, 의존성 역전 원칙)

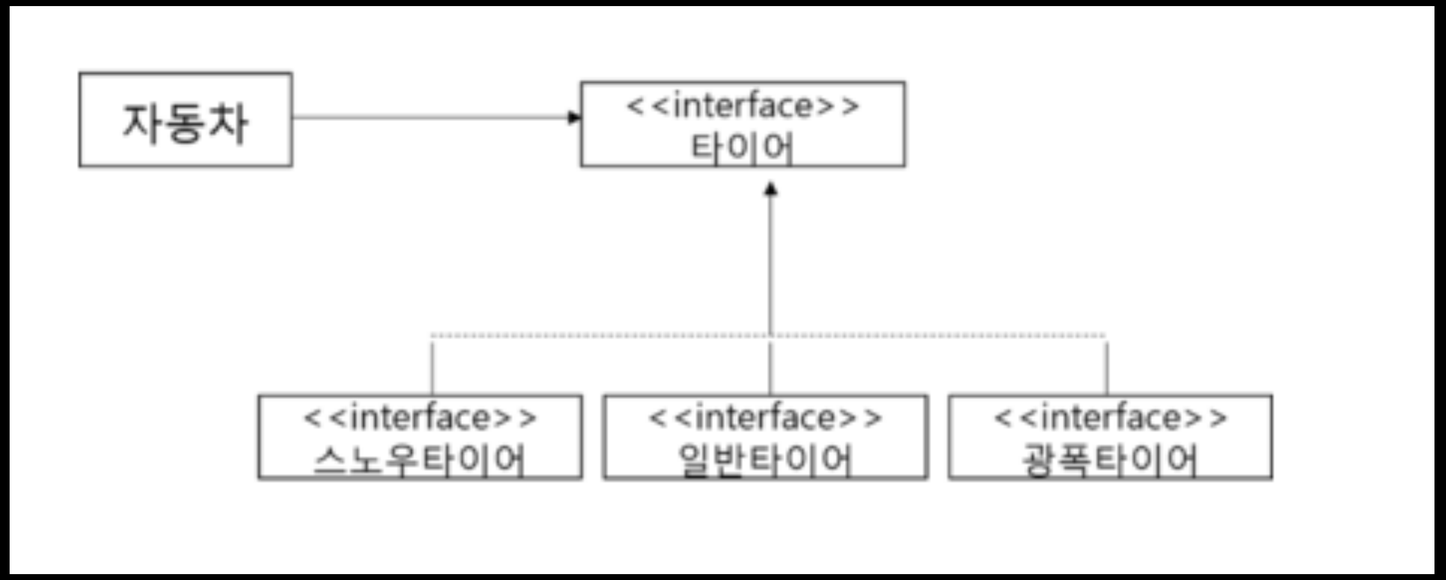
- 의존은 추상과 이루어져야 한다는 원칙
- 상위 레벨의 모듈은 하위 레벨들에 의존해선 안되며, 모든 것들은 추상에 의존해야 한다
- 의존 관계를 맺을 때, 변화하기 쉬운 것보다 변화하기 어려운 것에 의존해야 한다
- 객체지향 관점에서 변하기 어려운 추상적인 것들을 표현하는 수단은 인터페이스와 추상클래스를 사용하는것 (또는 추상화한 클래스)

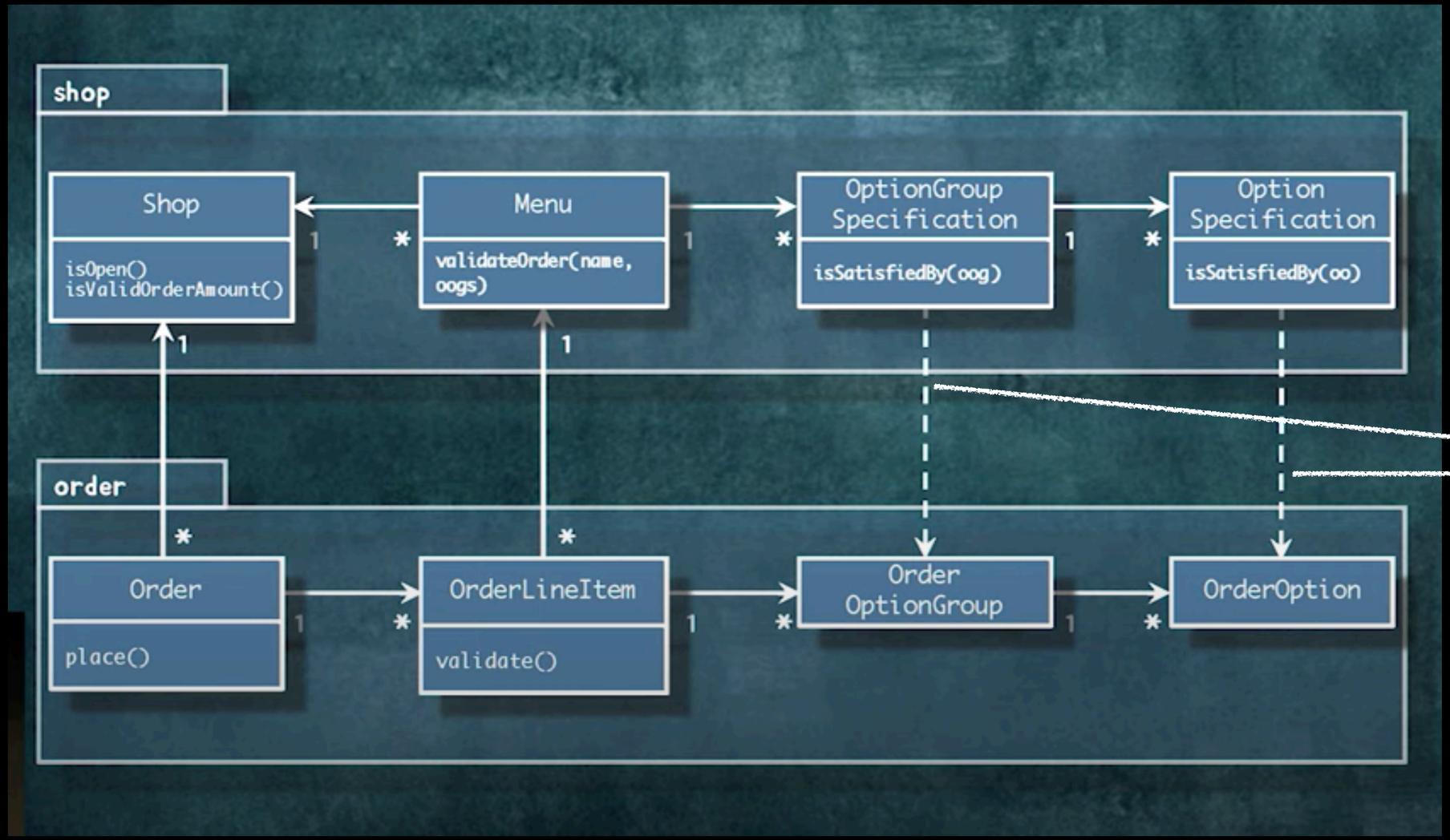
엄청 간단하게 보자면..

```
List<Product> products = new ArrayList<>();
```

→ 이것도 하나의 DIP, products 라는 ArrayList<Product> 타입의 참조변수를 ArrayList 의 추상타입에 의존

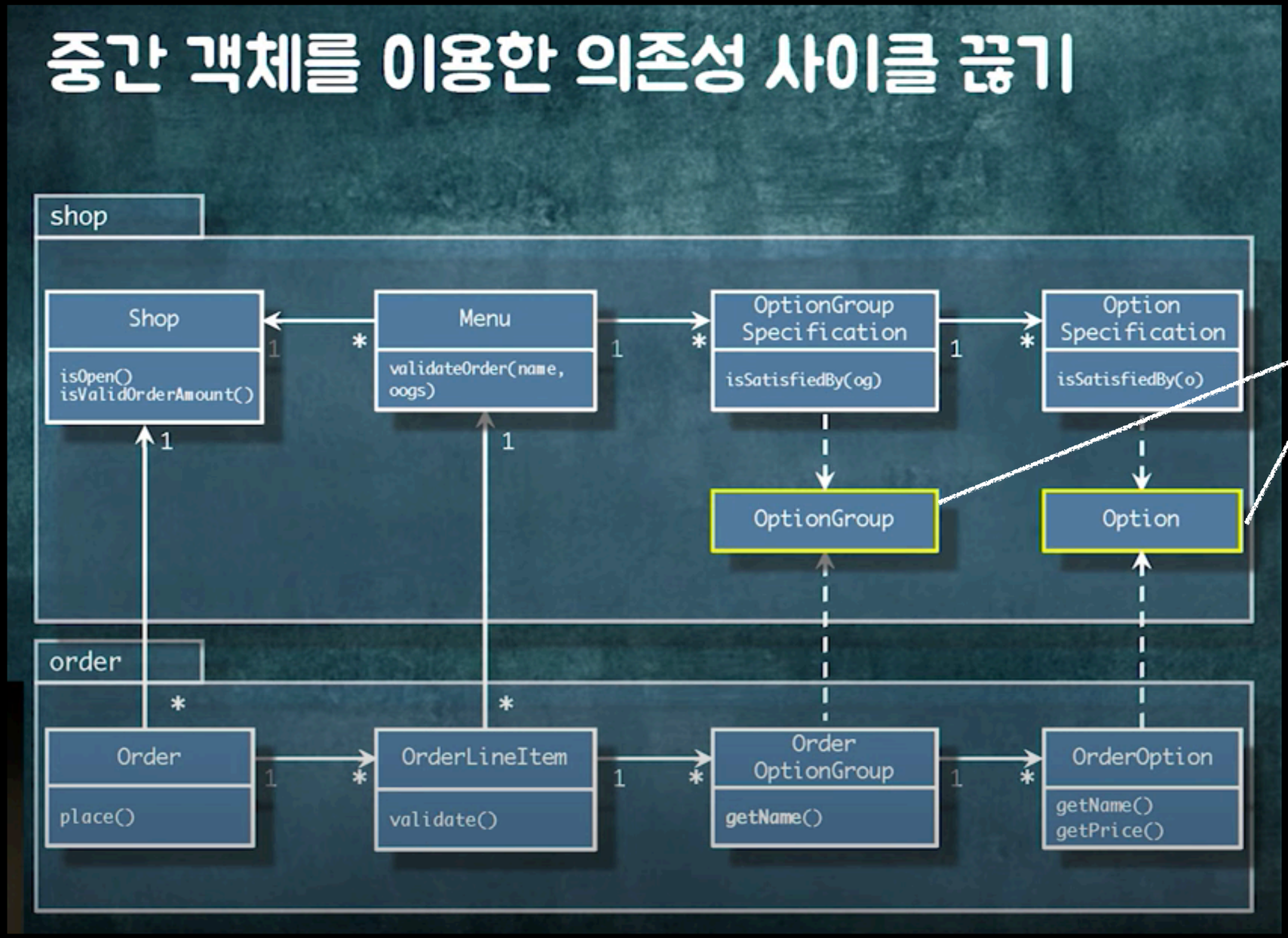
이런 것도..





[우아한 테크 세미나] 우아한객체지향 영상 참고
(https://www.youtube.com/watch?v=dJ5C4qRqAgA)

중간 객체를 이용한 의존성 사이클 끊기



Shop 패키지 내에 추상클래스를 두고 오더가 해당 클래스를 참조하도록

여기서의 추상 클래스는
abstract class 가 아닌
일반 클래스인데 조금 더 추상적인 클래스
(= 일반 클래스보다 잘 변하지 않는 클래스)

패턴이란?

: 일반적으로 발생하는 문제점들에 대한 일반화되고 재사용 가능한 솔루션

- 일반화된 해결책을 제공
- But, 모든 디자인 문제를 해결해주지 않는다
- GRASP 패턴은 개념적인 이야기

GRASP 패턴이란?

General Responsibility Assignment Software Patterns

- 책임에 대한 이야기
- 책임을 부여하는 원칙을 말하는 패턴
- 구체적인 구조는 없지만, 철학을 배울 수 있다
- 총 9가지 패턴

객체 디자인에서 가장 기본이 되는 것 중의 하나(원칙은 아닐지라도)는 책임을 어디에 둘지를 결정하는 것이다.

나는 십년 이상 객체를 가지고 일했지만 처음 시작할 때는 여전히 적당한 위치를 찾지 못한다.

늘 이런 점이 나를 괴롭혔지만, 이제는 이런 경우에 리팩토링을 사용하면 된다는 것을 알게 되었다.

- “리팩토링”, 마틴 파울러

1. Creator (소유권한)

문제정의:
- 누가 객체 A 를 생성하고 소유하는가?
- 어떤 클래스의 새로운 인스턴스를 생성하는 책임을 누가 가져야 할까?

해결방안:
- 다음중 적어도 하나 이상이라면 A 객체를 생성하는 책임을 객체 B 에 할당한다.

1) B 객체가 A 객체를 포함할 때
2) B 객체가 A 객체의 정보를 기록할 때
3) A 객체가 B객체의 일부일 때
4) B 객체가 A 객체를 긴밀하게 사용할 때
5) B 객체가 A 객체의 생성에 필요한 정보를 가지고 있을 때

2. Information Expert (정보 담당자)

문제정의:
- 객체에 책임을 할당하는 일반적인 원리는 무엇인가?

해결방안:
- 책임을 수행하기 위해 필요한 정보를 가장 많이 갖고 있는 객체에게 그 책임을 할당한다.

3. Low Coupling (느슨한 연결)

문제정의:
- 어떻게 변화에 대한 충격을 줄일 수 있을까?
- 어떻게 의존성을 줄여 변경에 유연하고, 재사용성을 높일 수 있을까?

해결방안:
- 불필요한 연결을 줄이도록 책임을 할당한다.

추가설명:
- 연결(Coupling)은 하나의 요소가 다른 요소와 얼마나 관련이 있는지를 나타내는 지표다. 느슨한 연결은 객체들끼리 서로 독립적이면서 분리되어 있는 것을 의미한다. 분리되어 있는 객체들은 다른 객체가 변하더라도 걱정할 필요가 없고, 무언가 깨지지 않는 것을 의미한다.

4. High Cohesion (높은 응집도)

문제정의:
- 객체 자체에 집중해서 객체를 이해하기 쉽고, 관리하기 편하고, 느슨한 연결을 추구하려면 어떻게 해야할까?

해결방안:
- 응집도가 높아지도록 책임을 할당한다.

추가설명:
- Low Coupling이 만족되면 High Cohesion도 만족된다.
- 응집도는 객체가 가진 모든 책임이 얼마나 관련이 높은지 나타내는 지표다. 다시 말해서 내부 요소에 있는 부분들이 서로 얼마나 관련이 높은지를 의미한다.

5. Controller (컨트롤러)

문제정의:
- 시스템 이벤트(사용자의 요청)를 처리하는 것은 누구의 책임인가?

해결방안:
- 시스템, 서브시스템으로 들어오는 외부 요청을 처리하는 객체를 만들어 사용한다.

추가설명:
- 만약 어떤 서브시스템 안에 있는 각 객체의 기능을 직접 사용한다면, Coupling이 증가되고, 서브시스템의 어떤 객체를 수정할 경우, 외부에 주는 충격이 크게 된다. 서브 시스템을 사용하는 입장에서 보면, 이 Controller 객체만 알고 있으면 되므로 사용하기 쉽다.

6. Polymorphism (다형성)

문제정의:
 - 타입을 다른 타입으로 대체할 방법은 어떻게 가능한가?

해결방안:
 - 타입(또는 클래스)에 따라 다른 동작이나 대체 수단은, 각 타입에서 해당 동작이 다형성을 갖도록 책임을 할당한다.

추가설명:
 - 만약 객체의 종류에 따라 행동이 바뀐다면 객체의 종류를 체크하는 조건문을 사용하지 말고, 다형성을 사용하라.

7. Pure Fabrication (순수 조립)

문제정의:
 - Information Expert에 근거하여, 정보가 많은 객체에 책임을 부여했는데 High Cohesion, Low Coupling이 위반될 때 어떻게 하는가?

해결방안:
 - 인위적으로 어떤 클래스를 만들어서, 문제가되는 책임만 모아 High Cohesion을 갖는 클래스가 되도록 만든다.

추가설명:
 - Database 관련된 작업을 하는 DAO(Data Access Object) 객체를 Pure Fabrication으로 볼 수 있다. 만약 유저정보를 DB에 삽입하려하면, Information Expert에 따르면 유저정보를 가지고있는 유저 객체가 DB에 접근하는 것이 맞지만, 이는 Low Coupling, High Cohesion을 위반한다. 왜냐하면, DB에 접근하는 책임이 여러 곳에 분산되기 때문이다. 따라서 DAO라는 인위적인 객체를 만들고, DB에 접근하는 책임을 할당한다.

8. Indirection (간접 참조)

문제정의:
- 두 객체 이상에서 직접적으로 연결을 피하도록 어디에 책임을 할당해야 할까?

해결방안:
- 두 개의 서비스나 컴포넌트를 직접 연결하지 말고, 중간 매개체에게 책임을 할당한다.

추가설명:
- 간접 참조를 사용하면 느슨한 연결이 되지만, 시스템의 가독성과 분별력을 떨어트린다. 컨트롤러 코드 상에서는 command 처리를 어느 객체가 담당하는지 알 수 없다. 트레이드 오프가 있다는 것을 늘 고려해야 한다.

9. Protected Variations (변화에 대한 보호)

문제정의:
 - 불안정적이고 변화에 따라서 다른 요소에 영향을 덜 주도록, 객체나 시스템을 설계하는 방법은 무엇인가?

해결방안:
 - 불안정적인 요소나 변화할 요소를 예측하고 분류해서, 안정적인 인터페이스를 갖도록 책임을 할당한다.

결론

- 한번에 완벽한 프로그램을 만들 순 없다
- 따라서 항상 수정되게 되어있으며
- 수정과 확장에 용이하게 설계/개발하는것이 최선의 방법
- SOLID 원칙과 GRASP 패턴은 위 내용을 가능하게 해주는 개념들

끝, 원칙과 책임을 항상 생각하고 지키려 노력하자