1. 객체지향의 다형성을 추구하여 만들면 깔끔한 소스코드가 만들어진다.

2. DI (dependency injection) 객체주입 방식으로 하면 깔끔해진다.

프로그램 기법

- 절차지향(Procedure Oriented)

- 객체지향(Object Oriented)

**객체지향**

1. 객체마다 자신만의 책임(responsibility) 을 가지며 그것은 단일책임원칙을 따라야지 된다

2. 서로의 객체끼리는 의존을 해서는 안 된다.(한곳의 구현 변경이 다른 곳에 변경을 가하지 않도록)

캡슐화를 하여 객체가 내부적으로 기능을 어떻게 구현하는지 감면 이를 통해 내부의 기능 구현이 변경되더라도 그 기능을 사용하는 코드는 영향을 받지 않도록 만들어준다.

**캡슐화**

1. Tell,Don’t Ask

- 데이터를 물어보지 않고 기능을 실행해달라고말한다

2. 데미테르 법칙(Law of Demeter)

- 메서드에서 생성한 객체의 메서드만 호출

- 파라미터로 받은 객체의 메서드만 호출

- 필드로 참조하는 객체의 메서드만 호출

**상속**

1. 다형성(Polymorphism)은 한객체가 여러가지(poly) 모습(morph)를 갖는 다는 것을 의미한다 한객체가 여러 타입을 가질수 있다는 것을 뜻한다.

**상속**

1. 상속은 Is-a 일때 부모와 같다 할 때 하는것이다. Has-a 일때는 하는 것이 아니다.

**상속보단 조립**

1. 상속을 하여 처리한다면 좀더 직관적으로 처리할수 는 있겠지만 윗 부모클래스가 변경이 된다면 그 여파는 고수라니 자식 클래스들에게 넘어갈것이다. 그렇기 때문에 DI 즉 객체주입을 처리하여 조립하여 처리하여라

**설계원칙:SOLID**

- 단일 책임원칙(Singel responsibility principle; SRP)

- 개방-폐쇄 원칙(Open-close principle; OCP)

- 리스코프 치환 원칙(Liskov substitution principle; LSP)

- 인터페이스 분리 원칙(Interface segregation principle; ISP)

- 의존 역전 원칙(Dependency inversion principle; DIP)

**단일 책임원칙(Singel responsibility principle; SRP)**

클래스는 단 한 개의 책임을 가져야 한다

클래스가 여러 책임을 갖게 되면 그 클래스는 각 책임마다 변경되는 이유가 발생하기 때문에 클래스가 한 개의 이유만 변경되면서 클래스는 한 개의 책임만을 가져야한다 이런 이유로 이원칙은 다른 말로 클래스를 변경하는 이유는 단 한 개여야 한다 고 표현한다.

객체에게 책임을 할당하는 것이 객체 설계의 기본인 만큼 단일 책임 원칙은 가장 중요한 원칙중 하나이다.

예) 통신을하여 파싱후 화면에 뿌려주는 프로그램을 만든다고 하자

하나의 객체에 소켓으로 데이터 가져와서 파싱한후 화면까지 보여주는 것이 한곳에 있다면

소켓 대신 html통신으로 가져오라고 변경됐다면 전혀 변경이 필요없는 부분까지 다시 재컴파일을 해야되는 아이러니한 상황이 발생된다 즉 책임별 객체를 나눠서 DI쪽으로 처리를 한다면 얼마나 변경할것이 없을것이다.

통신을 관장하는 객체만 재컴파일하면되니깐 말이다.

**개방-폐쇄 원칙(Open-close principle; OCP)**

확장에는 열려 있어야 하고 변경에는 닫혀 있어야 한다.

다시말해 기능을 변경하거나 확장할 수 있으면서, 그 기능을 사용하는 코드는 수정되지 않는다.

instanceof 와 같은 타입 확인 연산자가 사용된다면, if else 블록이 존재한다면

해당 코드는 개방 폐쇄원칙을 지키지 않을 가능성이 높다.

**리스코프 치환 원칙(Liskov substitution principle; LSP)**

상위 타입의 객체를 하위 타입의 객체로 치환해도 상위 타입을 사용하는 프로그램은 정상적으로 동작해야한다

SuperClass를 상속하는 SubClass가 있는데 다른 메서드에서 SupperClass를 인자로 받는다면 그하위 즉 SubClass 객체가 들어가도 아무런 오류없이 프로그램이 작동되어야 한다는 것이다.

**인터페이스 분리 원칙(Interface segregation principle; ISP)**

인터페이스는 그 인터페이스를 사용하는 클라이언트를 기준으로 분리해야 한다.

다시말해 기능별 인터페이스를 다르게 만들어서 서로의 영향없이 처리해야한다는것이다.

같은 인터페이스를 공유하게된다면 그 인터페이스가 변경이된다면 그인터페이스를 구현한 모든 클래스들은 재 컴파일이 이루어져야한다는것이다.,

**의존 역전 원칙(Dependency inversion principle; DIP)**

고수준 모듈은 저수준 모듈의 구현에 의존해서 안된다 저수준 모듈이 고수준 모듈에서 정의한 추상 타입에 의존해야 한다

고수준 모듈 : 단일 기능을 제공하는 모듈

저수준 모듈 : 고수준 모듈의 기능을 구현하기 위해 필요한 하위 기능의 실제 구현으로 정의할수 있다.

**전력 패턴**

구분할것들이 늘어나면 늘어날수록 if else 문을 늘리지말고

특정 부분을 추상화하여 각 콘크리트 클래스에서 상황에 맞는 할인 계산 알고리즘을 제공한다.

하나의 인터페이스로 추상화하여 처리한다 그러면 if else 문이 생기지 않는다.

DI쪽으로 콘크리트 객체를 받으면 좀더 좋겠죵 ㅎ

**템플릿 메서드(Template Method) 패턴**

중복되는 기능을 추상메서드 쪽을 빼고 상속받을 클래스에서 특정 메서드만 가지고 처리하도록 하는 것

즉 다시말해 메서드 추출을한다는것이다.

**상위 클래스가 흐름 제어 주체**

**이렇게 상위 클래승서 실행 시점이 제어되고 기본 구현을 제공하면서 하위 클래스에서 알맞게 확장할수 있는 메서드를 훅 메서드라고 부른다.**

**상태(State)패턴**

상태에 따라 동일한 기능 요청의 처리를 다르게함.

이또한 상위 클래스가 흐름 제어 주체를 하여 아래쪽에서 특정 훅메서드만 구현하는 방식으로 처리할수도있다.

**데코레이터(Decorator)패턴**

삭송을 이용한 기능 확장 방법이 쉽긴하지만 다양한 조합의 기능 확장이 요구될 때 클래스가 불필요하게 증가하는 문제발생한다.

이런 경우에 사용할 수 있는 패턴이 데코레이터(decorator) 패턴이다. 데코레이터 패턴은 상속이 아닌 위임을 하는 방식으로 기능을 확장해 나간다

**프록시(proxy)패턴**

**어댑터(Adapter)패턴**

**옵저버(Observer)패턴**

**미디에이터(Mediator)패턴**

**파사드(Facade)패턴**

**추상팩토리(Abstract Factory)패턴**

**컴포지트(Composite)패턴**