디자인 패턴 개요

목적

SW의 재사용성, 호환성, 유지 보수성을 보장하기 위함.

특징

디자인 패턴은 특정한 구현이 아닌 아이디어이다.

무조건 적용해야 한다! 가 아니고 추후에 재사용, 호환, 유지 보수시 발생하는 문제를

예방하거나 해결하기 위해 패턴을 만들어 둔 것.

원칙 - SOLID(객체지향 설계 원칙)

Robert C. Martin은 5가지 Software design principles을 정의하고 앞글자를 따서 SOLID라고 부른다.

1. Single Responsiblilty Principle : 하나의 클래스는 하나의 역할만 해야한다.
2. Open - Close Principle : 확장(상속)에는 열려있고, 수정에는 닫혀 있어야 한다.
3. Liskov Substitution Principle : 자식이 부모의 자리에 항상 교체될 수 있어야 한다.
4. Interface Segregation Principle : 인터페이스가 잘 분리되어, 클래스가 꼭 필요한 인터페이스만 구현하도록 해야한다.
5. Dependency Inversion Property : 상위 모듈이 하위 모듈에 의존하면 안된다, 모두 추상화에 의존하며, 추상화는 세부 사항에 의존하면 안된다.

분류

1. 생성 패턴(Creational) : 객체의 생성 방식 결정

   > ex) DB커넥션을 관리하는 Instance를 하나만 만들 수 있도록 제한, 불필요한 연결을 막음

2. 구조 패턴(Structural) : 객체간의 관계를 조직

   > ex) 2개의 인터페이스가 서로 호환되지 않을 때, 둘을 연결해주기 위해 새로운 클래스를 만들어 연결시킬 수 있도록 한다.

3. 행위 패턴(Behavioral): 객체의 행위를 조직, 관리 연합

   > ex) 하위 클래스에서 구현해야 하는 함수 및 알고리즘들을 미리 선언하여, 상속 시 이를 필수록 구현하도록 한다.

Design Smells

Design Smell이란 나쁜 디자인을 나타내는 증상 같은 것.

1. Rigidity (경직성)

- 시스템이 변경하기 어려움.
- 하나의 변경을 위해 다른것을 변경할 때 경직성이 높음.
- 경직성이 높으면 문제가 발생했을때 수정하기 힘듦.

2. Fragility (취약성)

- 취약성이 높다면 어떤 부분을 수정했을 때 다른 부분에 영향을 줌.
- 비용이 커지며 시스템의 credibility 또한 잃음.

3. Immobility (부동성)

- 부동성이 높다면 재사용하기 위해 시스템을 분리해 컴포넌트를 만드는 것이 어려움.
- 주로 개발자가 이전에 구현되었던 모듈과 비슷한 기능을 하는 모듈을 만드려 할 때 문제점을 발견함.

4. Viscosity (점착성)

- 점착성은 **디자인 점착성**과 **환경 점착성**으로 나눌 수 있음.
- 개발환경에 지나친 의존성을 가지는 상황을 말한다.
- 설계를 유지한 상태에서 변경하기 쉽도록 설계할 때 발생 -> 디자인 점착성
- 개발환경이 느리고 효율적이지 못할 때 나타난다 -> 환경 점착성

위의 Design Smell은 나쁜 디자인을 의미(스파게티 코드)

5. Needless Complexity (불필요한 복잡성)

- 당장 필요하지 않은 코드 구조가 존재하는 상황.
- 언젠가는 유용할지도 모르는 코드도 포함된 상황을 말한다.

6. Needless Repetition (불필요한 반복)

- 복붙으로 생상된 코드가 혼재하는 상황을 말한다.

7. Opacity (불투명성)

- 혼란스러운 표현, 읽고 이해하기 어렵다.
- 의도를 잘 표현하지 못한다
- 시간이 지남에 따라 발전하는 코드는 시간이 지날수록 점점 더 불투명해지는 경향이 있다.

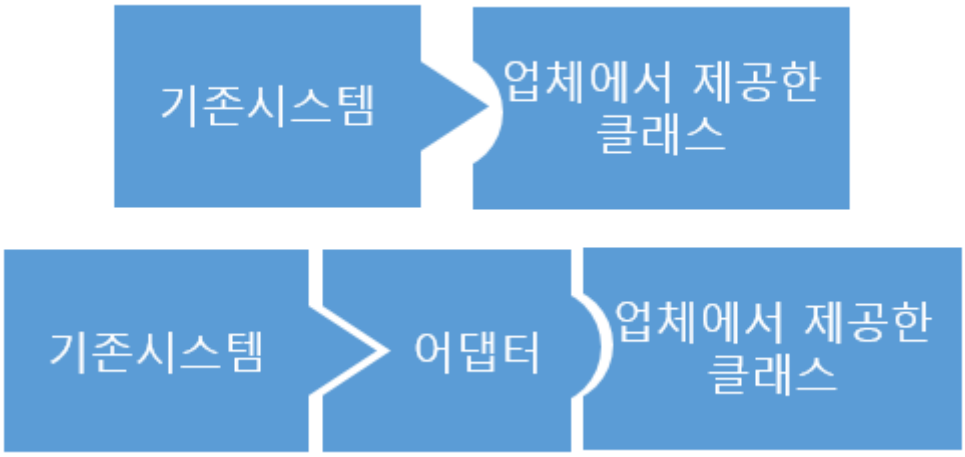
디자인 패턴 종류

Adapter Pattern

- 클래스를 바로 사용할 수 없는 경우가 종종 있다.(다른 곳에서 개발 or 수정할 수 없을 때)
- 그렇다면 중간에서 변환 역할을 해주는 클래스가 필요하다 -> 어댑터 패턴

- 상속을 통해 사용한다.
- 호환성이 없는 인터페이스를 사용하는 클라이언트 그대로 활용 가능하다.
- 향후 인터페이스가 바뀌더라도, 변경 내역은 어댑터에 캡슐화 되므로 클라이언트가 바뀔 필요가 없다.

즉 클래스의 인터페이스를 사용자가 기대하는 다른 인터페이스로 변환해준다.



Singleton Pattern

- 어플리케이션이 시작될 때, 어떤 클래스가 최초 한 번만 메모리를 할당하고 해당 메모리에 인스턴스를 만들어 사용하는 패턴.
- 즉, 인스턴스가 필요할 때, 똑같은 인스턴스를 만들지 않고 기존의 인스턴스를 활용하는 것.
- 생성자가 여러번 호출되더라도 실제로 생성되는 객체는 하나이며 최초로 생성된 이후 호출된 생성자는 이미 생성한 객체를 반환한다.
- java에서는 생성자를 private로 지정하여 다른곳에서 생성하지 못하도록 만들고, `getInstance()` 메서드를 통해 받아서 사용하도록 구현한다.

왜 쓸까요?

- 객체를 생성할 때마다 메모리 영역을 할당받아 사용해야하는데 한번의 new를 통해 객체를 생성한다면 메모리 낭비를 방지할 수 있다.
- 싱글톤으로 구현한 인스턴스는 **전역** 이므로 다른 클래스의 인스턴스들이 데이터를 공유할 수 있다는 장점이 있다.

언제 쓸까요?

- 주로 공통된 객체를 여러개 생성해서 사용해야하는 상황에 쓴다.
- ex) DB의 커넥션풀, 쓰레드풀, 캐시, 로그 기록 객체 ...
- 인스턴스가 절대적으로 한개만 존재한다는 것을 보증하고 싶을 때 사용

단점

객체 지향 설계 원칙 중 open - close principle이란 것이 존재한다.

싱글톤 인스턴스가 혼자 너무 많은 일을 하거나, 많은 데이터를 공유시키면 다른 클래스들 간의 결합도가 높아지게 되면서 Open - Close Principle을 위배하게 된다.

결합도가 높아지게 되면 유지보수가 힘들고 테스트도 원활하게 진행할 수 없는 문제점이 발생한다.

따라서, 반드시 싱글톤이 필요한 상황이 아니면 지양하는 것이 좋다.

멀티 스레드 환경에서 안전한 싱글톤 만들기

Initialization on demand holder idiom (holder에 의한 초기화)

클래스 안에 클래스(holder)를 두어 JVM클래스 로더 매커니즘과 클래스가 로드되는 시점을 이용한 방법

이러한 방법을 실제로 가장 많이 사용한다.(일반적인 싱글톤 클래스 사용 방법)

```
public class Something {
    private Something() {
    }

    private static class LazyHolder {
        public static final Something INSTANCE = new Something();
    }

    public static Something getInstance() {
        return LazyHolder.INSTANCE;
    }
}
```

- JVM의 클래스 초기화 과정에서 보장되는 원자적 특성을 이용해 싱글톤의 초기화 문제에 대한 책임을 JVM에게 떠넘기는걸 활용한다.
- 클래스 안에 선언한 클래스인 holder에서 선언된 인스턴스는 static이다. 따라서 클래스 로딩 시점에서 한번만 호출된다.
- final을 사용했기 때문에 다시 값이 할당되지 않는다.

Template Method Pattern

- 로직을 단계 별로 나눠야 하는 상황에서 적용한다.
- 단계 별로 나눈 로직들이 앞으로 수정될 가능성이 있을 경우 더 효율적이다.
- 즉, 동작 상 알고리즘의 프로그램 뼈대를 정의하는 행위 디자인 패턴이다.
- 특정 단계들을 다시 정의할 수 있게 해 준다.

조건

- 클래스는 abstract로 만든다.
- 단계를 진행하는 메소드는 수정이 불가능하도록 final 키워드를 추가한다.
- 각 단계들은 외부는 막고, 자식들만 활용할 수 있도록 protected로 선언한다.

예를 들어 A,B,C의 로직이 순서대로 실행되어야 하는 단계로 이루어져 있고 이 단계가 항상 유지되며 순서가 바뀔 일이 없다고 가정해 보았을 때, A,와 C는 변할 일이 없고 설계에 따라 B만 변한다고 해보면 아래와 같이 구현하면 된다.

```
abstract class Program{
    protected void A(){ System.out.println("A"); }
    abstract void B(){}
    protected void C(){ Ststem.out.println("C");}

    final void runProgram(){    // 상속 받은 클래스에서는 수정할 수 없다. 단계가 고정됨.
        this.A();
        this.B();
        this.C();
    }

}
```

```
class otherProgram extends Program{

    @Override
    void B(){   // 기능에 따라 B메서드를 오버라이드하여 사용한다.
        System.out.print("implememted logic");
    }

}
```

abstract와 interface의 차이는?

- abstract : 부모의 기능을 자식에서 확장시켜나가고 싶을 때 사용, **다중 상속이 안되므로 상황에 맞게 사용.**
- interface : 해당 클래스가 가진 함수의 기능을 활용하고 싶을 때 사용

Factory Method Pattern

- 부모 클래스에 알려지지 않은 구체 클래스를 생성하는 패턴
- 자식 클래스가 어떤 객체를 생성할지를 결정하도록 하는 패턴이기도 함.
- 즉, 객체를 만들어내느 부분을 서브 클래스에 위임하는 패턴이다.
- 부모 클래스 코드에 구체 클래스 이름을 감추기 위한 방법으로도 사용한다.

예시

아래처럼 로봇 객체를 만드는 추상클래스가 있다고 가정했을 때,

```
Robot (추상 클래스)
- SuperRobot
- PowerRobot

RobotFactory (추상 클래스)
- SuperRobotFactory
ModifiedSuperRobotFactory
```

Robot

각각의 클래스는 아래와 같은 구조라고 하면

```java
public abstract class Robot{
    public String getName();
}

public class SuperRobot extends Robot{
    @Override
    public String getName(){
        return "SuperRobot";
    }
}

public class PowerRobot extends Robot{
    @Override
    public String getName(){
        return "PowerRobot";
    }
}
```

Robot Factory

```java
// 기본 팩토리 클래스
public abstract class rRobotFactory{
    abstract Robot createRobot(String name);
}

// 기본 팩토리 클래스를 상속 받아 실제 로직을 구현한 팩토리
public class SuperRobotFactory extends RobotFactory{

    @Override
    Robot createRobot(String name){
        switch(name){
            case "super": return new SuperRobot();
            case "power": return new PowerRobot();
        }
        return null;
    }
}

// 로봇 클래스의 이름을 인자로 받아 직접 인스턴스를 만들어내는 팩토리
public class ModifiedSuperRobotFactory extends RobotFactory{

    @Override
    Robot createRobot(String name){
        try{
            Class<?> cls = Class.forName(name);
            Object obj = cls.newInstance();
            return(Robot)obj;
        } catch (Exception e){
            return null;
```

```
        }
    }

}
```

Test Code

```
public class FactoryMain{
    public static void main(String[] args){

        RobotFactory rf = new SuperRobotFactory();
        // 파라미터에 따라 생성되는 클래스가 다르다.
        Robot r1 = rf.createRobot("super");
        Robot r2 = rf.createRobot("power");

        RobotFactory mrf = new ModifiedSuperRobotFactory();

        // 클래스의 정확한 이름을 넣어주어야 한다.
        Robot r3 = rf.createRobot("pattern.factory.SuperRobot");
        Robot r4 = rf.createRobot("pattern.factory.PowerRobot");
    }
}
```

정리

객체 생성을 Factory 클래스에 위임하여 어떤 객체가 생성되었는지 신경쓰지 않고 반환된 객체를 사용만 하면 된다.

새로운 로봇이 추가되고 새로운 팩토리가 추가 된다 하더라도, 메인 프로그램에서 변경할 코드는 최소화 된다.

팩토리 메소드 패턴을 사용하는 이유는 클래스간의 결합도를 낮추기 위한 것이다.

결합도란 간단하게 클래스의 변경점이 생겼을 때 다른 클래스에 얼마나 영향을 주는가를 말한다.

팩토리 메소드 패턴을 사용하면 서브 클래스에 객체 생성을 위임함으로써 보다 효율적인 코드 제어를 할 수 있고 의존성을 제거하여 결과적으로 결합도를 낮출 수 있다.

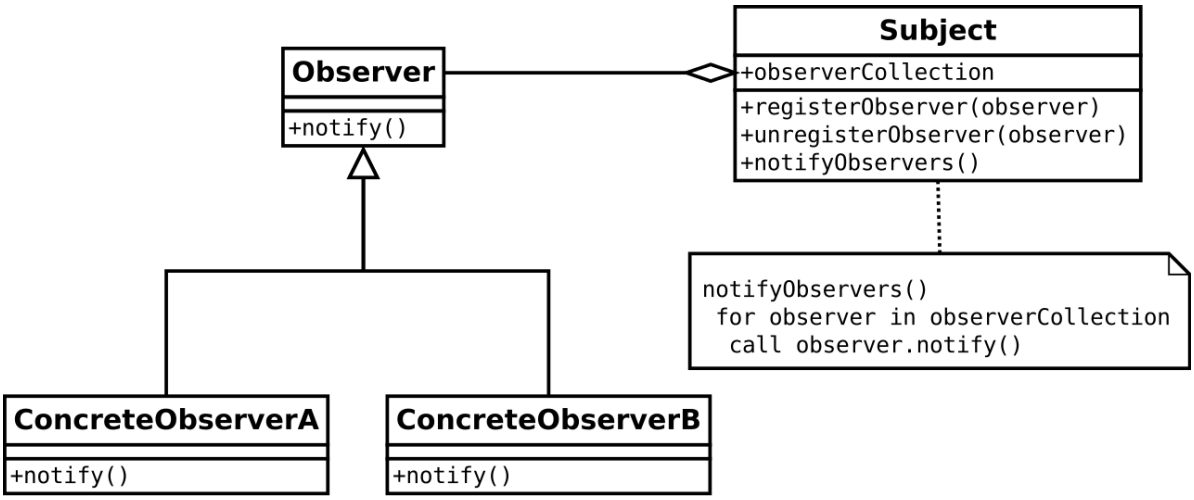
Observer Pattern

- 객체의 상태 변화를 관찰하는 관찰자
- 옵저버들의 목록을 객체에 등록하여 상태변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴
- 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다.
- 발행 / 구독 모델로 알려져 있기도 함.

구현

이 패턴의 핵심은 Observer, Listener, Subscriber 라 불리는 하나 이상의 객체를 관찰 대상이 되는객체에 등록시키는 것

각각의 옵저버들은 관찰 대상인 객체가 발생시키는 이벤트를 받아서 처리한다.



- 이벤트가 발생하면 각 옵저버는 콜백(Callback)을 받는다.
- notify 메서드는 관찰 대상이 발행한 메시지 이외에, 옵저버 자신이 생성한 인자값을 전달할 수도 있다.
- 각각의 파생 옵저버는 notify 함수를 오버라이딩하여 이벤트가 발생했을 때 처리할 각자의 동작을 정의해야 한다.
- 주체(subject)에는 일반적으로 등록, 제거 메서드가 있다. 이외에도 임시로 작동을 멈추거나 재개하는 메서드를 이용해 이벤트가 계속해서 있을 때 홍수같이 발생하는 요처응ㄹ 제어할 수 도 있다.

대표 사례

- 외부에서 발생한 이벤트(사용자의 입력 등)에 대한 응답.
- 객체의 속성 값 변화에 따른 응답. 종종 콜백은 속성 값 변화를 처리하기 위해 호출될 뿐 아니라 속성 값 또한 바꾼다. 따라서, 때때로 이벤트 연쇄(Callback Hell?)의 원인이 될 수 있다.

옵저버 패턴은 MVC(Model-View Controller) 패러다임과 자주 결합된다.

옵저버 패턴은 MVC에서 Model과 View 사이를 느슨히 연결하기 위해 사용된다.

대표적으로 Model에서 일어나는 이벤트를 통보받는 옵저버는 View의 내용을 바꾸는 스위치를 작동시킨다.

Strategy Pattern

전략 패턴(Strategy Pattern) 또는 정책 패턴(Policy Pattern)은 실행 중에 알고리즘을 선택할 수 있게 하는 행위 소프트웨어 디자인 패턴이다.

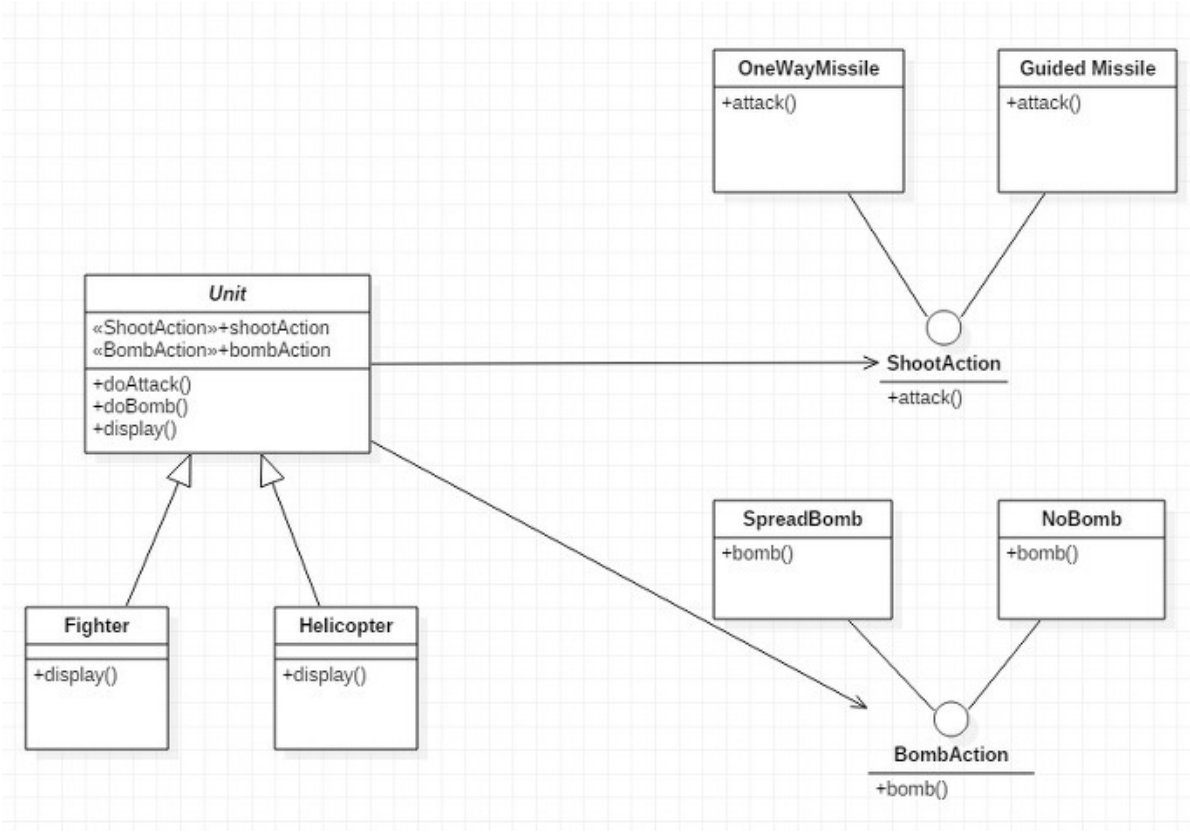
- 특정한 계열의 알고리즘들을 정의하고
- 각 알고리즘을 캡슐화하며
- 이 알고리즘들을 해당 계열 안에서 상호 교체가 가능하게 만든다.

새로운 로직을 추가하거나 변경할 때, 한번에 효율적으로 변경이 가능하다.

구현

상속은 무분별한 소스 중복이 일어날 수 있으므로 컴포지션을 활용한다.

인터페이스와 로직의 클래스와의 관계를 컴포지션하고, 유닛에서 상황에 맞는 로직을 쓰게끔 유도하는 것을 의미한다.



컴포지션

- 다른 객체의 인스턴스를 자신의 인스턴스 변수로 포함해서 메서드를 호출하는 기법
- 해당 인스턴스의 내부 구현이 바뀌더라도 영향을 받지 않는다.
- 또한, 다른 객체의 인스턴스이므로 인터페이스를 이용하면 Type을 바꿀 수 있다.

예시

```
[ 슈팅 게임을 설계하시오 ]
유닛 종류 : 전투기, 헬리콥터
유닛들은 미사일을 발사할 수 있다.
전투기는 직선 미사일을, 헬리콥터는 유도 미사일을 발사한다.
필살기로는 폭탄이 있는데, 전투기에는 있고 헬리콥터에는 없다.
```

위처럼 설계가 되어있다고 가정할 때,

1. 미사일을 쏘는 것과 폭탄을 사용하는것을 캡슐화 한다.

ShootAction과 BombAction으로 인터페이스를 선언하고, 각자 필요한 로직을 클래스로 만들어 implements한다.

2. 전투기와 헬리콥터를 묶을 Unit 추상 클래스를 만든다.

Unit에는 공통적으로 사용되는 메서드들이 들어있고, 미사일과 폭탄을 선언하기 위해 변수로 인터페이스들을 선언한다

전투기와 헬리콥터는 Unit클래스를 상속받고, 생성자에 맞는 로직을 정의해주면 된다.

```
class Fighter extends Unit {
    private ShootAction shootAction;
    private BombAction bombAction;

    public Fighter() {
        shootAction = new OneWayMissle();
        bombAction = new SpreadBomb();
    }
}

class Helicopter extends Unit {
    private ShootAction shootAction;
    private BombAction bombAction;

    public Helicopter() {
        shootAction = new GuidedMissle();
        bombAction = new NoBomb();
    }
}
```

정리

Strategy Pattern을 활용하면 로직을 독립적으로 관리하는 것이 편해진다.

로직에 들어가는 "행동"을 클래스로 선언하고, 인터페이스와 연결하는 방식으로 구성하는 것이다.

Composite Pattern

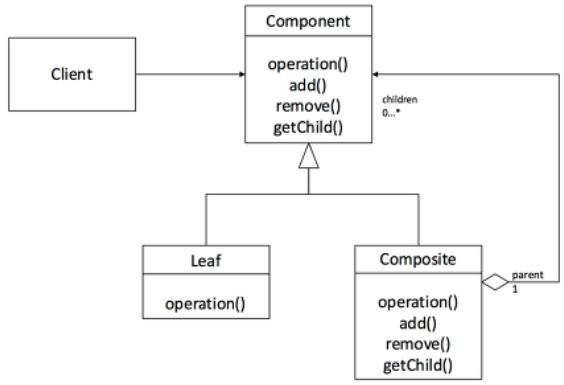
Object의 hierarchies를 표현하고 각각의 Object를 독립적으로 동일한 인터페이스를 통해 처리할 수 있게 하기 위한 디자인 패턴

객체간 관계를 트리구조로 구성하여 부분 - 전체 계층을 표현한다.

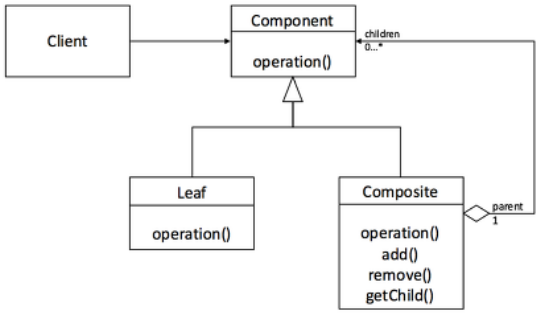
사용자가 단일 객체와 복합 객체 모두 동일하게 다루도록 한다.

구조

For Uniformity



For Type Safety



위 그림의 Leaf클래스와 Composite클래스를 같은 interface로 제어하기 위해 Component 클래스를 생성한다.

코드로 표현하면

```
public class Component {
    public void operation() {
        throw new UnsupportedOperationException();
    }
    public void add(Component component) {
        throw new UnsupportedOperationException();
    }

    public void remove(Component component) {
        throw new UnsupportedOperationException();
    }

    public Component getChild(int i) {
        throw new UnsupportedOperationException();
    }
}
```

꼭 위 코드처럼 모든 메서드가 포함되어야 하는 것은 아니다.

위의 코드는 사용하지 않는 메서드를 호출하면 exception을 발생하도록 처리한 것이다.

만약 Component 클래스에 Operation메서드만 정의되어 있다면 컴파일 시점에 오류를 알아 챌 수 있다는 장점이 있다.

Composite Pattern을 적용하면 각각의 클래스를 구분할 필요 없이 Component 클래스로 생각할 수 있다.

어떤 방식이 더 좋냐를 따지기에는 많은 것이 고려되어야 한다.

컴포지트 패턴은 이론적으로 안정성보다는 **일관성** 을 추구한다고 한다.