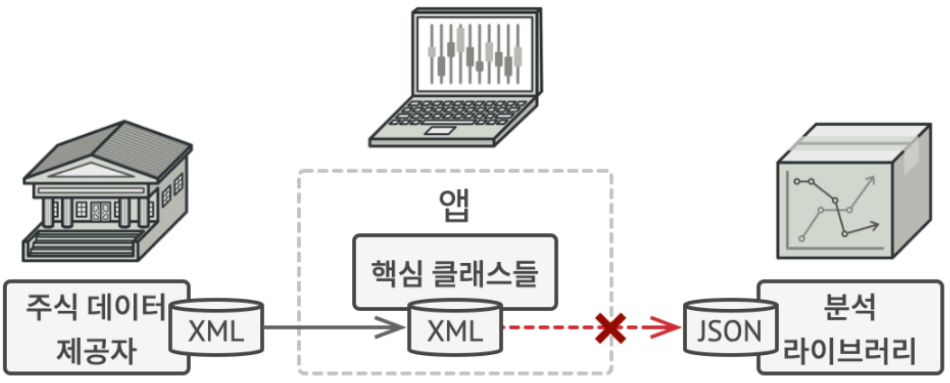
2회차 - 디자인 패턴

구조 패턴 5가지

어댑터

- 호환되지 않는 인터페이스를 가진 객체들이 협업할 수 있도록 하는 구조적 디자인 패턴
- 어댑터가 외부 인터페이스를 “감싸기” 때문에, Wrapper 패턴이라고도 부른다!
- 특정 클래스를 “래핑” 한다 == 특정 클래스의 인스턴스를 멤버 변수로 가진다 == 객체 합성

문제 제기



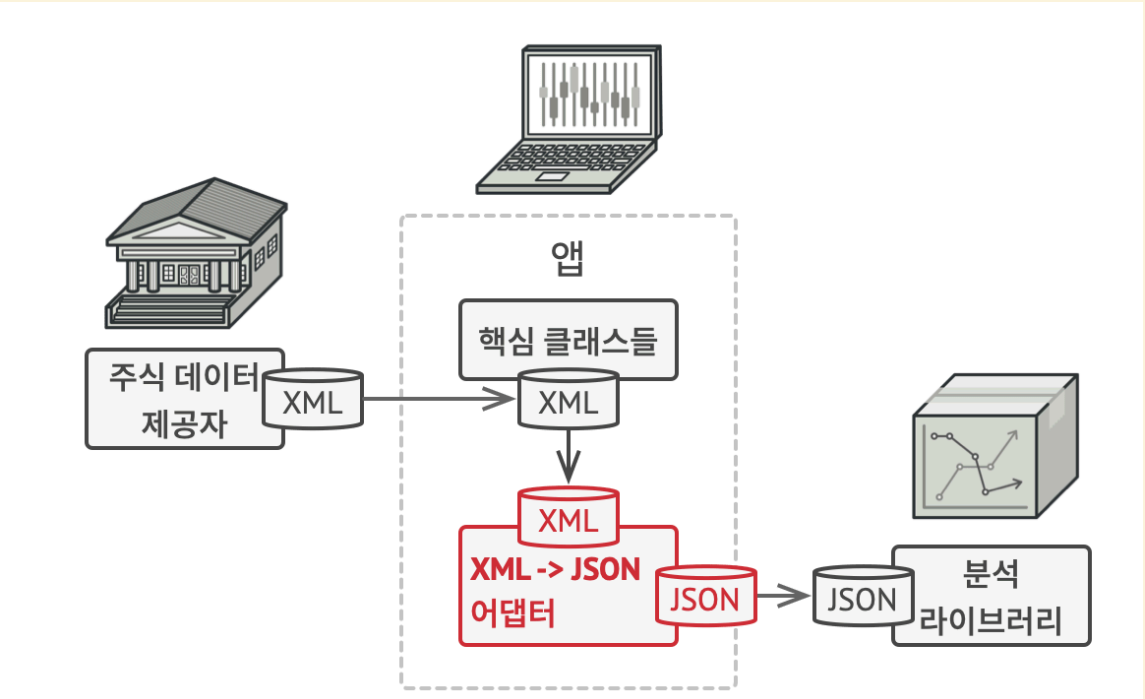
위 분석 라이브러리는 '있는 그대로' 사용할 수 없습니다. 왜냐하면 당신의 앱과 호환되지 않는 형식의 데이터를 기다리고 있기 때문입니다.

- JSON으로 분석라이브러리를 사용하려고 기존 코드를 고친다면..?

해결 : 어댑터 도입

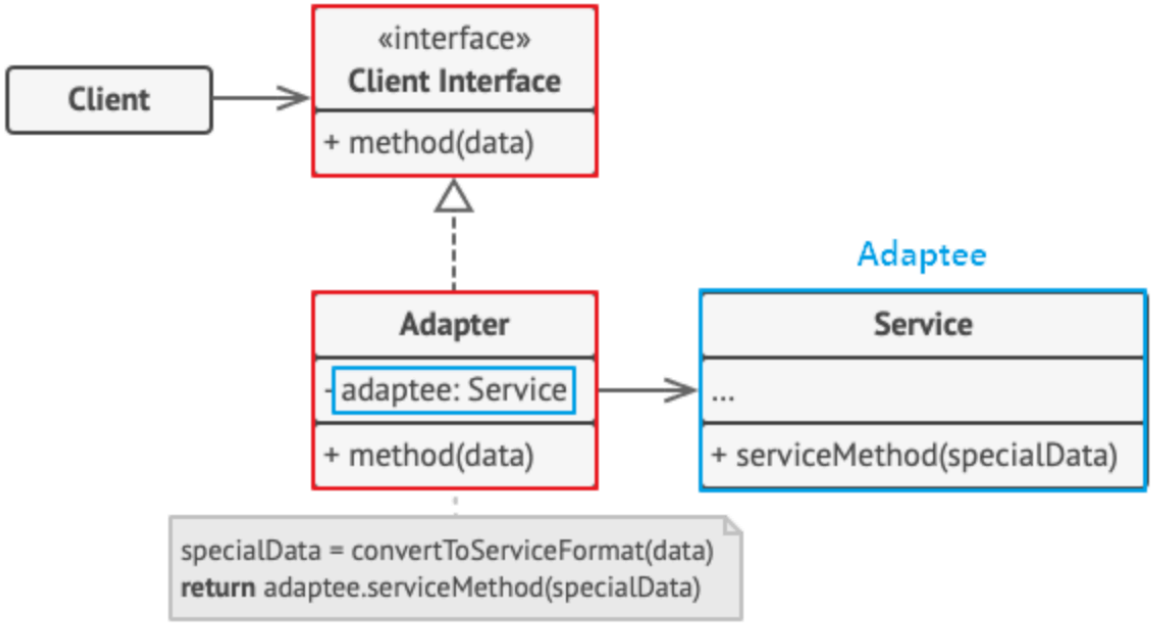
예시 - 콘센트 어댑터 (돼지코)

예시 - 주식데이터 분석 라이브러리 호환



구현1. 객체 어댑터

- 객체 합성 원칙 사용!
- 어댑터 역할 : 한 객체(Client, 기존 코드)의 인터페이스를 구현 + 다른 객체(Service, 외부 코드) 래핑(합성)



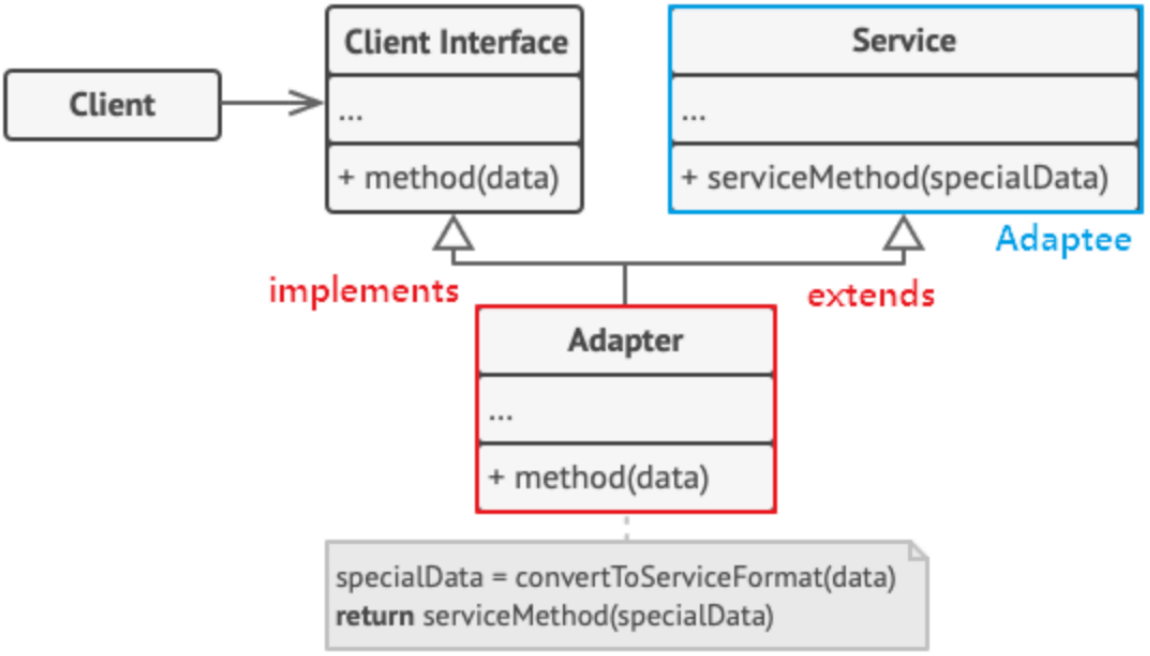
클라이언트 코드는 어댑터 클래스와 직접적으로 결합하지 않음 → **기존 클라이언트 코드를 변경하지 않고**, 새로운 유형의 어댑터들을 프로그램에 도입 가능!

💡 객체 합성 원칙?

- 객체들 간의 관계를 정의할 때, "상속보다 합성을 우선적으로 고려하라"
- 합성 : 다른 객체를 자신의 구성 요소로 포함시키기 ⇒ has-a 관계

구현2. 클래스 어댑터

- 상속 사용
- 어댑터 역할 : 동시에 두 객체(Client, Service)의 인터페이스를 상속
- C++처럼 다중 상속 지원하는 프로그래밍 언어에서만 구현 가능



💡 클래스 어댑터는 객체를 래핑할 필요 없다 → Service 클래스를 "상속" 받기 때문에, 바로 Service 클래스의 코드 재사용이 가능함

▼ 의사 코드 (객체 어댑터 방식)

```
// 220V 콘센트 (한국의 전압 시스템)
interface Korea220V {
    void connect220V();
}

// 110V 플러그 (미국의 전자제품)
class America110V {
    void connect110V() {
```

```
        System.out.println("미국 110V 전자제품이 연결됨");
    }
}

// 변압 어댑터 (110V ⇒ 220V)
class PowerAdapter implements Korea220V {
    private America110V device;

    public PowerAdapter(America110V device) {
        this.device = device;
    }

    @Override
    public void connect220V() {
        device.connect110V(); // 실제로 110V 기기를 연결
    }
}
```

```
// 사용 예시
public class Main {
    public static void main(String[] args) {
        America110V americanDevice = new America110V(); // 미국 110V 전자제
        Korea220V adapter = new PowerAdapter(americanDevice); // 220V →

        adapter.connect220V();
        // 출력:
        // "미국 110V 전자제품이 연결됨"
    }
}
```

적용

1. 외부 클래스를 사용하고 싶지만 그 인터페이스가 기존 코드와 호환되지 않을때

2. 부모 클래스에 공통 기능을 추가하려고 하는데,
    - 부모 클래스 수정이 불가하고
    - 기존 자식 클래스를 그대로 유지하면서
    - 새로운 기능을 추가하고자 할 때!

    ▼ 예시 코드

```
// 기존 클래스 (수정 불가능)
class ChildA {
    public String specificRequestA() {
        return "Child A의 요청 처리";
    }
}

class ChildB {
    public String specificRequestB() {
        return "Child B의 요청 처리";
    }
}

// 공통 인터페이스 (클라이언트가 기대하는 인터페이스)
interface Target {
    String request();
}

// 객체 어댑터: 기존 클래스를 포함(Composition)하여 변환
class AdapterA implements Target {
    private ChildA adaptee; // 포함 (Composition)

    public AdapterA(ChildA adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public String request() {
        return adaptee.specificRequestA(); // 기존 메서드를 호출하여 변환
    }
}
```

```
class AdapterB implements Target {
    private ChildB adaptee; // 포함 (Composition)

    public AdapterB(ChildB adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public String request() {
        return adaptee.specificRequestB(); // 기존 메서드를 호출하여 변환
    }
}

// 클라이언트 코드
public class Main {
    public static void main(String[] args) {
        ChildA childA = new ChildA();
        Target adapterA = new AdapterA(childA);
        System.out.println(adapterA.request()); // "Child A의 요청 처리"

        ChildB childB = new ChildB();
        Target adapterB = new AdapterB(childB);
        System.out.println(adapterB.request()); // "Child B의 요청 처리"
    }
}
```

장단점

장점

1. 단일 책임 원칙 : 프로그램의 비즈니스 로직에서 인터페이스 또는 데이터 변환 코드를 분리할 수 있음!
2. 개방/폐쇄 원칙 : 클라이언트 코드를 변경하지 않고 새로운 유형의 어댑터 도입 가능

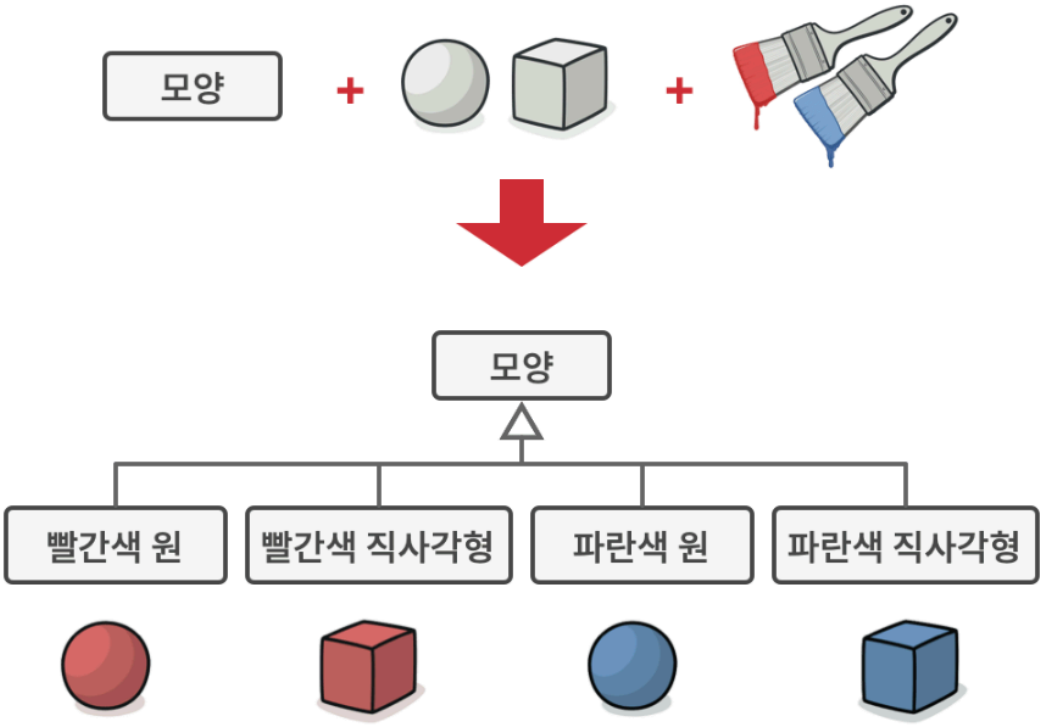
단점

1. 코드의 전반적인 복잡성 증가 : 다수의 새로운 인터페이스, 클래스들을 도입해야 하기 때문!

브리지

- 큰 클래스, 또는 밀접하게 관련된 클래스들의 집합을 두개의 개별 계층 구조로 나누기 → 각각 독립적으로 개발할 수 있도록 한다!

문제 제기

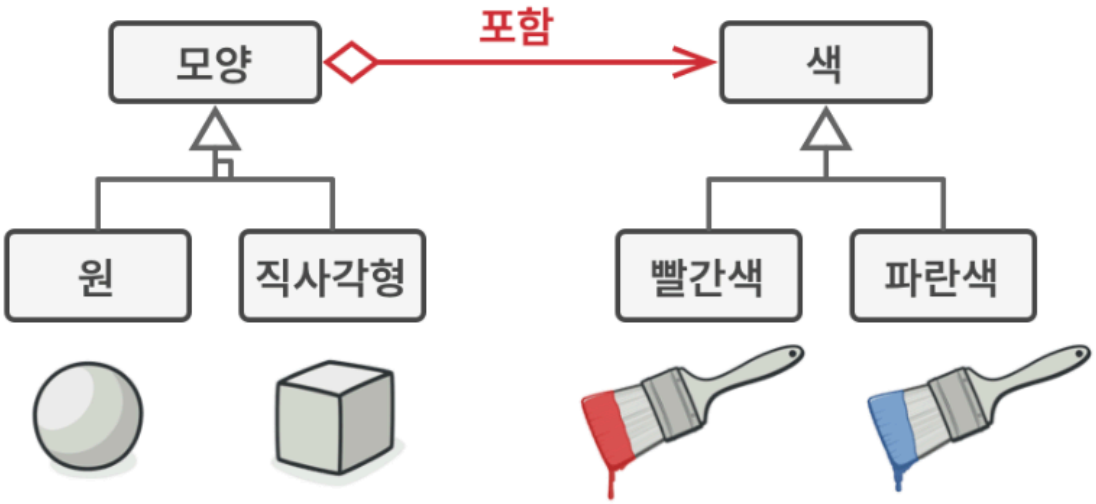


클래스 조합들의 수는 기하급수적으로 증가합니다.

- 기존에 존재하던 모양(원, 직사각형) 클래스에 색상 도입 → 모양 클래스를 '상속'받아 빨간색/파란색 원/직사각형 클래스 만들기!
- 벌써 4개의 클래스..

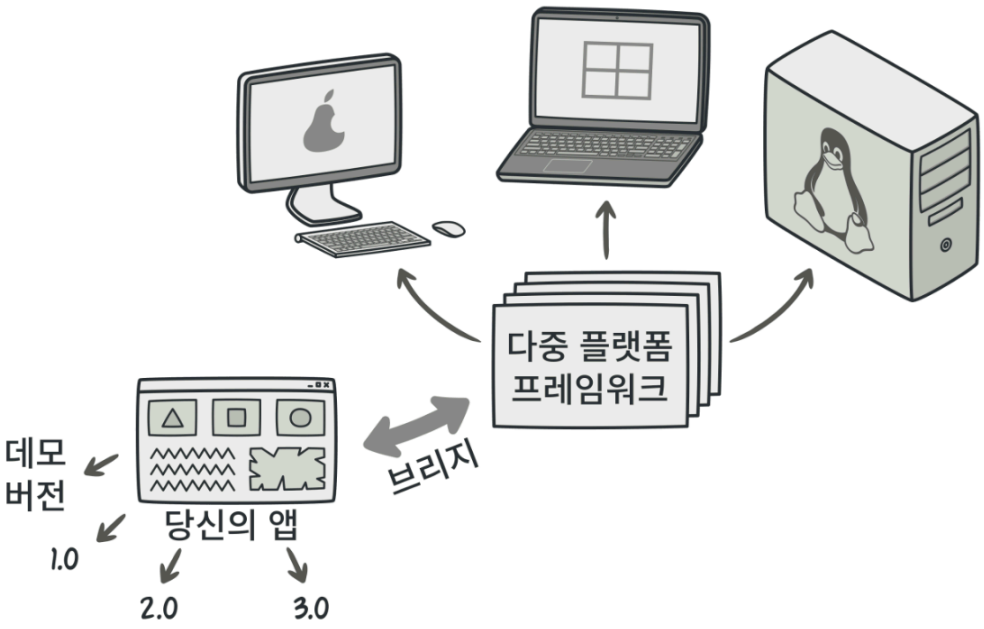
해결 : 상속 → 객체 합성

- 문제 원인 : 모양, 색상의 **두 독립적 차원** → 두 차원에서 모양 클래스를 확장하려고 하기 때문!
- 문제 해결 : 차원 중 하나를 별도의 클래스 계층구조로 추출 → 새 계층구조의 객체를 참조하도록 하자!



- '색' 코드를 별도의 자체 클래스로 추출
- '모양' 클래스는 **색 객체들 중 하나를 참조하는 필드**를 가지면 됨 → 이 '참조' 가 다리(브릿지) 역할을 한다!

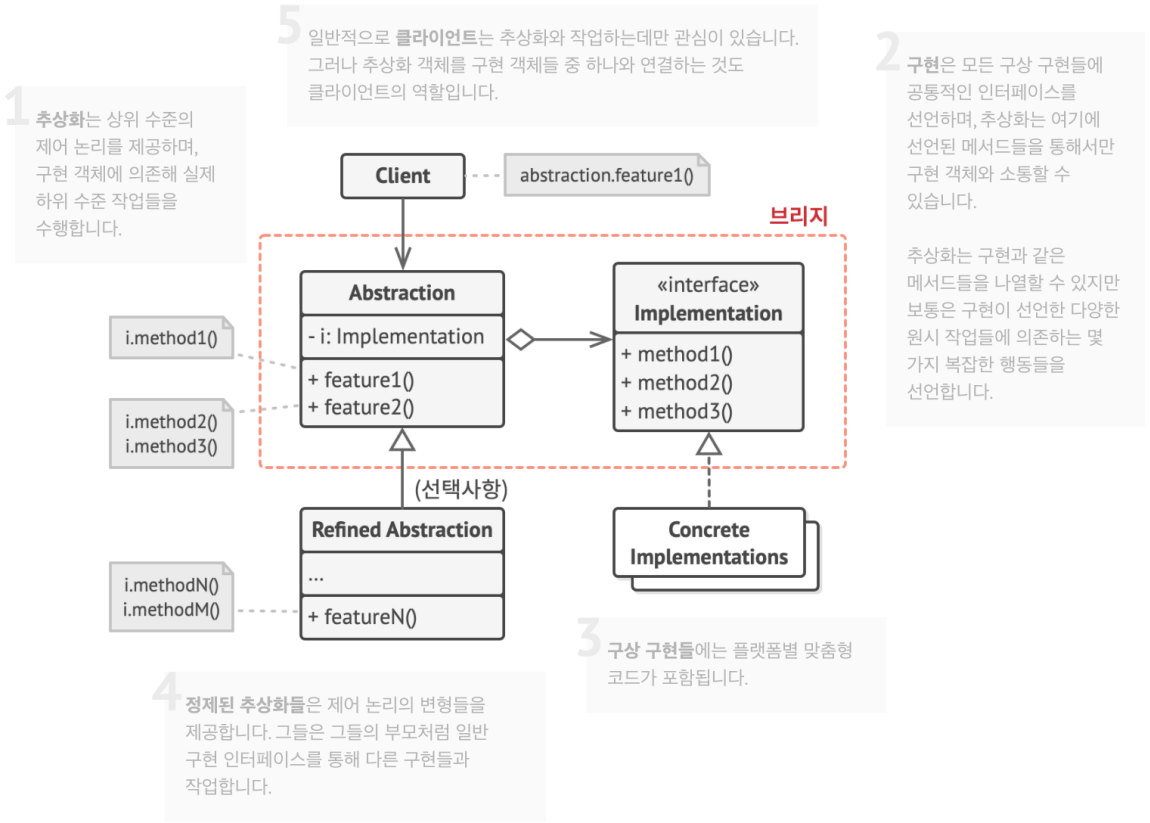
예시 : 다중 플랫폼 프레임워크



크로스 플랫폼 앱을 구성하는 방법 중 하나.

- 유저 인터페이스와 운영체제 코드(API)를 다른 차원으로 분리
- **앱의 버전과 운영체제는 독립적인 차원이다!**

구조



▼ 의사 코드

```
public interface Color {
    String fill();
}

public class Red implements Color {
    @Override
    public String fill(){
        return "빨간색";
    }
}

public class Blue implements Color {
    @Override
    public String fill(){
        return "파란색";
```

```
    }
}
```

```
public abstract class Brush {
    protected Color color;   // 객체 합성! => 브리지(다리) 역할

    protected Brush(Color color){
        this.color = color;
    }

    public abstract String draw();

}

public class HBPencil extends Brush {
    public static final String type = "[HB 연필]";

    public HBPencil(Color color) {
        super(color);
    }

    @Override
    public String draw(){
        return type + " " + color.fill();
    }
}

public class MonoLine extends Brush {
    public static final String type = "[모노라인]";

    public MonoLine(Color color) {
        super(color);
    }

    @Override
    public String draw(){
        return type + " " + color.fill();
```

```
    }
}
```

```
class BrushTest{

    void main(){
        Brush redBrush = new HBPencil(new Red());
        System.out.println(redBrush.draw());    // [HB 연필] 빨간색

        Brush blueBrush = new MonoLine(new Blue());
        System.out.println(blueBrush.draw());   // [모노라인] 파란색
    }
}
```

적용

1. 어떤 기능의 여러 변형을 가진 모놀리식 클래스를 나누고 정돈하려 할 때

> 💡 모놀리식 클래스 ?
>
> - “하나의 큰 덩어리로 이루어진” ⇒ 하나의 클래스가 너무 많은 역할과 책임을 가지고 있는 구조!

2. 여러 독립(직교) 차원에서 클래스를 확장해야 할 때
3. 런타임(실행 시간)에 동적으로 구현을 전환할 수 있어야 할 때
    - 런타임(실행 시점) ↔ 컴파일 타임(프로그램 작성 시점)

장단점

장점

1. 플랫폼 독립적인 클래스/앱을 만들 수 있음

2. 클라이언트 코드가 플랫폼 세부 정보에 노출되지 않음 : 상위 수준의 추상화를 통해 작동하기 때문
3. 개방/폐쇄 원칙 : 새로운 추상화들과 구현들을 상호 독립적으로 도입 가능
4. 단일 책임 원칙 : 브리지 패턴은 추상화와 구현을 분리하는 디자인 패턴! ⇒ 각 클래스가 하나의 역할만 수행하도록 설계됨

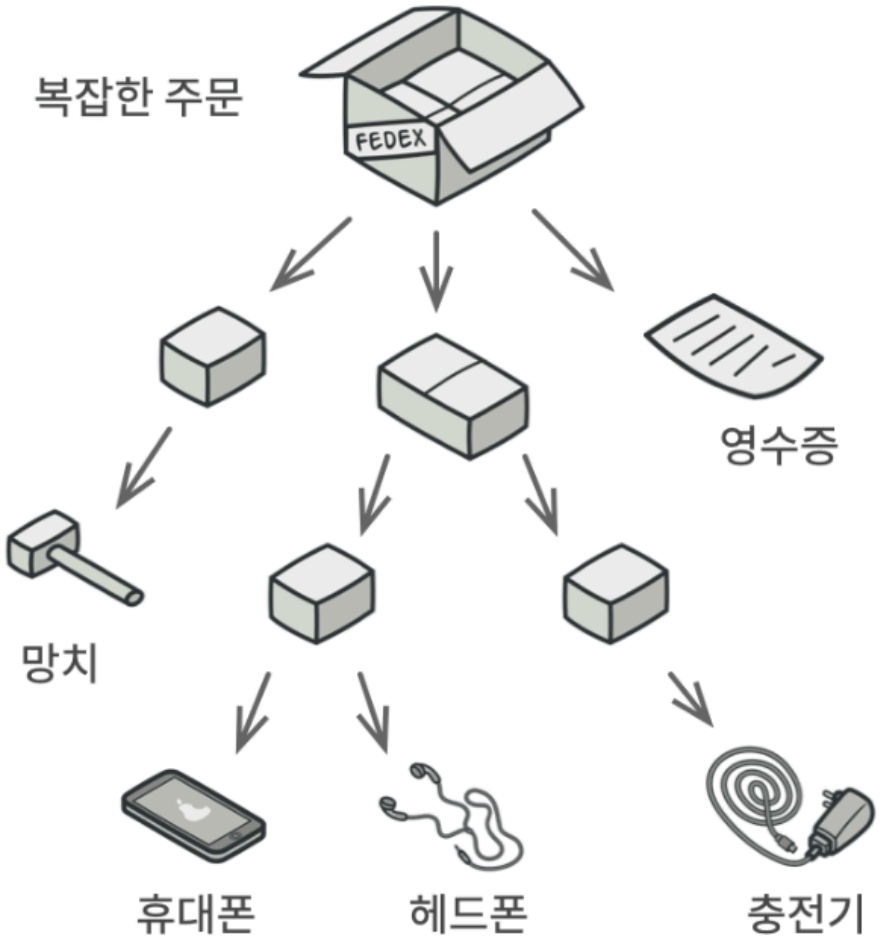
단점

1. 코드 복잡성 증가 가능성 : 결합도가 높은 클래스에 패턴을 적용하기 때문

복합체

- 객체들을 트리 구조로 구성 → 개별 객체들처럼 작업할 수 있도록 하자!
- "복합 객체"와 "단일 객체"를 동일한 컴포넌트로 취급 ⇒ 클라이언트가 둘을 구분할 필요 없이 동일한 인터페이스를 사용하도록!
    - 구현을 단순화

문제 제기



- 주문 총 가격 계산 어떻게 함?
- 모든 상자 풀고 제품 가격 더하면 되지 않음? → 상자의 중첩 수준 등 복잡한 세부 사항을 알고있어야 함
- 클라이언트는 그런거 생각 안하고 싶음..

해결 : 제품과 상자가, 총 가격을 계산하는 메서드를 선언하는 공통 인터페이스 사용

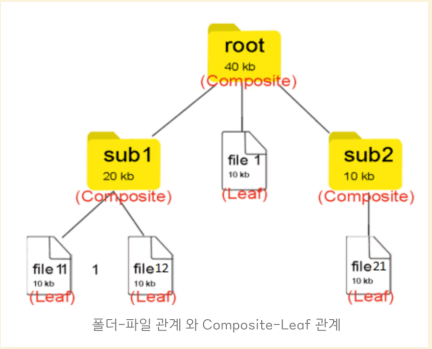
- 제품 : 제품 가격 반환
- 상자 : 포함된 항목들을 보고, 총 가격 반환
  - 포함된 항목 중 상자가 있으면? ⇒ 재귀 구현!
- 물건이 제품인지, 상자인지 신경쓰지 않아도 됨! → 공통 인터페이스를 통해 모두 같은 방식으로 처리하면 됨!

예시 : 군대

군대 구조의 예시.

- 명령을 내리면?
  - 계층구조 최상위에서 내려옴 → 모든 병사가 자신이 수행해야 할 작업을 알게 될때까지!

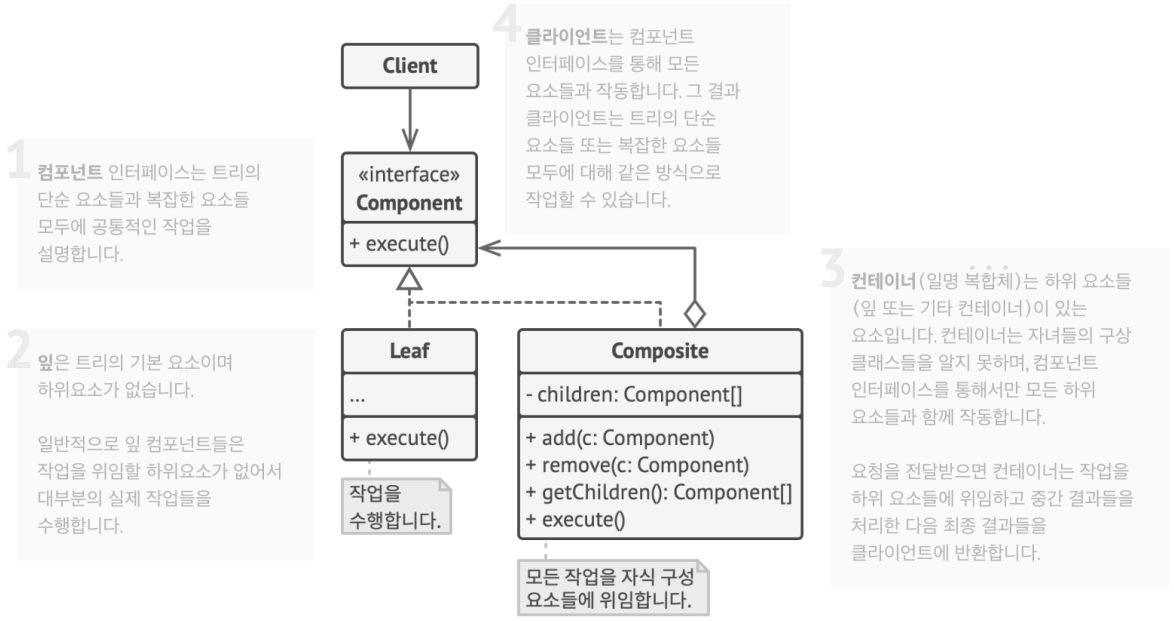
예시 : 파일 시스템



폴더-파일 관계 와 Composite-Leaf 관계

- 리눅스에서는 디렉토리와 파일이 모두 같은 파일로 취급된다!
  - 윈도우에서는 아님!

구조



▼ 의사 코드

```
interface Component {
    void operation();
}

class Leaf implements Component {

    @Override
    public void operation() {
        System.out.println(this + " 호출");
    }
}

class Composite implements Component {

    // Leaf 와 Composite 객체 모두를 저장하여 관리하는 내부 리스트
    List<Component> components = new ArrayList<>();

    @Override
    public void operation() {
        System.out.println(this + " 호출");
```

```
        // 내부 리스트를 순회하여, 단일 Leaf이면 값을 출력하고,
        // 또다른 서브 복합 객체이면, 다시 그 내부를 순회하는 재귀 함수 동작이 된다.
        for (Component component : components) {
            component.operation(); // 자기 자신을 호출(재귀)
        }
    }
}
```

적용

1. 트리 형태의 객체 구조 구현 시 사용
2. 클라이언트 코드가 단순 요소들과 복합 요소들을 모두 균일하게 처리하고 싶을 때

장단점

장점

1. 다형성과 재귀 : 복잡한 트리 구조를 편하게 작업
2. 개방/폐쇄 원칙 : 새로운 Leaf 클래스를 추가하더라도, 클라이언트는 추상화된 인터페이스만을 사용하면 됨!

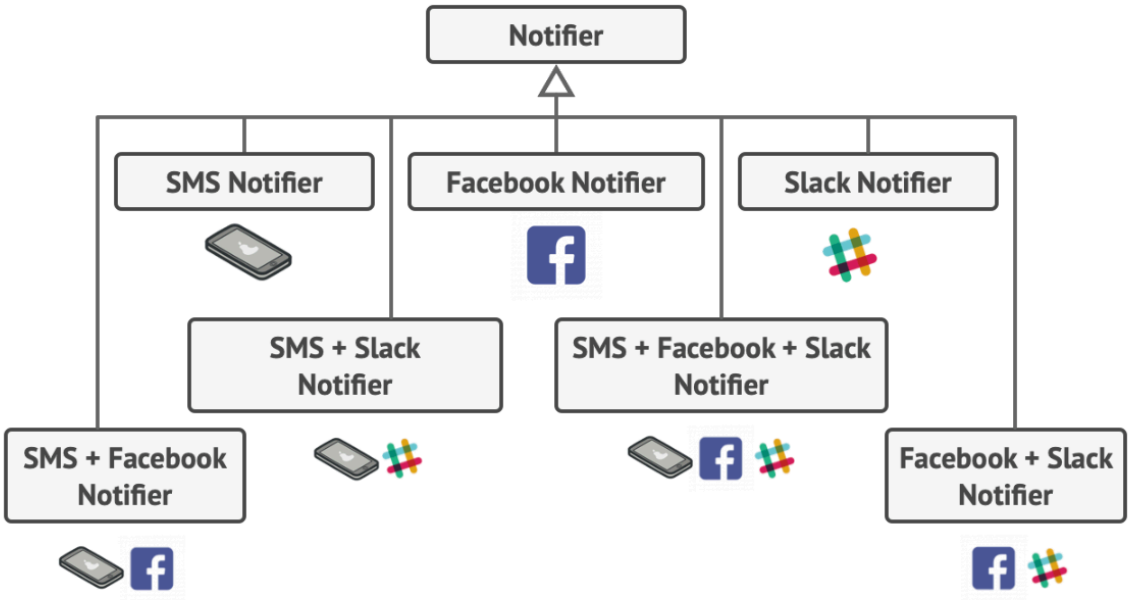
단점

1. 기능이 너무 다른 클래스들에는 애초에 공통 인터페이스 제공하기가 어려울수도!
2. 재귀 호출 특 : depth가 깊어지면 디버깅 어렵다

데코레이터

- 객체를, 새로운 기능을 포함한 특수 래퍼 객체들 내에 넣어서 위 행동들을 해당 객체들에 연결시키자!

문제 제기



자식 클래스들의 합성으로 인한 클래스 수의 폭발

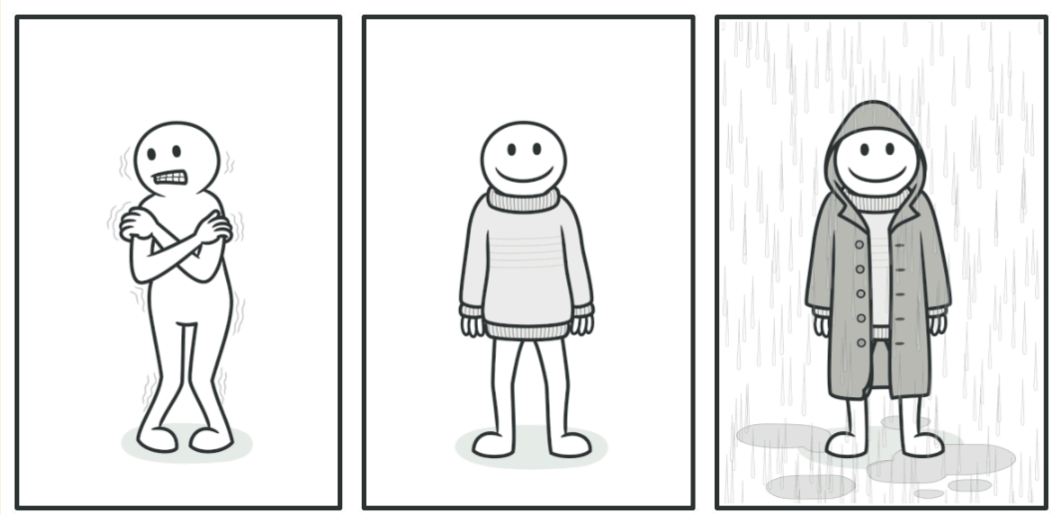
해결 : 집합 관계/합성

- 상속의 근본적 문제
  1. 상속은 정적이다 : 런타임 때 기존 객체의 행동을 변경할 수 없음!
  2. 다중 상속 불가 : 자바에서는 단일 상속만 가능함!
     - 애초에 확장에 좋은 형태가 아님
- 상속 대신 집합관계/합성 사용
  - 집합 관계 : 한 객체가 다른 객체에 대한 참조를 가짐 ⇒ has-a 관계

상속 관계 VS 집합 관계

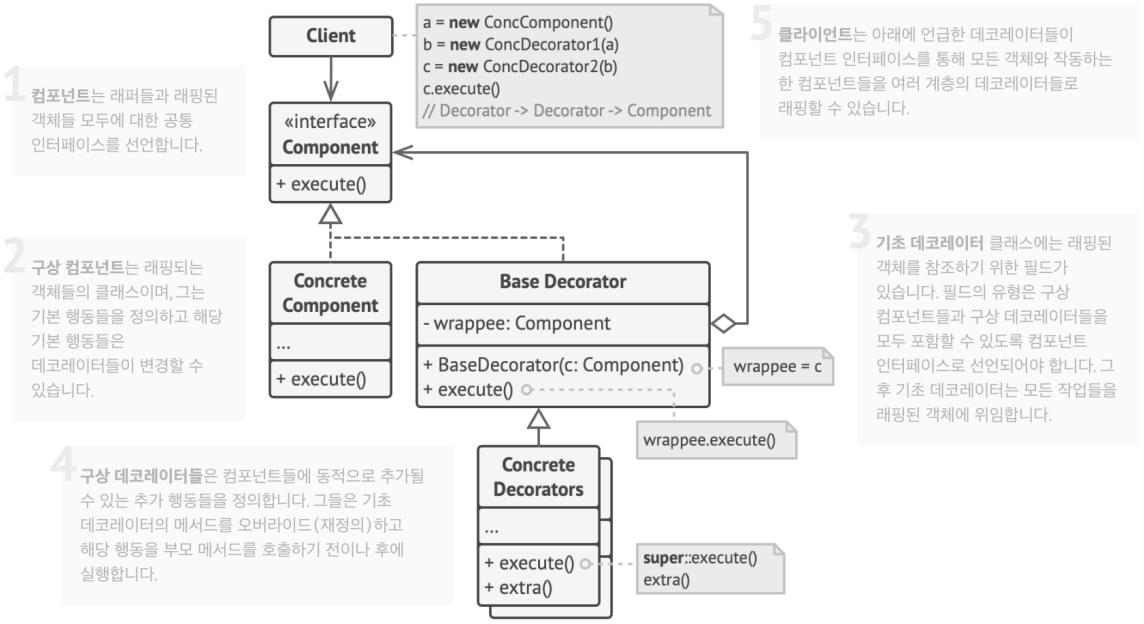
상속과 집합 관계의 차이점.

예시 : 옷 입기



- 옷 : 기초 행동을 확장함 ⇒ 하지만 필요하지 않으면 벗을 수 있다! (내 몸의 일부가 아니기 때문에)

구조



▼ 의사 코드

```
public interface ChristmasTree {
    //인터페이스이기 때문에 public abstract 생략
    void decorate();
}


//핵심 기능을 담당하는 클래스
public class ChristmasTreeImpl implements ChristmasTree {
    @Override
    public void decorate() {
        System.out.println("꾸미자 ");
    }
}


//부가 기능을 담는 클래스
public class TreeDecorator implements ChristmasTree{
    //핵심 기능을 담는 오브젝트를 주입받는다.
    private ChristmasTree tree;
```

```
    public TreeDecorator(ChristmasTree tree) {
        this.tree = tree;
    }

    @Override
    public void decorate() {
        tree.decorate();
    }
}

// 구체적인 부가 기능 클래스 : 꽃잎으로 꾸미기!
public class Garland extends TreeDecorator{
    public Garland(ChristmasTree tree) {
        super(tree);
    }

    @Override
    public void decorate() {
        super.decorate();
        decorateWithTreeDecorator();
    }

    private void decorateWithTreeDecorator(){
        System.out.println("꽃잎으로");
    }

}
```

```
//테스트 클래스(클라이언트)
public class DecoratorTest {

    @Test
    public void decoratorPatternTest(){
        // 부가기능 오브젝트에 핵심 기능 오브젝트를 주입해준다.
        //런타임시 동적으로 오브젝트의 관계가 형성돼 부가기능이 적용된다.
        ChristmasTree christmasTree = new Garland(new ChristmasTreeImpl()

        christmasTree.decorate();
```

```
        //결과:
        //꾸미자
        //꽃잎으로

        ChristmasTree christmasTree1 = new TreeTopper(new ChristmasTreeI

        christmasTree1.decorate();
        //결과:
        //꾸미자
        //트리 꼭대기 장식으로
    }
}
```

적용

1. 객체들을 사용하는 코드를 훼손하지 않으면서, 런타임에 추가 행동들을 객체들에 할당할 수 있어야 할때
2. 상속을 사용하여 객체의 행동을 확장하는 것이 어색하거나 불가능 할때

장단점

장점

1. 새 자식 클래스 없이 객체의 기능을 확장할 수 있음
2. 런타임에 객체들에서부터 책임을 추가/삭제 가능
3. 객체를 여러 데코레이터로 래핑 → 여러 행동들을 합성 가능
4. 단일 책임 원칙 : 모놀리식 클래스를 여러개의 작은 클래스들로 나눌 수 있음

단점

1. 래퍼들의 스택에서 특정 래퍼를 제거하기 어려움
2. 데코레이터의 행동이, 데코레이터 스택 내의 순서에 의존하지 않는 방식으로 구현하기 어려움
3. 계층들의 초기 설정 코드가 보기 흉할 수 있음...?

퍼사드

- 사용하기 복잡한 클래스 라이브러리에 대해 사용하기 편리하게 간편한 인터페이스를 구성하기 위한 패턴
- 클래스를 재정리
- 일반적으로 알고있는 “추상화”의 개념과 가까운 듯!

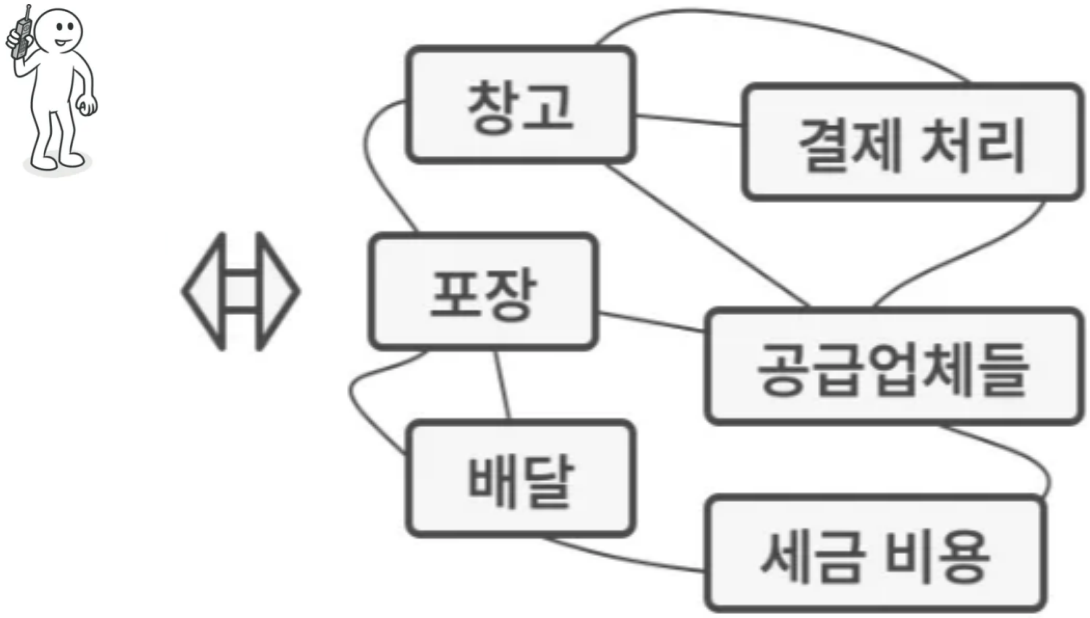
Facade

Facade라는 단어의 뜻은 건축물의 정면을 의미한다. 건축물의 정면은 보통 건축물의 이미지와 건축 의도를 나타내기 때문에 오래 전부터 특별한 디자인을 적용하여 의미를 부여했다.

이처럼 건축물 정면만 봐도 이 건물이 어떤 목적을 하는지 단번에 알수 있다는 특징을 차용하여 명명 지은 것이다.

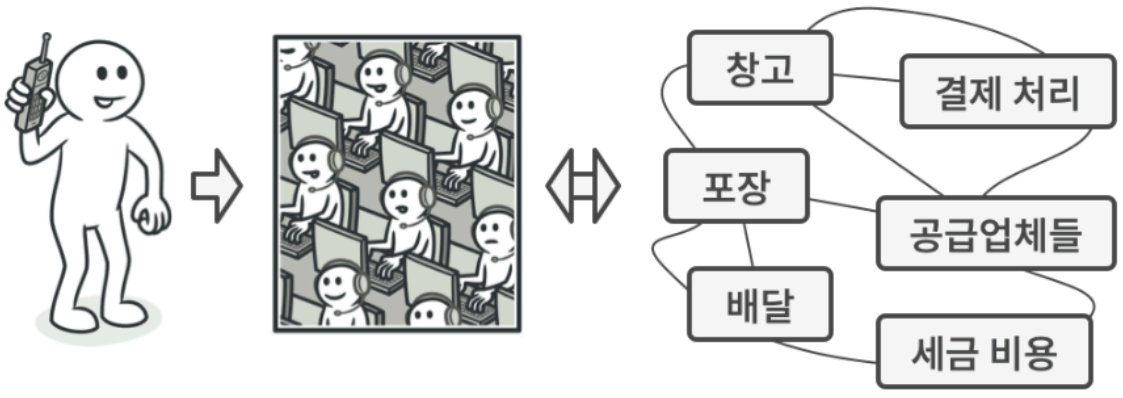
문제 제기

- 큰 규모의 라이브러리나 프레임워크는 사용하기 위해 많은 지식을 요구할 수 있음 (종속관계, 실행 순서 등등)
- 사용자 입장에서 이를 모두 습득한 후
  에 라이브러리/프레임워크를 사용하는
  것은 너무 어려운 일!



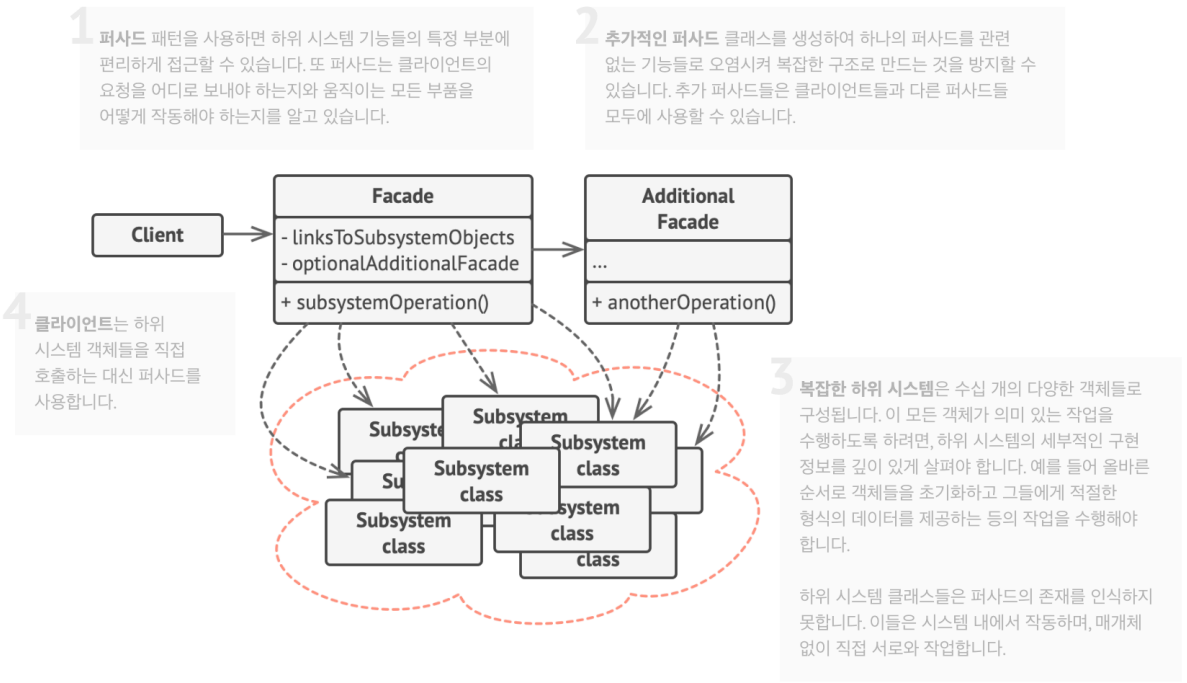
해결

- 사용자가 사용하고자 하는 기능들을 사용할 수 있는, 단순화 된 인터페이스를 제공



- 단순화된 인터페이스 : 상담원

구조



- 퍼사드 패턴은 클래스 구조가 정형화 되어있지 않다!!
  - 클래스 위치는 어떻고, 어떤 형식으로 위임을 해야되고 등등..
  - 그냥 퍼사드 클래스를 만들어서, 적절히 기능 집약화만 해주면 됨

Additional Facade

퍼사드 패턴은 재귀적으로 적용 가능!

아래 계층에서 만들어진 퍼사드들을 합친 퍼사드를 만들고, 또 그런 퍼사드들을 합쳐서 다른 퍼사드를 만들고..

퍼사드를 재귀적으로 구성하면 시스템은 보다 편리해짐!

적용

1. 복잡한 하위 시스템에 대한, 제한적이지만 간단한 인터페이스가 필요할 때
2. 하위 시스템을 계층들로 구성하려는 경우

장단점

장점

1. 복잡한 하위 시스템에서 코드를 별도로 분리할 수 있음!
2. 외부에서 시스템을 사용하기 쉬워짐

단점

1. 퍼사드가 앱의 모든 클래스에 결합된 God 객체가 될 수 있다 (..?)
2. 퍼사드 클래스 자체가 서브시스템에 대한 의존성을 가져서, 의존성을 완전히는 피할 수 없다
3. 추가적인 코드가 늘어난다 → 유지 보수 측면에서 공수가 더 많이 들어간다