10. 데커레이터 패턴

학습목표

학습목표

- 독립적인 추가 기능의 조합 방법 이해하기
- 데커레이터 패턴을 통한 기능의 조합 방법 이해하기
- 사례 연구를 통한 데커레이터 패턴의 핵심 특징 이해하기

Decorate의 사전적 의미

- [동사] 장식하다, 꾸미다
- 참고 : Decorator는 네이버 사전에 의하면 (주택) 도장 및 도배업자를 의미

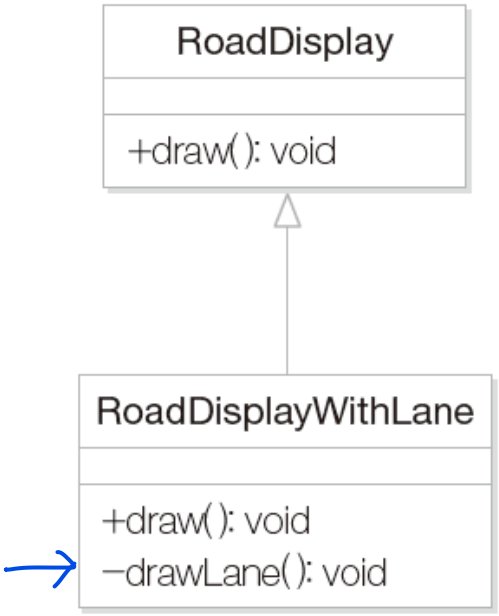
데커레이터 패턴의 특징

- 일반적으로 데커레이터 패턴을 설명할 때 “객체에 동적으로 새로운 책임을 추가할 수 있다”라고 함
- 하지만, Decorate라는 단어의 의미에 집중하여 마치 Subway에서 샌드위치를 살 때 내 마음대로 빵도 고르고 토핑도 고르고 소스도 골라 하나의 샌드위치를 완성하듯 제공하는 여러 기능을 상황에 맞게 조합하여 그 때 그 때 다르게 실행할 수 있도록 지원하는 패턴이라고 이해하는 것이 조금 더 패턴에 대해 이해하기 쉬울 것

10.1 도로 표시 방법 조합하기

- 도로 표시
  - RoadDisplay 클래스: 기본 도로 표시 기능을 제공하는 클래스
  - RoadDisplayWithLane 클래스: 기본 도로 표시에 추가적으로 차선을 표시하는 클래스

그림 10-1 기본 도로 및 차선을 표시하는 RoadDisplay와 RoadDisplayWithLane 클래스의 설계



소스 코드

코드 10-1

```
public class RoadDisplay {  // 기본 도로 표시 클래스
   public void draw() {
      System.out.println("도로 기본 표시") ;
   }
}
```

```
public class RoadDisplayWithLane extends RoadDisplay {  // 기본 도로 표시 + 차선 표시 클래스
   public void draw() {
      super.draw(); // 상위 클래스 즉 RoadDisplay의 draw 메서드를 호출해서 기본 도로를 표시
      drawLane();
   }
   private void drawLane() {
      System.out.println("차선 표시") ;
   }
}
```

```
public class Client {
   public static void main(String[] args) {
      RoadDisplay road = new RoadDisplay() ;
      road.draw() ;  // 기본 도로만 표시

      RoadDisplay roadWithLane = new RoadDisplayWithLane() ;
      roadWithLane.draw() ; // 기본 도로 + 차선 표시
   }
}
```

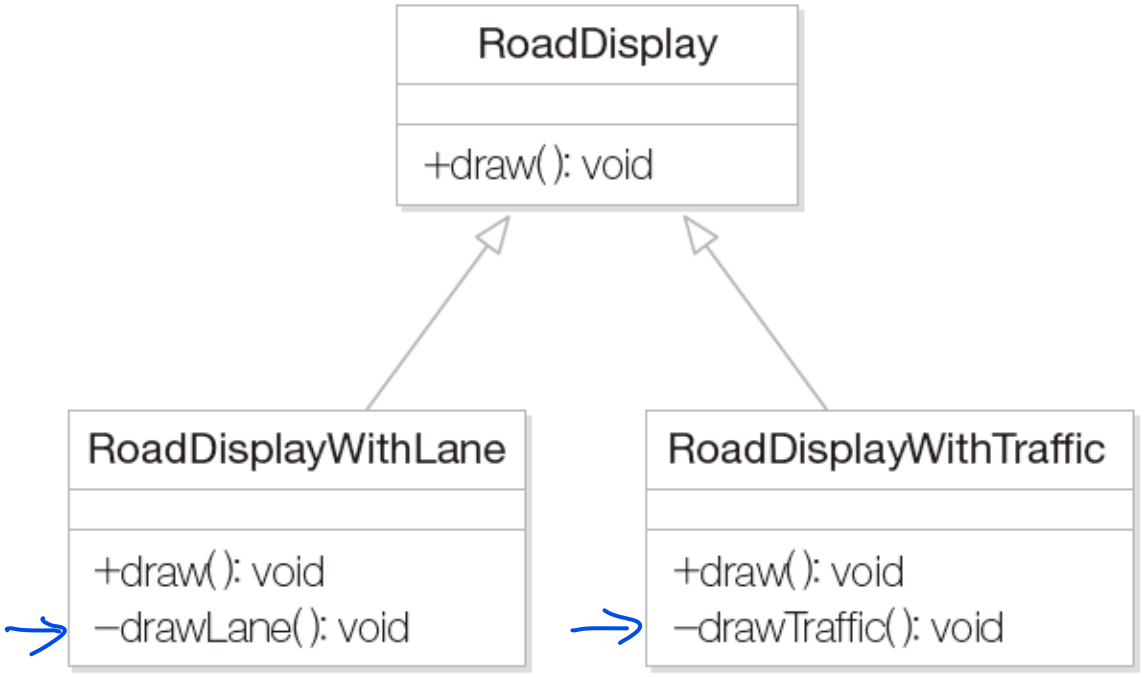
10.2 문제점

- 또다른 추가적인 도로 표시 기능을 구현하고 싶다면 어떻게 해야 하는가? 예를 들어 기본 도로 표시에 교통량을 표시하고 싶다면?
- 뿐만 아니라 여러가지 추가 기능의 조합하여 제공하고 싶다면 어떻게 해야 하는가? 예를 들어 기본 도료 표시에 차선 표시 기능과 교통량 표시 기능을 함께 제공하고 싶다면?

10.2.1 또다른 도로 표시 기능을 추가로 구현하는 경우

❖ 기본 도로 표시에 추가적으로 교통량을 표시하는 경우

그림 10-2 기본 도로 및 교통량을 표시하는 RoadDisplayWithTraffic 클래스의 설계



10.2.1 또다른 도로 표시 기능을 추가로 구현하는 경우

코드 10-2

```
public class RoadDisplayWithTraffic extends RoadDisplay {
   public void draw() {
      super.draw();
      drawTraffic();
   }
   private void drawTraffic() {
      System.out.println("교통량 표시") ;
   }
}
```

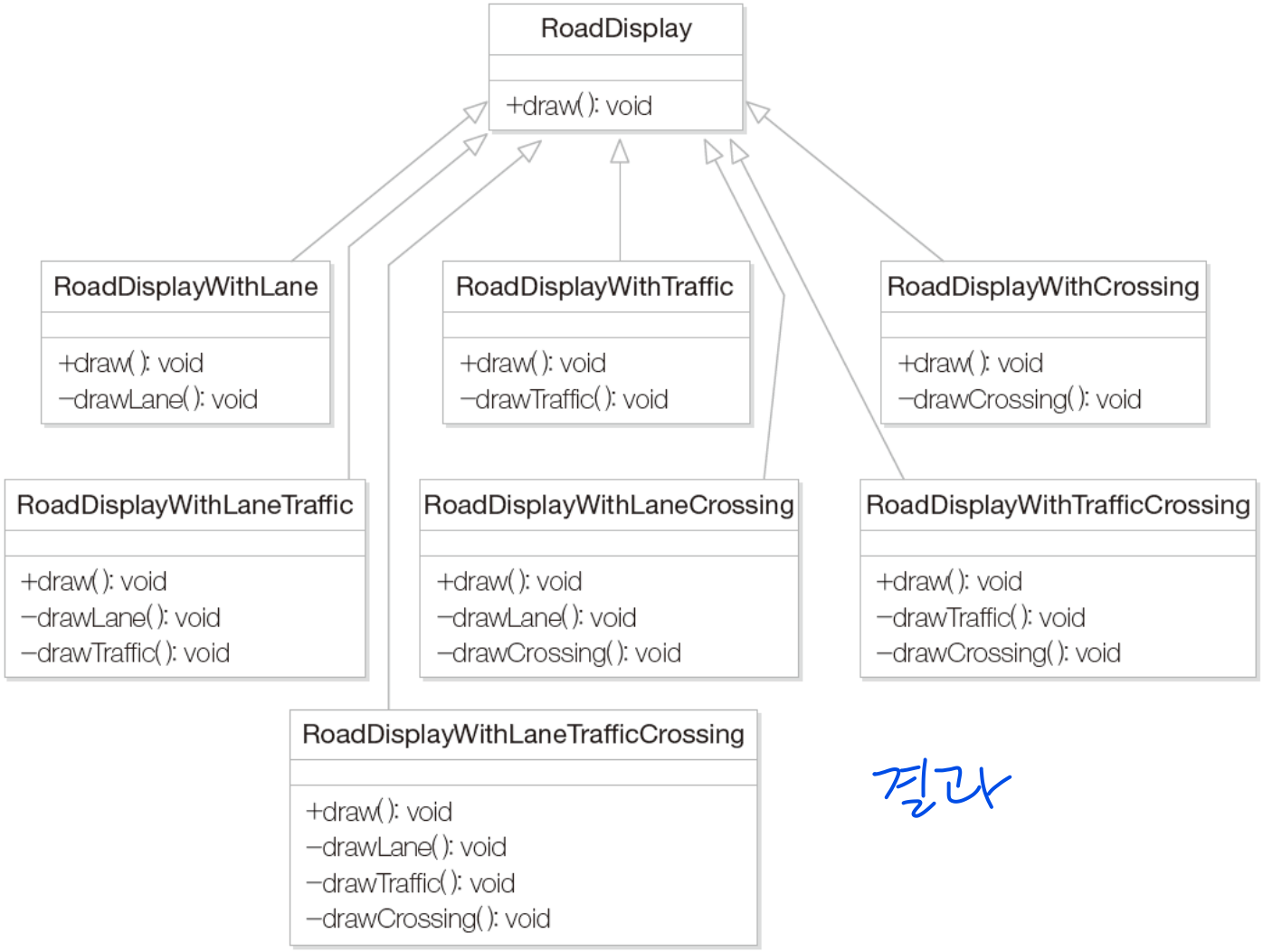
10.2.2 여러가지 추가 기능을 조합해야 하는 경우

경우	기본 기능 도로 표시	추가 기능			클래스 이름
		차선 표시	교통량 표시	교차로 표시	
1	√				RoadDisplay
2	√	√			RoadDisplayWithLane
3	√		√		RoadDisplayWithTraffic
4	√			√	RoadDisplayWithCrossing
5	√	√	√		RoadDisplayWithLaneTraffic
6	√	√		√	RoadDisplayWithLaneCrossing
7	√		√	√	RoadDisplayWithTrafficCrossing
8	√	√	√	√	RoadDisplayWithLaneTrafficCrossing

클래스 이름이 너무 많아진다

10.2.2 여러가지 추가 기능을 조합해야 하는 경우

그림 10-3 상속을 이용한 추가 기능 조합의 설계

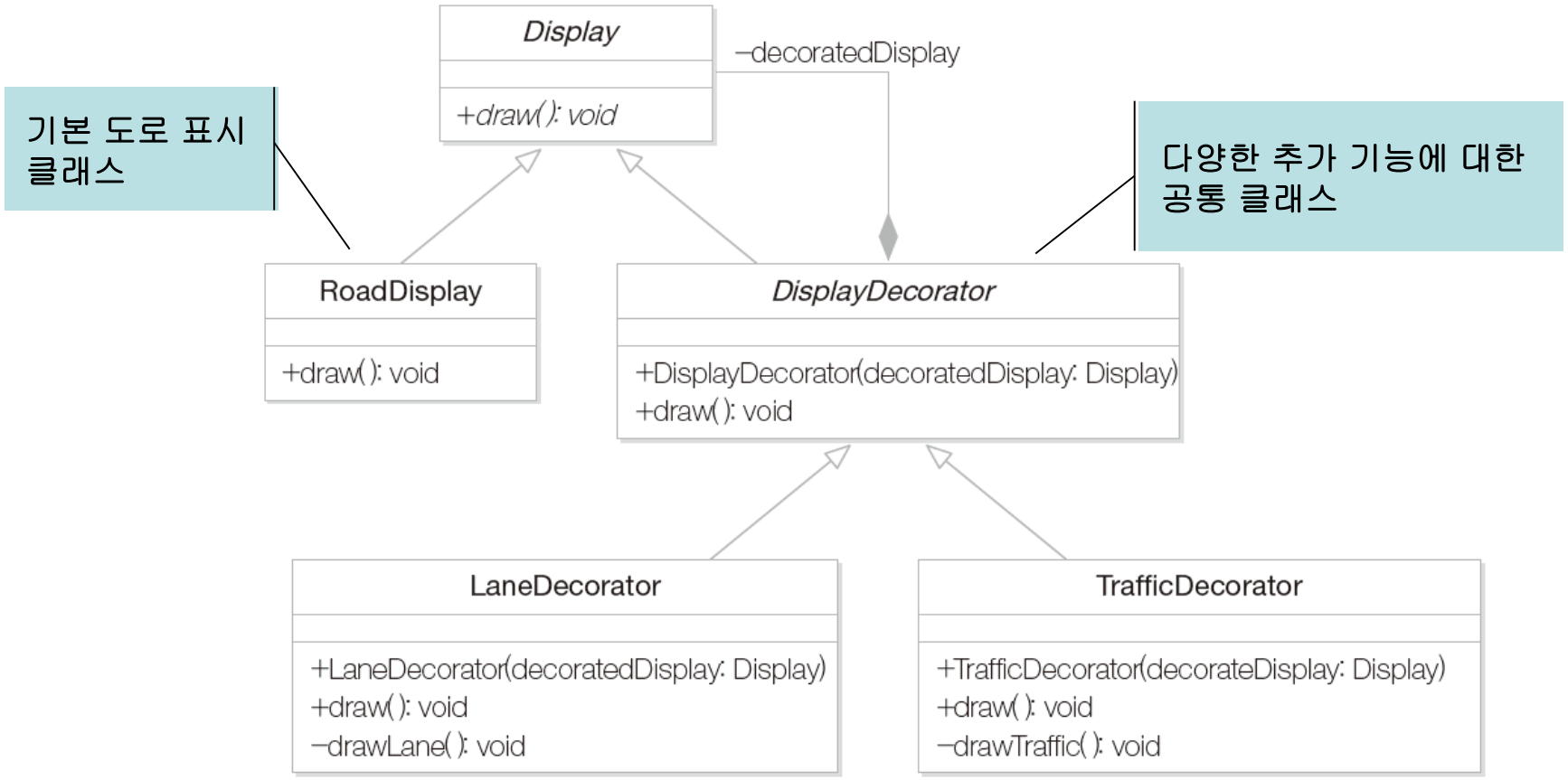


결과

10.3. 해결책

❖ 추가 기능 별로 개별적인 클래스를 설계하고 이를 조합

그림 10-4 개선된 추가 기능 조합의 설계



10.3. 해결책: 소스 코드

코드 10-3

```
public abstract class Display {
    public abstract void draw() ;
}
```

```
public class RoadDisplay extends Display {  // 기본 도로 표시 클래스
   public void draw() {
      System.out.println("도로 기본 표시") ;
   }
}
```

```
// 다양한 추가 기능에 대한 공통 클래스
public class DisplayDecorator extends Display {
   private Display decoratedDisplay ;
   public DisplayDecorator(Display decoratedDisplay) {
      this.decoratedDisplay = decoratedDisplay ;
   }
   public void draw() {
      decoratedDisplay.draw() ;
   }
}
```

10.3. 해결책: 소스 코드

코드 10-3

```
public class LaneDecorator extends DisplayDecorator {  // 차선표시를 축하는 클래스
   public LaneDecorator(Display decoratedDisplay) { // 기존 표시 클래스의 설정
      super(decoratedDisplay);
   }
   public void draw() {
      super.draw(); // 설정된 기존 표시 기능을 수행
      drawLane() ; // 추가적으로 차선을 표시
   }
   private void drawLane() { System.out.println("₩t차선 표시") ; }
}
```

```
public class TrafficDecorator extends DisplayDecorator { // 교통량 표시를 추가하는 클래스
   public TrafficDecorator(Display decoratedDisplay) { // 기존 표시 클래스의 설정
      super(decoratedDisplay);
   }
   public void draw() {
      super.draw();  // 설정된 기존 표시 기능을 수행
      drawTraffic() ; // 추가적으로 교통량을 표시
   }
   private void drawTraffic() { System.out.println("₩t교통량 표시") ;              }
}
```

10.3. 해결책: 소스 코드

코드 10-4

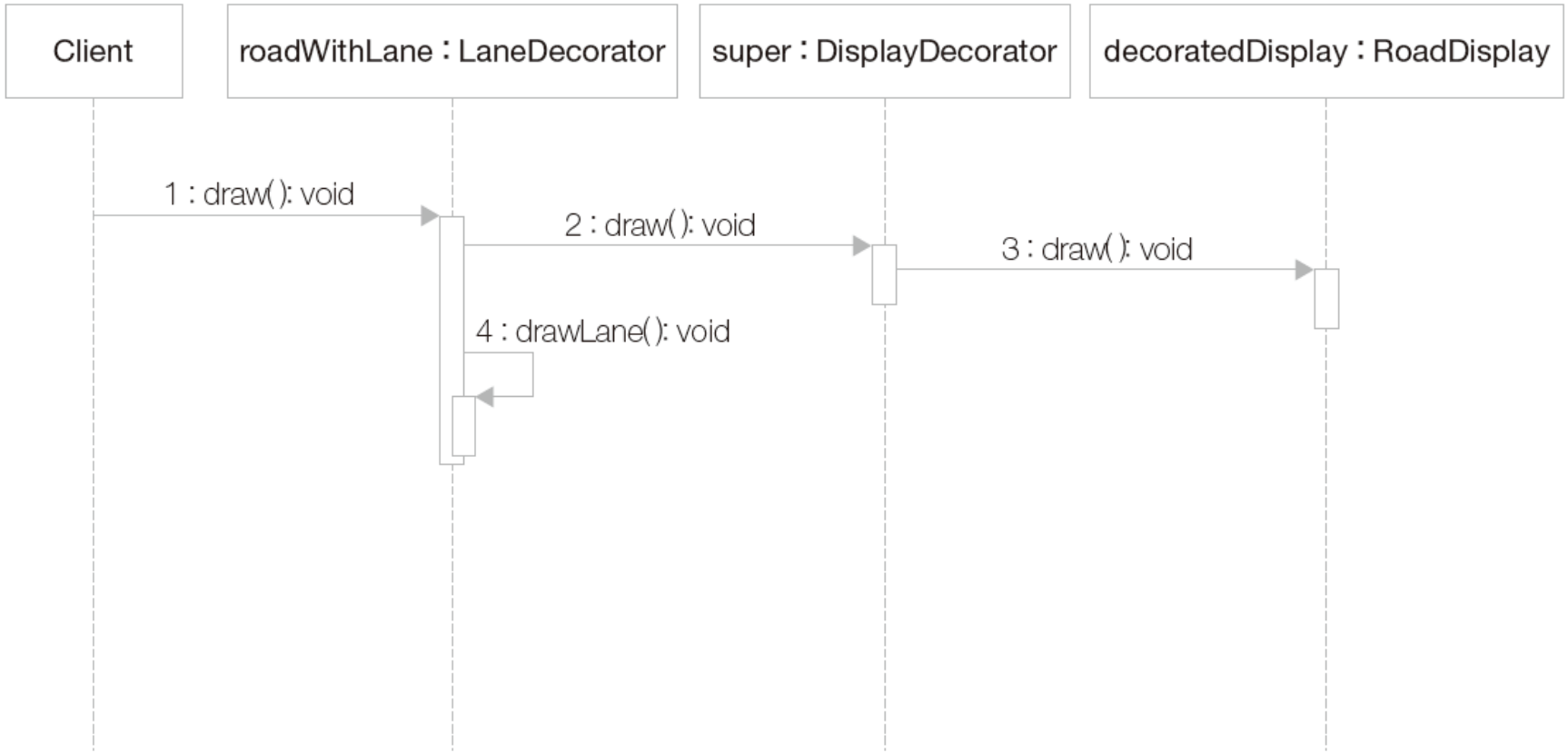
```
public class Client {
   public static void main(String[] args) {
      Display road = new RoadDisplay() ;
      road.draw() ;  // 기본 도로 표시

      Display roadWithLane = new LaneDecorator(new RoadDisplay()) ;
      roadWithLane.draw() ;  // 기본 도로 표시 + 차선 표시

      Display roadWithTraffic = new TrafficDecorator(new RoadDisplay()) ;
      roadWithTraffic.draw() ;  // 기본 도로 표시 + 교통량 표시
}
```

Client 클래스는 동일한 Display 클래스만을 통해서 일관성 있는 방식으로 도로 정보를 표시

```
도로 기본 표시
도로 기본 표시
        차선 표시
도로 기본 표시
        교통량 표시
```

10.3. 해결책

그림 10-5 roadWithLane 객체의 draw 메서드 동작



10.3. 해결책: 소스 코드

코드 10-5

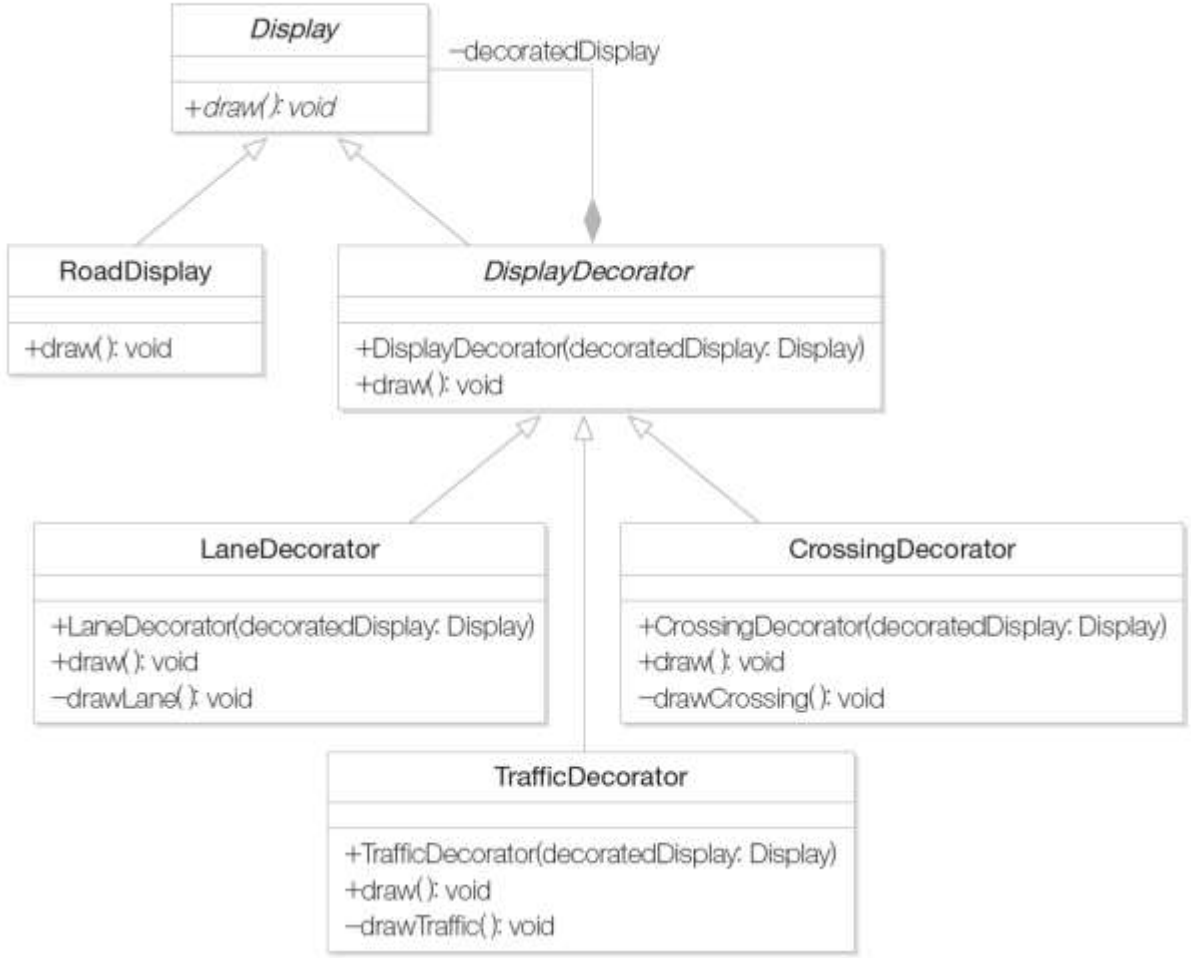
```
public class Client {
   public static void main(String[] args) {
   Display roadWithLaneAndTraffic =
      new TrafficDecorator(
         new LaneDecorator(
            new RoadDisplay())) ;
      roadWithLaneAndTraffic.draw() ;
   }
}
```

다른 방법

도로 기본 표시
차선 표시
교통량 표시

교차로 기능 추가

그림 10-6 LaneDecorator, TrafficDecorator, CrossingDecorator의 관계



교차로 기능 추가

코드 10-6

```
public class CrossingDecorator extends DisplayDecorator {
   public CrossingDecorator(Display decoratedDisplay) {
      super(decoratedDisplay);
   }
   public void draw() {
      super.draw();
      drawCrossing() ;
   }
   private void drawCrossing() {
      System.out.println("₩t횡단보도 표시") ;
   }
}
```

교차로 기능 추가

코드 10-7

```
public class Client {
   public static void main(String[] args) {
    Display roadWithCrossingAndTrafficAndLane
     = new CrossingDecorator(
           new TrafficDecorator(
              new LaneDecorator(
                 new RoadDisplay())));
   roadWithCrossingAndTrafficAndLane.draw() ;
   }
}
```

도로 기본 표시
차선 표시
교통량 표시
횡단보도 표시

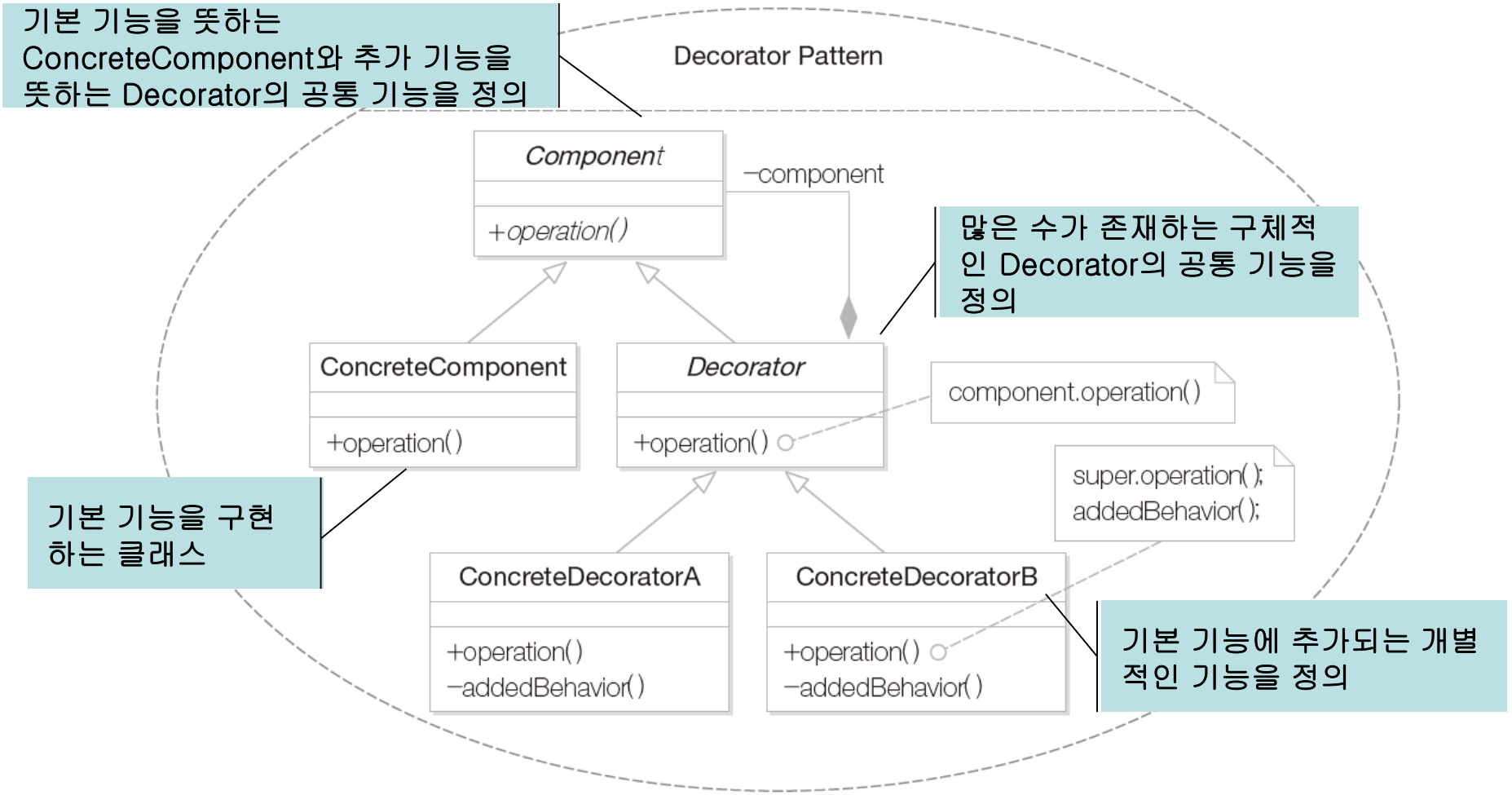
10.4 데커레이터 패턴

- 기본 기능에 추가할 수 있는 기능의 종류가 많은 경우

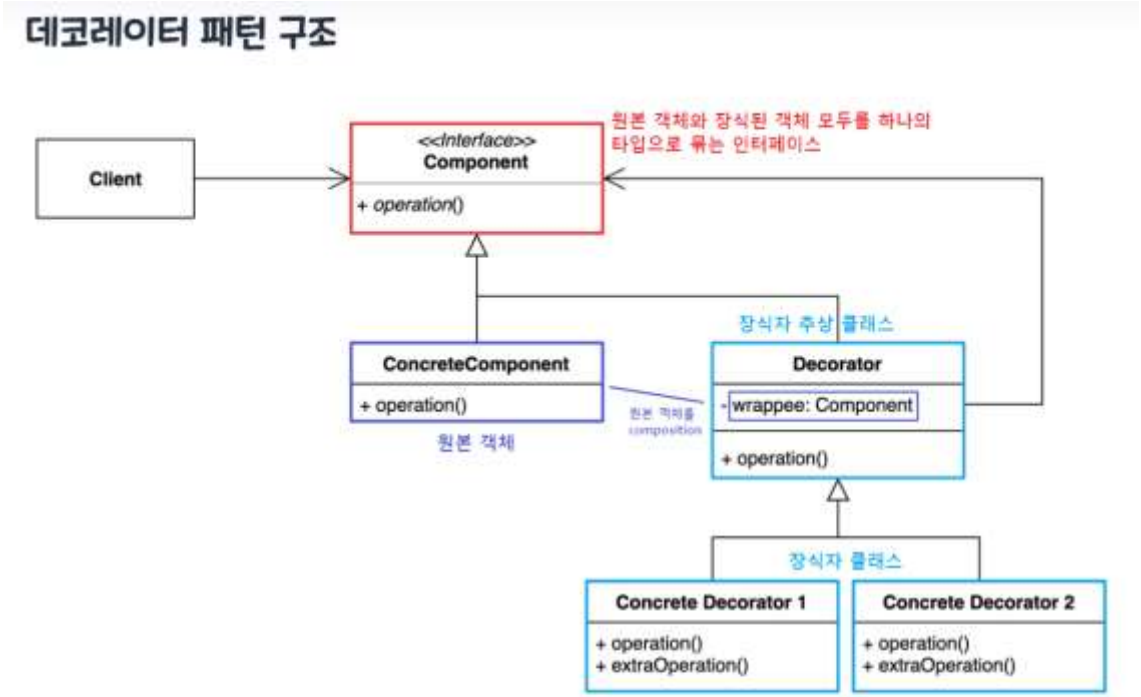
데코레이터 패턴은 기본 기능에 추가될 수 있는 많은 수의 부가 기능에 대해서 다양한 조합을 동적으로 구현할 수 있는 패턴이다.

10.4 데커레이터 패턴

그림 10-7 데커레이터 패턴의 컬레보레이션



10.4 데커레이터 패턴

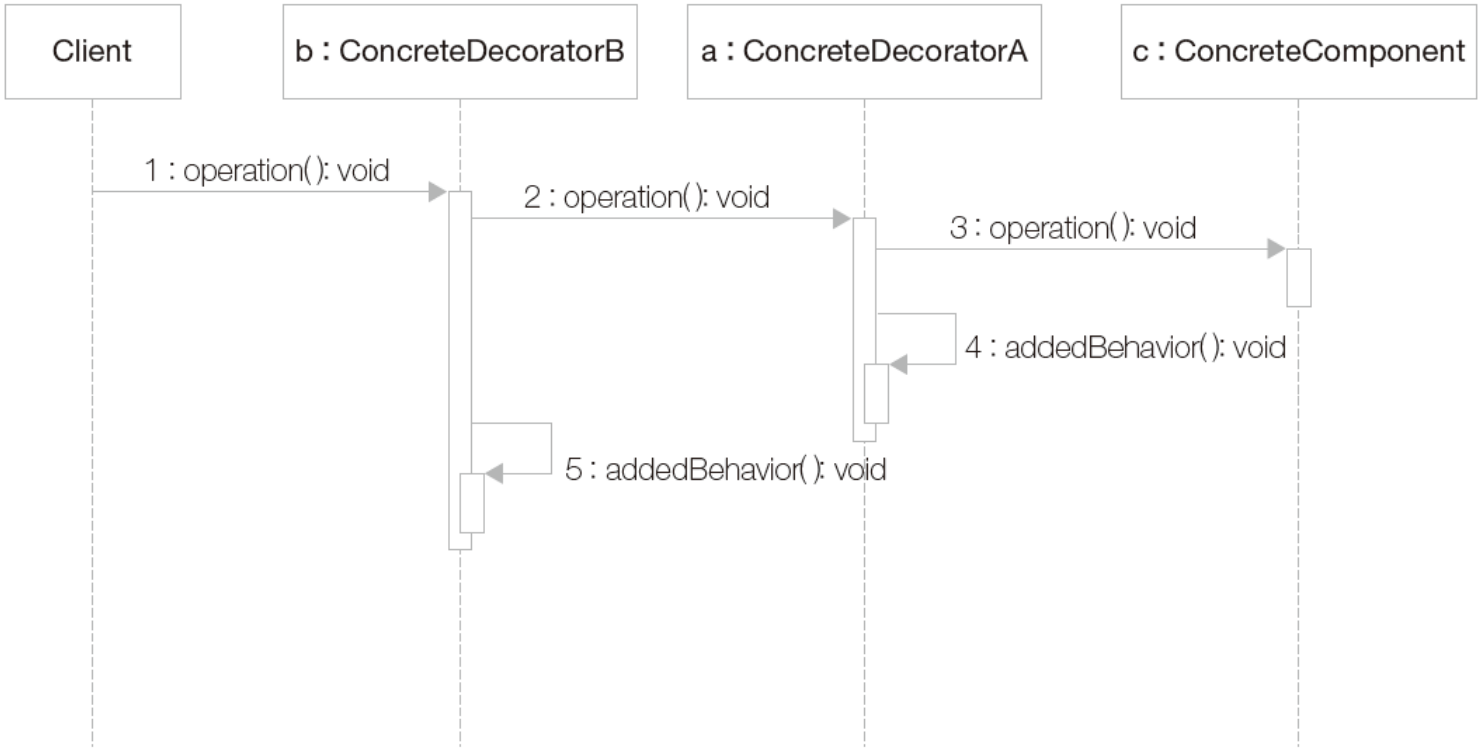


- Component (Interface) : 원본 객체와 장식된 객체 모두를 묶는 역할
- ConcreteComponent : 원본 객체 (데코레이팅 할 객체)
- Decorator : 추상화된 장식자 클래스
  - 원본 객체를 합성(composition)한 wrappee 필드와 인터페이스의 구현 메소드를 가지고 있다
- ConcreteDecorator : 구체적인 장식자 클래스
  - 부모 클래스가 감싸고 있는 하나의 Component를 호출하면서 호출 전/후로 부가적인 로직을 추가할 수 있다.

https://inpa.tistory.com/entry/GOF-%F0%9F%92%A0-데코레이터Decorator-패턴-제대로-배워보자/

9.4 데커레이터 패턴

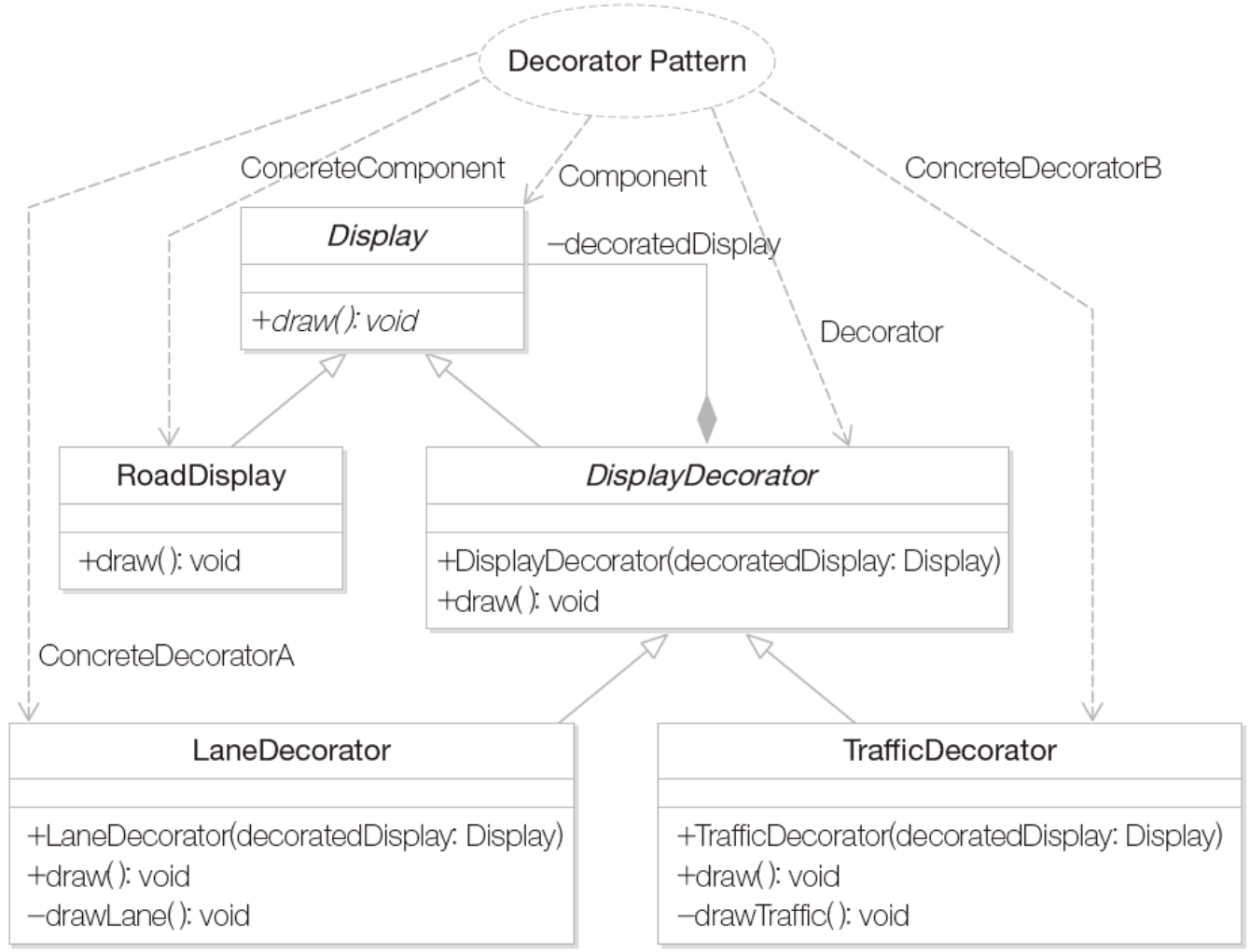
그림 10-8 데커레이터 패턴의 순차 다이어그램




```
Component c = new ConcreteComponent() ;
Decorator a = new ConcreteDecoratorA(c) ;
Decorator b = new ConcreteDecoratorB(a) ;
```

데커레이터 패턴의 적용

그림 10-9 데커레이터 패턴을 도로 표시 예제에 적용한 경우



또 다른 예시

- 피자가게
  - 본래 제공되는 피자 메뉴들
  - 이외에 고객들이 추가로 얹어달라고 요청하는 토핑들
  - 피자메뉴+토핑의 경우의 수만큼 클래스를 만들면 너무 많아질 것
  - 피자를 component로 두고, 토핑을 decorator로 만들어서 구현해보자!

*출처 – https://huisam.tistory.com/entry/DecoratorPattern

피자 추상 클래스

- 여기서 protected는 Pizza를 상속받는 클래스에서 description에 접근할 수 있도록 하기 위함(private는 상속을 통한 재사용성에는 제약이 있음)
  - 즉, protected는 상속과 접근성을 고려하여 사용됨

```
abstract class Pizza {
    protected String description = "Original Pizza";

    public String getDescription() {
        return description;
    }

    public abstract int getCost();
}
```

Concrete Component 클래스들

```
class CheesePizza extends Pizza {
    public CheesePizza() {
        this.description = "CheesePizza";
    }

    @Override
    public int getCost() {
        return 10000;
    }
}
```

```
class CombinationPizza extends Pizza {
    public CombinationPizza() {
        this.description = "CombinationPizza";
    }

    @Override
    public int getCost() {
        return 12000;
    }
}
```

Decorator

- ToppingDecorator의 파라미터가 final인 이유는 생성자에서 전달받은 Pizza 객체가 ToppingDecorator 객체 생성 후에도 변경되지 않도록 보장하기 위함임(불변성을 위함)

```
abstract class ToppingDecorator extends Pizza {
    protected Pizza pizza;

    ToppingDecorator(final Pizza pizza) {
        this.pizza = pizza;
    }

    public abstract String getDescription();
}
```

Concrete Decorator 클래스들

```
class FreshTomato extends ToppingDecorator {
    public FreshTomato(final Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.description + " with FreshTomato";
    }

    @Override
    public int getCost() {
        return 300 + pizza.getCost();
    }
}
```

Concrete Decorator 클래스들

```
class Olive extends ToppingDecorator {
    public Olive(final Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.description + " with Olive";
    }

    @Override
    public int getCost() {
        return 400 + pizza.getCost();
    }
}
```

Client 클래스

```
public class PizzaOrder {
    public static void main(String[] args) {
        // 기본 CheesePizza 생성
        Pizza cheesePizza = new CheesePizza();
        System.out.println(cheesePizza.getDescription() + " Cost: " + cheesePizza.getCost());

        // CheesePizza에 FreshTomato 토핑 추가
        Pizza cheeseWithTomato = new FreshTomato(cheesePizza);
        System.out.println(cheeseWithTomato.getDescription() + " Cost: " + cheeseWithTomato.getCost());

        // CheesePizza에 FreshTomato와 Olive 토핑 추가
        Pizza cheeseWithTomatoAndOlive = new Olive(cheeseWithTomato);
        System.out.println(cheeseWithTomatoAndOlive.getDescription() + " Cost: " + cheeseWithTomatoAndOlive.getCost());

        // CombinationPizza에 Olive 토핑 추가
        Pizza combinationWithOlive = new Olive(new CombinationPizza());
        System.out.println(combinationWithOlive.getDescription() + " Cost: " + combinationWithOlive.getCost());
    }
}
```

스트래티지 패턴과 데커레이터 패턴의 차이

- Strategy 패턴목적: 객체의 행동을 동적으로 변경하기 위해 사용됨. 특정 알고리즘 또는 행동을 캡슐화하고, 실행 시점에 객체가 그 중 하나를 선택하게 함.
- 구조:
  - Context: 전략을 사용하는 주체 역할. 인터페이스만 알면 되며, 세부 알고리즘에 대해 몰라도 됨
  - Strategy 인터페이스: 구현할 행동(알고리즘)의 인터페이스를 정의
  - Concrete Strategy: 실제 알고리즘을 구현한 클래스들
- 특징:
  - 주로 행동(알고리즘)을 캡슐화하여 바꿔치기할 수 있음
  - 클래스를 변경하지 않고 객체의 행동을 바꾸는 것이 핵심
  - 예: 결제 수단을 선택하는 PaymentStrategy, 정렬 방식이 다른 SortStrategy 등

스트래티지 패턴과 데커레이터 패턴의 차이

- 목적: 객체에 새로운 기능을 추가할 때 사용. 기능을 추가하되 원본 객체를 수정하지 않고, 여러 데코레이터를 조합하여 기능을 확장할 수 있음
- 구조:
  - Component 인터페이스: 공통 기능을 정의하는 기본 인터페이스
  - Concrete Component: 실제 기능을 구현하는 클래스
  - Decorator: 기능을 추가하는 래퍼 클래스. 기본 기능을 가지며, 원하는 추가 기능을 포함.
  - Concrete Decorator: 실제로 기능을 추가하는 구현 클래스
- 특징:
  - 동적으로 객체의 기능을 추가할 수 있으며, 객체를 계층 구조로 감쌀 수 있음
  - 주로 기능을 확장하거나 추가할 때 사용됨
  - 예: 커피에 첨가물을 추가하는 CoffeeDecorator 또는 UI 컴포넌트에 스타일을 추가하는 BorderDecorator 등.

스트래티지 패턴과 데커레이터 패턴의 차이

- 차이점 요약

패턴	Strategy Pattern	Decorator Pattern
목적	행동(알고리즘) 선택 및 변경	객체에 기능 추가
구조	Context + 여러 Strategy	기본 객체 + 여러 Decorator
사용 예	정렬 방식 선택, 결제 방식 선택	커피에 재료 추가, UI 스타일링

- 이 두 패턴의 핵심 차이는 전략을 바꿀지 아니면 기능을 추가할지에 있음

데커레이터 패턴의 장점

- 객체에 동적으로 기능 추가
  - 기존 클래스를 수정하지 않고도 기능을 추가할 수 있음. 객체를 여러 데코레이터로 감싸써, 실행 중에 동적으로 기능을 확장할 수 있음
- 유연한 기능 조합
  - 데코레이터를 중첩하여 여러 기능을 조합할 수 있음. 예를 들어, 커피 객체에 MilkDecorator와 SugarDecorator를 순차적으로 추가하면, 유연하게 "우유와 설탕이 들어간 커피"를 표현할 수 있음
- 단일 책임 원칙(Single Responsibility Principle) 준수
  - 기능이 추가될 때마다 새로운 클래스를 만들지 않고, 데코레이터를 통해 독립적으로 기능 분리 가능. 각 데코레이터는 한 가지 기능에 집중하게 되므로, 유지보수가 용이해짐
- 기존 코드 수정 없이 확장 가능
  - 클라이언트 코드에서 원래 객체와 데코레이터 객체 모두 동일한 인터페이스를 사용하므로, 클라이언트 코드를 수정할 필요 없이 새로운 데코레이터를 추가할 수 있음

데커레이터 패턴의 단점

- 객체 생성이 복잡해질 수 있음
  - 데코레이터가 많아지면 객체의 계층 구조가 깊어져 복잡해질 수 있음. 여러 데코레이터를 중첩하면, 디버깅이나 유지보수가 어려워질 수 있음
- 데코레이터 관리가 어려움
  - 데코레이터를 여러 개 사용할 경우, 각 데코레이터의 적용 순서에 따라 최종 결과가 달라질 수 있음. 모든 데코레이터의 조합을 고려해야 하므로, 체계적인 관리가 필요함
- 객체 식별의 어려움
  - 데코레이터를 여러 번 감싼 경우, 원래 객체를 식별하기 어려울 수 있음. 특정 데코레이터를 찾거나 제거하려면 복잡한 로직이 필요할 수 있음
- 많은 수의 작은 클래스 생성
  - 기능이 증가할 때마다 새로운 데코레이터 클래스를 생성해야 하므로, 클래스 수가 많아질 수 있음. 클래스가 너무 많으면 전체 코드베이스가 복잡해지고, 프로젝트 구조가 복잡해질 수 있음

과제2

- 커맨드패턴과 옵서버패턴의 사례를 각 1개씩 생각하여 코드로 구현하고 제출
- 기한 : 11월 17일(일) 밤 12시까지
  - 앞으로 남은 패턴이 생각보다 많아 매주 과제가 부여될 수 있습니다.