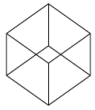
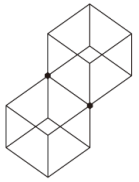
# 10. 데커레이터 패턴



JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



# 학습목표

학습목표

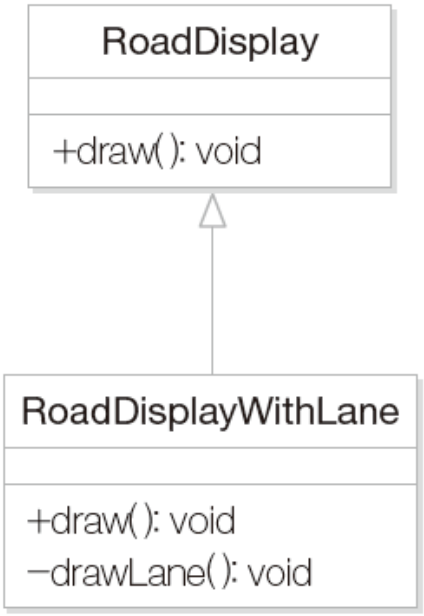
- 독립적인 추가 기능의 조합 방법 이해하기
- 데커레이터 패턴을 통한 기능의 조합 방법 이해하기
- 사례 연구를 통한 데커레이터 패턴의 핵심 특징 이해하기

# 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
   }
}
```

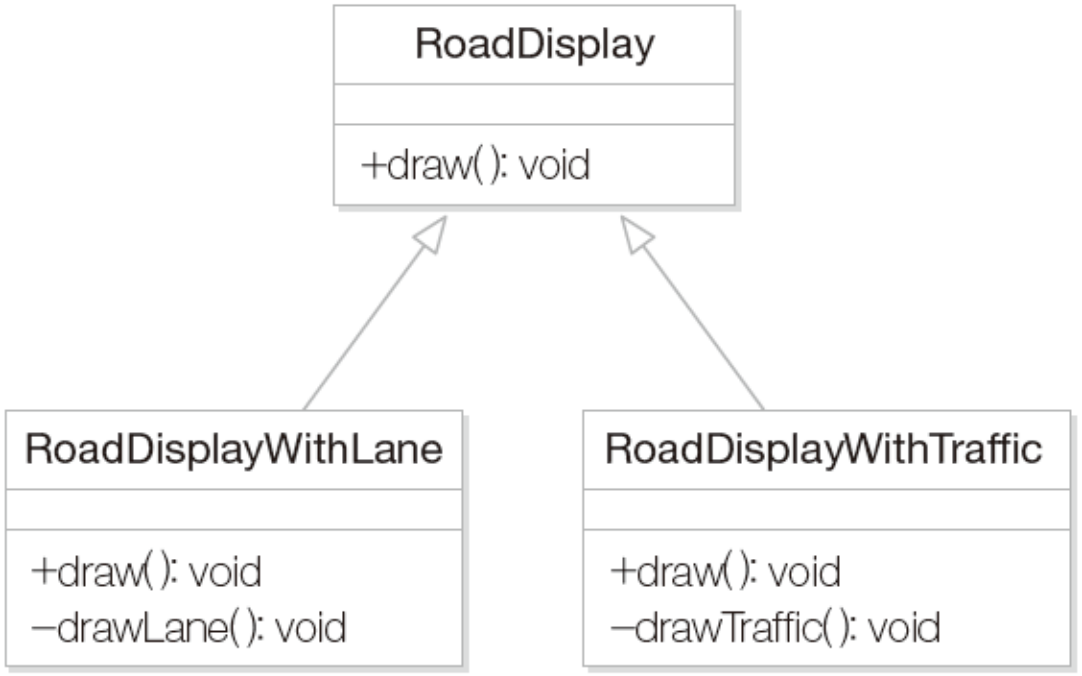
# 10.2 문제점

- 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 도로 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

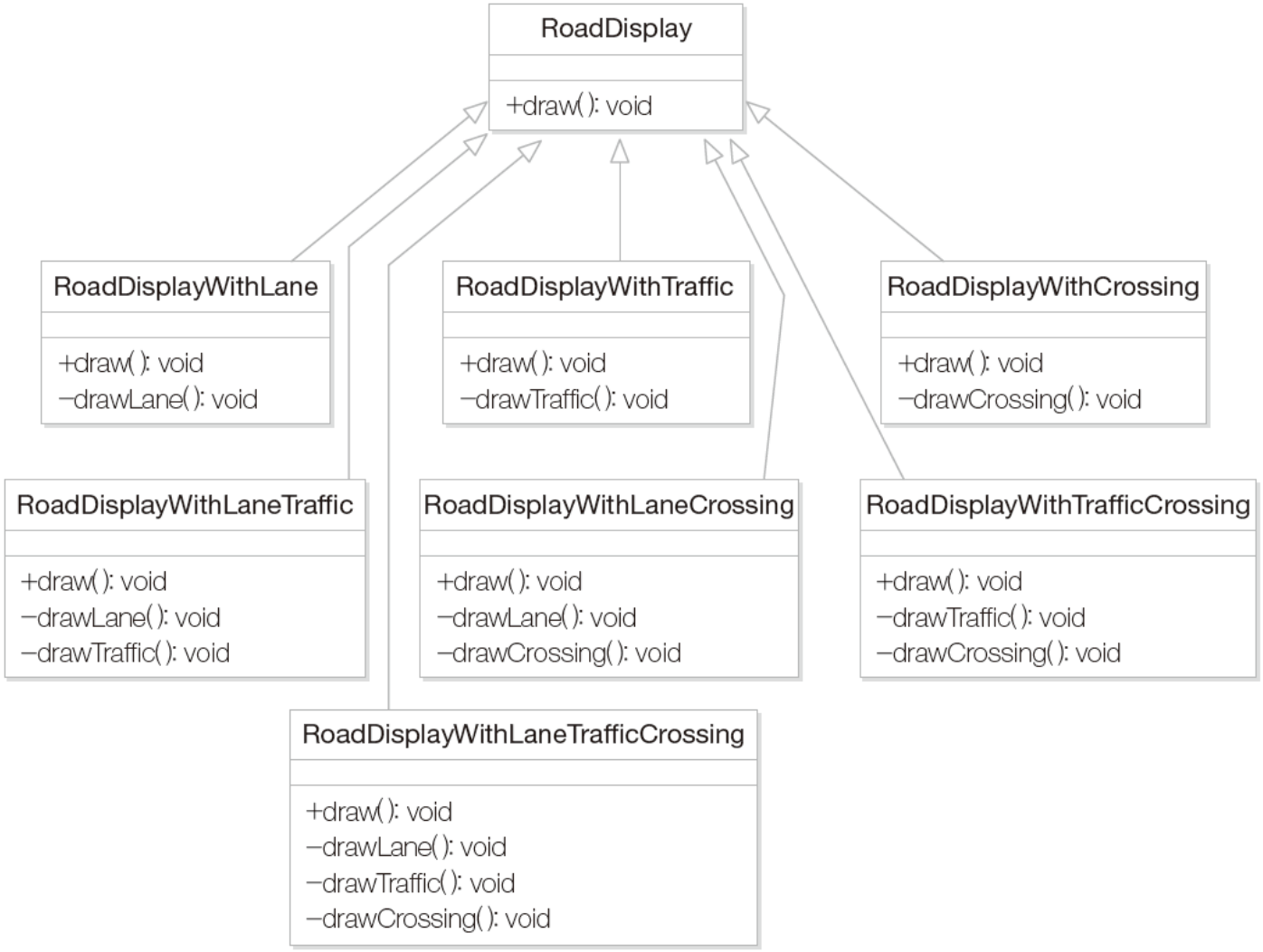
```
public class RoadDisplayWithTraffic extends RoadDisplay {
   public void draw() {
      super.draw();
      drawTraffic();
   }
   private void drawTraffic() {
      System.out.println("교통량 표시") ;
   }
}
```

# 10.2.2 여러가지 추가 기능을 조합해야 하는 경우

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

# 10.2.2 여러가지 추가 기능을 조합해야 하는 경우

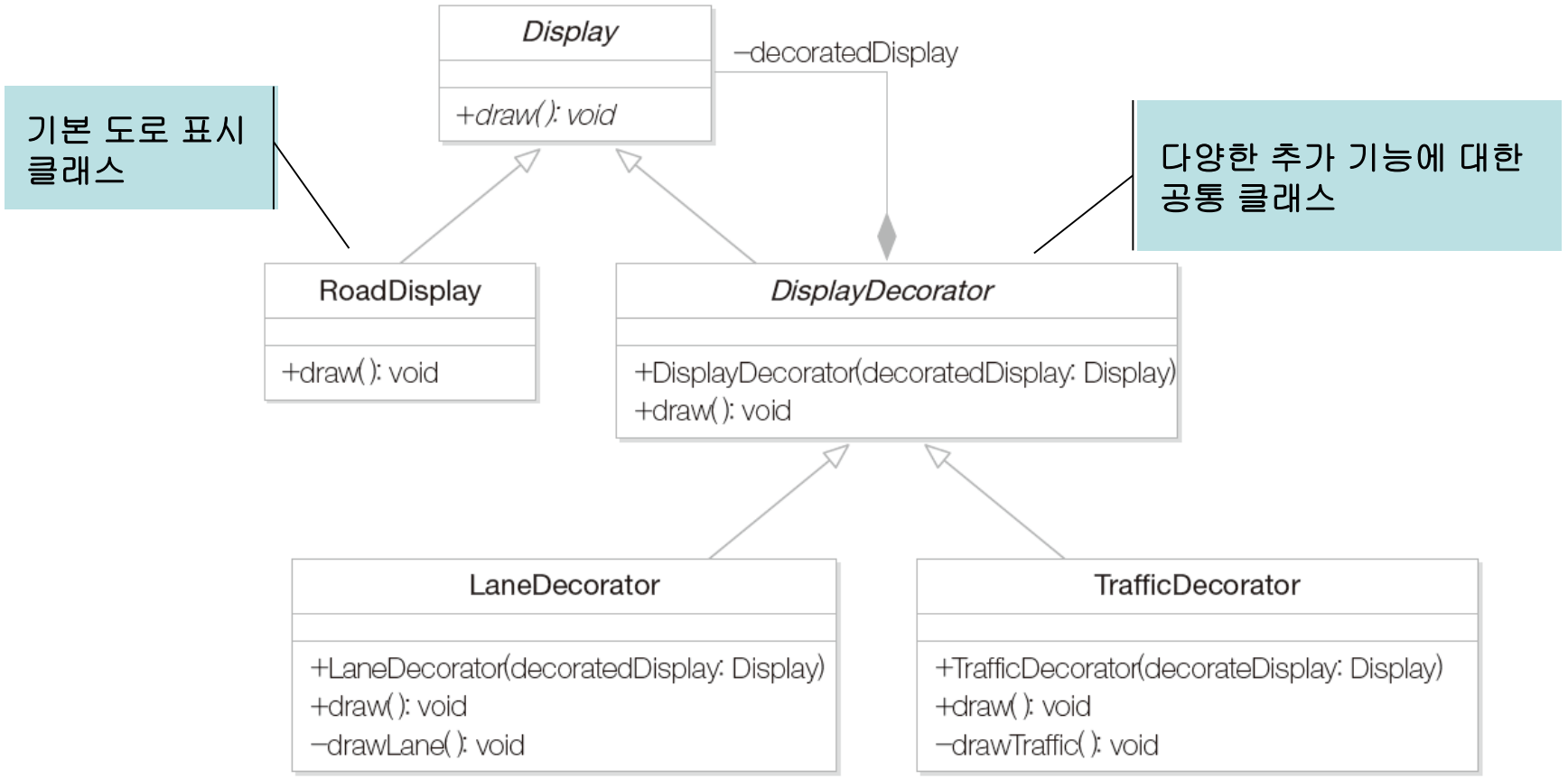
그림 10-3 상속을 이용한 추가 기능 조합의 설계



# 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

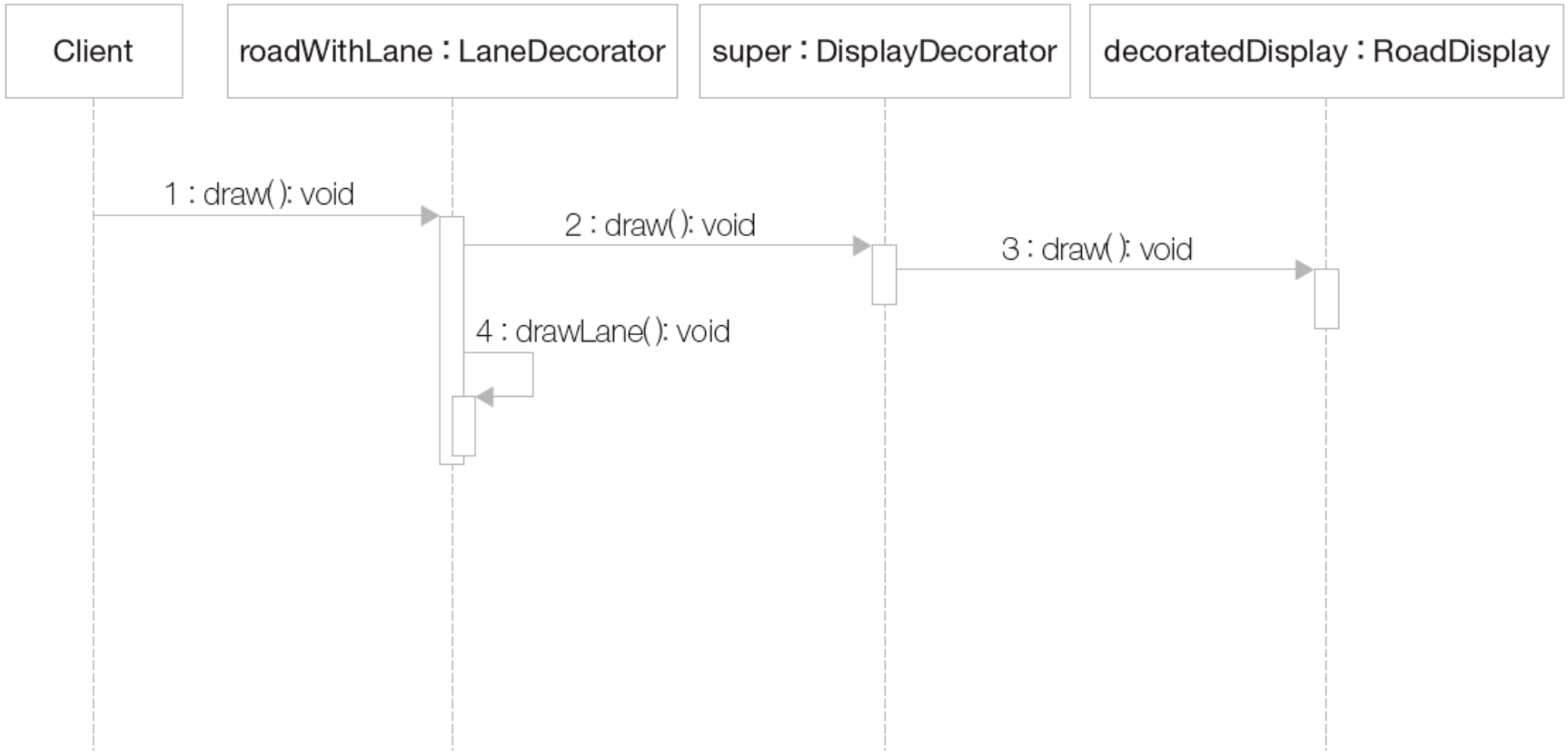
```
public class Client {
   public static void main(String[] args) {
      Display road = new RoadDisplay() ;
      road.draw() ;  // 기본 도로 표시

      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

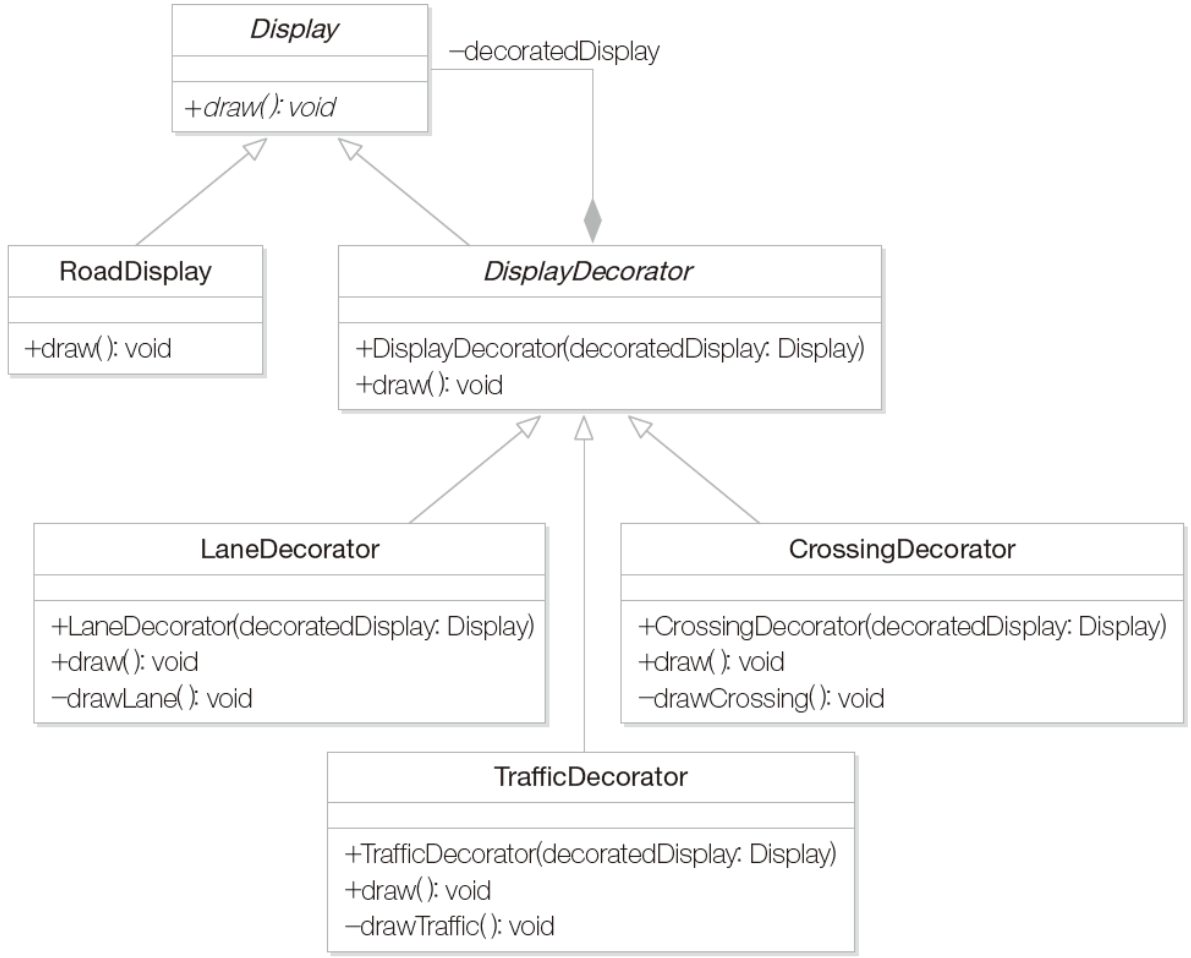
코드 10-5

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

```
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 roadWithLaneAndTraffic =
      new LaneDecorator(
         new TrafficDecorator(
            new CrossingDecorator(
               new RoadDisplay())));
      roadWithCrossingAndTrafficAndLane.draw() ;
   }
}
```

```
도로 기본 표시
            횡단보도 표시
            교통량 표시
            차선 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

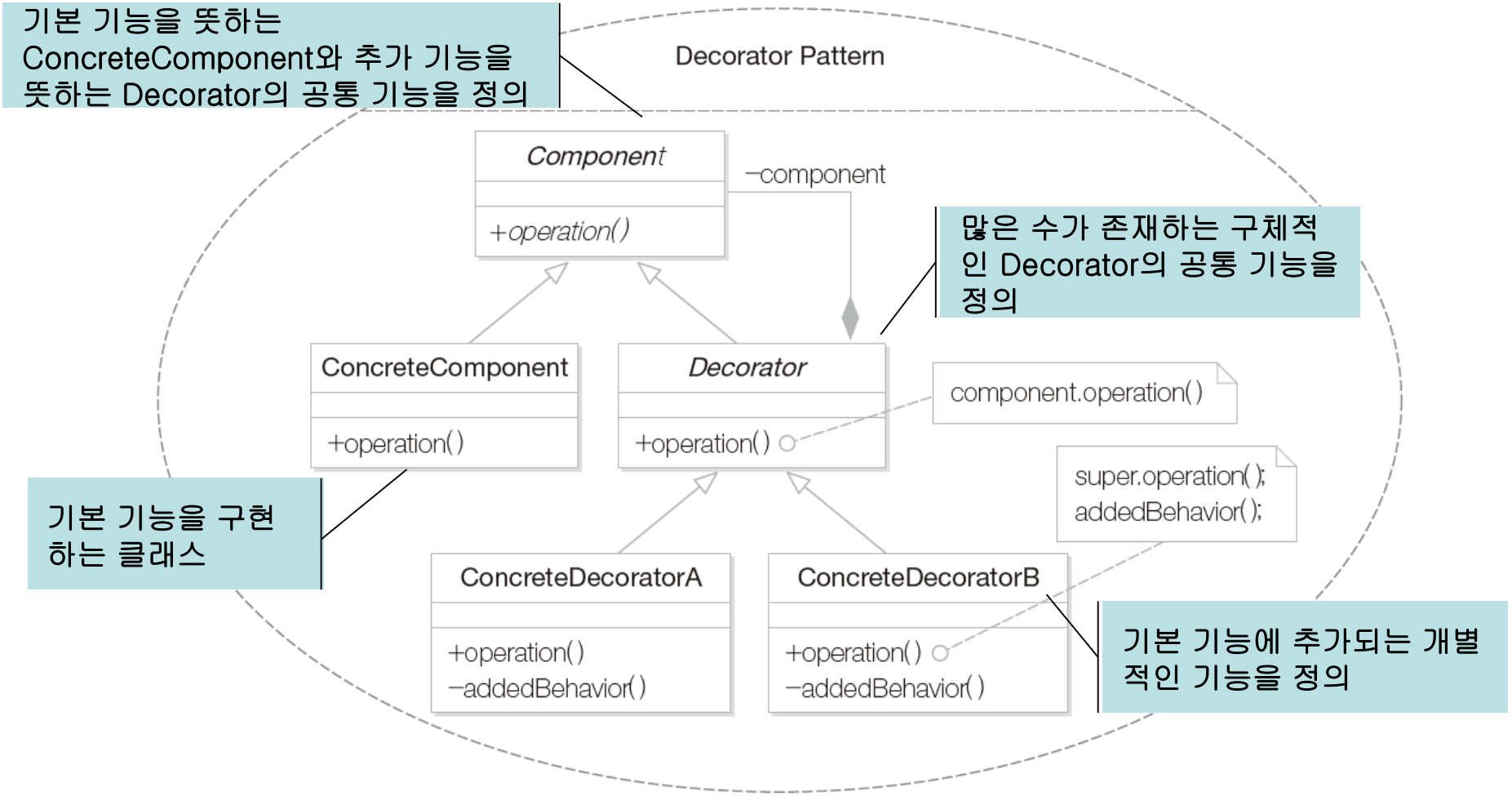
코드 10-8

```
public class Client {
   public static void main(String[] args) {
      Display road = new RoadDisplay() ;
      for ( String decoratorName : args ) {
         if ( decoratorName.equalsIgnoreCase("Lane"))
            road = new LaneDecorator(road) ;
         if ( decoratorName.equalsIgnoreCase("Traffic"))
            road = new TrafficDecorator(road) ;
         if ( decoratorName.equalsIgnoreCase("Crossing"))
            road = new CrossingDecorator(road) ;
      }
      road.draw() ;
   }
}
```

| 인자 예 | Traffic Lane | Crossing Lane Traffic |
|---|---|---|
| 실행 결과 | 도로 기본 표시<br>교통량 표시<br>차선 표시 | 도로 기본 표시<br>횡단보도 표시<br>차선 표시<br>교통량 표시 |

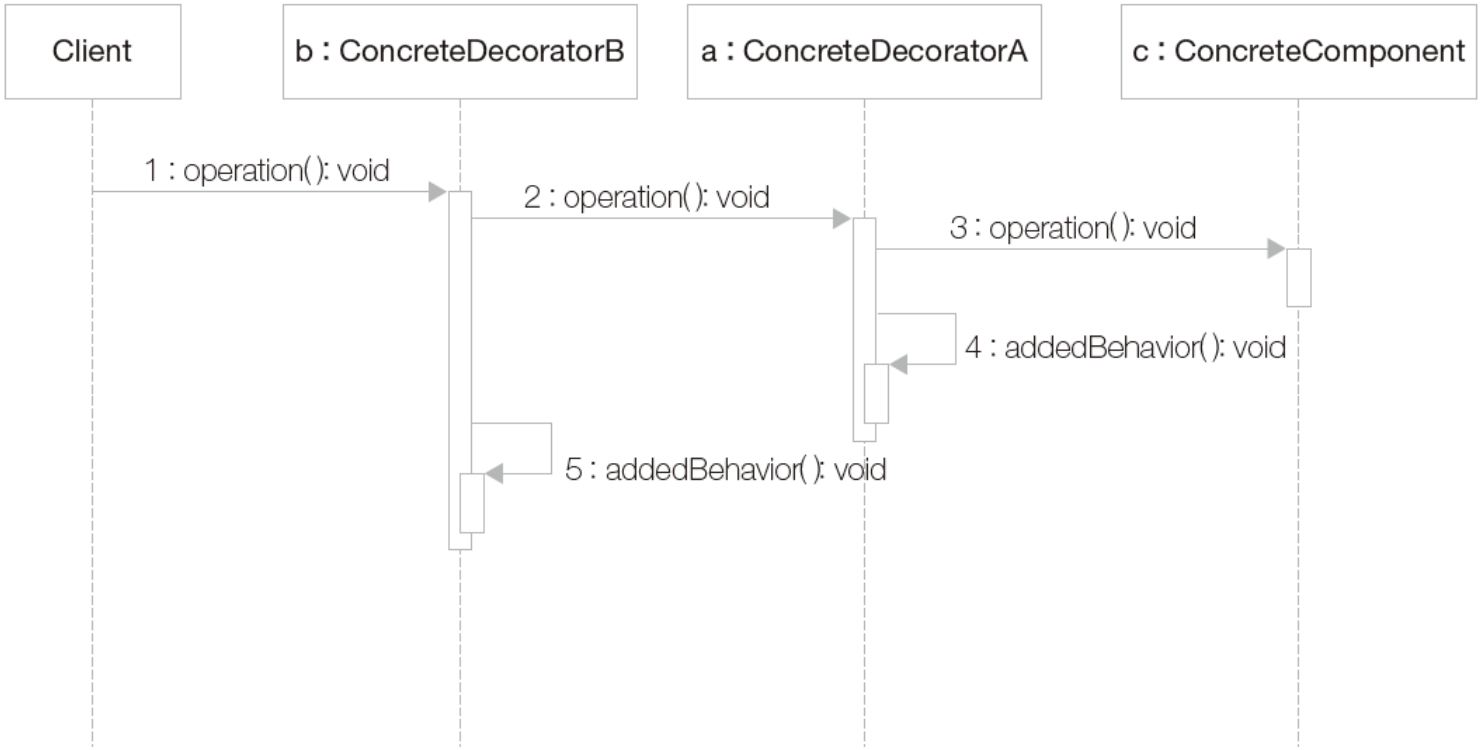
# 10.4 데커레이터 패턴

그림 10-7 데커레이터 패턴의 컬레보레이션



# 9.4 데커레이터 패턴

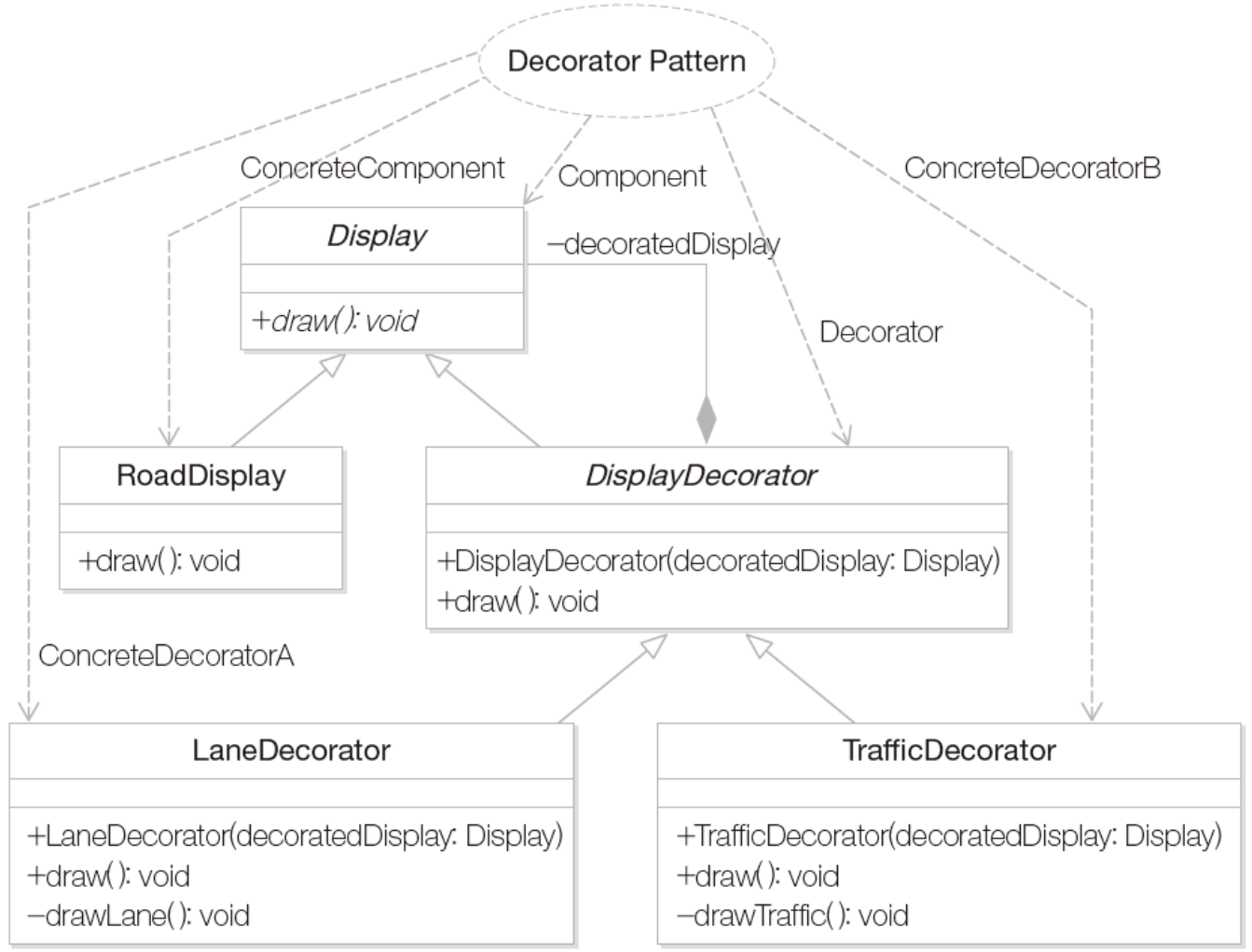
그림 10-8 데커레이터 패턴의 순차 다이어그램




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

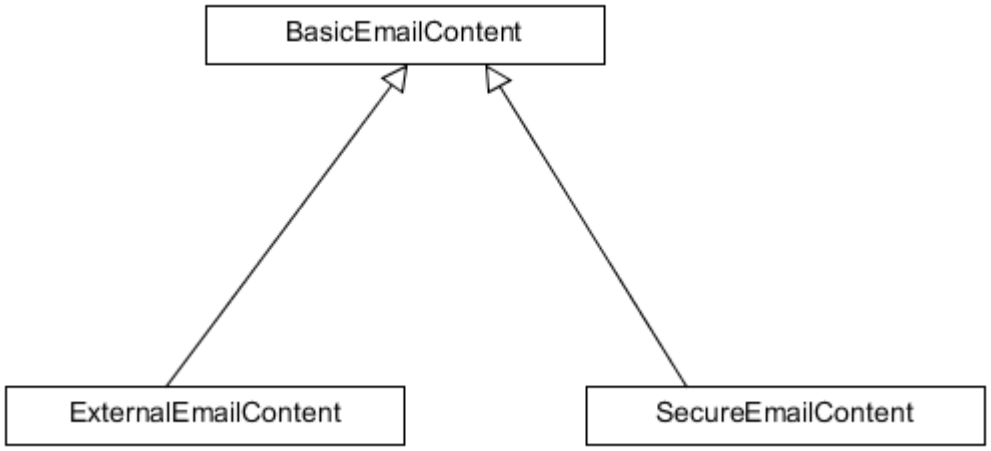
# 데커레이터 패턴의 적용

그림 10-9 데커레이터 패턴을 도로 표시 예제에 적용한 경우



# 실습

- 다음 이메일 내용과 관련된 기능들을 구현해보자(10장 연습문제 1번)
  - 단순하게 문자열로 이메일 내용 저장
  - 이메일 내용에 책임제한조항(disclaimer)을 추가하는 기능
  - 이메일 내용을 암호화는 기능
- 상속을 이용한 설계



# BasicEmailContent

```
public class BasicEmailContent {
    private String content;

    public BasicEmailContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

# ExternalEmailConten

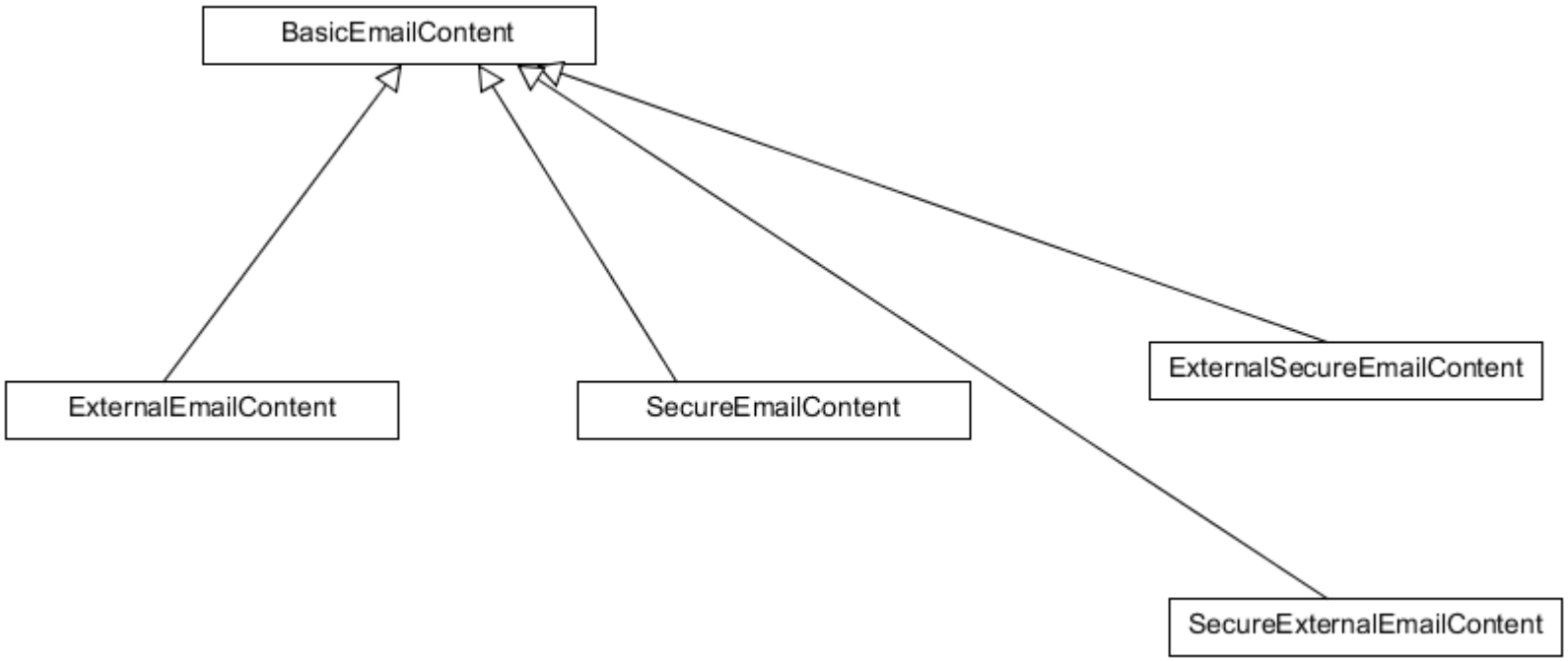
```
public class ExternalEmailContent extends BasicEmailContent {
    public ExternalEmailContent(String content) {
        super(content);
    }

    public String getContent() {
        String content = super.getContent();
        String externalContent = addDisclaimer(content);
        return externalContent;
    }

    private String addDisclaimer(String content) {
        return content + "Company Disclaimer";
    }
}
```

# SecureEmailContent

```
public class SecureEmailContent extends BasicEmailContent {
    public SecureEmailContent(String content) {
        super(content);
    }

    public String getContent() {
        String content = super.getContent();
        String encryptedContent = encrypt(content);
        return encryptedContent;
    }

    private String encrypt(String content) {
        return content + "Encrypted";
    }
}
```

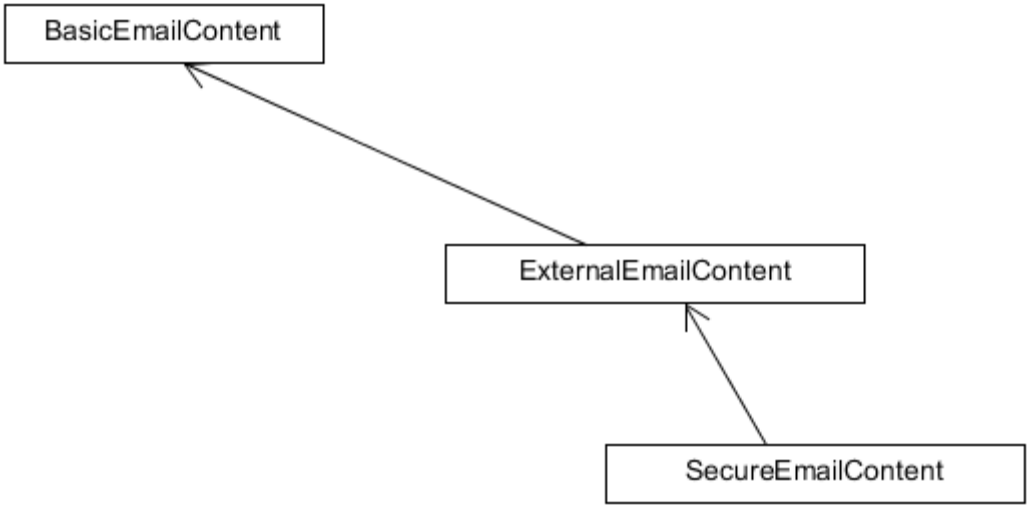
# 문제점

- 기능을 재사용하기 위해 상속을 이용한 설계
  - 코드의 중복
  - 이미 있는 기능들을 조합하여 제공하더라도 새로운 클래스 생성 필요



# 문제 해결

❖ 기능을 재사용하기 위해 상속 대신 연관관계 사용



# BasicEmailContent

```
public class BasicEmailContent {
    private String content;

    public BasicEmailContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

# ExternalEmailContent

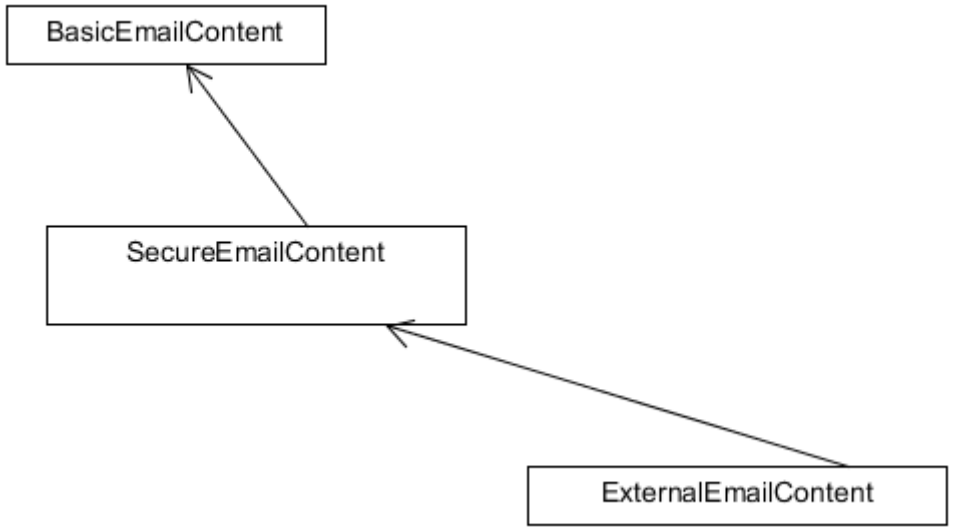
```
public class ExternalEmailContent  {
    private BasicEmailContent basicEmailContent;

    public ExternalEmailContent(BasicEmailContent basicEmailContent) {
        this.basicEmailContent = basicEmailContent;
    }

    public String getContent() {
        String content = basicEmailContent.getContent();
        String externalContent = addDisclaimer(content);
        return externalContent;
    }

    private String addDisclaimer(String content) {
        return content + "Company Disclaimer";
    }
}
```

# SecureEmailContent

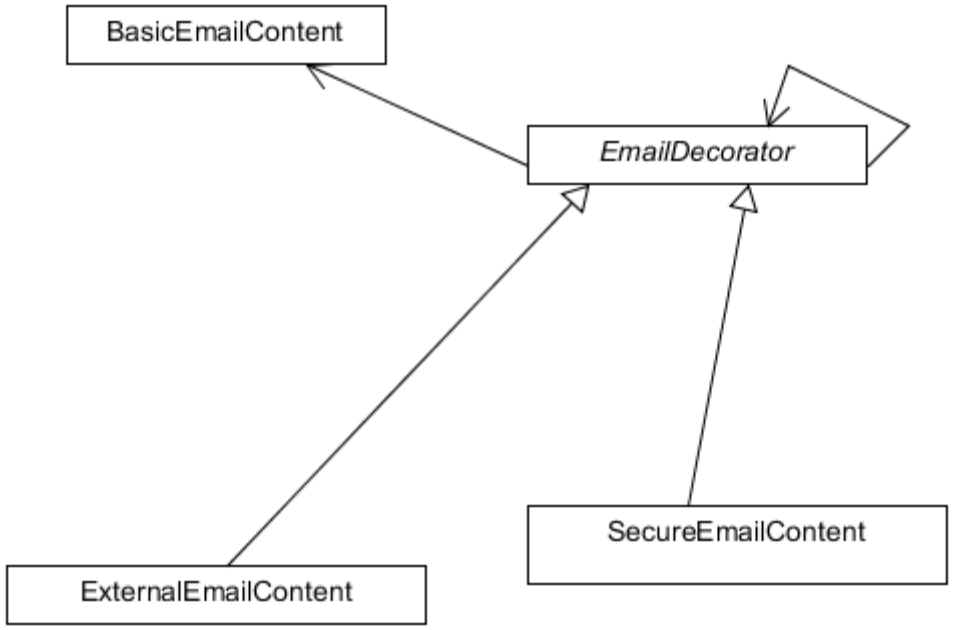
```
public class SecureEmailContent  {
    private ExternalEmailContent externalEmailContent;

    public SecureEmailContent(ExternalEmailContent externalEmailContent) {
        this.externalEmailContent = externalEmailContent;
    }

    public String getContent() {
        String content = externalEmailContent.getContent();
        String encryptedContent = encrypt(content);
        return encryptedContent;
    }

    private String encrypt(String content) {
        return content + "Encrypted";
    }
}
```

# 문제점

- 새로운 기능의 조합에 대해 기존의 코드를 변경해야 한다.



# 새로운 설계


BasicEmailContent
EmailDecorator
ExternalEmailContent
SecureEmailContent

# BasicEmailContent

```
public class BasicEmailContent {
    private String content;

    public BasicEmailContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

# EmailDecorator

```
public abstract class EmailDecorator {
   private EBasicEmailContent basicEmailContent;
   private EmailDecorator decorated;

    public ExternalEmailContent(BasicEmailContent basicEmailContent,
EmailDecorator decorated) {
        this.basicEmailContent= basicEmailContent;
        this.decorated = decorated;
    }

    public String getContent() {
        if (basicEmailContent!=null)
            return basicEmailContent.getContent();
        else return decorated.getContent();
    }

}
```

# ExternalEmailContent

```
public class ExternalEmailContent extends EmailDecorator {

    public ExternalEmailContent(BasicEmailContent basicEmailContent,
EmailDecorator decorated) {
        super(basicEmailContent, decorated);
    }

    public String getContent() {
        String content = super.getContent();
        String externalContent = addDisclaimer(content);
        return externalContent;
    }

    private String addDisclaimer(String content) {
        return content + "Company Disclaimer";
    }
}
```

# SecureEmailContent

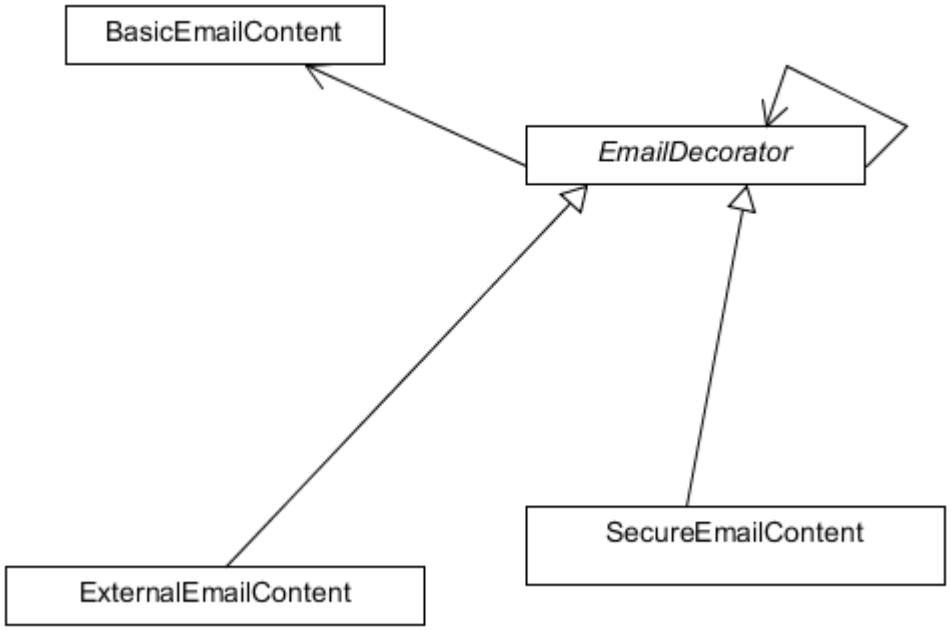
```
public class SecureEmailContent extends EmailDecorator {

    public SecureEmailContent(BasicEmailContent basicEmailContent,
ExternalEmailContent decorated) {
        super(basicEmailContent, decorated);
    }

    public String getContent() {
        String content = super.getContent();
        String encryptedContent = encrypt(content);
        return encryptedContent;
    }

    private String encrypt(String content) {
        return content + "Encrypted";
    }
}
```

# Main

```
public class Main {
    public static void main(String[] args) {
        BasicEmailContent b = new BasicEmailContent("Hello");
        ExternalEmailContent e = new ExternalEmailContent(b, null);
        SecureEmailContent content = new SecureEmailContent(null, e);
        System.out.println(content.getContent());
    }
}
```

# 문제점

- 동시에 2종류의 연관관계 존재
  - 부가기능에서 기본 기능으로의 연관관계
  - 부가가능간의 연관 관계



# 테코레이터 패턴

코드는 E-class에서 다운로드 가능