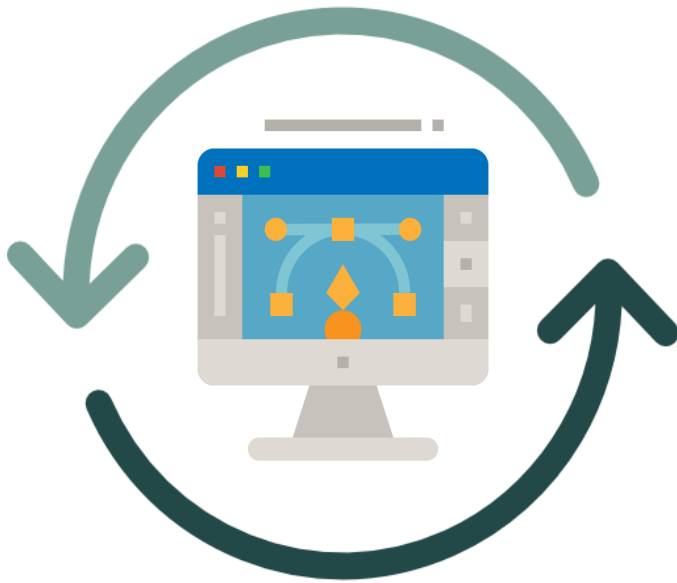


Design Patterns

주요 디자인 패턴

1 디자인 패턴이란?

1 디자인 패턴이란?



객체 지향 설계

재설계를 최소화하면서 요구 사항의 변화를 수용할 수 있도록 만들어 준다.

객체 지향 설계를 하다보면 ...

이전과 비슷한 상황에서 사용했던 설계를 재사용하는 경우가 종종 발생

반복적으로 사용되는 설계는

클래스, 객체의 구성, 객체 간 메시지 흐름에서 **일정 패턴**을 갖는다.

1 디자인 패턴이란?

패턴을 잘 습득한다면?

- ✓ 상황에 맞는 올바른 설계를 더 빠르게 적용할 수 있다.
- ✓ 각 패턴의 장단점을 통해서 설계를 선택하는데 도움을 얻을 수 있다.
- ✓ 설계 패턴에 이름을 붙임으로써 시스템의 문서화, 이해, 유지 보수에 도움을 얻을 수 있다.

자주 사용되는 패턴들의 다양한 책이 존재하는데,
그 중에서도 GoF의 디자인 패턴은 많은 프로그래머들에게 도움을 주었다.

↓
20여 개에 이르는 패턴을 정리하고 있다.

1 디자인 패턴이란?

✓ 자주 사용되면서도 다른 패턴에 비해 상대적으로 이해하기 쉬운 패턴을 알아보자

- 전략 패턴 / 템플릿 메서드 패턴 / 상태 패턴
 - 데코레이터 패턴 / 프록시 패턴 / 어댑터 패턴
 - 옵저버 패턴 / 미디에이터 패턴 / 파사드 패턴
 - 추상 팩토리 패턴 / 컴포지트 패턴
 - + 널(Null) 객체 패턴(GoF 패턴엔 속해 있지 않음)
- 총 12개

2 전략(Strategy) 패턴

2 전략(Strategy) 패턴

✓ 한 과일 매장이 상황에 따라 다른 **가격 할인 정책**을 적용하고 있다고 생각해보자

- 첫 손님 할인 정책
- 덜 신선한 과일 할인 정책

가격을 계산하는 모듈에
가격 할인을 적용하기 위한
if-else 블록이 포함되어 있다.

```
public class Calculator {  
  
    public int calculate(boolean firstGuest, List<Item> items) {  
        int sum = 0;  
  
        for (Item item : items) {  
            if (firstGuest) {  
                sum += (int) (item.getPrice() * 0.9); // 첫 손님 10% 할인  
            } else if (!item.isFresh()) {  
                sum += (int) (item.getPrice() * 0.8); // 덜 신선한 것 20% 할인  
            } else {  
                sum += item.getPrice();  
            }  
        }  
  
        return sum;  
    }  
}
```

2 전략(Strategy) 패턴

✓ 간단하게 보이지만 문제가 있다.

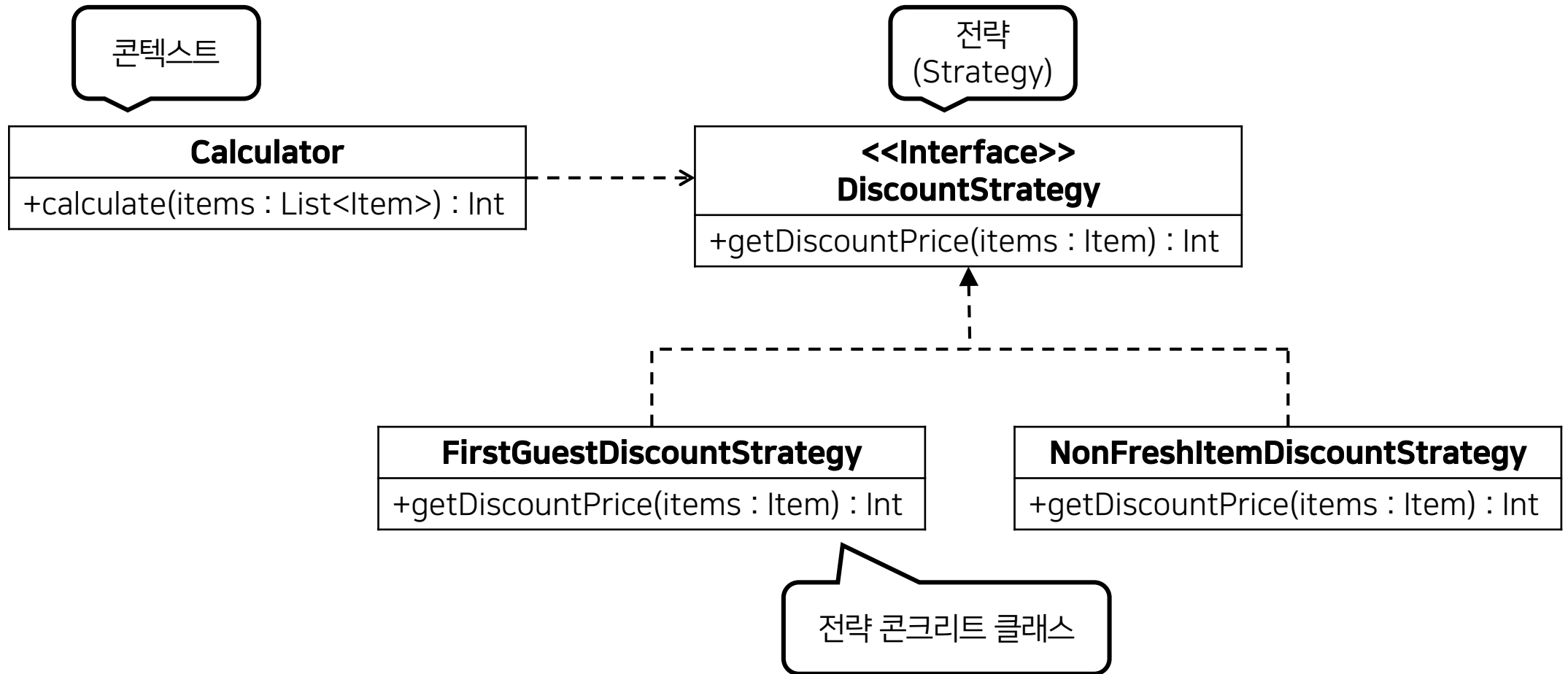
- 서로 다른 계산 정책들이 한 코드에 섞여있다.
 - ▶ 코드 분석이 어려워짐
 - ▶ calculate 메서드를 수정하는 것이 어려워짐

```
public int calculate(boolean firstGuest, boolean lastGuest, List<Item> items) {  
    int sum = 0;  
  
    for (Item item : items) {  
        if (firstGuest) {  
            sum += (int) (item.getPrice() * 0.9);  
        } else if (!item.isFresh()) {  
            sum += (int) (item.getPrice() * 0.8);  
        } else if (lastGuest) {  
            sum += (int) (item.getPrice() * 0.5);  
        } else {  
            sum += item.getPrice();  
        }  
    }  
}
```

정책이 추가된다면...?

2 전략(Strategy) 패턴

- ✓ 해결하기 위한 방법 중 하나는 가격 할인 정책을 **별도 객체로 분리**하는 것



2 전략(Strategy) 패턴

✓ 전략 패턴을 적용한 Calculator

```
public class Calculator {  
  
    private DiscountStrategy discountStrategy;  
  
    public Calculator(DiscountStrategy discountStrategy) {  
        this.discountStrategy = discountStrategy;  
    }  
  
    public int calculate(List<Item> items) {  
        int sum = 0;  
  
        for (Item item : items) {  
            sum += discountStrategy.getDiscountPrice(item);  
        }  
  
        return sum;  
    }  
}
```

기존 Calculator

```
public class Calculator {  
  
    public int calculate(boolean firstGuest, List<Item> items) {  
        int sum = 0;  
  
        for (Item item : items) {  
            if (firstGuest) {  
                sum += (int) (item.getPrice() * 0.9); // 첫 손님 10% 할인  
            } else if (!item.isFresh()) {  
                sum += (int) (item.getPrice() * 0.8); // 덜 신선한 것 20% 할인  
            } else {  
                sum += item.getPrice();  
            }  
        }  
  
        return sum;  
    }  
}
```

2 전략(Strategy) 패턴

✓ 전략 인터페이스

```
public interface DiscountStrategy {  
    int getDiscountPrice(Item item);  
}
```



```
public interface DiscountStrategy {  
    int getDiscountPrice(Item item);  
    int getDiscountPrice(int totalPrice);  
}
```

```
public interface ItemDiscountStrategy {  
    int getDiscountPrice(Item item);  
}  
public interface TotalPriceDiscountStrategy {  
    int getDiscountPrice(int totalPrice);  
}
```

2 전략(Strategy) 패턴

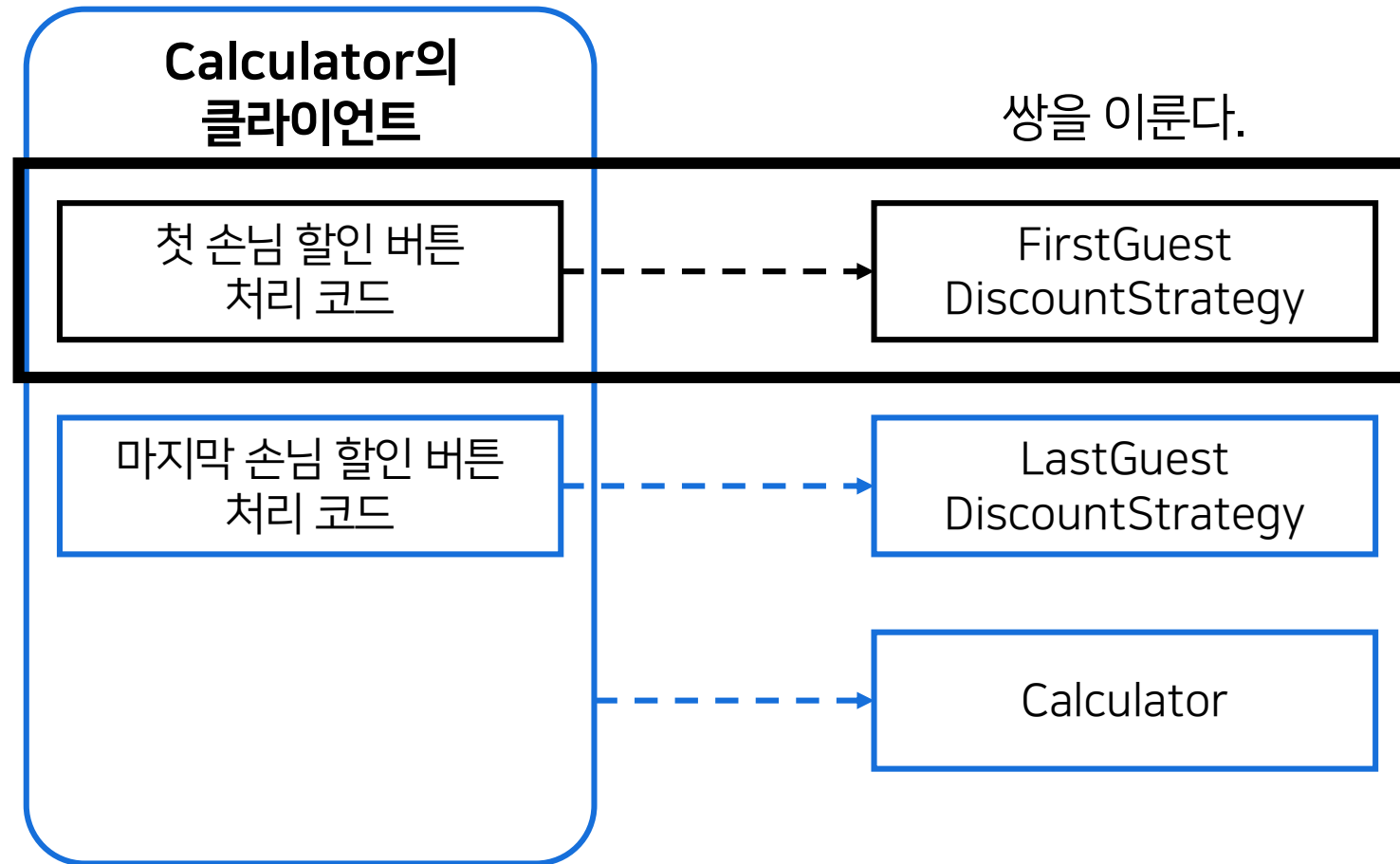
- ✓ DiscountStrategy 인터페이스를 구현한 콘크리트 클래스

```
public class FirstGuestDiscountStrategy implements DiscountStrategy {  
  
    @Override  
    public int getDiscountPrice(Item item) {  
        return (int) (item.getPrice() * 0.9);  
    }  
  
}
```

- ✓ 손님의 할인 적용과 계산을 처리하는 코드

```
private DiscountStrategy strategy;  
  
public void onFirstGuestButtonClick() {  
    // 첫 손님 할인 버튼 누를 때 생성 됨  
    strategy = new FirstGuestDiscountStrategy();  
}  
  
public void onCalculationButtonClick() {  
    // 계산 버튼 누를 때 실행 됨  
    Calculator cal = new Calculator(strategy);  
    int price = cal.calculate(items);  
    // ...  
}
```

2 전략(Strategy) 패턴



2 전략(Strategy) 패턴

- ✓ 마지막 손님 대폭 할인, 덜 신선한 과일 할인
정책 추가시



개방 폐쇄 원칙을 따르는 구조를 갖게 된다.

```
private DiscountStrategy strategy;

public void onFirstGuestButtonClick() {
    // 첫 손님 할인 버튼 누를 때 생성 됨
    strategy = new FirstGuestDiscountStrategy();
}

public void onLastGuestButtonClick() {
    // 마지막 손님 대폭 할인 버튼 누를 때 생성 됨
    strategy = new LastGuestDiscountStrategy();
}

public void onNonFreshItemDiscountStrategy() {
    // 덜 신선한 과일 할인 버튼 누를 때 생성 됨
    strategy = new NonFreshItemDiscountStrategy();
}

public void onCalculationButtonClick() {
    // 계산 버튼 누를 때 실행 됨
    Calculator cal = new Calculator(strategy);
    int price = cal.calculate(items);
    // ...
}
```

2 전략(Strategy) 패턴

- ✓ if-else로 구성된 코드 블록이 비슷한 기능을 수행하는 경우, 전략 패턴을 적용하면 쉽게 확장 가능한 코드로 변경할 수 있다.

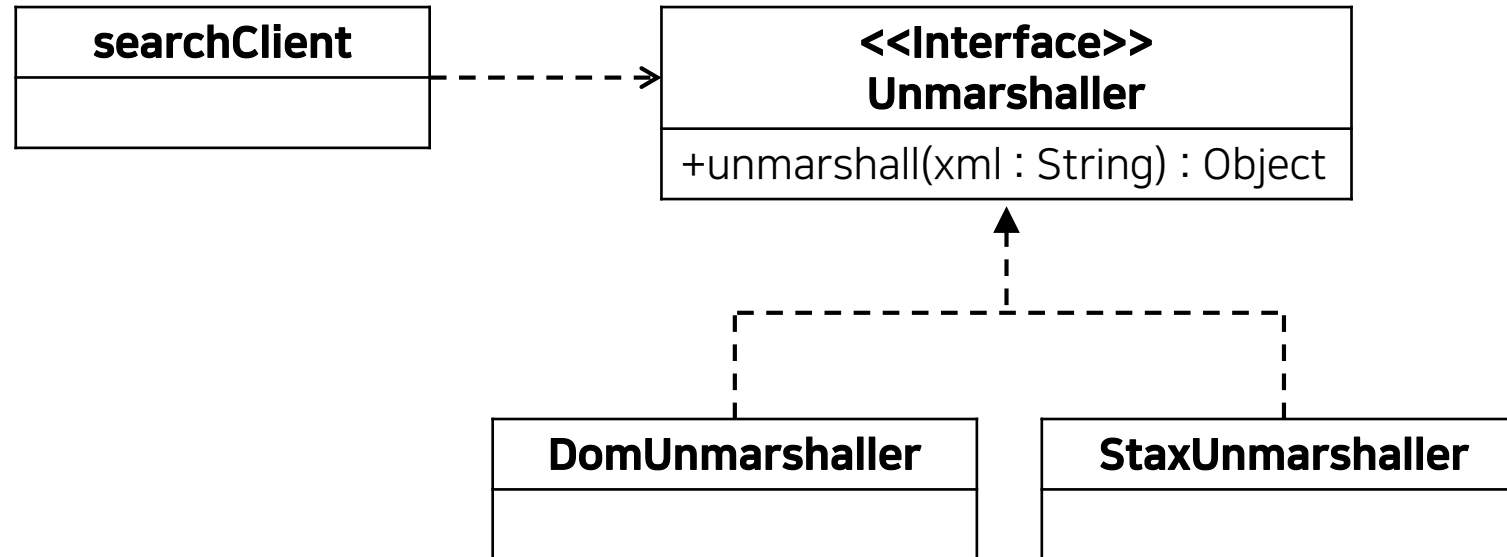
```
public class Calculator {  
  
    public int calculate(boolean firstGuest, List<Item> items) {  
        int sum = 0;  
  
        for (Item item : items) {  
            if (firstGuest) {  
                sum += (int) (item.getPrice() * 0.9); // 첫 손님 10% 할인  
            } else if (!item.isFresh()) {  
                sum += (int) (item.getPrice() * 0.8); // 덜 신선한 것 20% 할인  
            } else {  
                sum += item.getPrice();  
            }  
        }  
  
        return sum;  
    }  
}
```



```
public class Calculator {  
  
    private DiscountStrategy discountStrategy;  
  
    public Calculator(DiscountStrategy discountStrategy) {  
        this.discountStrategy = discountStrategy;  
    }  
  
    public int calculate(List<Item> items) {  
        int sum = 0;  
  
        for (Item item : items) {  
            sum += discountStrategy.getDiscountPrice(item);  
        }  
  
        return sum;  
    }  
}
```

2 전략(Strategy) 패턴

- ✓ 전략 패턴은 **동일한 기능**의 알고리즘 변경이 필요할 때 사용



3 템플릿 메서드(Template Method) 패턴

3 템플릿 메서드(Template Method) 패턴

- ✓ 프로그램을 구현하다 보면, **완전히 동일한 절차**를 가진 코드를 작성할 때가 있다.

```
public class DbAuthenticator {  
    public Auth authenticate(String id, String pw) {  
        // 사용자 정보로 인증 확인  
        User user = userDao.selectById(id);  
        boolean auth = user.equalPassword(pw);  
        // 인증 실패시 익셉션 발생  
        if (!auth) {  
            throw createException();  
        }  
        // 인증 성공시, 인증 정보 제공  
        return new Auth(id, user.getName());  
    }  
  
    private AuthException createException() {  
        return new AuthException();  
    }  
}
```

```
public class LdapAuthenticator {  
    public Auth authenticate(String id, String pw) {  
        // 사용자 정보로 인증 확인  
        boolean lauth = ldapClient.authenticate(id, pw);  
        // 인증 실패시 익셉션 발생  
        if (!lauth) {  
            throw createException();  
        }  
        // 인증 성공시, 인증 정보 제공  
        LdapContext ctx = ldapClient.find(id);  
        return new Auth(id, ctx.getAttribute(name: "name"));  
    }  
  
    private AuthException createException() {  
        return new AuthException();  
    }  
}
```

3 템플릿 메서드(Template Method) 패턴

✓ 템플릿 메서드 패턴은 **두 가지**로 구성된다.

1. 실행 과정을 구현한 상위 클래스

- 기능을 구현하는 메서드를 제공
- 구현 일부 단계는 추상 메서드를 호출하는 방식

동일한 실행과정

차이가 나는 부분은 추상 메서드로 구현

```
public abstract class Authenticator {  
    // 템플릿 메서드  
    public Auth authenticate(String id, String pw) {  
        if (!doAuthenticate(id, pw)) {  
            throw createException();  
        }  
  
        return createAuth(id);  
    }  
  
    protected abstract boolean doAuthenticate(String id, String pw);  
  
    private RuntimeException createException() {  
        throw new AuthException();  
    }  
  
    protected abstract Auth createAuth(String id);  
}
```

3 템플릿 메서드(Template Method) 패턴

✓ 템플릿 메서드 패턴은 **두 가지**로 구성된다.

2. 실행 과정의 **일부 단계를 구현**한 **하위 클래스**

- 다른 메서드만 알맞게 재정의해 주면 된다.



코드 중복 문제를 제거할 수 있다.

```
public class LdapAuthenticator extends Authenticator {  
  
    @Override  
    protected boolean doAuthenticate(String id, String pw) {  
        return ldapClient.authenticate(id, pw);  
    }  
  
    @Override  
    protected Auth createAuth(String id) {  
        LdapContext ctx = ldapClient.find(id);  
        return new Auth(id, ctx.getAttribute("name"));  
    }  
}
```

3 상위 클래스가 흐름 제어 주체

- ✓ 템플릿 메서드 패턴은 상위 클래스에서 흐름 제어를 한다.

일반적인 경우, 하위 타입이 상위 타입의 기능을 재사용할지 여부를 결정하여 흐름제어

```
public class SuperCar extends ZetEngine {  
    @Override  
    public void turnOn() {  
        // 하위 클래스에서 흐름 제어  
        if (notReady) {  
            beep();  
        } else {  
            super.turnOn();  
        }  
    }  
}
```

재정의해서 흐름을 제어한다.

3 상위 클래스가 흐름 제어 주체

✓ 템플릿 메서드 패턴은 상위 클래스에서 흐름 제어를 한다.

```
public abstract class Authenticator {  
    // 템플릿 메서드  
    public Auth authenticate(String id, String pw) {  
        if (!doAuthenticate(id, pw)) {  
            throw createException();  
        }  
  
        return createAuth(id);  
    }  
  
    protected abstract boolean doAuthenticate(String id, String pw);  
  
    private RuntimeException createException() {  
        throw new AuthException();  
    }  
  
    protected abstract Auth createAuth(String id);  
}
```

모든 실행 흐름을 제어하고 있다.

상위타입

```
public class LdapAuthenticator extends Authenticator {  
  
    @Override  
    protected boolean doAuthenticate(String id, String pw) {  
        return ldapClient.authenticate(id, pw);  
    }  
  
    @Override  
    protected Auth createAuth(String id) {  
        LdapContext ctx = ldapClient.find(id);  
        return new Auth(id, ctx.getAttribute("name"));  
    }  
}
```

하위타입

3 흑(hook) 메서드

- ✓ 하위 클래스는 **기본 구현**을 **알맞게 재정의**할 수도 있다.
- ex) 안드로이드에서 비동기 처리를 위한 기능을 제공하는 AsyncTask 클래스

추상 메서드 호출

구현 메서드 호출

추상 메서드

빈 구현을 갖는 메서드
(흑(hook)메서드)

```
public abstract class AsyncTask<Params, Progress, Result> {

    public AsyncTask() {
        mWorker = new WorkerRunnable<Params, Result>() {
            public Result call() throws Exception {
                mTaskInvoked.set(true);
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                return postResult(doInBackground(mParams));
            }
        };
    }

    public final AsyncTask<Params, Progress, Result> executeOnExecutor(
        Executor exec, Params... params) {
        // ...
        mStatus = Status.RUNNING;
        onPreExecute();
        mWorker.mParams = params;
        exec.execute(mFuture);
        return this;
    }

    protected abstract Result doInBackground(Params... params);

    protected void onPreExecute() { // 하위 클래스의 확장 지점
    }

    // ... 기타 코드
}
```

3 템플릿 메서드와 전략 패턴의 조합

✓ 템플릿 메서드와 전략 패턴을 **함께** 사용한다면?

변경되는 부분을 실행할 객체를 전달받는 형식

```
public <T> T execute(TransactionCallback<T> action)
    throws TransactionException {
    // 일부 코드들...
    TransactionStatus status = this.transactionManager.getTransaction(this);
    T result;

    try {
        result = action.doInTransaction(status);
    } catch (RuntimeException ex) {
        rollbackOnException(status, ex);
        throw ex;
    }

    // ... 기타 다른 익셉션 처리 코드
    this.transactionManager.commit(status);
    return result;
}
```

- 상속이 아닌 **조립의 방식**으로
템플릿 메서드 패턴을 **활용**할 수 있다.

대표적인 예)

스프링 프레임워크의
Template로 끝나는 클래스들

```
transactionTemplate.execute(new TransactionCallback<String>() {
    public String doInTransaction(TransactionStatus status) {
        // 트랜잭션 범위 안에서 실행될 코드
    }
});
```

execute() 메서드를 사용하는 코드는 기능을 구현한 객체를 전달한다.

3 템플릿 메서드와 전략 패턴의 조합

✓ 장점

- 상속에 기반을 둔 템플릿 메서드 구현과 비교해서 **유연함**을 갖는다.
템플릿 메서드에서 사용할 객체를 교체할 수 있는 장점을 가진다.
(상속을 통한 재사용은 클래스가 불필요하게 증가할 수 있고, 런타임에 교체할 수 없다.)

✓ 단점

- 확장 기능을 제공하려면 구현이 복잡해진다.
(상속 방식의 경우 훅(hook) 메서드를 재정의 하는 방법으로 쉽게 하위 클래스에서 확장 기능 제공)

✓ Note

- 엄격하게 GoF 정의를 적용하면 전략패턴에 가깝다.
(실제 GoF 패턴에서 템플릿 메서드 패턴은 **상속에 기반**한 패턴으로 정의한다.)

4 상태(State) 패턴

4 상태(State) 패턴

- ✓ 단일 상품을 판매하는 자판기에 들어갈 소프트웨어 개발 요구시

동작 방식의 최초 요구 사항

동 작	조 건	실 행	결 과
동전을 넣음	동전 없음이면	금액을 증가	제품 선택 가능
동전을 넣음	제품 선택 가능이면	금액을 증가	제품 선택 가능
제품 선택	동전 없음이면	아무 동작 하지 않음	동전 없음 유지
제품 선택	제품 선택 가능이면	제품 주고 잔액 감소	잔액 있으면 제품 선택 가능 잔액 없으면 동전 없음

4 상태(State) 패턴

- ✓ 조건에 따라 다른 코드를 실행해야 한다고 판단하여 코드를 작성

```
public class VendingMachine {  
  
    public static enum State {NOCOIN, SELECTABLE,}  
  
    private State state = State.NOCOIN;  
  
    public void insertCoin(int coin) {  
        switch (state) {  
            case NOCOIN:  
                increaseCoin(coin);  
                state = State.SELECTABLE;  
                break;  
            case SELECTABLE:  
                increaseCoin(coin);  
        }  
    }  
}
```

```
public void select(int producId) {  
    switch (state) {  
        case NOCOIN:  
            // 아무 것도 하지 않음  
            break;  
        case SELECTABLE:  
            provideProduct(producId);  
            decreaseCoin();  
            if (hasNoCoin()) {  
                state = State.NOCOIN;  
            }  
        }  
    }  
  
    // ... increaseCoin, provideProduct, decreaseCoin 구현  
}
```

4 상태(State) 패턴

- ✓ 새로운 요구사항이 들어온다.
 - 자판기에 제품이 없는 경우, 동전을 넣으면 바로 동전을 되돌려준다.

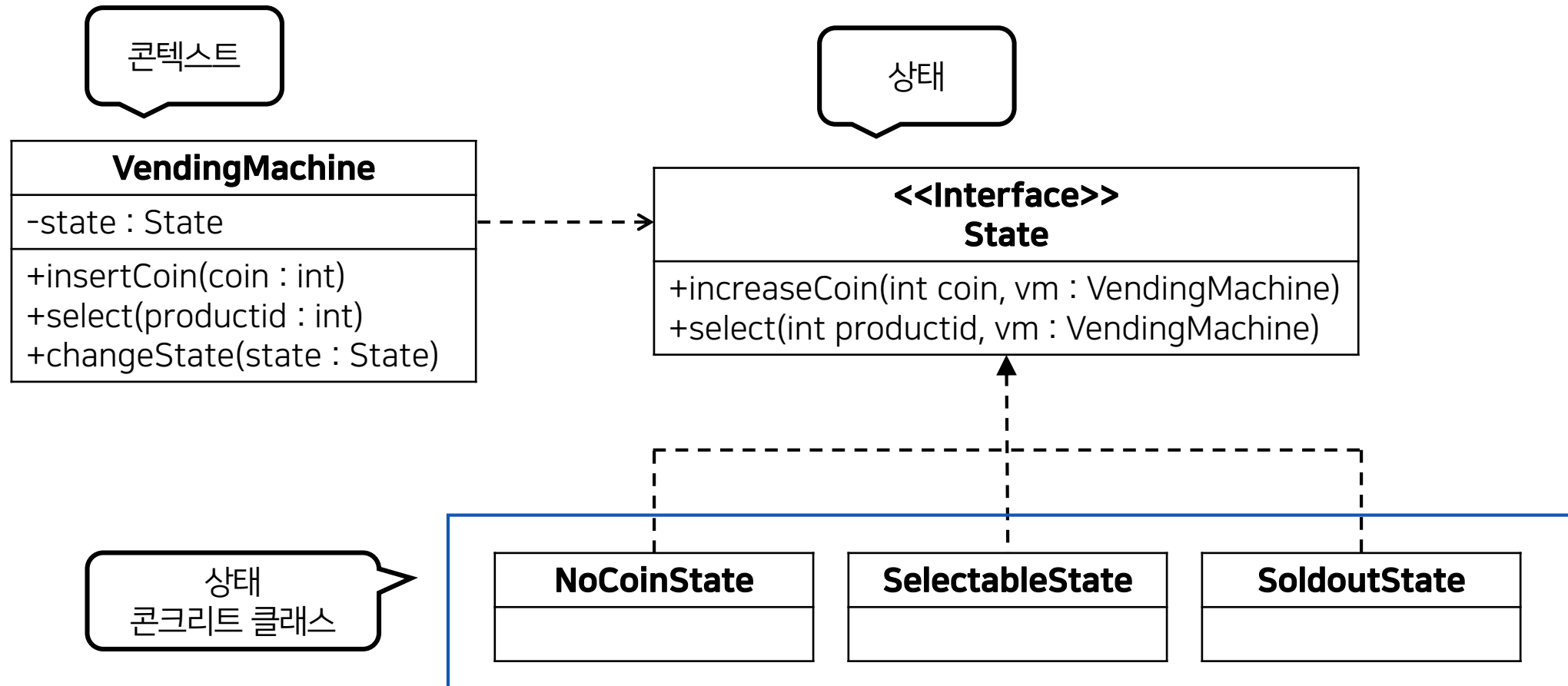
```
public class VendingMachine {  
  
    public static enum State {NOCOIN, SELECTABLE, SOLDOUT,}  
  
    private State state = State.NOCOIN;  
  
    public void insertCoin(int coin) {  
        switch (state) {  
            case NOCOIN:  
                increaseCoin(coin);  
                state = State.SELECTABLE;  
                break;  
            case SELECTABLE:  
                increaseCoin(coin);  
                break;  
            case SOLDOUT:  
                returnCoin();  
        }  
    }  
}
```

동일한 구조이다.
상태가 많아질수록 복잡해져
코드 변경이 어려워진다.

```
public void select(int producId) {  
    switch (state) {  
        case NOCOIN:  
            // 아무 것도 하지 않음  
            break;  
        case SELECTABLE:  
            provideProduct(producId);  
            decreaseCoin();  
            if (hasNoCoin()) {  
                state = State.NOCOIN;  
            }  
        case SOLDOUT:  
            // 아무 것도 하지 않음  
    }  
  
    // ... increaseCoin, provideProduct, decreaseCoin, returnCoin 구현  
}
```

4 상태(State) 패턴

✓ 상태를 별도 타입으로 분리하고, 각 상태 별로 알맞은 하위 타입을 구현한다.



4 상태(State) 패턴

✓ **컨텍스트**는 상태 객체에 **처리를 위임**하는 방식으로 구현한다.

```
public class VendingMachine {  
  
    private State state;  
  
    public VendingMachine() {  
        state = new NoCoinState();  
    }  
  
    public void insertCoin(int coin) {  
        state.increaseCoin(coin, vendingMachine: this); // 상태 객체에 위임  
    }  
  
    public void select(int productId) {  
        state.select(productId, vendingMachine: this); // 상태 객체에 위임  
    }  
  
    public void changeState(State newState) {  
        this.state = newState;  
    }  
  
    // ... 기타 다른 기능  
}
```

4 상태(State) 패턴

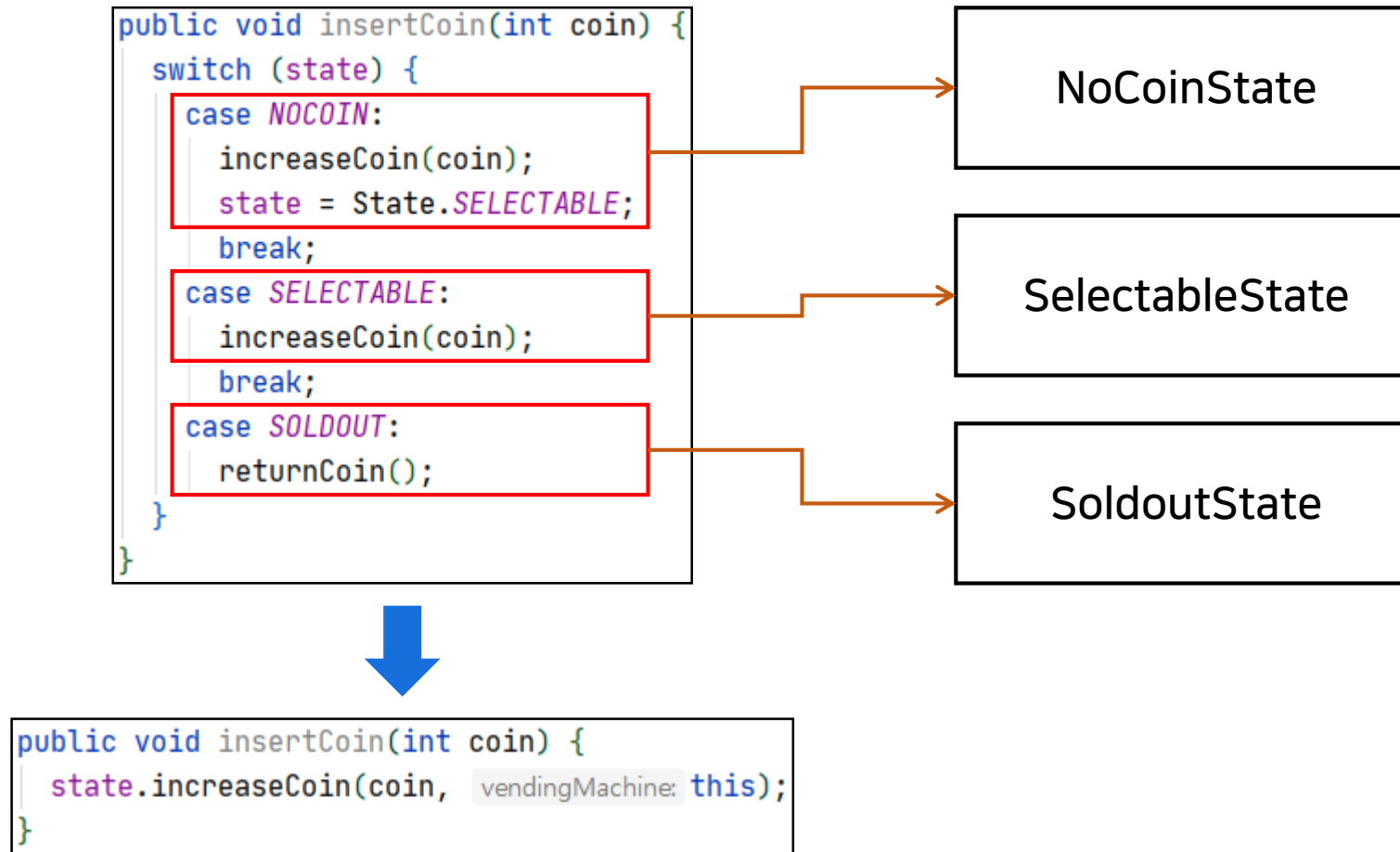
✓ 상태패턴을 적용하여 상태별 동작 구현 코드가 구현 클래스로 이동한다.

```
public class NoCoinState implements State {  
  
    @Override  
    public void increaseCoin(int coin, VendingMachine vm) {  
        vm.increaseCoin(coin);  
        vm.changeState(new SelectableState());  
    }  
  
    @Override  
    public void select(int productId, VendingMachine vm) {  
        SoundUtil.beep();  
    }  
}
```

```
public class SelectableState implements State {  
  
    @Override  
    public void increaseCoin(int coin, VendingMachine vm) {  
        vm.increaseCoin(coin);  
    }  
  
    @Override  
    public void select(int productId, VendingMachine vm) {  
        vm.provideProduct(productId);  
        vm.decreaseCoin();  
  
        if (vm.hasNoCoin()) {  
            vm.changeState(new NoCoinState());  
        }  
    }  
}
```


4 상태(State) 패턴

- ✓ **컨텍스트**의 코드가 **간결**해지고, 변경의 **유연함**을 얻는다.



4 상태 변경은 누가?

✓ 상태변경의 주체가 상태 객체

```
public class NoCoinState implements State {  
  
    @Override  
    public void increaseCoin(int coin, VendingMachine vm) {  
        vm.increaseCoin(coin);  
        // 상태 객체에서 콘텍스트의 상태 변경  
        vm.changeState(new SelectableState());  
    }  
}
```

```
public class SelectableState implements State {  
  
    @Override  
    public void select(int productId, VendingMachine vm) {  
        vm.provideProduct(productId);  
        vm.decreaseCoin();  
  
        if (vm.hasNoCoin()) { // 상태 변경을 위해, vm 객체의 동전을 확인  
            vm.changeState(new NoCoinState());  
        }  
    }  
}
```

4 상태 변경은 누가?

✓ 상태변경의 주체가 콘텍스트

```
public class VendingMachine {  
  
    private State state;  
  
    public VendingMachine() {  
        state = new NoCoinState();  
    }  
  
    public void insertCoin(int coin) {  
        state.increaseCoin(coin, vendingMachine: this);  
        if (hasCoin()) {  
            changeState(new SelectableState()); // 콘텍스트에서 상태 변경  
        }  
    }  
  
    public void select(int productId) {  
        state.select(productId, vendingMachine: this);  
        if (state.isSelectable() && hasNoCoin()) {  
            changeState(new NoCoinState()); // 콘텍스트에서 상태 변경  
        }  
    }  
}
```

```
} private void changeState(State newState) {  
    | this.state = newState;  
}  
  
} private boolean hasCoin() {  
    | // ...  
}  
  
} private boolean hasNoCoin() {  
    | // ...  
}  
  
// 기타 다른 기능  
  
}
```

4 상태 변경은 누가?

✓ 상태변경의 주체가 컨텍스트

```
public class SelectableState implements State {  
  
    @Override  
    public void select(int productId, VendingMachine vm) {  
        vm.provideProduct(productId);  
        vm.decreaseCoin();  
  
        if (vm.hasNoCoin()) { // 상태 변경을 위해, vm 객체의 동전을 확인  
            vm.changeState(new NoCoinState());  
        }  
    }  
}
```



```
public class SelectableState implements State {  
  
    // 상태 객체는 자신이 할 작업만 처리한다.  
    @Override  
    public void select(int productId, VendingMachine vm) {  
        vm.provideProduct(productId);  
        vm.decreaseCoin();  
    }  
}
```

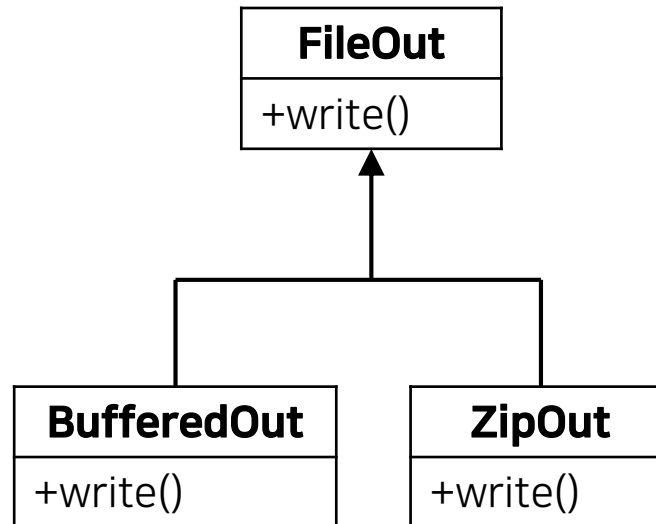
4 상태 변경은 누가?

- ✓ 상태 변경을 누가 할지는 주어진 **상황**에 알맞게 정해 주어야 한다.
- **컨텍스트**에서 상태를 변경하는 방식
 - 비교적 상태 수가 적고 상태 변경 규칙이 거의 바뀌지 않는 경우에 유리
 - 상태 종류의 변경이 많거나 상태 변경 처리 코드가 복잡해지면 유연함이 떨어진다.
- **상태 객체**에서 컨텍스트의 상태를 변경하는 방식
 - 컨텍스트에 영향을 주지 않으면서 상태를 추가하거나 상태 변경 규칙을 바꿀 수 있다.
 - 상태 변경 규칙이 여러 클래스에 분산되어 있어 클래스가 많아질수록 규칙을 파악하기 어렵다.

5 데코레이터(Decorator) 패턴

5 데코레이터(Decorator) 패턴

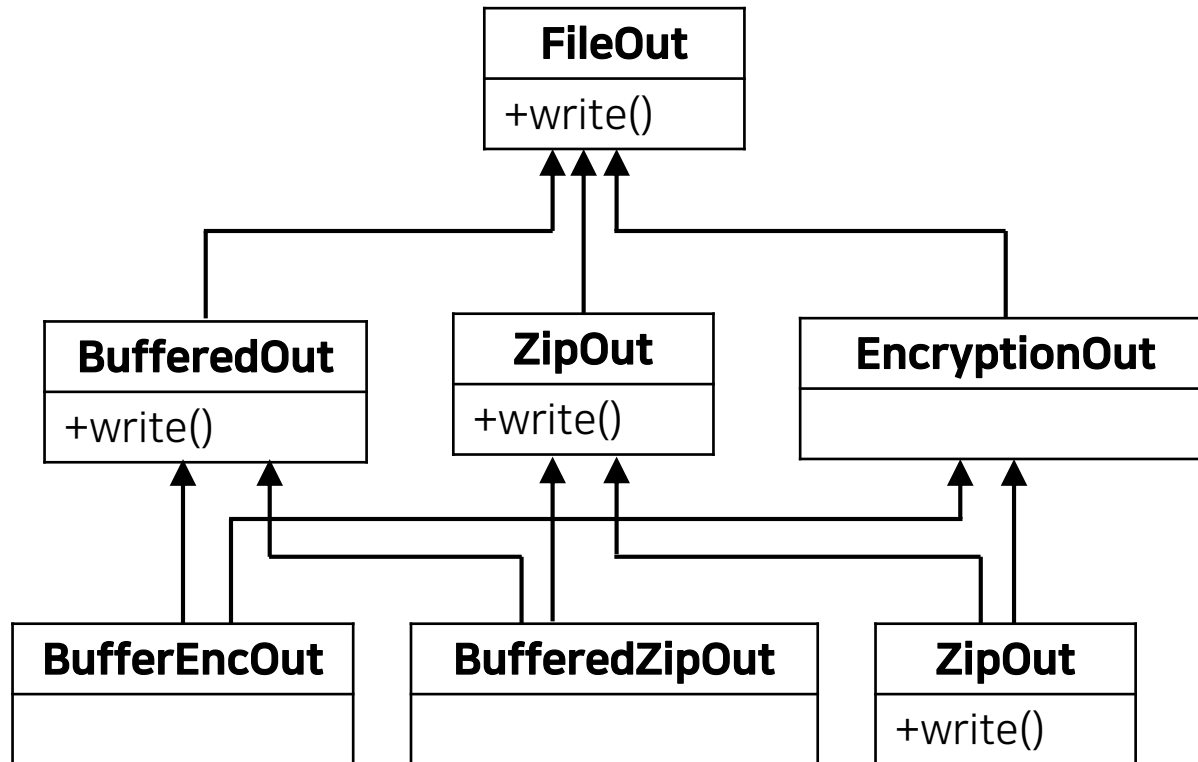
✓ 상속은 기능을 확장하는 방법을 제공한다.



상속을 이용해 기능을 확장

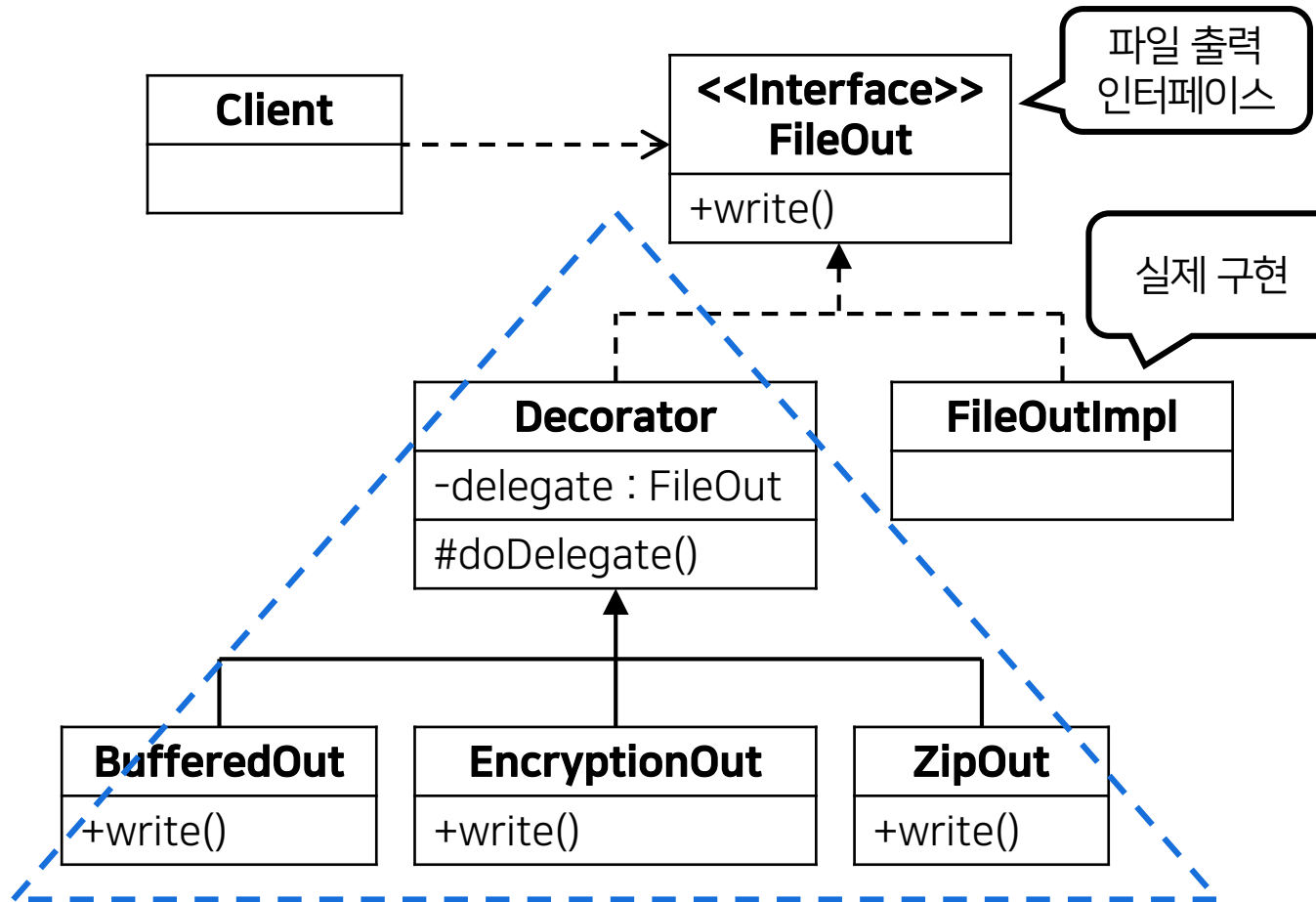
5 데코레이터(Decorator) 패턴

✓ 상속을 이용하면 클래스가 증가하고 계층 구조가 복잡해진다.



5 데코레이터(Decorator) 패턴

- ✓ 상속이 아닌 위임을 하는 방식으로 기능을 확장해 나간다.



5 데코레이터(Decorator) 패턴

```
public abstract class Decorator implements FileOut {  
  
    private FileOut delegate; // 위임 대상  
  
    public Decorator(FileOut delegate) {  
        this.delegate = delegate;  
    }  
  
    protected void doDelegate(byte[] data) {  
        delegate.write(data); // delegate에 쓰기 위임  
    }  
}
```

Decorator

```
public class EncryptionOut extends Decorator {  
  
    public EncryptionOut(FileOut delegate) {  
        super(delegate);  
    }  
  
    public void write(byte[] data) {  
        byte[] encryptedData = encrypt(data);  
        super.doDelegate(encryptedData);  
    }  
  
    private byte[] encrypt(byte[] data) {  
        // ...  
    }  
}
```

EncryptionOut

5 데코레이터(Decorator) 패턴

```
FileOut delegate = new FileOutImpl();  
FileOut fileOut = new EncryptionOut(delegate);  
fileOut.write(data);
```

```
public class EncryptionOut extends Decorator {  
  
    public EncryptionOut(FileOut delegate) {  
        super(delegate);  
    }  
  
    public void write(byte[] data) {  
        byte[] encryptedData = encrypt(data);  
        super.doDelegate(encryptedData);  
    }  
  
    private byte[] encrypt(byte[] data) {  
        // ...  
    }  
}
```

```
public abstract class Decorator implements FileOut {  
  
    private FileOut delegate; // 위임 대상  
  
    public Decorator(FileOut delegate) {  
        this.delegate = delegate;  
    }  
  
    protected void doDelegate(byte[] data) {  
        delegate.write(data); // delegate에 쓰기 위임  
    } FileOutImpl의 write()  
}
```

✓ 기존 FileOutImpl 기능에 새로운 기능을 추가해준다.

5 데코레이터(Decorator) 패턴

✓ 데코레이터 패턴의 장점

- 데코레이터를 조합하는 방식으로 기능을 확장할 수 있다.

```
FileOut delegate = new FileOutImpl();  
FileOut fileOut = new EncryptionOut(new ZipOut(delegate));  
fileOut.write(data);
```

- 기능 적용 순서의 변경도 쉽다.

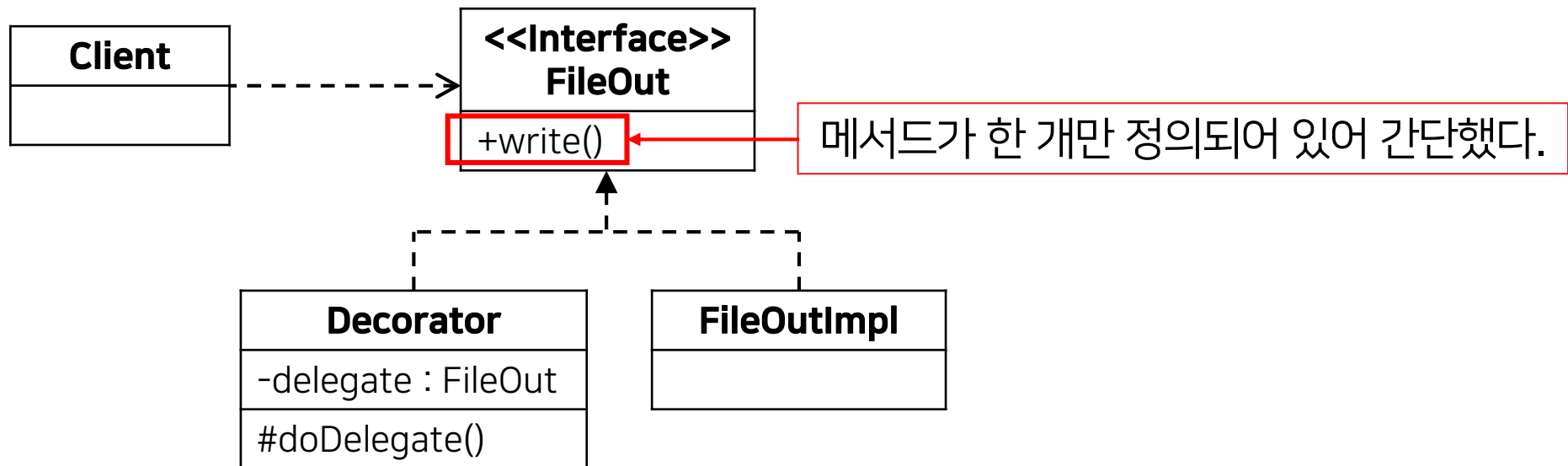
```
// 버퍼 -> 암호화 -> 압축 -> 파일 쓰기  
FileOut fileOut = new BufferedOut(new EncryptionOut(new ZipOut(delegate)));  
  
// 암호화 -> 압축 -> 버퍼 -> 파일 쓰기  
FileOut fileOut = new EncryptionOut(new ZipOut(new BufferedOut(delegate)));
```

데코레이터 패턴은 단일 책임 원칙을 지킬 수 있도록 만들어 준다.

5 데코레이터 패턴을 적용할 때 고려할 점

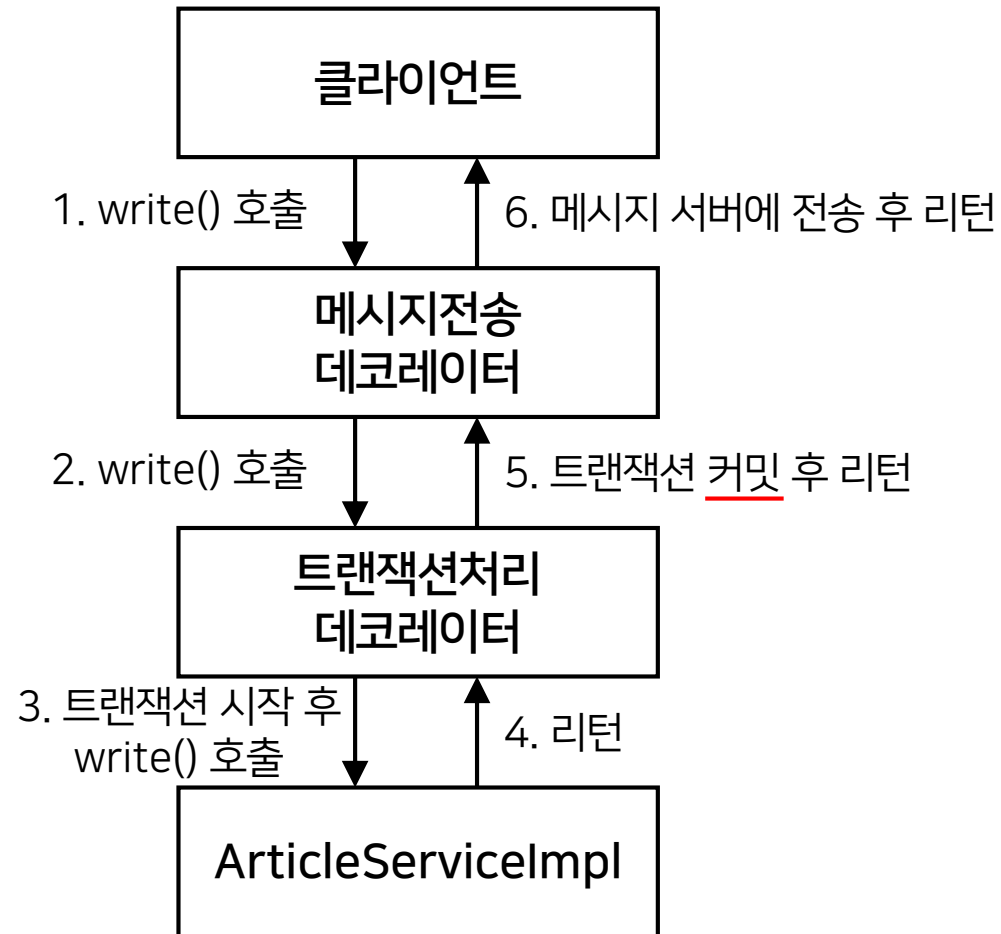
✓ 데코레이터 대상이 되는 타입의 **기능 개수**

- 데코레이터 대상이 되는 타입에 정의되어 있는 메서드가 증가하면 데코레이터 구현도 복잡해진다.
(데코레이터 대상의 예 : 앞에서 했던 FileOut)



5 데코레이터 패턴을 적용할 때 고려할 점

- ✓ 데코레이터 객체가 비정상적으로 동작할 때 어떻게 처리할 것인가?



5 데코레이터 패턴의 단점

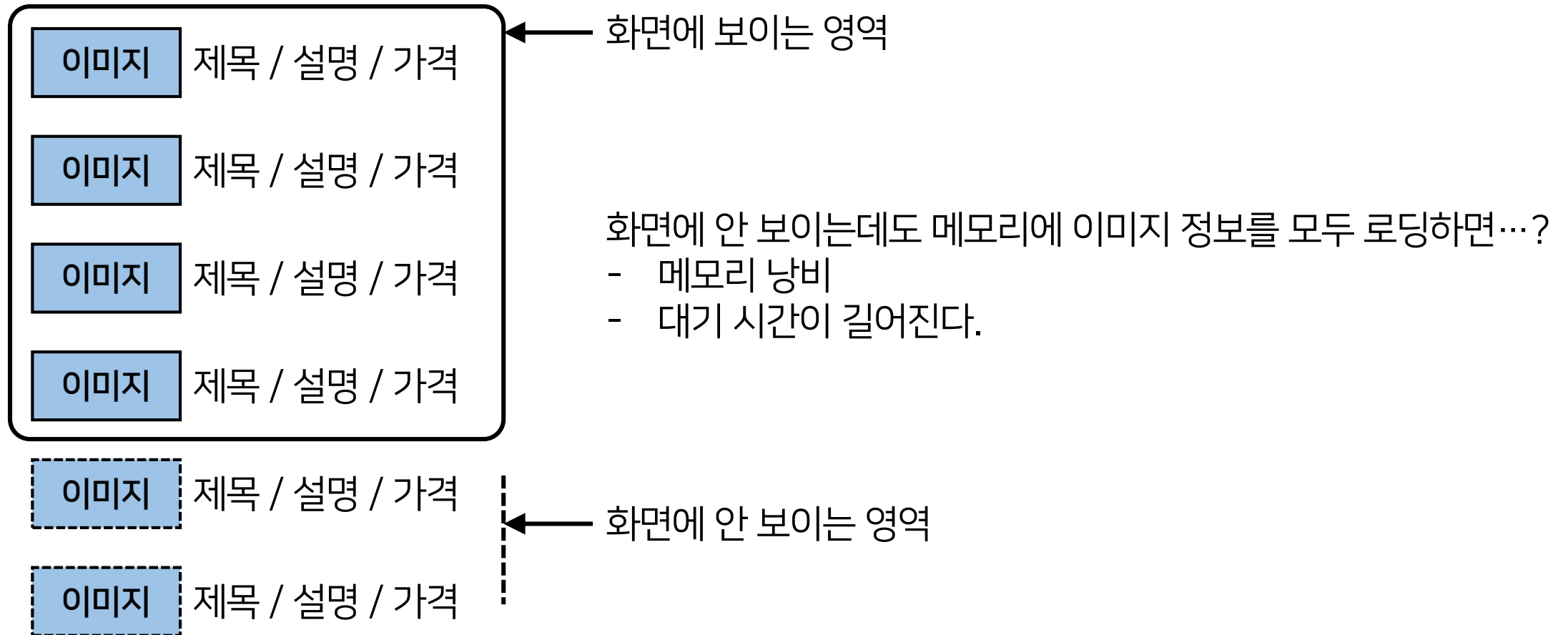
- ✓ 데코레이터 객체와 실제 구현 객체의 **구분**이 되지 않는다.

```
public class ImageSource {  
  
    public void writeTo(FileOut out) {  
        | out.write(imageData);  
    }  
  
    // ...  
}
```

6 프록시(Proxy) 패턴

6 프록시(Proxy) 패턴

✓ 제품 목록을 보여주는 GUI 프로그램



6 프록시(Proxy) 패턴

✓ 이미지가 **실제**로 화면에 보여질 때 이미지 데이터를 로딩하자.

- 가장 **쉬운** 방법

필요 시점에 이미지를 로딩하는 DynamicLoadingImage 클래스 추가(Image 클래스를 이용)

→ Image 클래스 대신 DynamicLoadingImage를 사용하게 만든다.

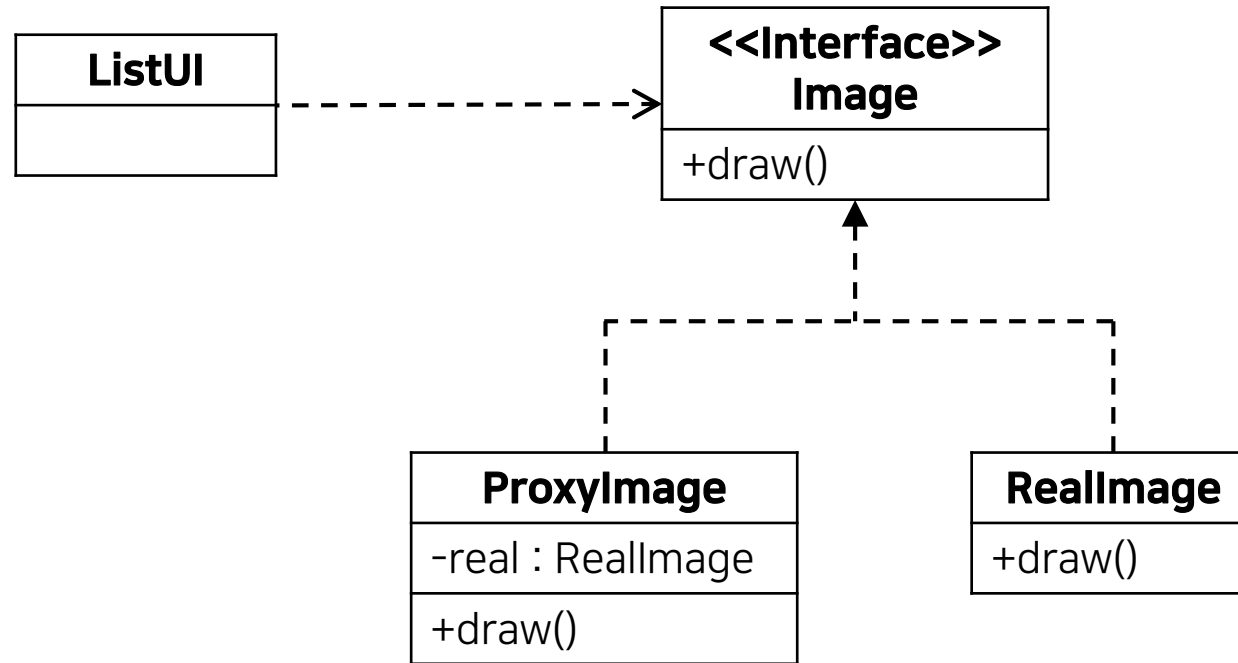


- 이미지 로딩 방식을 **변경**해야 한다면?

→ ListUI 코드를 변경해야 한다.

6 프록시(Proxy) 패턴

- ✓ ListUI 변경 없이 이미지 로딩 방식을 교체할 수 있도록 해주는 패턴



6 프록시(Proxy) 패턴

✓ ProxyImage 클래스 구현

```
public class ProxyImage implements Image {  
  
    private String path;  
    private RealImage image;  
  
    public ProxyImage(String path) {  
        this.path = path;  
    }  
  
    @Override  
    public void draw() {  
        if (image == null) {  
            image = new RealImage(path); // 최초 접근 시 객체 생성  
        }  
        image.draw(); // RealImage 객체에 위임  
    }  
}
```

6 프록시(Proxy) 패턴

✓ ListUI 클래스

```
public class ListUI {  
  
    private List<Image> images;  
  
    public ListUI(List<Image> images) {  
        this.images = images;  
    }  
  
    public void onScroll(int start, int end) {  
        // 스크롤 시, 화면에 표시되는 이미지를 표시  
        for (int i = start; i <= end; i++) {  
            Image image = images.get(i);  
            image.draw();  
        }  
    }  
  
    // ...  
}
```

6 프록시(Proxy) 패턴

```
public class ListUI {  
  
    private List<Image> images;  
  
    public ListUI(List<Image> images) {  
        this.images = images;  
    }  
  
    public void onScroll(int start, int end) {  
        // 스크롤 시, 화면에 표시되는 이미지를 표시  
        for (int i = start; i <= end; i++) {  
            Image image = images.get(i);  
            image.draw();  
        }  
    }  
  
    // ...  
}
```

```
public class ProxyImage implements Image {  
  
    private String path;  
    private RealImage image;  
  
    public ProxyImage(String path) {  
        this.path = path;  
    }  
  
    @Override  
    public void draw() {  
        if (image == null) {  
            image = new RealImage(path); // 최초 접근 시 객체 생성  
        }  
        image.draw(); // RealImage 객체에 위임  
    }  
}
```

6 프록시(Proxy) 패턴

- ✓ 상위 4개는 바로 이미지를 로딩, 나머지는 보여지는 순간 로딩할 때

```
List<String> paths = null; // 이미지 경로 목록을 가져온다.
List<Image> images = new ArrayList<Image>(paths.size());
for (int i = 0; i < paths.size(); i++) {
    if (i <= 4) {
        images.add(new RealImage(paths.get(i)));
    } else {
        images.add(new ProxyImage(paths.get(i)));
    }
}

// 이미지 로딩 정책 변경이 ListUI에 영향을 주지 않는다.
ListUI listUI = new ListUI(images);
```

RealImage 객체와 ProxyImage 객체를 섞어서 ListUI에 전달하면 된다.

6 프록시(Proxy) 패턴

✓ 필요한 순간 실제 객체를 생성해 주는 프록시

- 가상 프록시(Virtual Proxy)  `public class ProxyImage implements Image {`

```
private String path;  
private RealImage image;
```

```
public ProxyImage(String path) {  
    this.path = path;  
}
```

```
@Override  
public void draw() {  
    if (image == null) {  
        image = new RealImage(path); // 최초 접근 시 객체 생성  
    }  
    image.draw(); // RealImage 객체에 위임  
}
```

✓ 가상 프록시 외의 프록시

- 보호 프록시**(Protection Proxy)
실제 객체에 대한 접근을 제어하는 프록시
(접근 권한이 있는 경우만 실제 객체의 메서드 실행)
- 원격 프록시**(Remote Proxy)
다른 프로세스에 존재하는 객체에 접근할 때 사용
(내부적으로 IPC나 TCP통신을 이용해서 실행)
- 등등...

6 프록시 패턴을 적용할 때 고려할 점

✓ **실제** 객체를 누가 생성할 것인가?

- **가상** 프록시는 필요한 순간 실제 객체를 생성하는 경우가 많다.
→ 가상 프록시에서 실제 생성할 객체의 타입을 사용하게 된다.
(ex. ProxyImage 클래스에서 직접 RealImage 타입을 사용)
- **보호** 프록시는 보호 프록시 객체를 생성할 때 **실제 객체를 전달**하면 된다.
→ 실제 객체의 타입을 알 필요 없이 추상 타입을 사용하면 된다.

6 프록시 패턴을 적용할 때 고려할 점

- ✓ 위임 방식이 아닌 **상속**을 사용해서 프록시를 구현할 수도 있다.
- 만약 특정 기능을 **관리자만** 실행할 수 있어야 할 경우 → 보호 프록시를 사용할 수 있다.

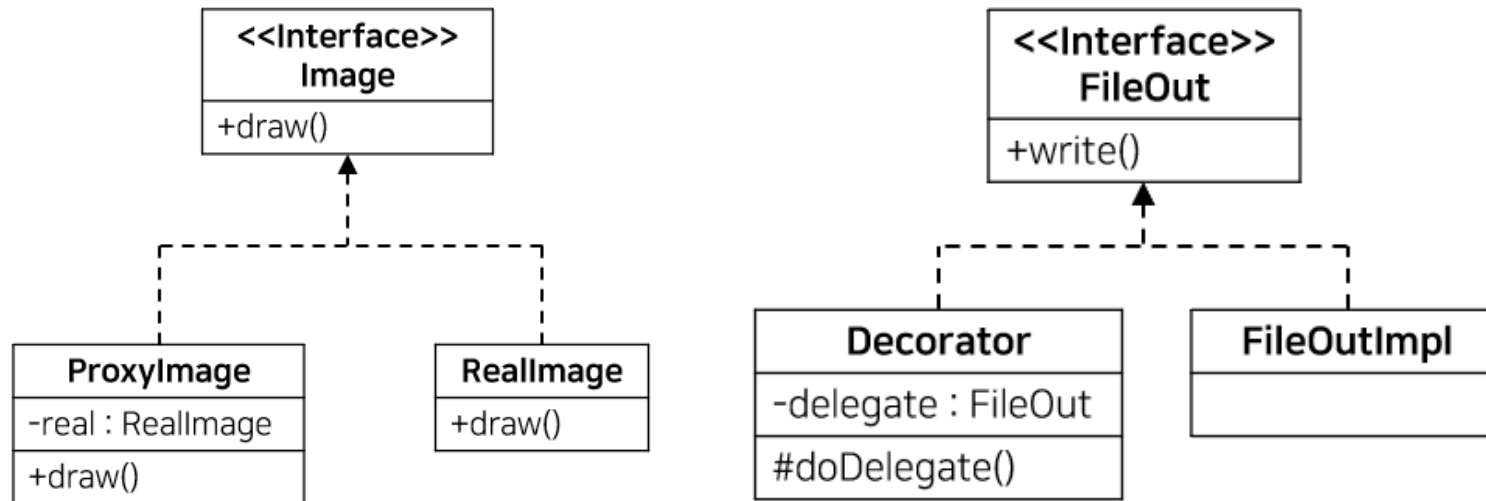
```
public class ProtectedService extends Service {  
  
    @Override  
    public void someMethod() {  
        if (!CurrentContext.getAuth().isAdmin()) {  
            throw new AccessDeniedException();  
        }  
        super.someMethod();  
    }  
}
```

- 상속 방식은 위임 방식에 비해 구조가 단순해 구현이 비교적 쉽다.
→ 하지만 객체를 생성하는 순간 실제 객체가 생성되기 때문에 가상 프록시를 구현하기엔 부적합

6 프록시(Proxy) 패턴

✓ Note

- 위임 기반의 프록시 패턴 구현은 데코레이터 패턴의 구현과 매우 유사하다.
→ 의도에서 분명한 차이가 있다.



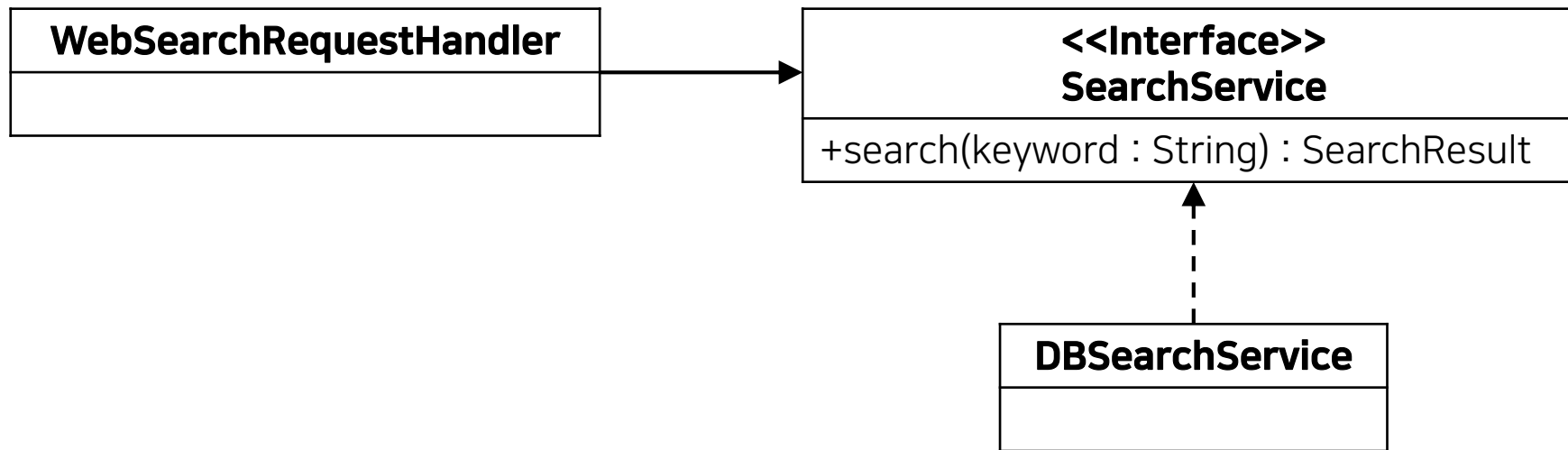
- 프록시 패턴은 실제 객체에 대한 접근을 제어하는데 초점이 맞춰져 있다.
 - 반면 데코레이터 패턴은 기존 객체의 기능을 확장하는데 초점을 맞추고 있다.

클래스 이름을 부여할 때 의도에 맞는 단어를 선택하자.

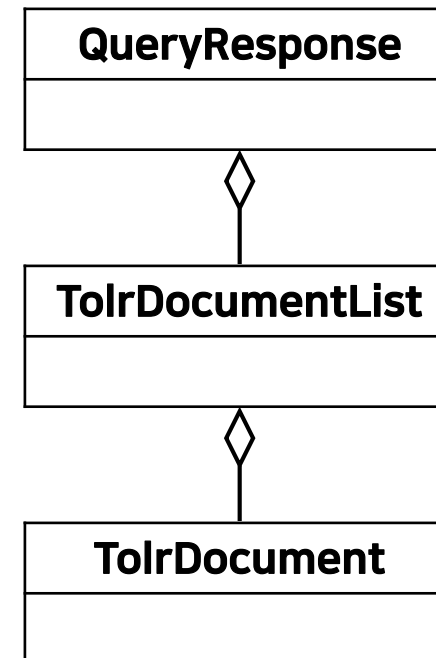
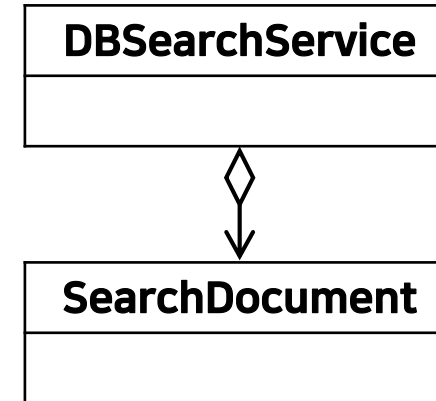
7 어댑터(Adapter) 패턴

7 어댑터(Adapter) 패턴

- ✓ DB를 이용해 통합 검색 기능을 추가하는 상황

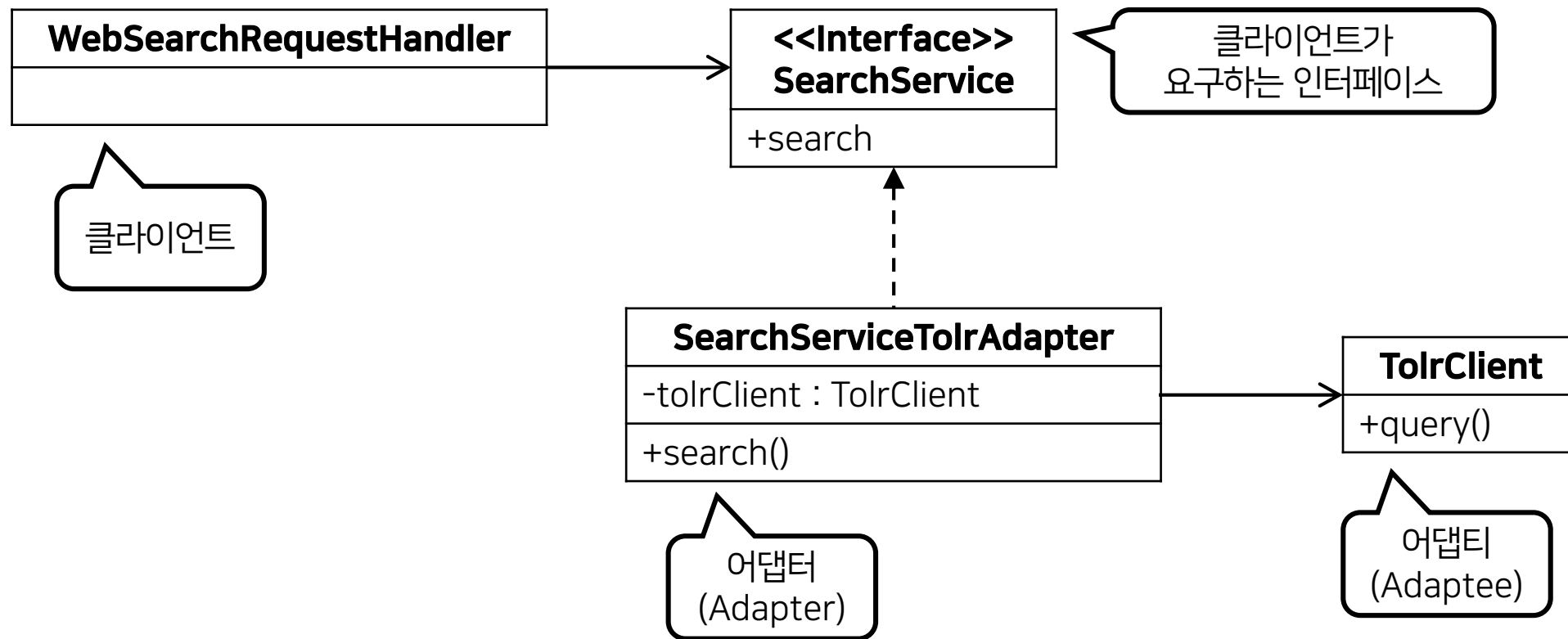


7 어댑터(Adapter) 패턴



7 어댑터(Adapter) 패턴

✓ 어댑터 패턴의 구조

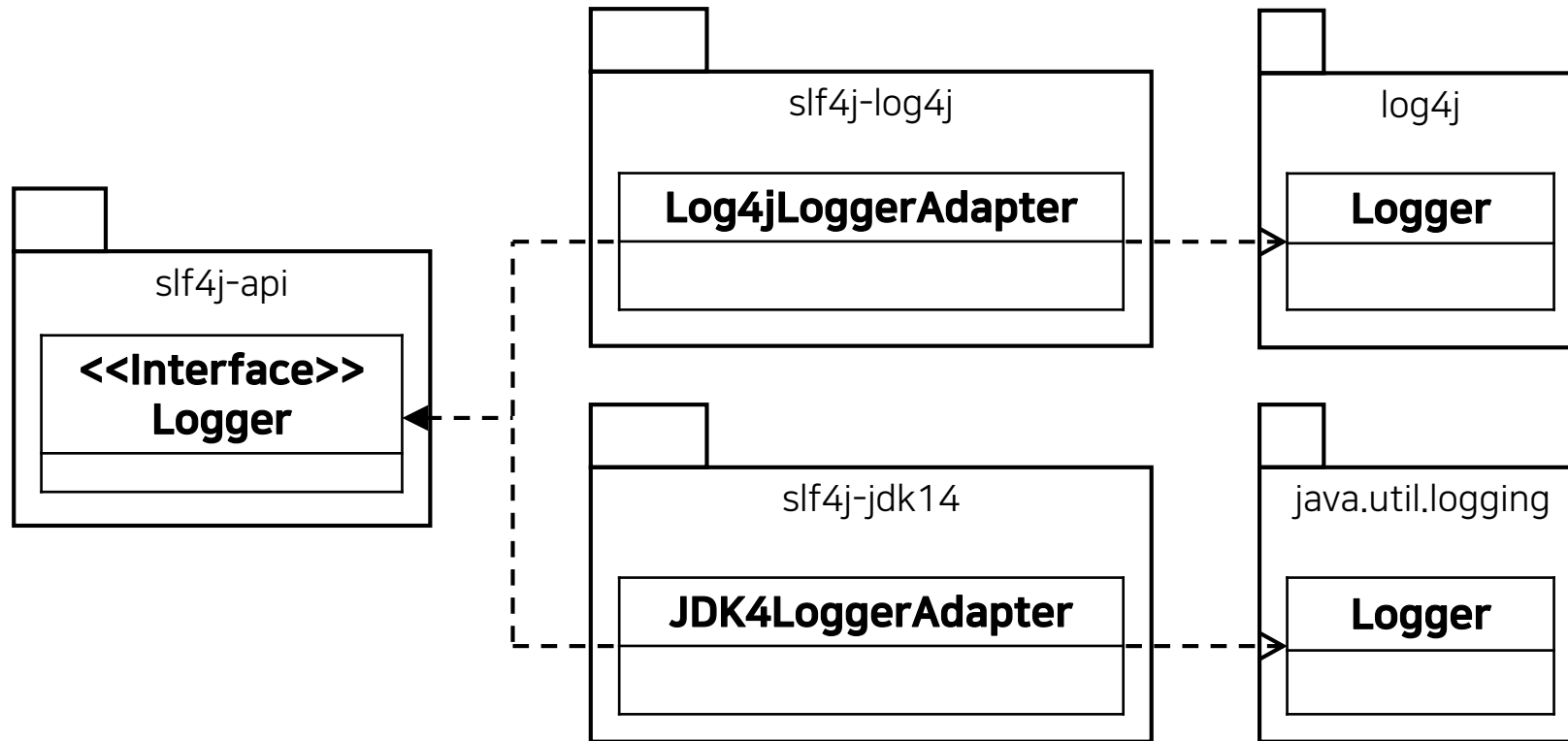


7 어댑터(Adapter) 패턴

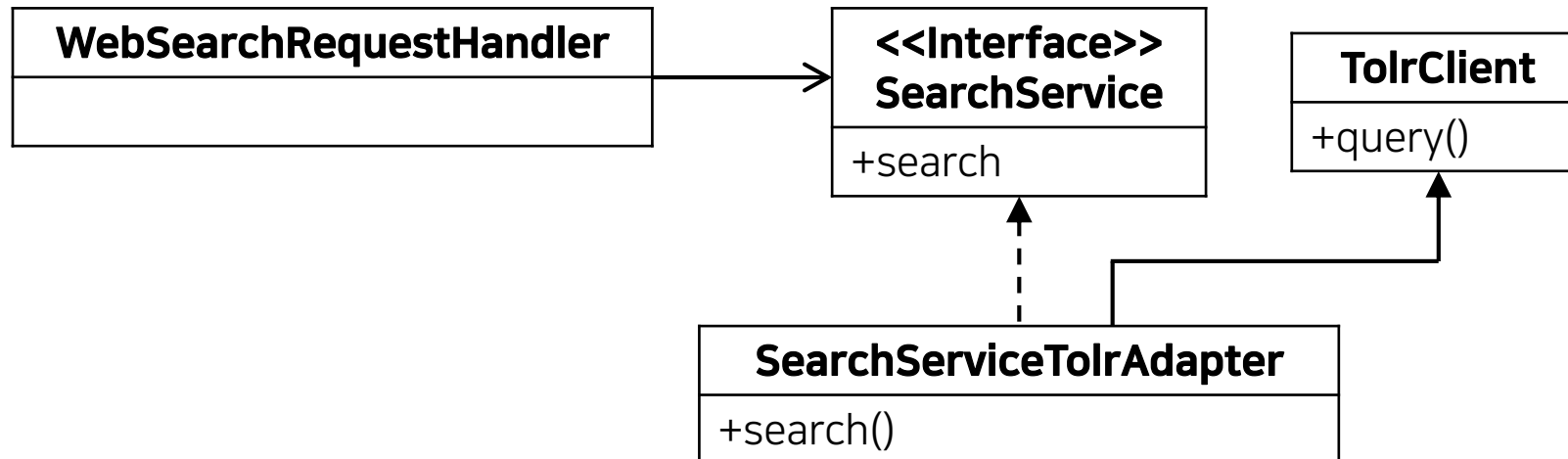
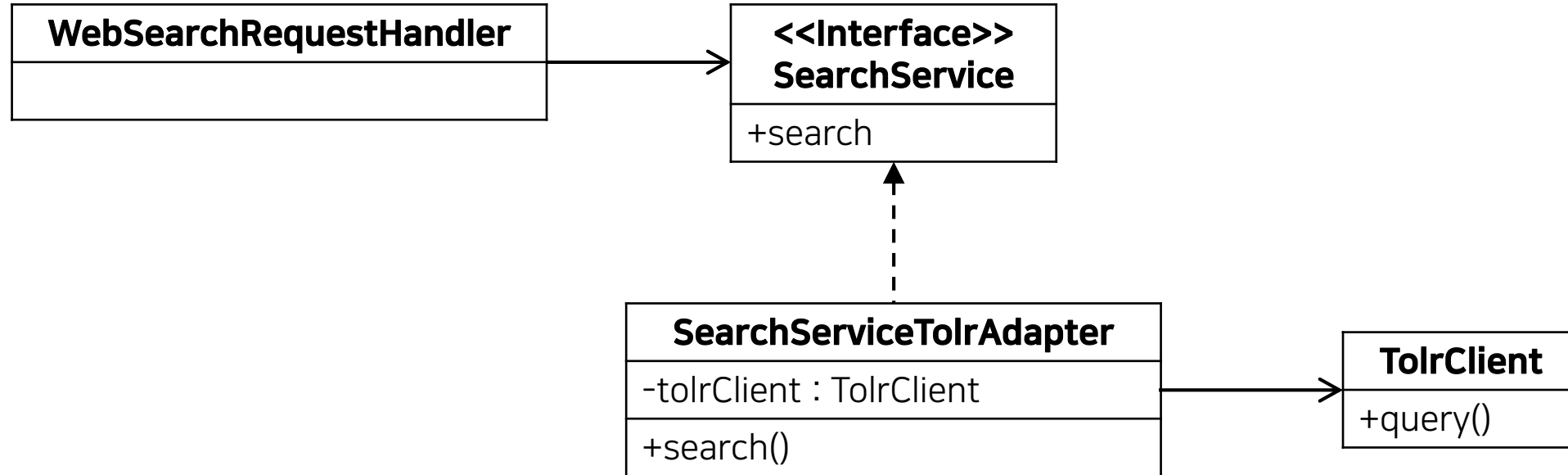
```
public class SearchServiceTolrAdapter implements SearchService {  
    private TolrClient tolrClient = new TolrClient();  
  
    public SearchResult search(String keyword) {  
        // keyword를 tolrClient가 요구하는 형식으로 변환  
        TolrQuery tolrQuery = new TolrQuery(keyword);  
  
        // TolrClient 기능 실행  
        QueryResponse response = tolrClient.query(tolrQuery);  
  
        // TolrClient의 결과를 SearchResult로 변환  
        SearchResult result = convertToResult(response);  
        return result;  
    }  
  
    private SearchResult convertToResult(QueryResponse response) {  
        List<TolrDocument> tolrDocs =  
            response.getDocumentList().getDocuments();  
  
        List<SearchDocument> docs = new ArrayList<>();  
  
        for (TolrDocument tolrDoc : tolrDocs) {  
            docs.add(new SearchDocument(tolrDoc.getId(), ...));  
        }  
  
        return new SearchResult(docs);  
    }  
}
```


7 어댑터(Adapter) 패턴

- ✓ 어댑터 패턴이 적용된 예 - SLF4J라는 로깅 API



7 어댑터(Adapter) 패턴



7 어댑터(Adapter) 패턴

- ✓ 상속을 이용할 시 상위 클래스의 메서드를 호출

```
public class SearchServiceTolrAdapter implements SearchService {  
  
    private TolrClient tolrClient = new TolrClient();  
  
    public SearchResult search(String keyword) {  
        // keyword를 tolrClient가 요구하는 형식으로 변환  
        TolrQuery tolrQuery = new TolrQuery(keyword);  
  
        // TolrClient 기능 실행  
        QueryResponse response = tolrClient.query(tolrQuery);  
  
        // TolrClient의 결과를 SearchResult로 변환  
        SearchResult result = convertToResult(response);  
        return result;  
    }  
}
```

조립

```
public class SearchServiceTolrAdapter extends TolrClient  
    implements SearchService {  
  
    public SearchResult search(String keyword) {  
        // keyword를 tolrClient가 요구하는 형식으로 변환  
        TolrQuery tolrQuery = new TolrQuery(keyword);  
  
        // TolrClient 기능 실행  
        QueryResponse response = super.query(tolrQuery);  
  
        // TolrClient의 결과를 SearchResult로 변환  
        SearchResult result = convertToResult(response);  
        return result;  
    }  
}
```

상속

8 옵저버(Observer) 패턴

8 옵저버(Observer) 패턴

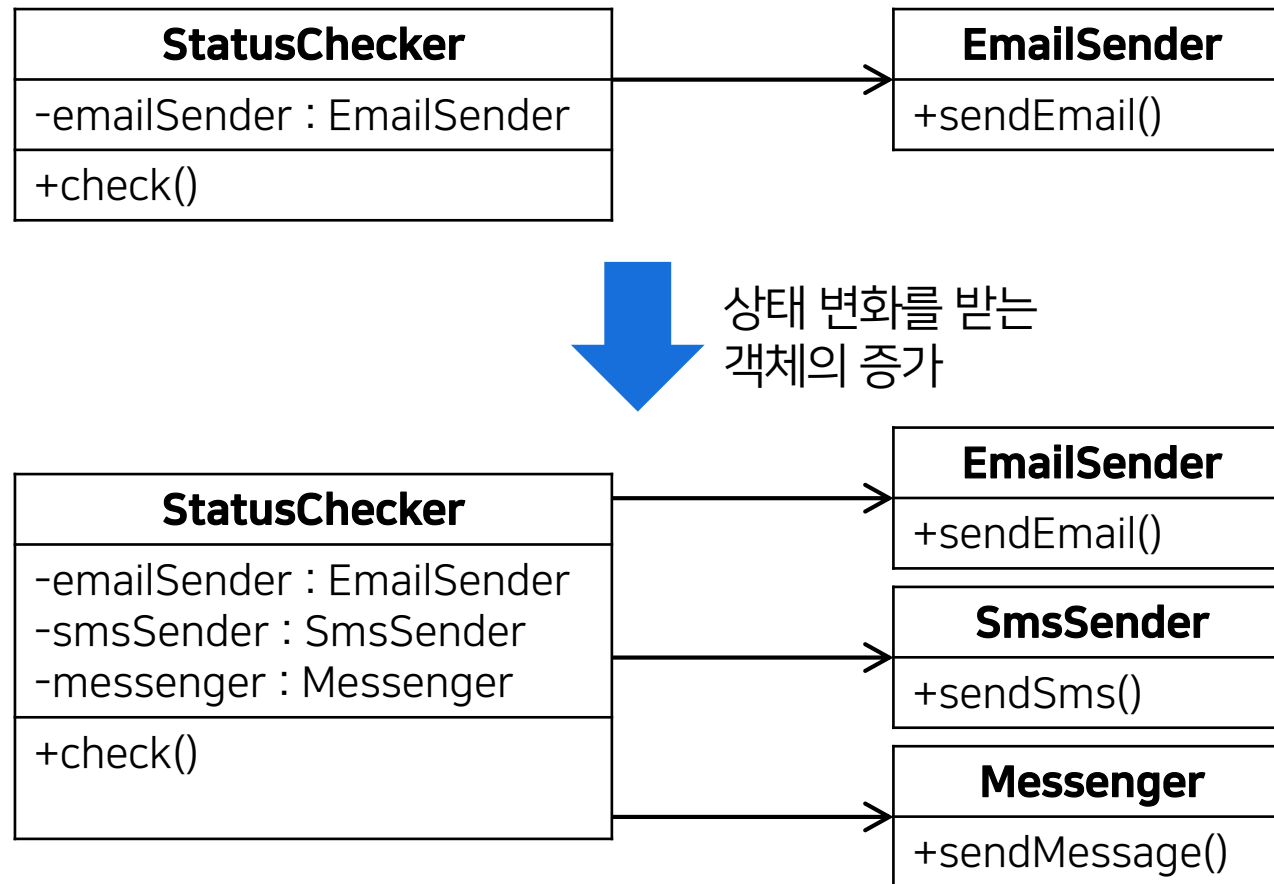
- ✓ 상태를 확인해서 알리는 시스템

```
public class StatusChecker {  
  
    private EmailSender emailSender;  
  
    public void check() {  
        Status status = loadStatus();  
  
        if (status.isNotNormal()) {  
            emailSender.sendEmail(status);  
        }  
    }  
}
```

```
public class StatusChecker {  
  
    private EmailSender emailSender;  
    private SmsSender smsSender;  
  
    public void check() {  
        Status status = loadStatus();  
  
        if (status.isNotNormal()) {  
            emailSender.sendEmail(status);  
            smsSender.sendSms(status);  
        }  
    }  
}
```

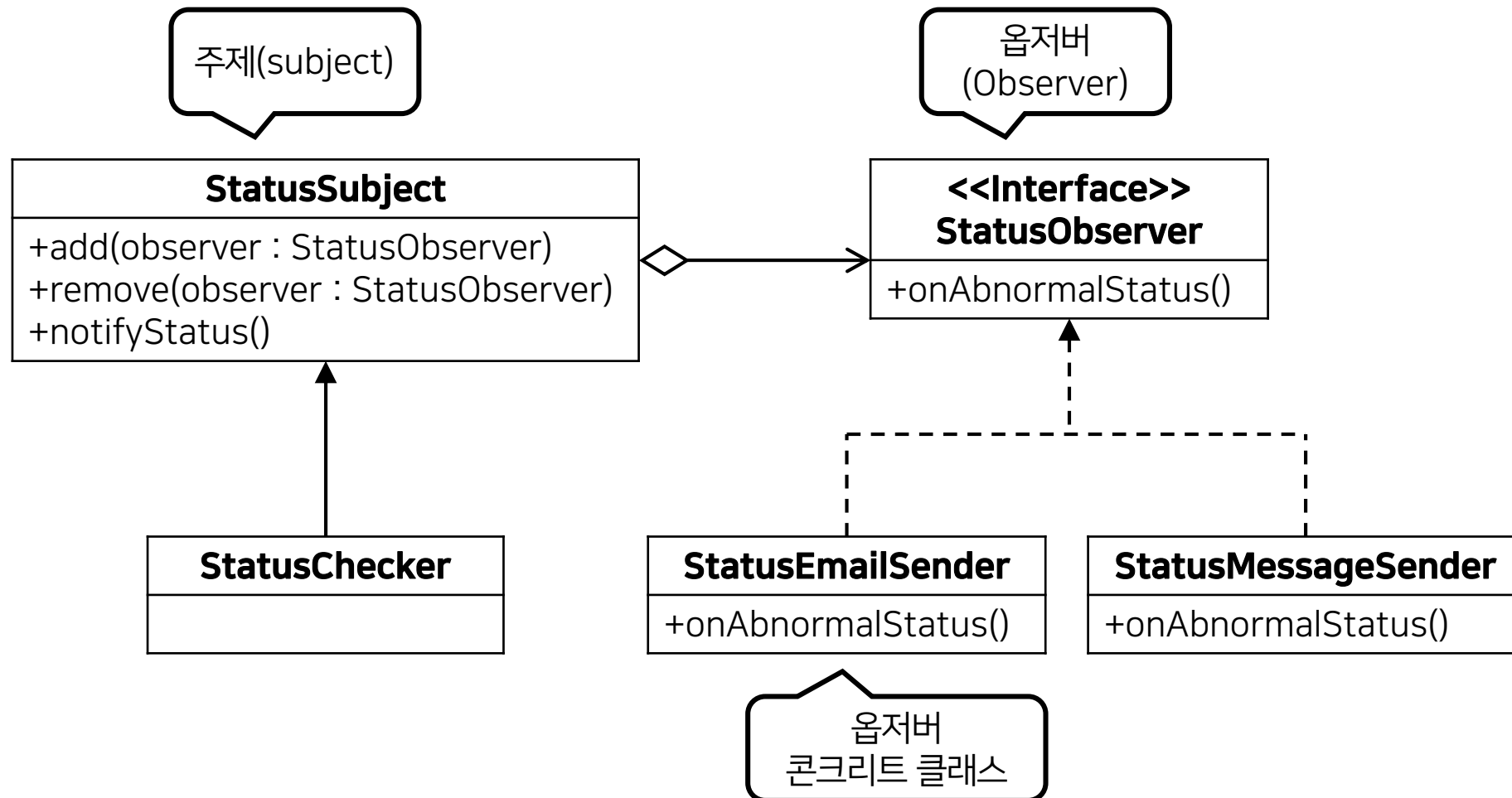
8 옵저버(Observer) 패턴

✓ 요구가 계속 들어오면...?



8 옵저버(Observer) 패턴

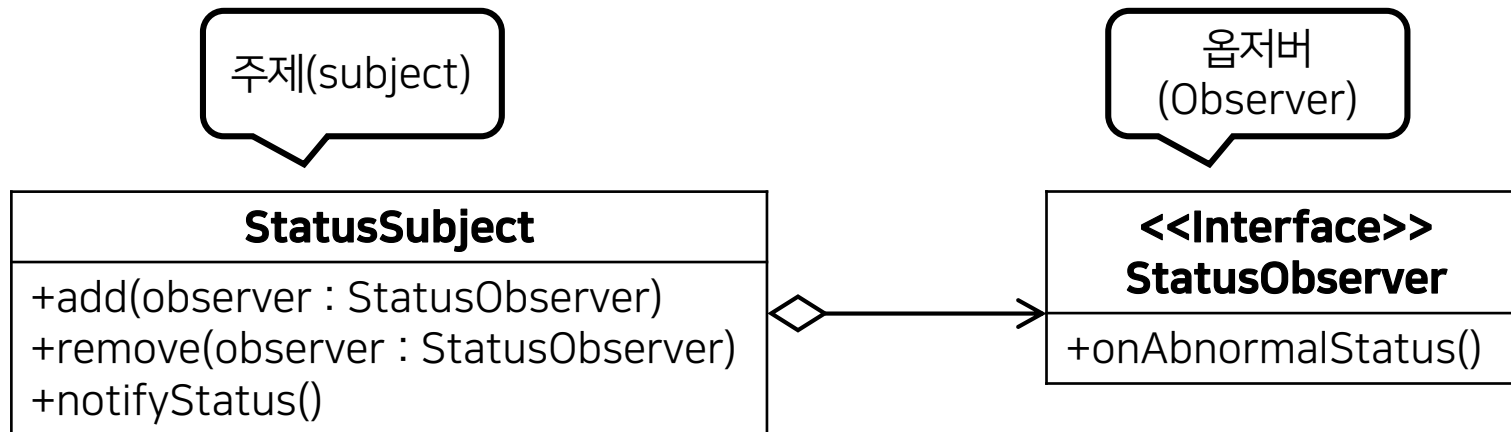
✓ 옵저버 패턴의 구조



8 옵저버(Observer) 패턴

✓ 주제 객체의 두 가지 책임

- 옵저버 목록을 관리하고, 옵저버를 등록하고 제거할 수 있는 메서드를 제공
- 상태의 변경이 발생하면 등록된 옵저버에 변경 내역을 알린다.



8 옵저버(Observer) 패턴

✓ 주제 클래스 구현

```
public abstract class StatusSubject {  
  
    private List<StatusObserver> observers = new ArrayList<>();  
  
    public void add(StatusObserver observer) {  
        observers.add(observer);  
    }  
  
    public void remove(StatusObserver observer) {  
        observers.remove(observer);  
    }  
  
    public void notifyStatus(Status status) {  
        for (StatusObserver observer : observers) {  
            observer.onAbnormalStatus(status);  
        }  
    }  
}
```

```
public class StatusChecker extends StatusSubject {  
  
    public void check() {  
        Status status = loadStatus();  
  
        if (status.isNotNormal()) {  
            super.notifyStatus(status);  
        }  
    }  
  
    private Status loadStatus() {  
        // ...  
    }  
}
```

8 옵저버(Observer) 패턴

✓ 옵저버 인터페이스 구현

```
public interface StatusObserver {  
    void onAbnormalStatus(Status status);  
}
```

```
public class StatusEmailSender implements StatusObserver {  
  
    @Override  
    public void onAbnormalStatus(Status status) {  
        sendEmail(status);  
    }  
  
    private void sendEmail(Status status) {  
        // ... 이메일 전송 코드  
    }  
}
```

8 옵저버(Observer) 패턴

- ✓ 옵저버 객체를 주제 객체에 등록

```
StatusChecker checker = new StatusChecker();  
checker.add(new StatusEmailSender()); // 옵저버로 등록
```

```
StatusChecker checker = new StatusChecker();  
  
// 새로운 타입의 옵저버가 추가되어도 StatusChecker 코드는 바뀌지 않는다.  
StatusObserver faultObserver = new FaultStatusSMSSender();  
checker.add(faultObserver);  
checker.add(new StatusEmailSender());
```

8 옵저버 객체에게 상태 전달 방법

- ✓ 옵저버 객체가 기능 수행을 위해 주제 객체의 **상태**가 필요할 수 있다.

```
// 주제 객체가 옵저버 호출시 상태 값 전달
public abstract class StatusSubject {

    private List<StatusObserver> observers = new ArrayList<>();

    // ...

    public void notifyStatus(Status status) {
        for (StatusObserver observer : observers) {
            observer.onAbnormalStatus(status); // 상태를 옵저버에 전달
        }
    }
}
```

```
// 옵저버는 파라미터로 전달받은 상태 값을 사용
public class FaultStatusSMSSender implements StatusObserver {

    @Override
    public void onAbnormalStatus(Status status) {
        if (status.isFault()) { // 전달받은 상태 값을 사용
            sendSMS(status);
        }
    }

    private void sendSMS(Status status) {
        // ...
    }
}
```

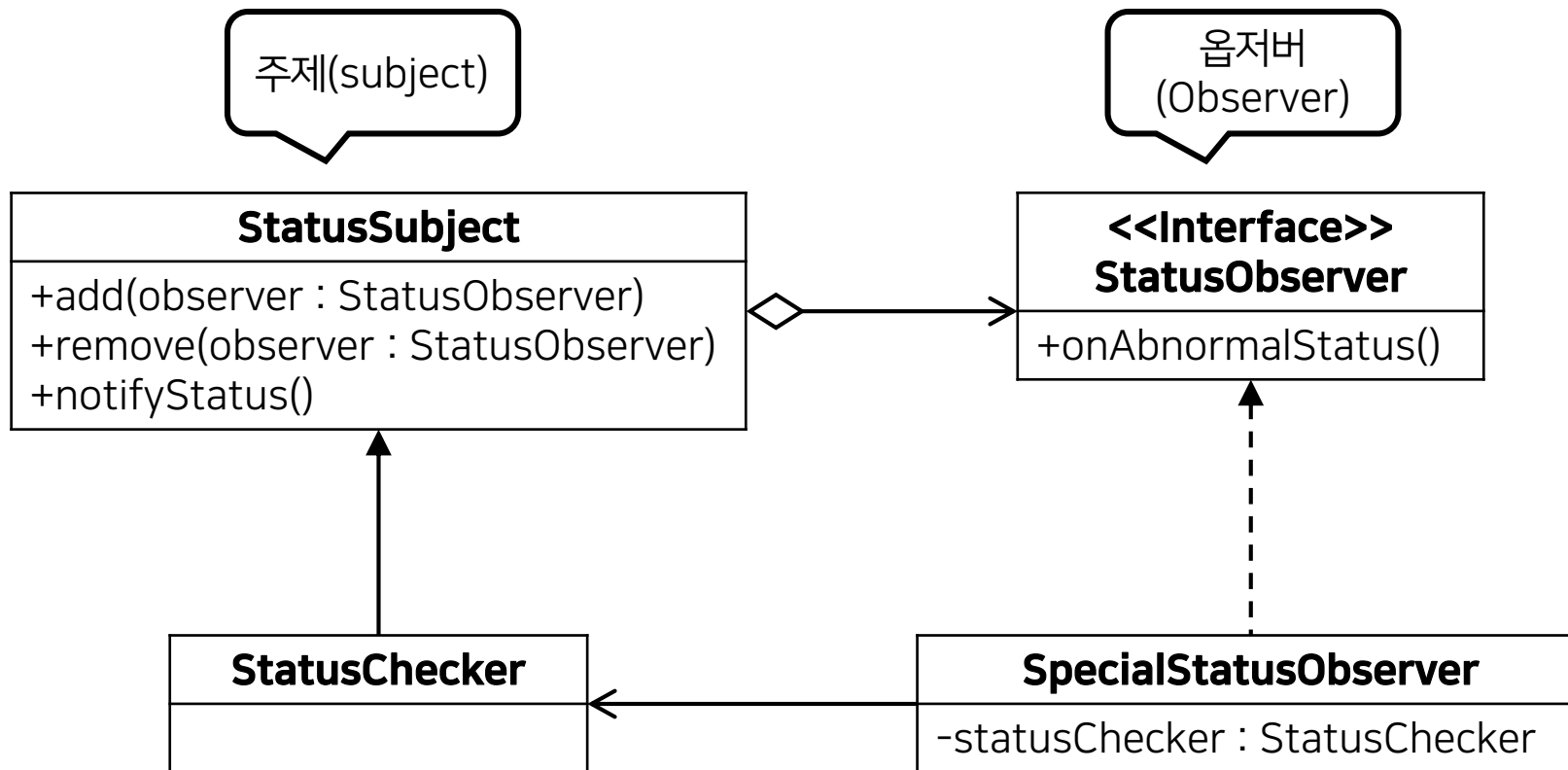
8 옵저버 객체에게 상태 전달 방법

- ✓ 옵저버 객체가 주제 객체에 직접 **접근**하는 방법도 있다.

```
public class SpecialStatusObserver implements StatusObserver {  
  
    private StatusChecker statusChecker;  
    private Siren siren;  
  
    public SpecialStatusObserver(StatusChecker statusChecker) {  
        | this.statusChecker = statusChecker;  
    }  
  
    @Override  
    public void onAbnormalStatus(Status status) {  
        | // 특정 타입의 주제 객체에 접근  
        | if (status.isFault() && statusChecker.isContinuousFault()) {  
        | | siren.begin();  
        | }  
    }  
}
```

8 옵저버 객체에게 상태 전달 방법

- ✓ 옵저버 클래스는 주제 클래스에 대해 의존을 가질 수 있다.



8 옵저버에서 주제 객체 구분

- ✓ 옵저버 패턴이 가장 **많이** 사용되는 영역
 - GUI 프로그래밍 영역
 - 버튼이 눌릴 때 로그인 기능을 호출한다고 하면?
 - ▶ 버튼이 주제 객체, 로그인 모듈을 호출하는 객체가 옵저버가 된다.

```
public class MyActivity extends Activity implements View.OnClickListener {  
  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
        // ...  
        Button loginButton = getViewById(R.id.main_loginbtn);  
        loginButton.setOnClickListener(this);  
    }  
  
    @Override  
    public void onClick(View view) { // OnClickListener의 메서드  
        login(id, password);  
    }  
}
```

8 옵저버에서 주제 객체 구분

- ✓ 한 개의 옵저버 객체를 여러 주제 객체에 등록할 수도 있다.

```
public class MyActivity extends Activity implements View.OnClickListener {
```

```
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
        // ...  
        // 두 개의 버튼에 동일한 OnClickListener 객체 등록  
        Button loginButton = (Button) findViewById(R.id.main_loginbtn);  
        loginButton.setOnClickListener(this);  
        Button logoutButton = (Button) findViewById(R.id.main_logoutbtn);  
        logoutButton.setOnClickListener(this);  
    }
```

@Override

```
public void onClick(View view) { // OnClickListener의 메서드  
    // 주제 객체를 구분할 수 있는 방법 필요  
    if (view.getId() == R.id.main_loginbtn) {  
        login(id, password);  
    } else if (view.getId() == R.id.main_logoutbtn) {  
        logout();  
    }  
}
```

@Override

```
public void onClick(View view) { // OnClickListener의 메서드  
    // 주제 객체를 구분할 수 있는 방법 필요  
    if (view == loginButton) {  
        login(id, password);  
    } else if (view == logoutButton) {  
        logout();  
    }  
}
```


8 옵저버에서 주제 객체 구분

- ✓ 주제 클래스는 옵저버 객체를 **관리**하기 위한 기능을 제공한다.
 - 한 주제에 대한 **다양**한 구현 클래스가 존재한다면?
 - ▶ 옵저버 관리 / 통지기능을 제공하는 추상 클래스로 불필요한 동일한 코드중복을 방지한다.

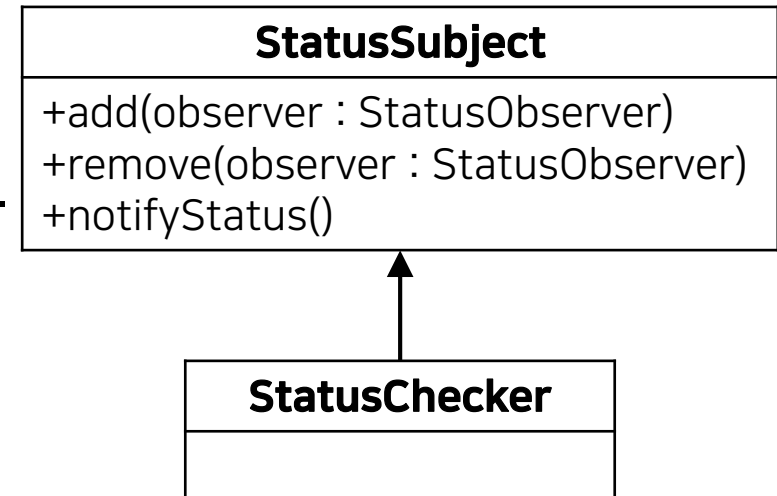
```
public void add(StatusObserver observer) {}
```

StatusSubject 클래스

```
public void setOnClickListener(OnClickListener o) {}
```

View 클래스

- 주제 클래스가 **한 개**뿐이라면?
 - ▶ 옵저버 관리를 위한 추상 클래스를 따로 만들 필요는 없다.



8 옵저버 패턴 구현의 고려 사항

- ✓ 옵저버 패턴을 구현할 때 **고려**해야 하는 사항
 - 주제 객체의 통지 기능 실행 **주체**
 - 옵저버 인터페이스의 **분리**
 - **통지 시점**에서의 주제 객체 상태
 - 옵저버 객체의 실행 **제약 조건**

8 옵저버 패턴 구현의 고려 사항

- ✓ 주제 객체의 통지 기능 실행 **주체**

```
public class StatusChecker extends StatusSubject {  
    public void check() {  
        Status status = loadStatus();  
  
        if (status.isNotNormal()) {  
            super.notifyStatus(status); // StatusChecker가 옵저버에 대한 통지 요청  
        }  
    }  
}
```

8 옵저버 패턴 구현의 고려 사항

- ✓ 주제 객체의 통지 기능 실행 주체

```
StatusChecker checker1 = ...;
StatusChecker checker2 = ...;

checker1.check();
checker2.check();

if (checker1.isLastStatusFault() && checker2.isLastStatusFault()) {
    checker1.notifyStatus();
    checker1.notifyStatus();
}
```

8 옵저버 패턴 구현의 고려 사항

✓ 옵저버의 인터페이스 **개수**

```
public interface EventObserver {  
    void onClick(View v);  
    void onScroll(View v);  
    void onTouch(View v);  
    // ...  
}
```

하나의 옵저버 인터페이스에 몰려있음

```
public class OnlyClickObserver implements EventObserver {  
  
    @Override  
    public void onClick(View v) {  
        // ... 이벤트 처리 코드  
    }  
  
    @Override  
    public void onScroll(View v) {  
        /* 빈 구현 */  
    }  
  
    @Override  
    public void onTouch(View v) {  
        /* 빈 구현 */  
    }  
    // ... 다른 메서드의 빈 구현  
}
```

8 옵저버 패턴 구현의 고려 사항

- ✓ 통지 시점에서 주제 객체의 상태에 결함이 없어야 한다.

```
public class AnySubject extends SomeSubject {  
  
    @Override  
    public void changeState(int newValue) {  
        super.changeState(newValue); // 상위 ChangeState()에서 옵저버에 통지  
        // 아래 코드가 실행되기 전에 옵저버가 상태를 조회  
        if (isStateSome()) {  
            state += newValue;  
        }  
    }  
}
```

옵저버 객체는 완전하지 못한 상태값을 조회하게 된다.

8 옵저버 패턴 구현의 고려 사항

- ✓ 템플릿 메서드 패턴을 사용해서 **완전한** 상태 값을 사용하도록 한다.

```
// 상위 클래스
public class SomeSubject {
    // 템플릿 메서드로 구현
    public void changeState(int newValue) {
        internalChangeState(newState);
        notifyObserver();
    }

    protected void internalChangeState(int newState) {
        // ...
    }
}
```

```
// 하위 클래스
public class AnySubject extends SomeSubject {
    // internalChangeState() 메서드 실행 이후, 옵저버에 통지
    @Override
    protected void internalChangeState(int newValue) {
        super.internalChangeState(newValue);
        if (isStateSome()) {
            state += newValue;
        }
    }
}
```

8 옵저버 패턴 구현의 고려 사항

- ✓ 옵저버 객체의 실행에 대한 **제약 규칙**을 정해야 한다.

```
private void notifyToObserver() {  
    for (StatusObserver o : observers) {  
        o.onStatusChange();  
    }  
}  
  
public void changeState(int newState) {  
    internalChagneState(newState);  
    notifyToObserver();  
}
```

모든 옵저버 객체의 onStatusChange() 메서드 종료를 기다려야 한다.

8 옵저버 패턴 구현의 고려 사항

✓ 이 외에 **여러**가지 생각해 볼 만한 고려 사항들이 있다.

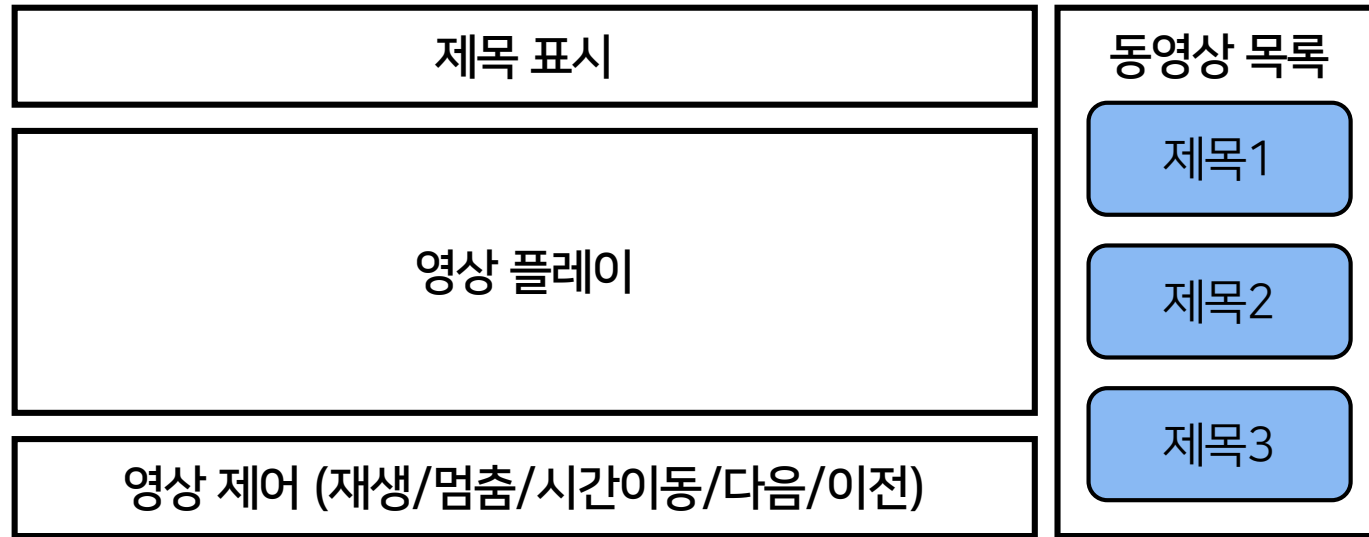
- ex1) 옵저버 객체에서 주제 객체의 상태를 다시 변경하면 어떻게 구현할 것인가?
- ex2) 옵저버 자체를 비동기로 실행하는 문제
- 등등...

▶ 이러한 문제는 주어진 **상황**에 따라 대답이 달라질 수 있다.

9 미디어이터(Mediator) 패턴

9 미디어이터(Mediator) 패턴

✓ 비디오 플레이어를 만든다고 가정

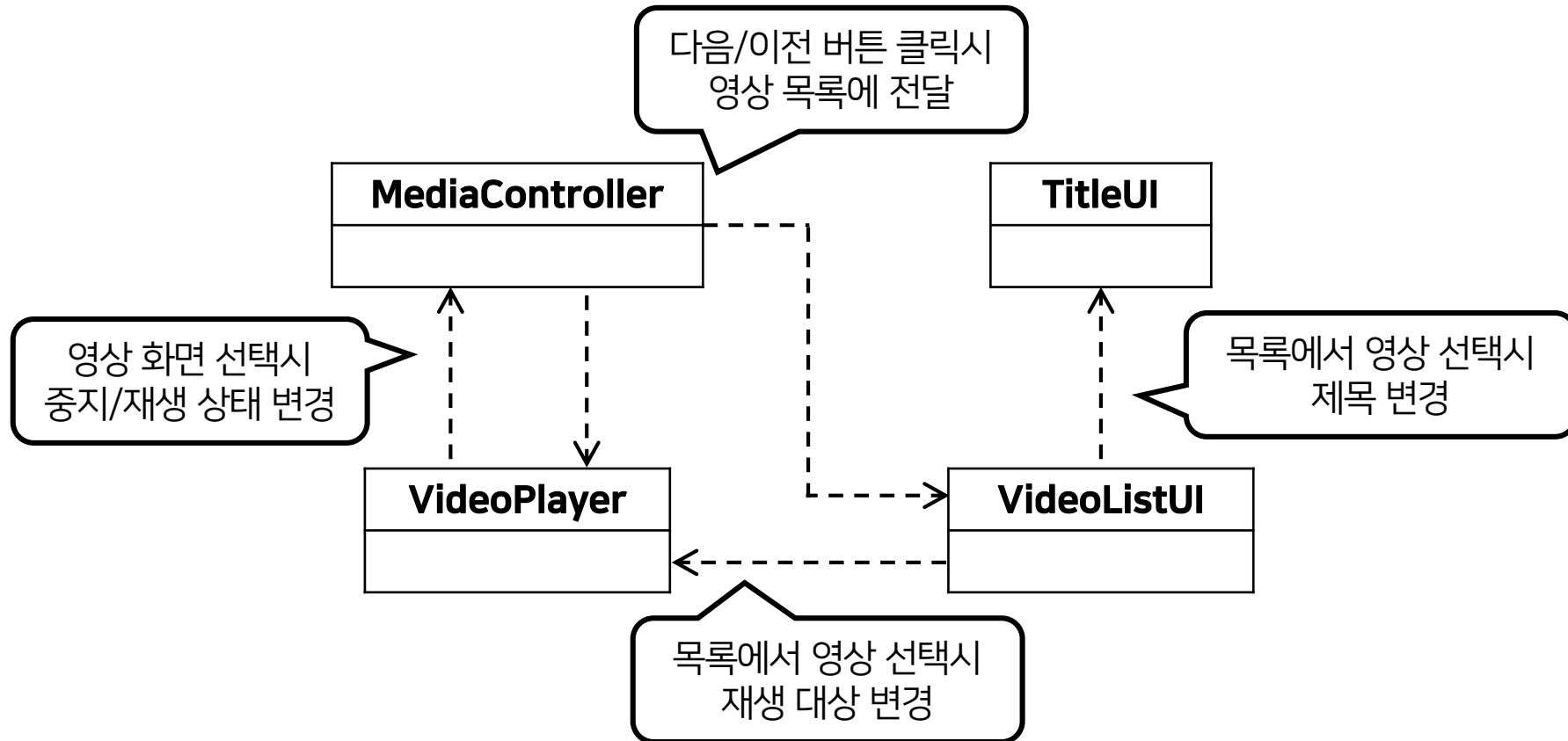


■ 주요 기능

- ▶ 목록에서 특정 제목 클릭시, 해당 제목과 동영상을 표시한다.
- ▶ 재생/멈춤을 누를시 플레이 영역은 재생하거나 멈추고, 시간이동 조작시 알맞은 시점으로 이동하여 재생
- ▶ 다음/이전을 누를시 영상 목록에서 다음이나 이전 영상을 재생
- ▶ 영상 플레이 화면 터치시 플레이가 멈추고, 다시 터치하면 재생. 제어 부분의 UI도 중지/시작 모양으로 변경

9 미디어이터(Mediator) 패턴

✓ 각 클래스의 의존 구조



몇 가지 단점 중 하나로, **재사용**이 어렵다.

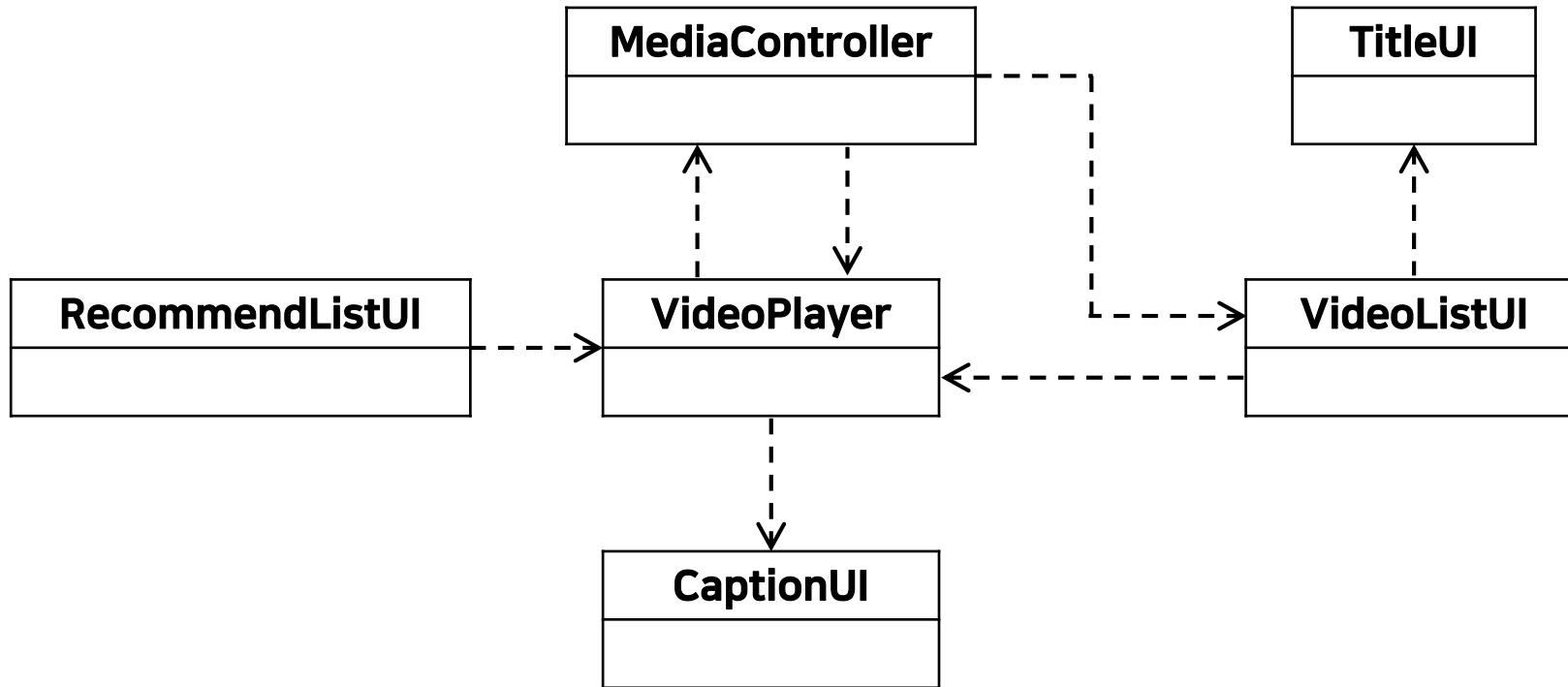
9 미디어이터(Mediator) 패턴

✓ 또 다른 단점으로, **클래스가 증가할수록** 개별 클래스의 **수정**이 어려워진다.

- 요구 사항 추가시

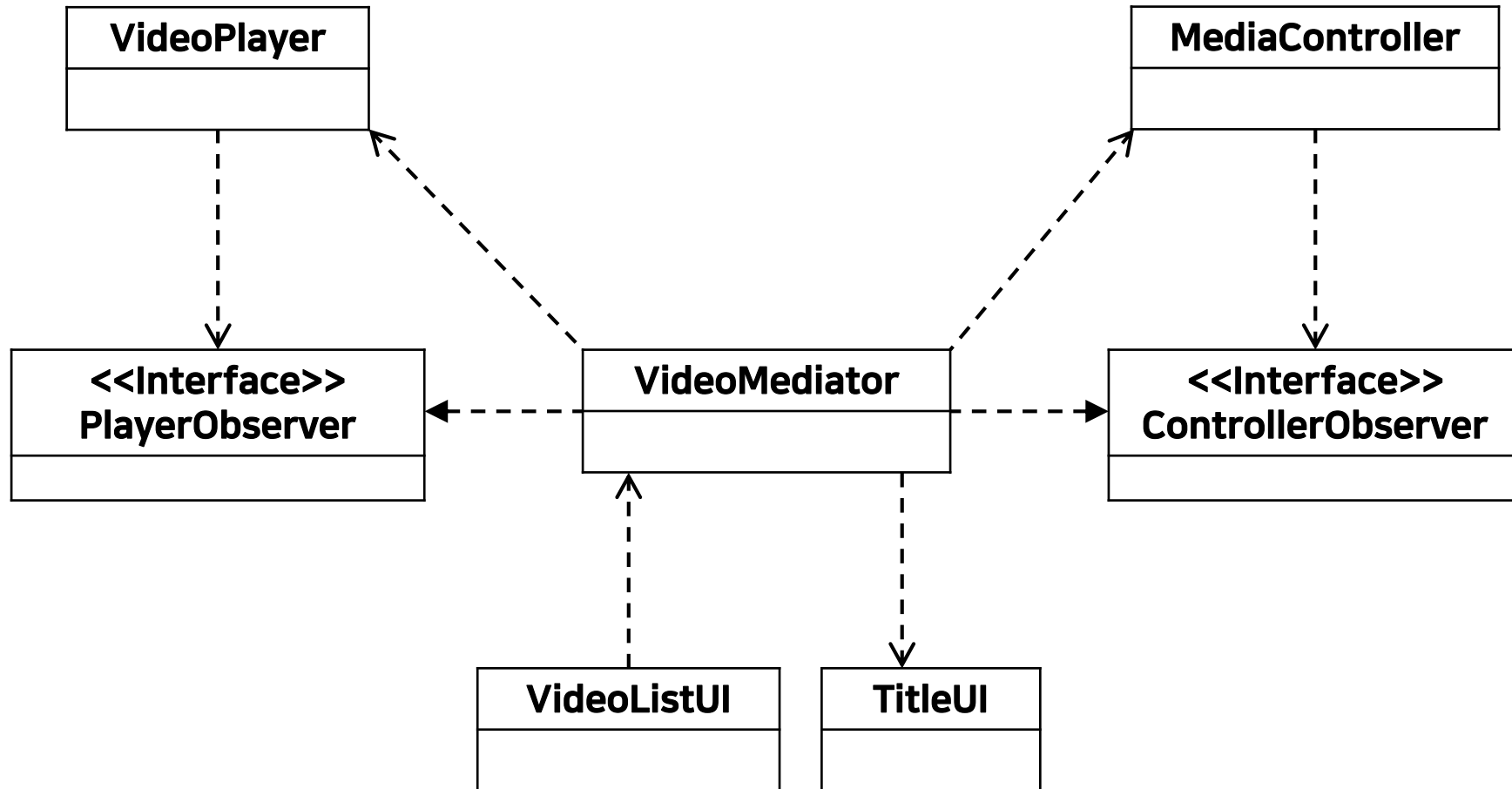
ex1) 플레이 화면 좌측에 시스템이 추천한 영상 목록을 표시

ex2) 영상 자막 존재 시, 자막 표시



9 미디어이터(Mediator) 패턴

- ✓ 미디어이터 패턴을 적용한 비디오 플레이어 구조



9 미디어이터(Mediator) 패턴

- ✓ 협업 객체는 미디어이터에 요청한다.

```
// VideoListUI
private VideoMediator videoMediator;

public void onSelectedItem(int selectedIdx) {
    VideoInfo videoInfo = videoList.get(selectedIdx);
    videoMediator.selectVideo(videoInfo.getFile());
}
```

9 미디어이터(Mediator) 패턴

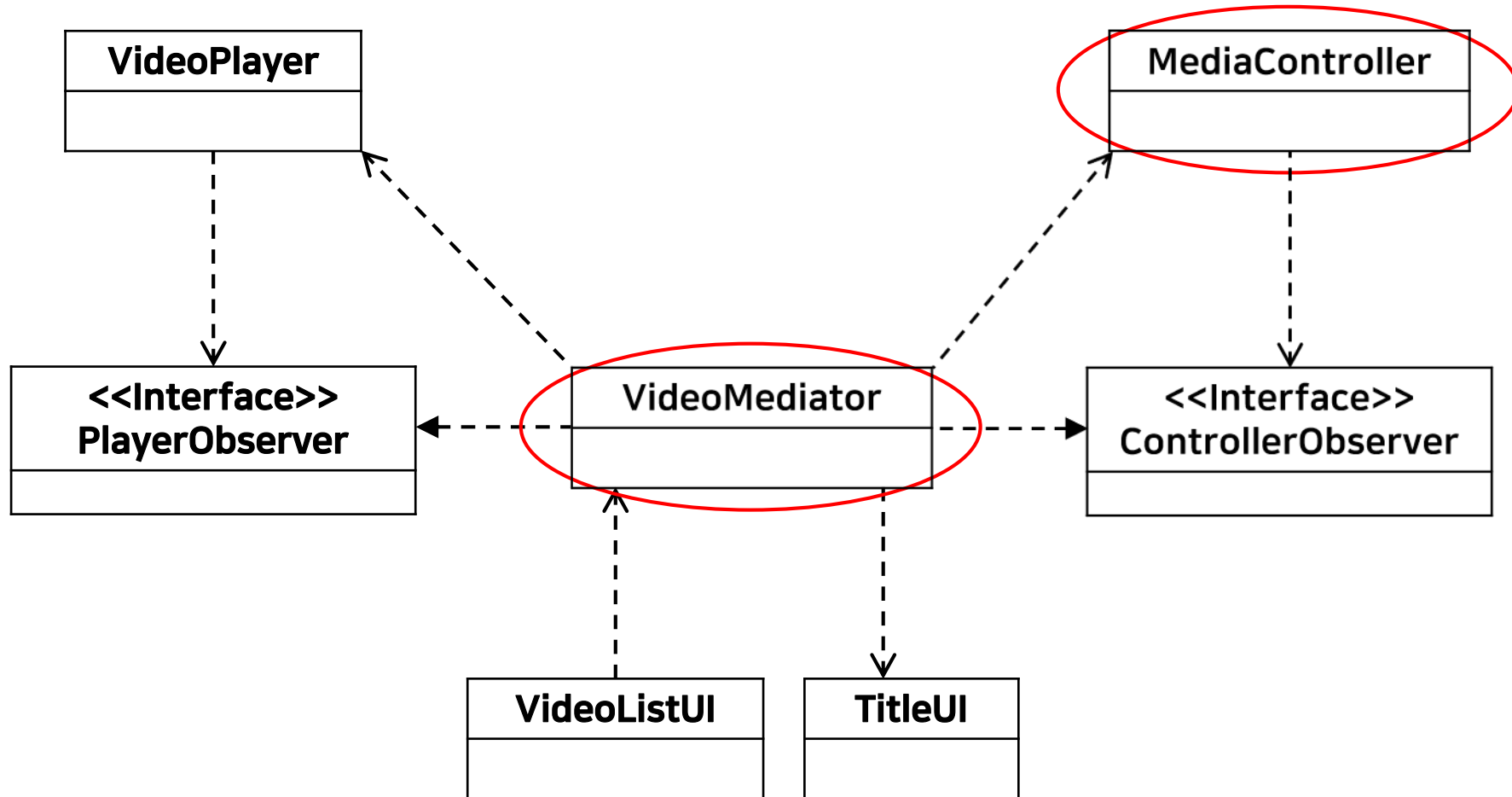
- ✓ 요청받은 미디어이터 객체는 협업 객체들에게 요청을 전달

```
// VideoMediator
private VideoPlayer videoPlayer;
private TitleUI titleUI;

public void selectVideo(File videoFile) {
    // 미디어이터는 다른 협업 객체에게 요청을 전달
    videoPlayer.play(videoFile);
    titleUI.setTitle(videoFile);
}
```

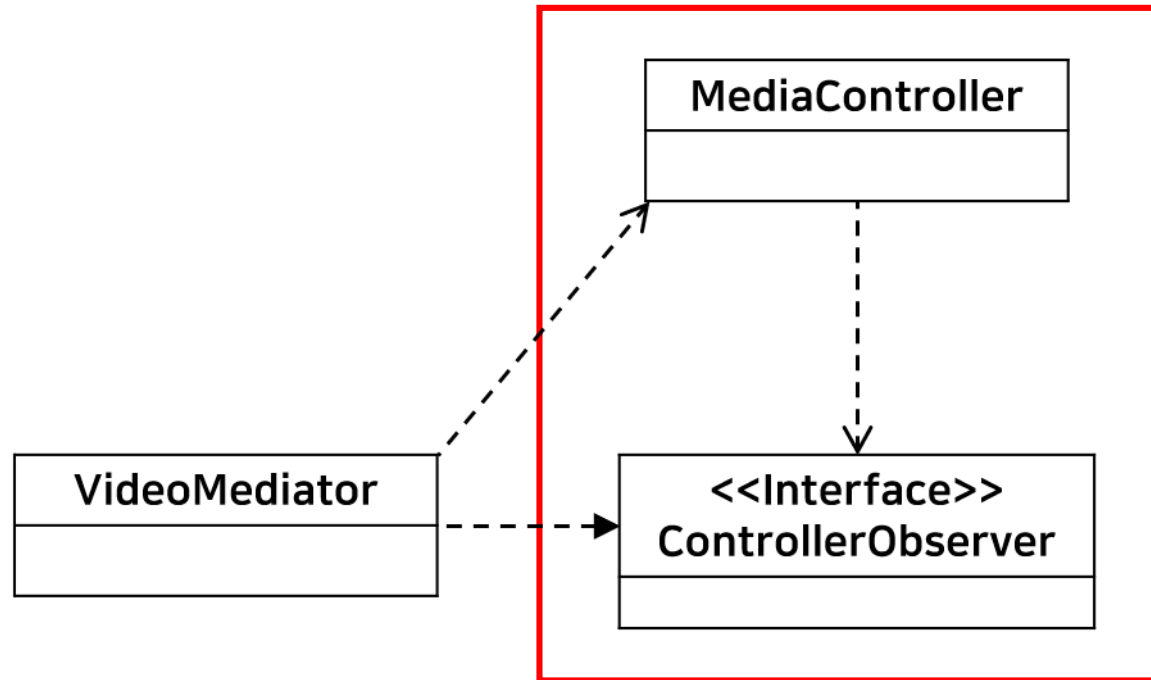

9 미디어이터(Mediator) 패턴

✓ 의존이 역전된다.



9 미디어이터(Mediator) 패턴

- ✓ 옵저버 패턴 적용으로 협업 클래스의 재사용을 높일 수 있다.

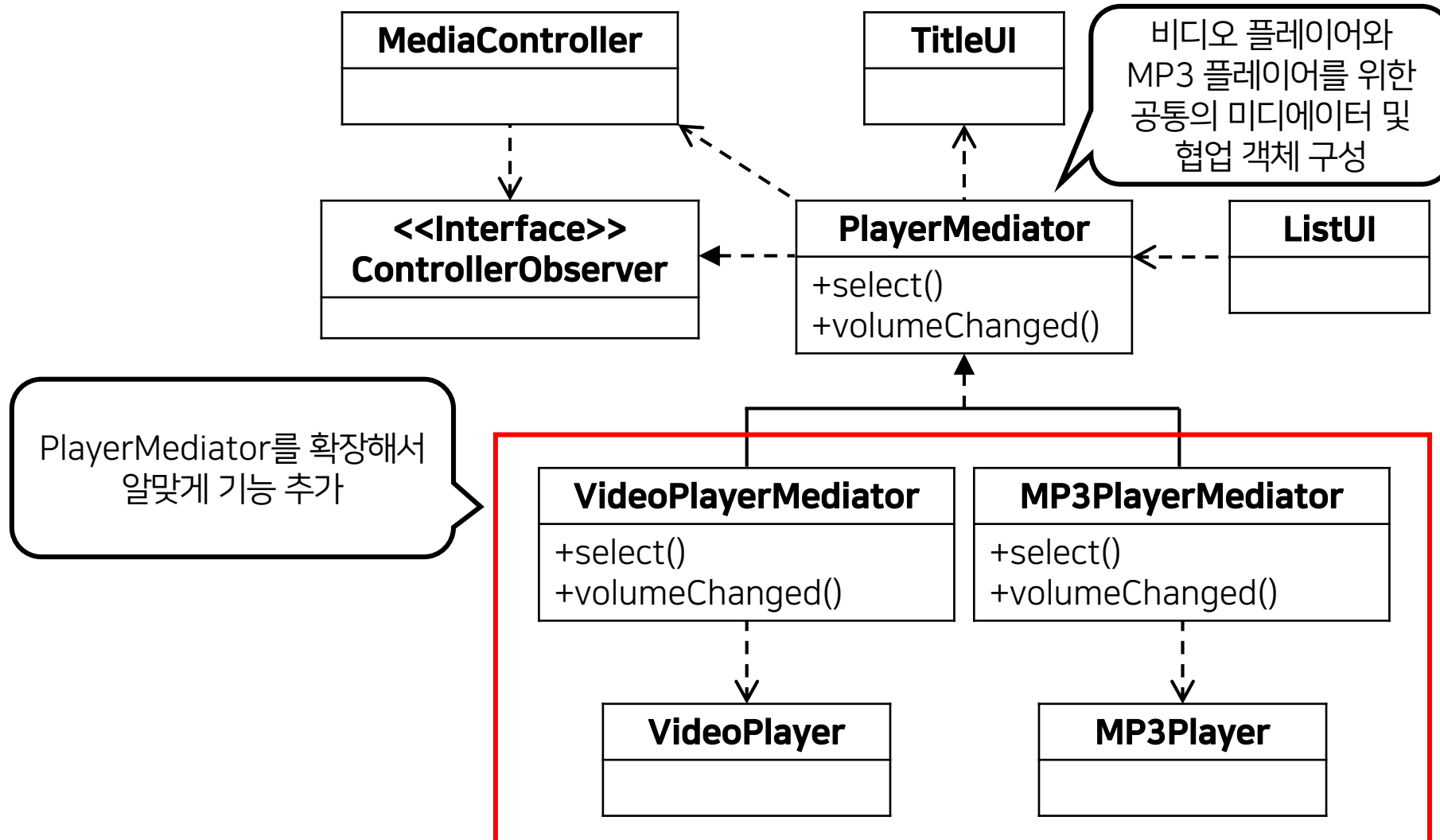


9 미디어이터(Mediator) 패턴

- 각 협업 클래스에 흩어져 있는 흐름 제어를 미디어이터로 모은다.
 - ▶ 각 협업 클래스의 코드는 단순해진다.
- 각 협업 클래스는 다른 협업 클래스에 의존하지 않는다.
 - ▶ 개별 협업 클래스를 수정/확장하거나 재사용하기가 쉬워진다.
- 미디어이터에 각 협업 객체의 흐름 제어 코드가 모여 있다.
 - ▶ 전체 협업 객체 간의 메시지 흐름을 이해/수정하고 확장하는 것을 쉽게 만들어준다.

9 추상 미디어이터 클래스의 재사용

- 협업 객체 간 동일한 메시지 흐름이 다른 기능에서 **반복** 사용될 경우



9 추상 미디어이터 클래스의 재사용

- 추상 미디어이터 클래스는 협업 객체 간 메시지 흐름을 **재사용**할 수 있도록 해준다.

```
public abstract class PlayerMediator implements ControllerObserver {
    private MediaController mediaController;
    private TitleUI titleUI;

    public PlayerMediator() {
        this.mediaController = new MediaController();
        this.mediaController.addObserver(this);
        this.titleUI = new TitleUI();
        // ...
    }

    public void select(File file) {
        titleUI.setTitle(file);
    }

    // ...
    // volumeChanged 의 구현은 제공하지 않음
    // MediaController 객체의 볼륨 조절 이벤트 발생시
    // volumeChagned() 메서드 호출
}
```

9 추상 미디어이터 클래스의 재사용

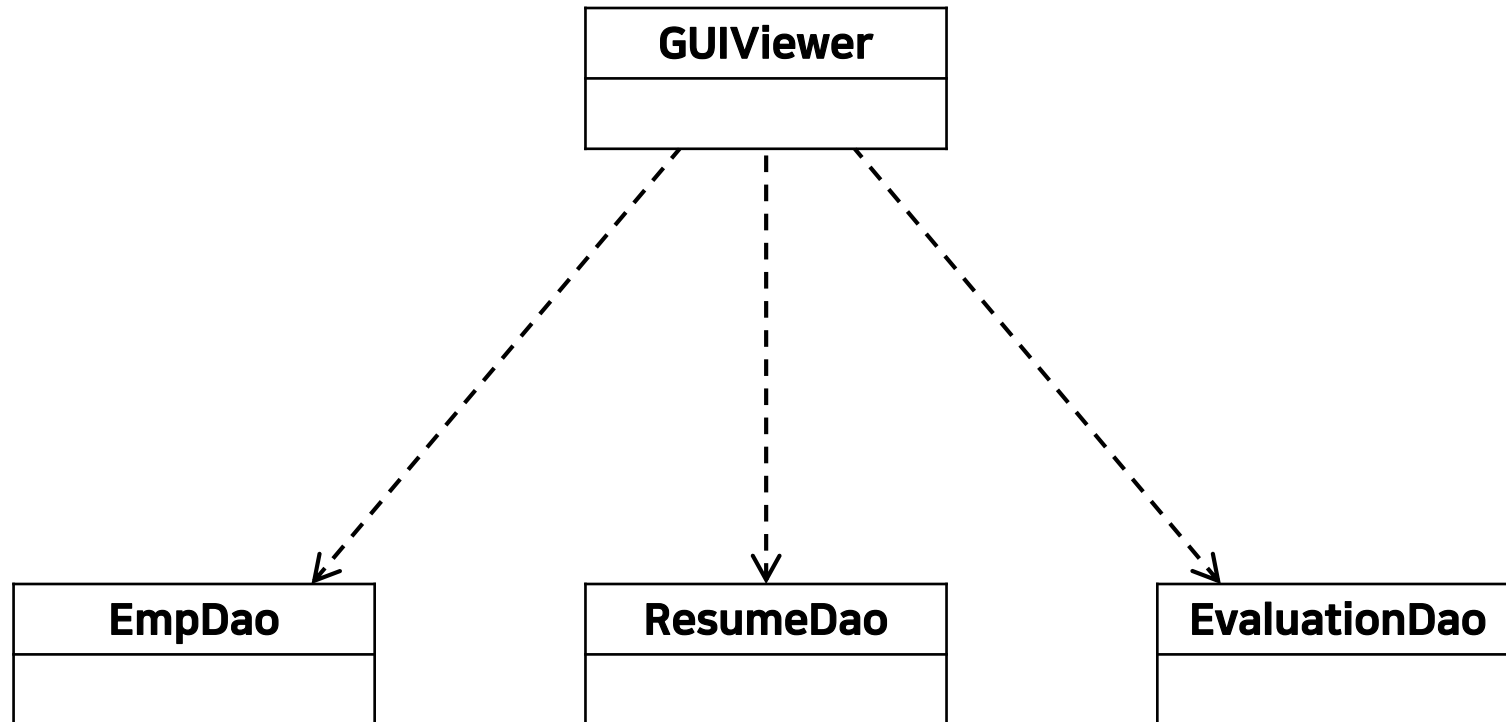
- PlayerMediator 클래스를 재사용하면서 필요한 기능 **확장**

```
public class VideoPlayerMediator extends PlayerMediator {  
    private VideoPlayer videoPlayer;  
  
    public VideoPlayerMediator() {  
        super();  
        this.videoPlayer = new VideoPlayer();  
    }  
  
    @Override  
    public void select(File file) {  
        videoPlayer.play(file);  
        super.select(file); // 상위 미디어이터에 정의된 협업 기능 재사용  
    }  
  
    // 하위 미디어이터에서 새로운 협업 기능 구현  
    public void volumeChanged(int volume) {  
        videoPlayer.changeVolume(volume);  
    }  
}
```

10 파사드(Façade) 패턴

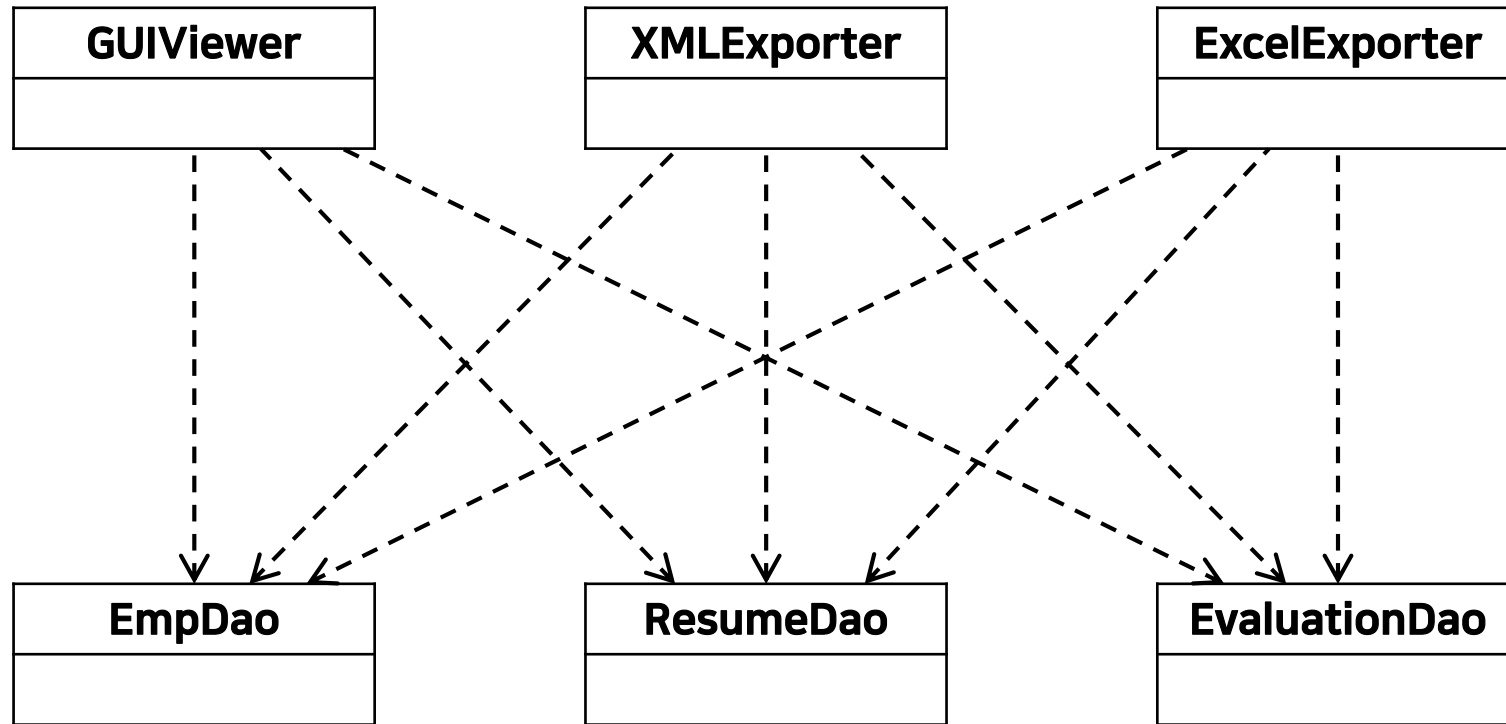
10 파사드(Façade) 패턴

- ✓ 파사드 패턴 적용 전, GUIViewer는 개별 Dao 객체에 접근



10 파사드(Façade) 패턴

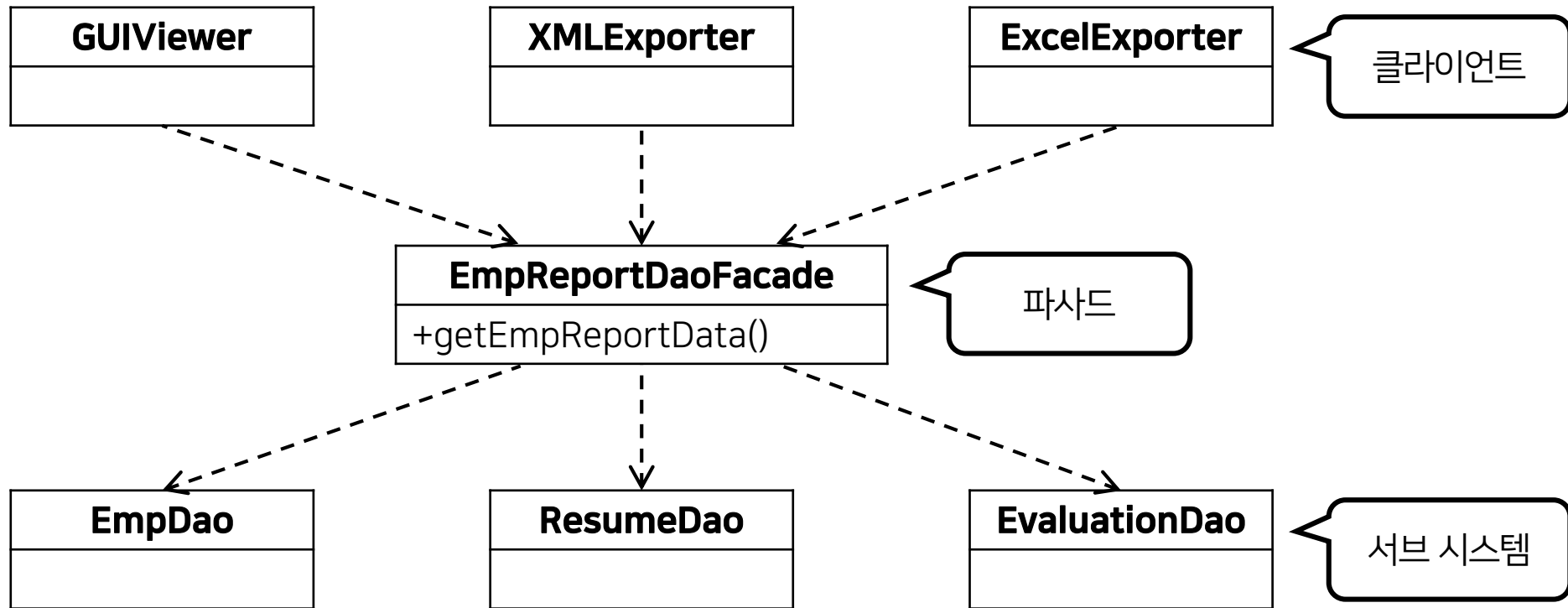
✓ 요구사항이 들어올 시 구조



GUIViewer, XMLExporter, ExcelExporter 사이에서 코드 중복이 발생

10 파사드(Façade) 패턴

✓ 파사드 패턴 적용

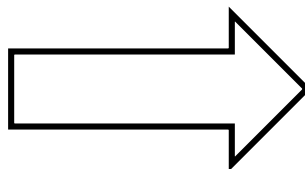


10 파사드(Façade) 패턴

✓ 클라이언트 코드가 **간결**해진다.

```
public class GuiViewer {  
  
    public void display() {  
        // ...  
        Emp emp =  
            empDao.select(id);  
        // ...  
        Resume resume =  
            resumeDao.select(id);  
        // ...  
        Evaluation eval =  
            evaluationDao.select(id);  
        // ...  
    }  
}
```

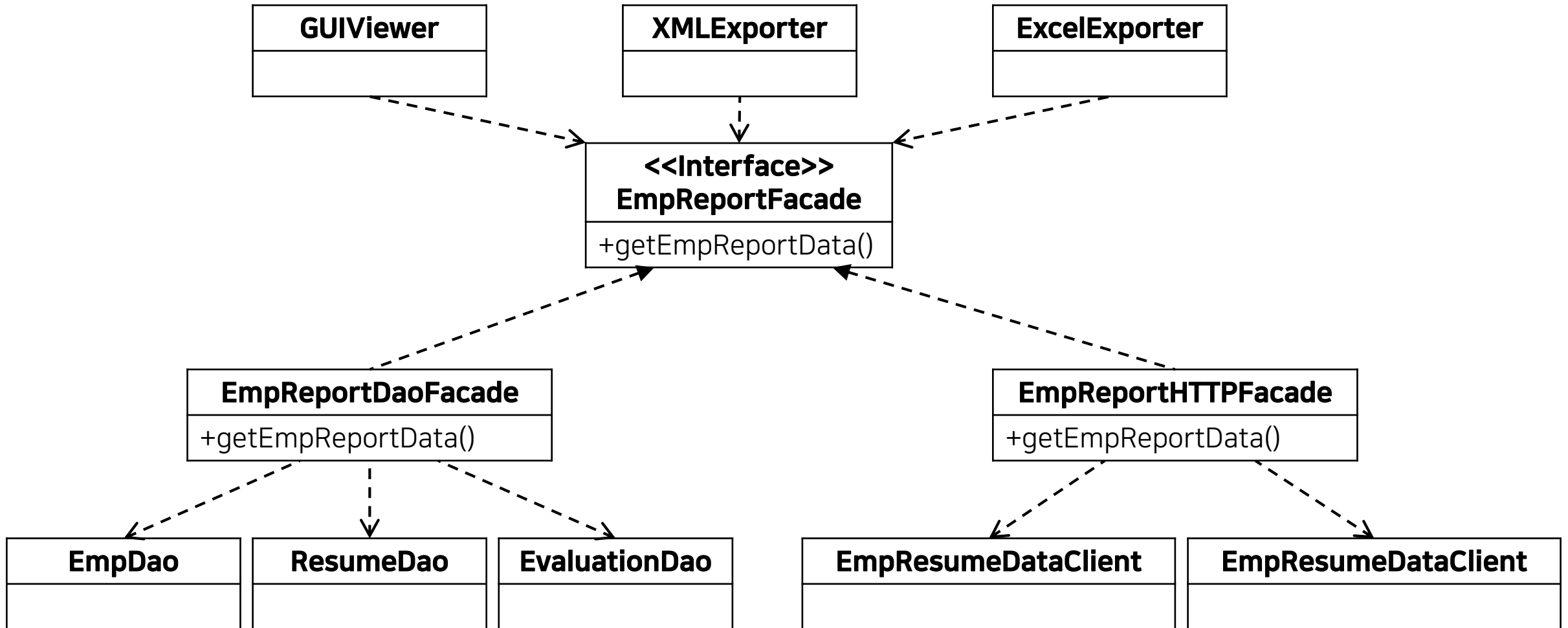
```
public class ExcelExporter {  
  
    public void export(File file) {  
        // ...  
        Emp emp =  
            empDao.select(id);  
        // ...  
        Resume resume =  
            resumeDao.select(id);  
        // ...  
        Evaluation eval =  
            evaluationDao.select(id);  
        // ...  
    }  
}
```



```
public class ExcelExporter {  
  
    public void export(File file) {  
        // ...  
        EmpReport rep =  
            empReportDaoFacade.select(id);  
        // ...  
    }  
}  
  
public class GuiViewer {  
  
    public void display() {  
        // ...  
        EmpReport rep =  
            empReportDaoFacade.select(id);  
        // ...  
    }  
}
```

10 파사드 패턴의 장점과 특징

- ✓ 파사드는 서브 시스템의 **교체**를 **쉽게** 만들어 준다.



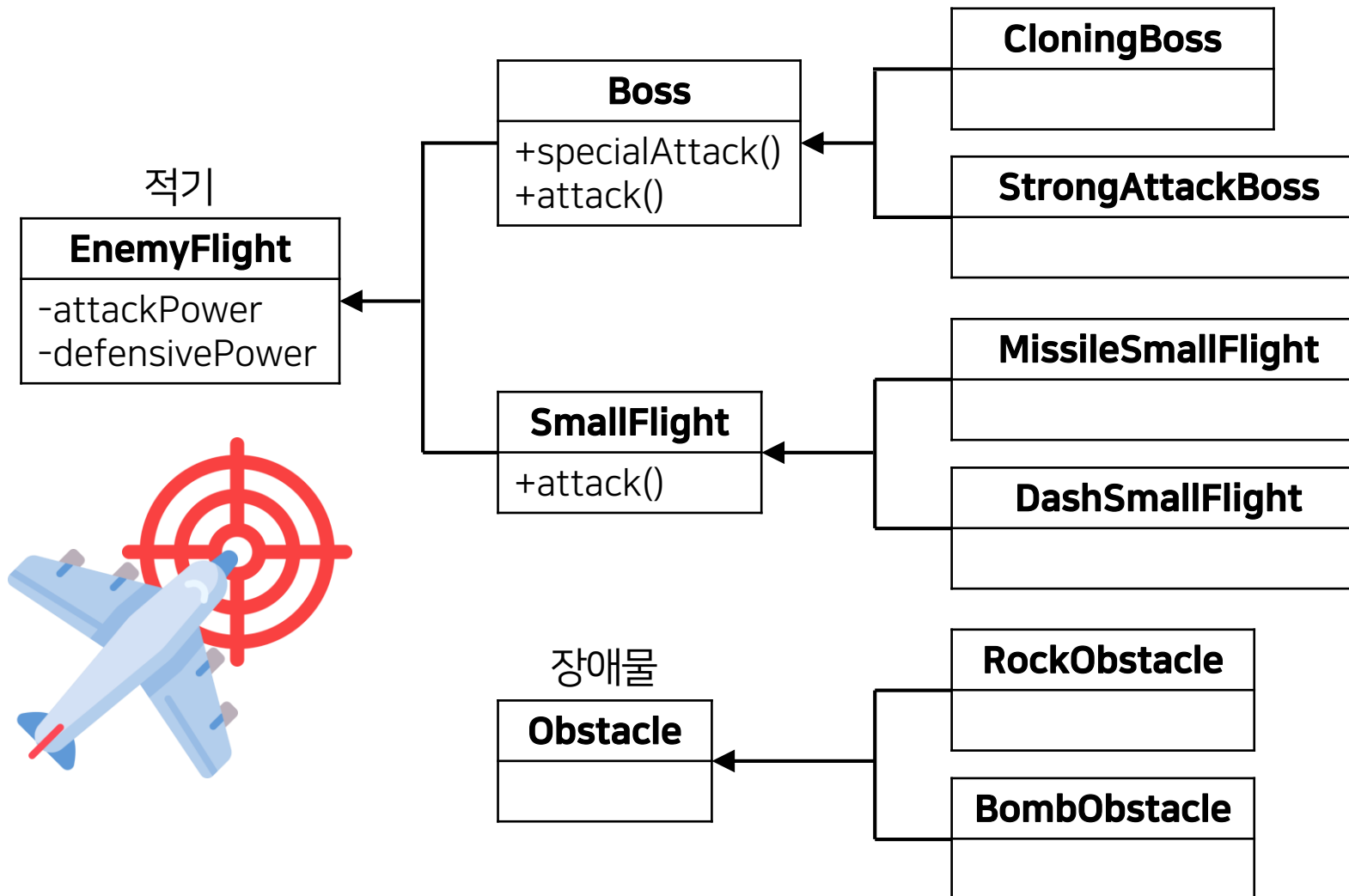
10 파사드 패턴의 장점과 특징

- ✓ 파사드 패턴은 서브 시스템에 대한 직접적인 접근을 무조건 막는건 아니다.
 - ▶ 단지 여러 클라이언트에 중복된 서브 시스템 사용을 파사드로 추상화할 뿐
- ✓ 다수의 클라이언트에 공통된 기능은 파사드를 통해 쉽게 서브 시스템을 사용
 - ▶ 보다 세밀한 제어가 필요한 경우, 서브 시스템에 직접 접근하는 방식을 선택할 수 있다.

11 추상 팩토리(Abstract Factory) 패턴

11 추상 팩토리(Abstract Factory) 패턴

- ✓ 비행기를 조종/미사일을 발사해 적을 잡는 슈팅 게임을 만든다고 가정



11 추상 팩토리(Abstract Factory) 패턴

- ✓ 게임 플레이를 진행하는 Stage 클래스의 코드

// Stage 클래스

```
private void createEnemies() {
    for (int i = 0; i <= ENEMY_COUNT; i++) {
        if (stageLevel == 1) {
            enemies[i] = new DashSmallFlight(attackPower: 1, defensivePower: 1);
        } else if (stageLevel == 2) {
            enemies[i] = new MissileSmallFlight(attackPower: 1, defensivePower: 1);
        }
    }

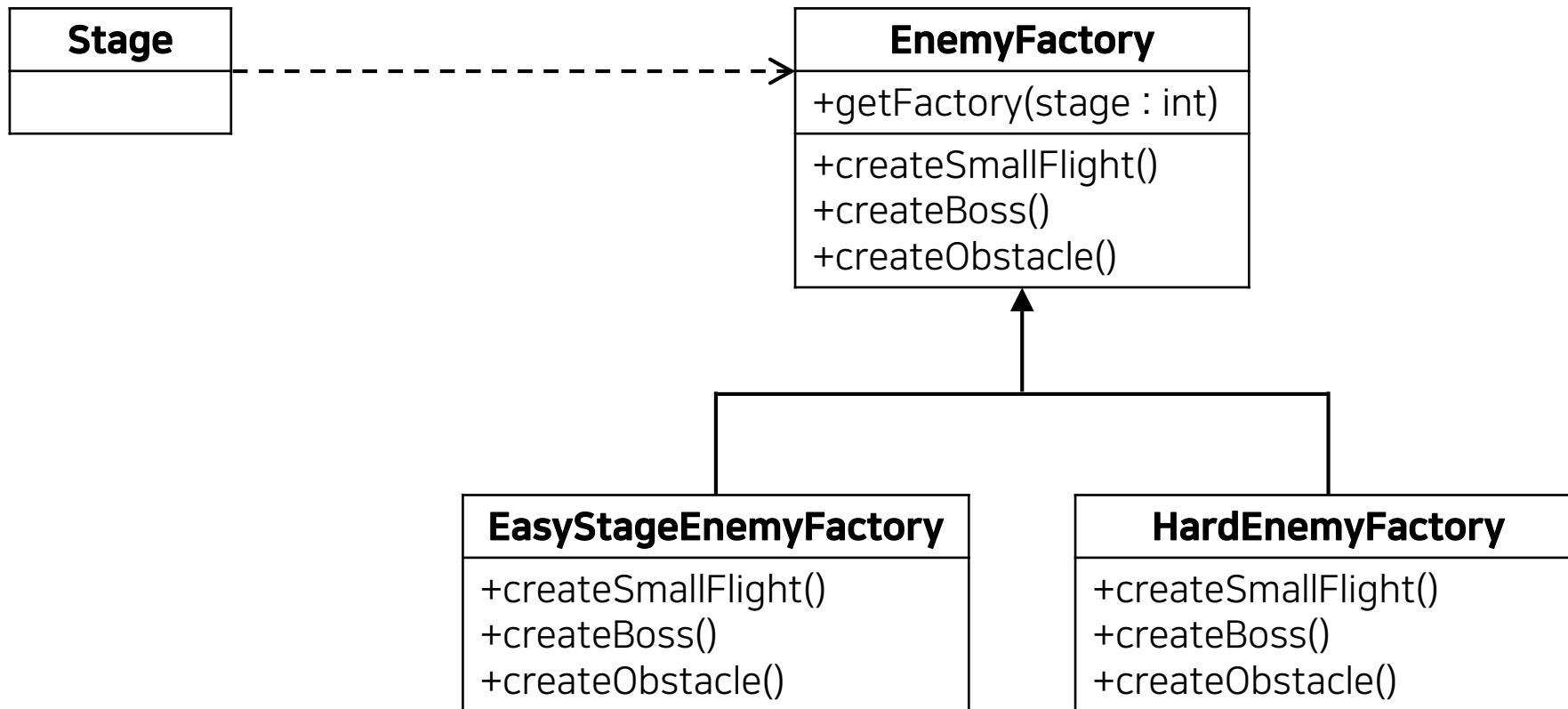
    if (stageLevel == 1) {
        boss = new StrongAttackBoss(attackPower: 1, defensivePower: 10);
    } else {
        boss = new CloningBoss(attackPower: 5, defensivePower: 20);
    }
}
```

```
private void createObstacle() {
    for (int i = 0; i < OBSTACLE_COUNT; i++) {
        if (stageLevel == 1) {
            obstacles[i] = new RockObstacle();
        } else {
            obstacles[i] = new BombObstacle();
        }
    }
}
```

적과 장애물 생성을 Stage 클래스에서 하게된다.

11 추상 팩토리(Abstract Factory) 패턴

- ✓ 객체 군을 생성하는 책임을 갖는 타입을 별도로 분리한다.



11 추상 팩토리(Abstract Factory) 패턴

✓ 추상 팩토리 패턴의 추상 팩토리 구현

```
public abstract class EnemyFactory {  
  
    public static EnemyFactory getFactory(int level) {  
        if (level == 1) {  
            return EasyStageEnemyFactory();  
        } else {  
            return HardEnemyFactory();  
        }  
    }  
}  
  
// 객체 생성을 위한 팩토리 메서드  
public abstract Boss createBoss();  
public abstract SmallFlight createSmallFlight();  
public abstract Obstacle createObstacle();  
  
}
```

11 추상 팩토리(Abstract Factory) 패턴

✓ 콘크리트 팩토리 클래스 구현

```
public class EasyStageEnemyFactory extends EnemyFactory {  
  
    @Override  
    public Boss createBoss() {  
        return new StrongAttackBoss();  
    }  
  
    @Override  
    public SmallFlight createSmallFlight() {  
        return new DashSmallFlight();  
    }  
  
    @Override  
    public Obstacle createObstacle() {  
        return new RockObstacle();  
    }  
}
```

```
public class HardEnemyFactory extends EnemyFactory {  
  
    @Override  
    public Boss createBoss() {  
        return new CloningBoss();  
    }  
  
    @Override  
    public SmallFlight createSmallFlight() {  
        return new MissileSmallFlight();  
    }  
  
    @Override  
    public Obstacle createObstacle() {  
        return new BombObstacle();  
    }  
}
```

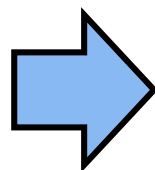
11 추상 팩토리(Abstract Factory) 패턴

✓ Stage 클래스는 추상 팩토리 타입을 이용해 객체를 생성한다.

```
// 기존 Stage 클래스
private void createEnemies() {
    for (int i = 0; i <= ENEMY_COUNT; i++) {
        if (stageLevel == 1) {
            enemies[i] = new DashSmallFlight(attackPower: 1, defensivePower: 1);
        } else if (stageLevel == 2) {
            enemies[i] = new MissileSmallFlight(attackPower: 1, defensivePower: 1);
        }
    }

    if (stageLevel == 1) {
        boss = new StrongAttackBoss(attackPower: 1, defensivePower: 10);
    } else {
        boss = new CloningBoss(attackPower: 5, defensivePower: 20);
    }
}

private void createObstacle() {
    for (int i = 0; i < OBSTACLE_COUNT; i++) {
        if (stageLevel == 1) {
            obstacles[i] = new RockObstacle();
        } else {
            obstacles[i] = new BombObstacle();
        }
    }
}
```



```
// 팩토리를 사용하도록 바뀐 Stage 클래스
private EnemyFactory enemyFactory;

public Stage(int level) {
    enemyFactory = EnemyFactory.getFactory(level);
}

private void createEnemies() {
    for (int i = 0; i <= ENEMY_COUNT; i++) {
        enemies[i] = enemyFactory.createSmallFlight();
    }

    boss = enemyFactory.createBoss();
}

private void createObstacle() {
    for (int i = 0; i < OBSTACLE_COUNT; i++) {
        obstacles[i] = enemyFactory.createObstacle();
    }
}
```

11 추상 팩토리(Abstract Factory) 패턴

```
private EnemyFactory enemyFactory;  
public Stage(int level) {  
    // EnemyFactory 객체를 구함  
    enemyFactory = EnemyFactory.getFactory(level);  
}
```

```
public abstract class EnemyFactory {  
  
    public static EnemyFactory getFactory(int level) {  
        if (level == 1) {  
            // 적 생성 규칙 변경 시, 새로운 팩토리 클래스를 만들면 된다.  
            return new SomethingNewEnemyFactory();  
        } else {  
            return new HardEnemyFactory();  
        }  
    }  
}
```

11 추상 팩토리(Abstract Factory) 패턴

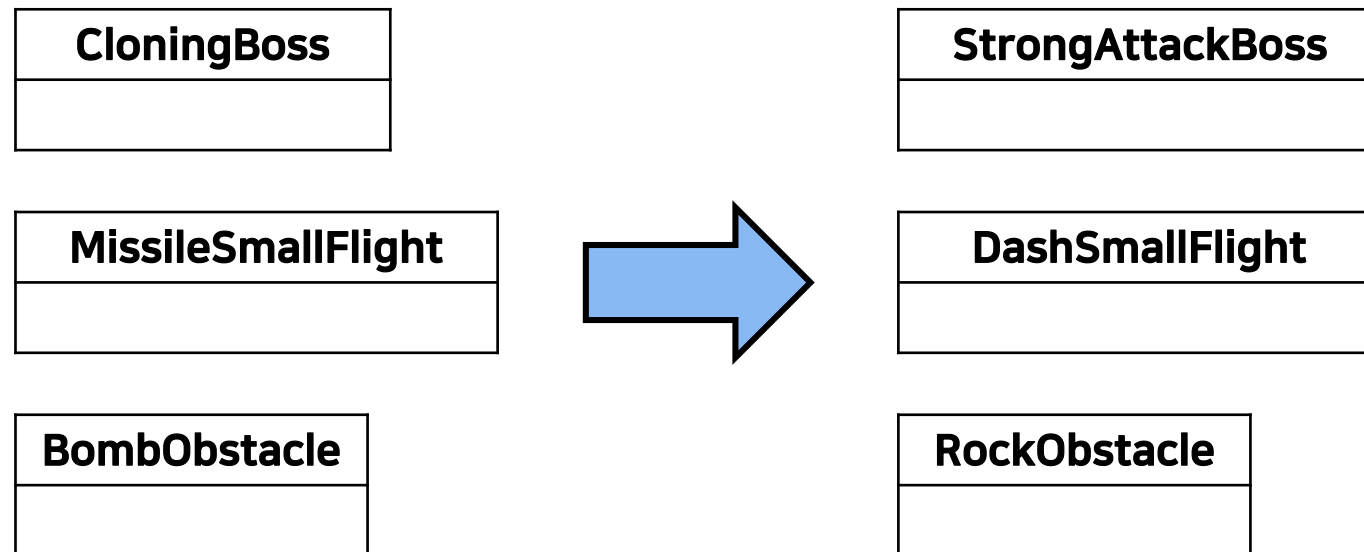
- ✓ 팩토리 객체는 DI(의존성 주입)을 사용해도 된다.

```
public class Stage {  
  
    private EnemyFactory enemyFactory;  
  
    // DI를 적용하면 팩토리를 구하는 기능을 EnemyFactory에 구현할 필요가 없다.  
    public Stage(int level, EnemyFactory enemyFactory) {  
        this.level = level;  
        this.enemyFactory = enemyFactory;  
    }  
    // ...  
}
```

11 추상 팩토리(Abstract Factory) 패턴

✓ 추상 팩토리 패턴의 장점

- 클라이언트에 영향을 주지 않으면서 사용할 제품(객체) 군을 교체할 수 있다.
 - 예제에서 Stage 클래스가 쓰는 제품을 변경해도 Stage 클래스는 영향을 받지 않는다.
 - Stage 클래스가 사용하는 콘크리트 팩토리 객체를 변경해도 Stage 클래스는 영향을 받지 않는다.



11 추상 팩토리(Abstract Factory) 패턴

✓ 프로토타입 방식의 팩토리

```
public class Factory {  
  
    private ProductA productAProto;  
    private ProductB productBProto;  
  
    public Factory(ProductA productAProto, ProductB productBProto) {  
        this.productAProto = productAProto;  
        this.productBProto = productBProto;  
    }  
  
    public ProductA createA() {  
        return (ProductA) productAProto.clone();  
    }  
  
    public ProductB createB() {  
        return (ProductB) productBProto.clone();  
    }  
}
```


11 추상 팩토리(Abstract Factory) 패턴

✓ 프로토타입 방식의 팩토리

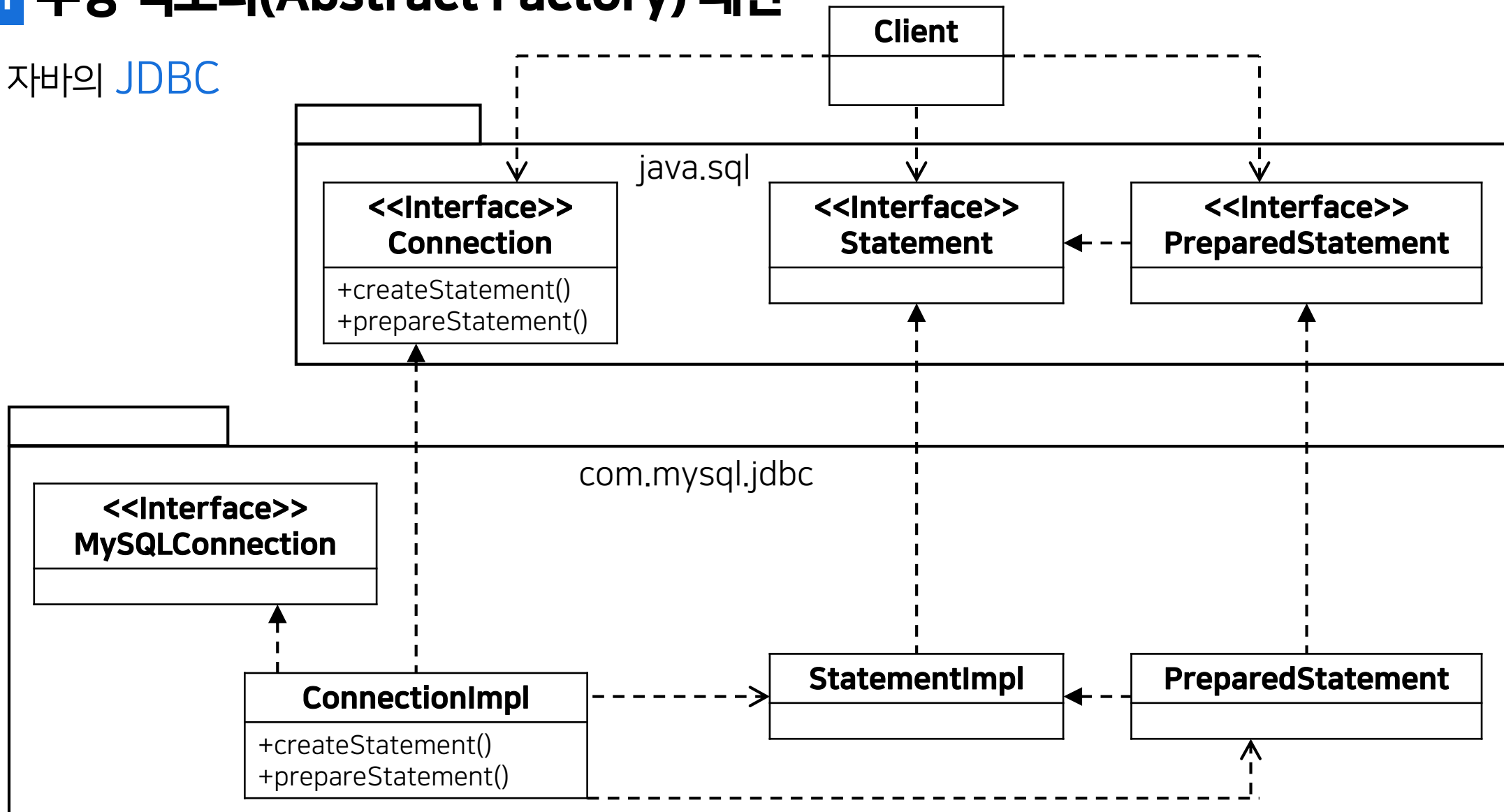
```
// 객체 군 1을 위한 팩토리 객체  
Factory family1Factory = new Factory(new HighProductA(), new HighProductB());  
ProductA a = family1Factory.createA(); // HighProductA 객체 복제본 생성  
  
// 객체 군 2를 위한 팩토리 객체  
Factory family2Factory = new Factory(new LowProductA(), new LowProductB());  
ProductB b = family2Factory.createB(); // LowProductB 객체 복제본 생성
```

구현이 쉬워진다.

하지만 제품 객체의 생성 규칙이 복잡할 경우, 적용할 수 없다.

11 추상 팩토리(Abstract Factory) 패턴

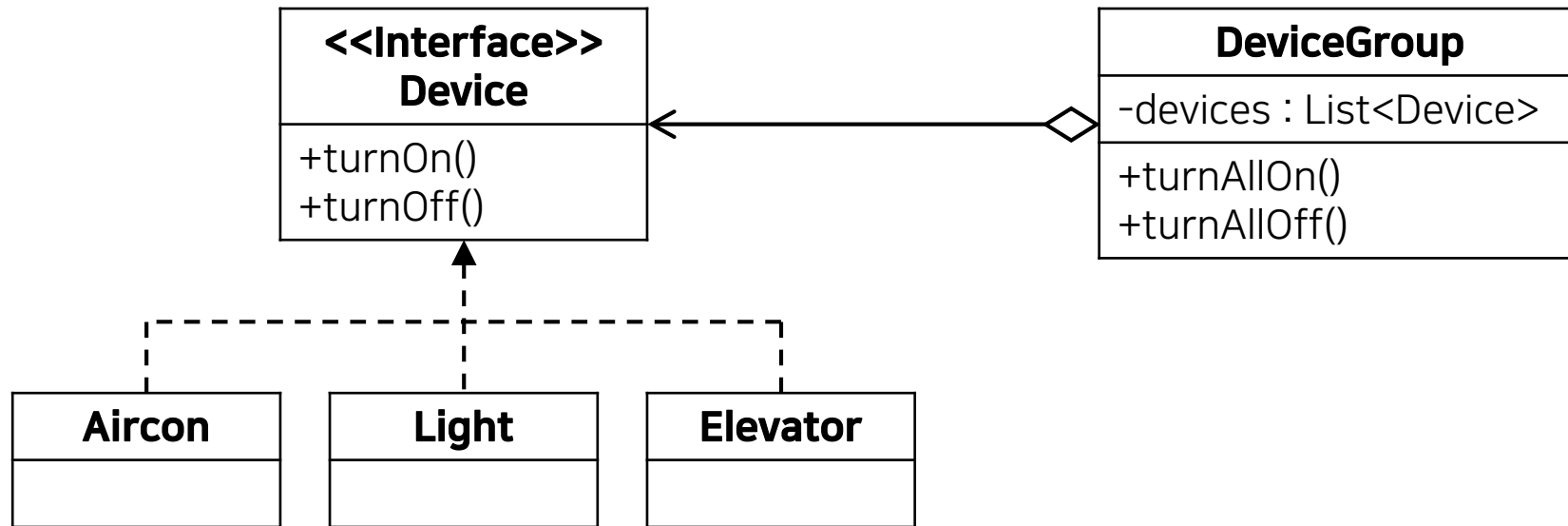
자바의 JDBC



12 컴포지트(Composite) 패턴

12 컴포지트(Composite) 패턴

- ✓ 빌딩의 장비들의 전원을 관리하는 제어 프로그램

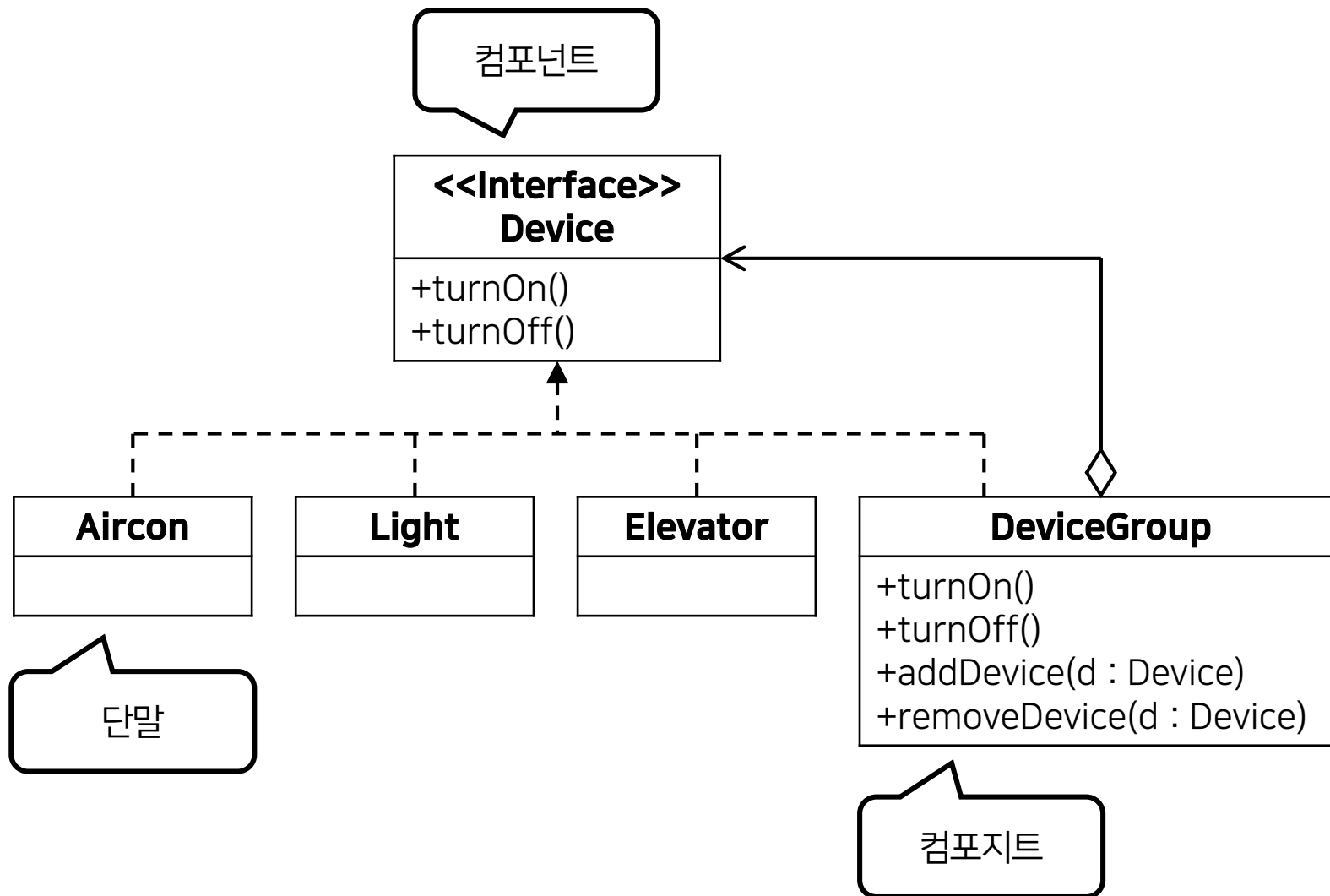


12 컴포지트(Composite) 패턴

- ✓ 장비들의 전원을 제어하는 코드

```
public class PowerController {  
  
    public void turnOn(Long deviceId) {  
        Device device = findDeviceById(deviceId);  
        device.turnOn();  
    }  
  
    // turnGroupOn()과 turnOn()은 개별/그룹 차이를 빼면 동일한 기능  
    public void turnGroupOn(Long groupId) {  
        DeviceGroup group = findGroupById(groupId);  
        group.turnAllOn();  
    }  
}
```

12 컴포지트(Composite) 패턴



12 컴포지트(Composite) 패턴

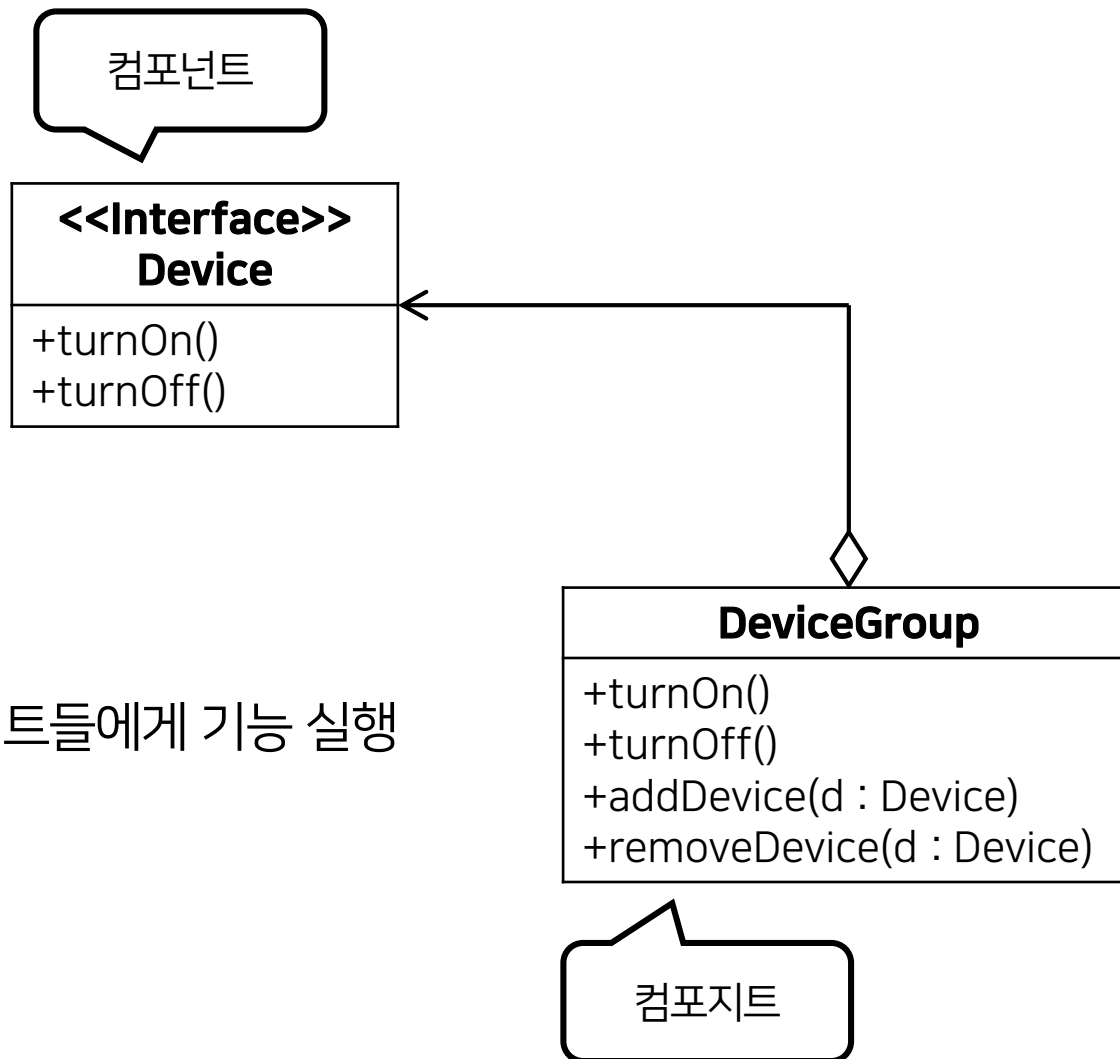
✓ 컴포지트가 가지는 책임

- 컴포넌트 그룹을 관리한다.

- 컴포지트에 기능 실행을 요청하면?

▶ 컴포지트는 포함하고 있는 컴포넌트들에게 기능 실행

요청을 위임



12 컴포지트(Composite) 패턴

```
public class DeviceGroup implements Device {  
  
    private List<Device> devices = new ArrayList<>();  
  
    public void addDevice(Device d) {  
        devices.add(d);  
    }  
  
    public void removeDevice(Device d) {  
        devices.remove(d);  
    }  
  
    public void turnOn() {  
        for (Device device : devices) {  
            device.turnOn(); // 관리하는 Device 객체들에게 실행 위임  
        }  
    }  
  
    public void turnOff() {  
        for (Device device : devices) {  
            device.turnOff(); // 관리하는 Device 객체들에게 실행 위임  
        }  
    }  
}
```


12 컴포지트(Composite) 패턴

- ✓ DeviceGroup 객체에 Device를 등록

```
Device device1 = ...;  
Device device2 = ...;  
DeviceGroup group = new DeviceGroup();  
group.addDevice(device1);  
group.addDevice(device2);  
  
group.turnOn(); // device1과 device2의 turnOn() 실행
```

12 컴포지트(Composite) 패턴

- ✓ 전체냐 부분이나에 **상관 없이** 기능을 실행할 수 있다.

```
public class PowerController {  
  
    public void turnOn(Long deviceId) {  
        // device의 실제 타입이 DeviceGroup인지 여부에 상관없이 동작  
        Device device = findDeviceById(deviceId);  
        device.turnOn();  
    }  
  
    /* 필요 없어진 메서드  
    public void turnGroupOn(Long groupId) {  
        DeviceGroup group = findGroupById(groupId);  
        group.turnAllOn();  
    }  
    */  
}
```

12 컴포지트(Composite) 패턴

- ✓ 컴포지트에 다른 컴포지트를 등록할 수 있다.

```
// 각 층의 Light 객체를 모은 그룹 생성
DeviceGroup firstFloorLightGroup = ...; // 1층의 모든 Light를 포함
DeviceGroup secondFloorLightGroup = ...; // 2층의 모든 Light를 포함
// ...

// 각 층의 DeviceGroup 객체를 하나의 DeviceGroup에 다시 등록
DeviceGroup allLightGroup = new DeviceGroup();
allLightGroup.add(firstFloorLightGroup);
allLightGroup.add(secondFloorLightGroup);
// ...

allLightGroup.turnOff(); // 모든 층의 Light 객체의 turnOff() 실행
```

12 컴포지트 패턴 구현의 고려 사항

- ✓ 컴포넌트를 관리하는 인터페이스를 어디서 구현할지에 대한 여부

```
DeviceGroup group = new DeviceGroup();  
group.addDevice(device1);  
group.addDevice(device2);
```

```
public void addDeviceTo(Device device, Integer toDeviceId) {  
    Device composite = findDevice(toDeviceId);  
    composite.addDevice(device);  
}
```

12 컴포지트 패턴 구현의 고려 사항

- ✓ 컴포넌트를 관리하는 인터페이스를 어디서 구현할지에 대한 여부

```
public abstract class Device {  
  
    public void addDevice(Device d) {  
        | throw new CanNotAddException("추가할 수 없음");  
    }  
  
    public void removeDevice(Device d) {  
        | // 아무 것도 하지 않음  
    }  
  
    public abstract void turnOn();  
    public abstract void turnOff();  
  
}
```

```
public class DeviceGroup extends Device {  
  
    @Override  
    public void addDevice(Device d) {  
        | devices.add(d);  
    }  
  
    @Override  
    public void removeDevice(Device d) {  
        | devices.remove(d);  
    }  
  
    // ...  
}
```

12 컴포지트 패턴 구현의 고려 사항

- ✓ 컴포넌트를 관리하는 인터페이스를 어디서 구현할지에 대한 여부

```
public void addDeviceTo(Device device, Integer toDeviceId) {  
    Device composite = findDevice(toDeviceId);  
  
    try {  
        composite.addDevice(device);  
    } catch (CanNotAddException ex) {  
        // ... 추가할 수 없는 경우의 처리  
    }  
}
```

컴포지트가 아닌 객체라면 예외발생

12 컴포지트 패턴 구현의 고려 사항

- ✓ 컴포넌트를 관리하는 인터페이스를 어디서 구현할지에 대한 여부

```
public abstract class Device {  
  
    // ...  
  
    public boolean canContain(Device device) {  
        return false;  
    }  
}
```

```
// GroupDevice에서 알맞게 재정의  
public class DeviceGroup extends Device {  
  
    // ...  
  
    @Override  
    public boolean canContain(Device device) {  
        return true;  
    }  
}
```

12 컴포지트 패턴 구현의 고려 사항

- ✓ 컴포넌트를 관리하는 인터페이스를 어디서 구현할지에 대한 여부

```
public void addDeviceTo(Device device, Integer toDeviceId) {  
    Device composite = findDevice(toDeviceId);  
  
    try {  
        composite.addDevice(device);  
    } catch (CanNotAddException ex) {  
        // ... 추가할 수 없는 경우의 처리  
    }  
}
```



```
public void addDeviceTo(Device device, Integer toDeviceId) {  
    Device composite = findDevice(toDeviceId);  
  
    if (composite.canContain(device)) {  
        composite.addDevice(device);  
        return;  
    }  
    // ... 추가할 수 없는 경우의 처리  
}
```


12 컴포지트 패턴 구현의 고려 사항

- ✓ 컴포넌트를 관리하는 인터페이스를 어디서 구현할지에 대한 여부

```
public class OnlyAirconContainingGroup extends DeviceGroup {  
  
    @Override  
    public boolean canContain(Device device) {  
        return (device instanceof Aircon);  
    }  
  
}
```

13 널(Null) 객체 패턴

13 널(Null) 객체 패턴

- ✓ 고객 상태에 따른 할인을 가정

```
public Bill createBill(Customer customer) {  
    Bill bill = new Bill();  
    // ... 사용 내역 추가  
    bill.addItem(new Item("기본사용요금", price));  
    bill.addItem(new Item("할부금", somePrice));  
  
    // 특별 할인 내역 추가  
    SpecialDiscount specialDiscount = specialDiscountFactory.create(customer);  
    if (specialDiscount != null) { // 특별 할인 대상인 경우만 처리  
        specialDiscount.addDetailTo(bill);  
    }  
}
```

13 널(Null) 객체 패턴

```
public void someMethod(MyObject obj) {  
    if (obj != null) {  
        obj.someOperation();  
    }  
    // ...  
    callAnyMethod(obj, other);  
}
```

```
private void callAnyMethod(MyObject obj, Other other) {  
    if (other.someOp()) {  
        if (obj != null) {  
            obj.process();  
        }  
    } else {  
        if (obj != null) {  
            obj.processOther();  
        }  
    }  
}
```

13 널(Null) 객체 패턴

- ✓ 널(Null) 객체 패턴은 null을 리턴하지 않고 null을 대신할 객체를 리턴한다.
→ null 검사 코드를 없앨 수 있다.
- ✓ 널(Null) 객체 패턴 구현방법
 - null 대신 사용될 클래스를 구현한다.
(상위 타입을 상속받고, 아무 기능도 수행하지 않는다.)
 - null을 리턴하는 대신, null을 대체할 클래스의 객체를 리턴한다.

13 널(Null) 객체 패턴

- ✓ null일 때, 대신 사용될 클래스의 객체를 리턴

```
public class NullSpecialDiscount extends SpecialDiscount {
```

```
    @Override
```

```
    public void addDetailTo(Bill bill) {
```

```
        // 아무 것도 하지 않음
```

```
    }
```

```
}
```

```
public class SpecialDiscountFactory {
```

```
    public SpecialDiscount create(Customer customer) {
```

```
        if (checkNewCustomer(customer)) {
```

```
            return new NewCustomerSpecialDiscount();
```

```
        }
```

```
        // ... 다른 코드 실행
```

```
        // 특별 할인이 없을 때, null 대신 NullSpecialDiscount 객체 리턴
```


```
        return new NullSpecialDiscount();
```

```
    }
```

13 널(Null) 객체 패턴

✓ 코드가 간결해진다.

```
public Bill createBill(Customer customer) {  
    Bill bill = new Bill();  
    // ... 사용 내역 추가  
  
    // 특별 할인 내역 추가  
    // 특별 할인 대상이 아닐 경우 NullSpecialDiscount 객체 리턴  
    SpecialDiscount specialDiscount = specialDiscountFactory.create(customer);  
    specialDiscount.addDetailTo(bill); // null 검사 불필요  
    // ...  
    return bill;  
}
```



```
SpecialDiscount specialDiscount = specialDiscountFactory.create(customer);  
if (specialDiscount != null) { // 특별 할인 대상인 경우만 처리  
    specialDiscount.addDetailTo(bill);  
}
```