PRESENTATION

18주차 주제 디자인 패턴

디자인 패턴 ~중재자 패턴~

by mun

디자인패턴

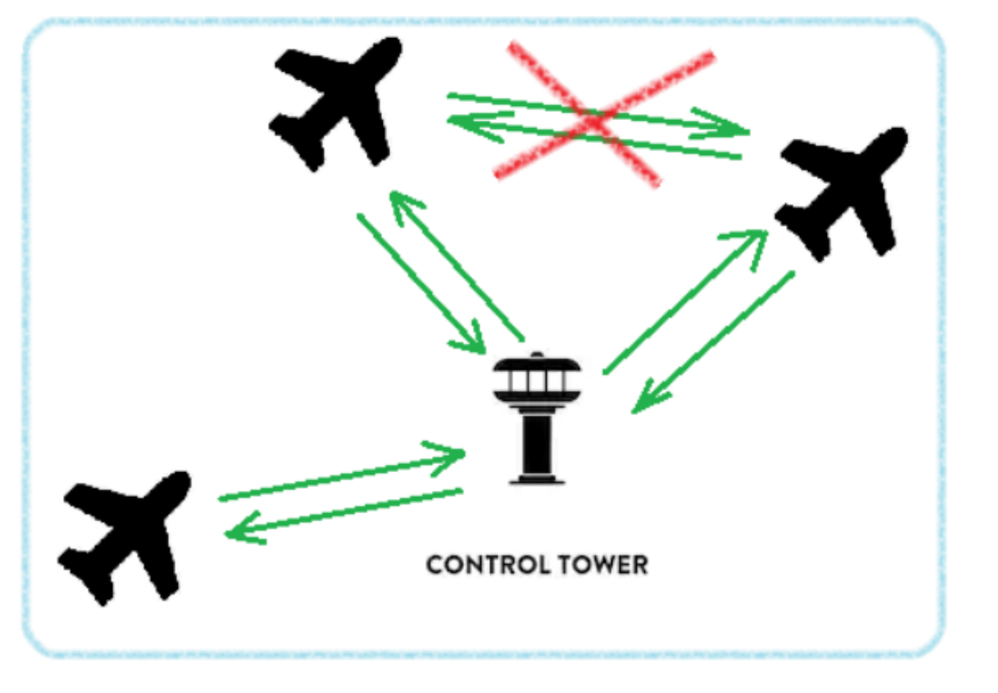
설계 문제에 대한 해답을 문서화하기 위해 고안된 형식 방법이다.

프로그램 개발에서 자주 나타나는 과제를 해결하기 위한 방법 중 하나로,
과거의 소프트웨어 개발 과정에서 발견된 설계의 노하우를 축적하여 이름을 붙여,
이후에 재이용하기 좋은 형태로 특정의 규약을 묶어서 정리한 것이다. (위키백과)

프로그램을 설계할 때 발생했던 문제점들을 객체 간의 상호 관계 등을 이용하여 해결할 수 있도록 하나의 '규약' 형태로 만들어 놓은 것을 의미한다.

중재자 패턴 Mediator Pattern

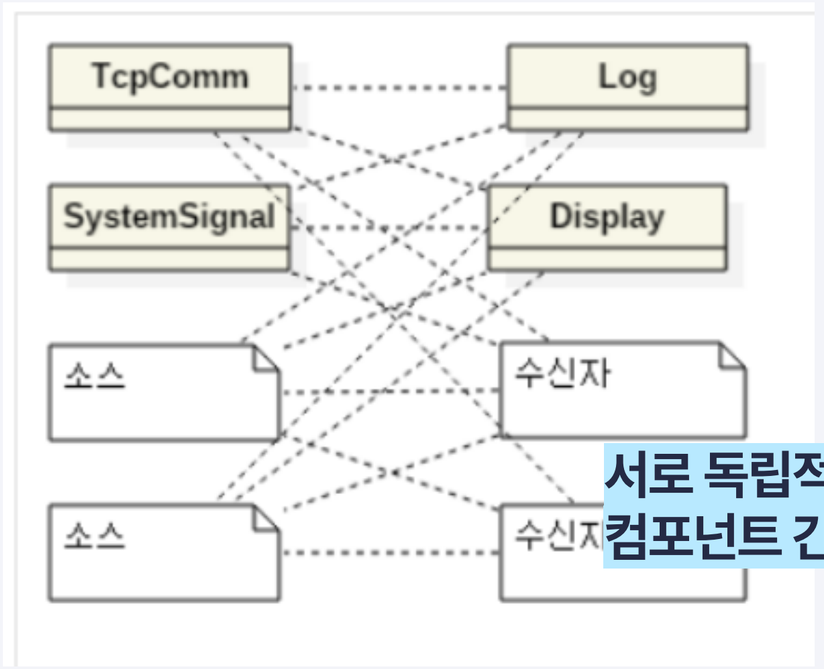
모든 클래스간의 복잡한 로직(상호작용)을 캡슐화하여
하나의 클래스에 위임하여 처리하는 패턴



객체 간의 직접 통신을 제한하고
중재자 객체를 통해서만 협력하도록 한다

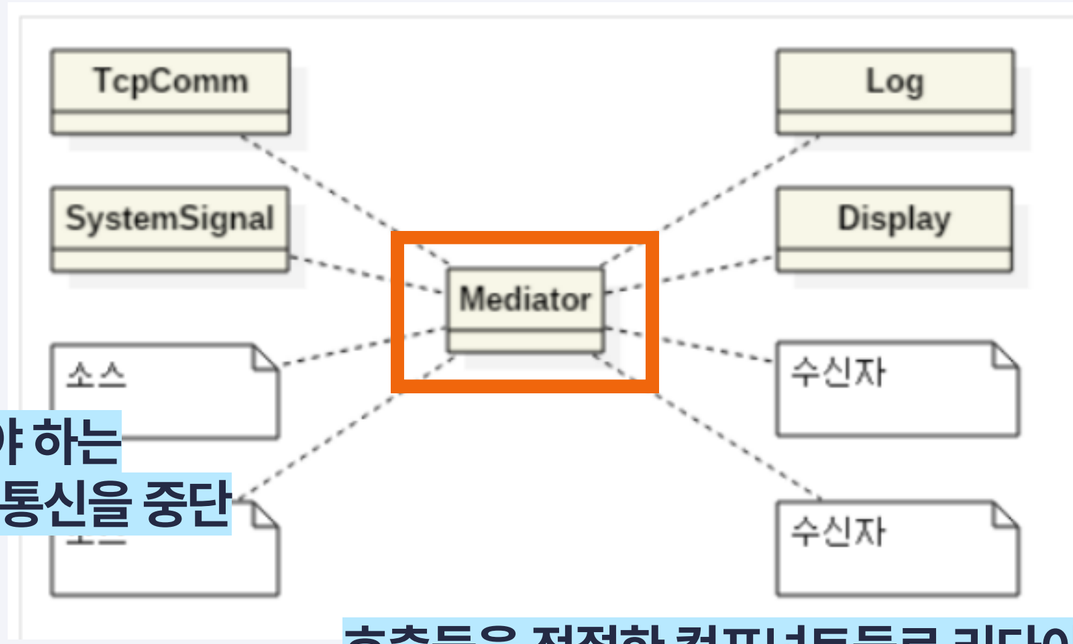
상호작용을 하기 위한
복잡한 의존 관계를 단순화

중재자 패턴 Mediator Pattern



서로 독립적으로 작동해야 하는
컴포넌트 간의 모든 직접 통신을 중단

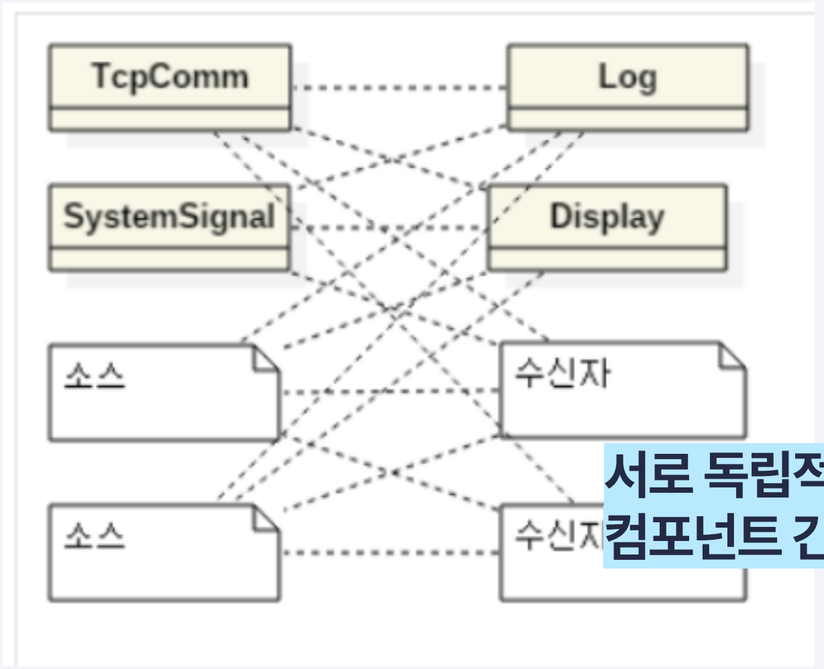
컴포넌트들은 동료 컴포넌트들과
결합되는 대신 단일 중재자 클래스에만 의존



호출들을 적절한 컴포넌트들로 리다이렉션하는
특수 중재자 객체를 호출하여 간접적으로 협력

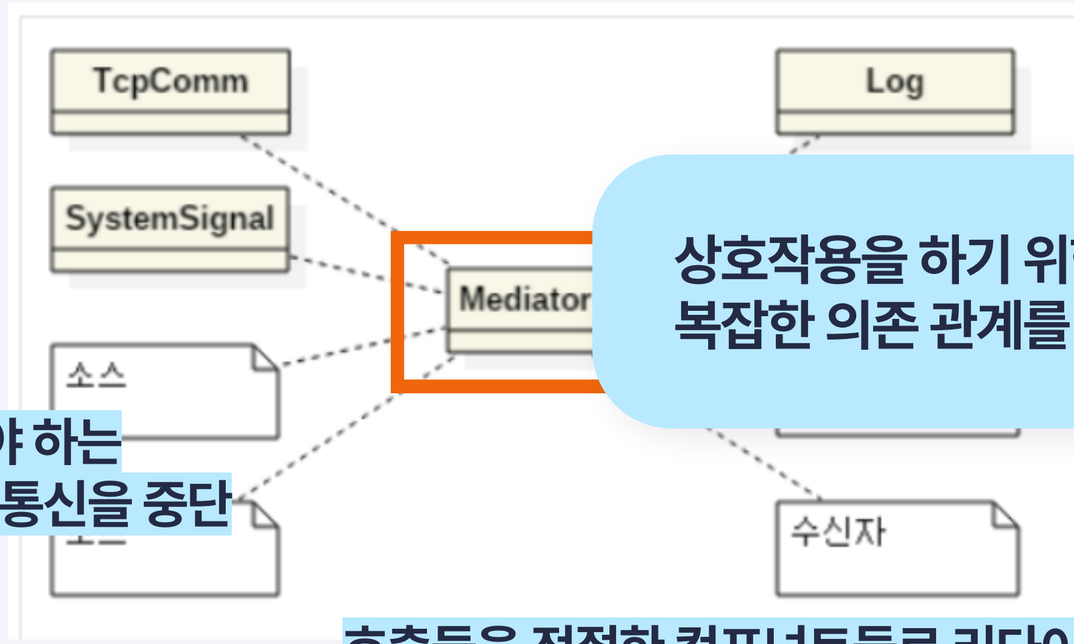
https://effectiveprogramming.tistory.com/entry/Mediator-%ED%8C%A8%ED%84%B4

중재자 패턴 Mediator Pattern



서로 독립적으로 작동해야 하는
컴포넌트 간의 모든 직접 통신을 중단

컴포넌트들은 동료 컴포넌트들과
결합되는 대신 단일 중재자 클래스에만 의존

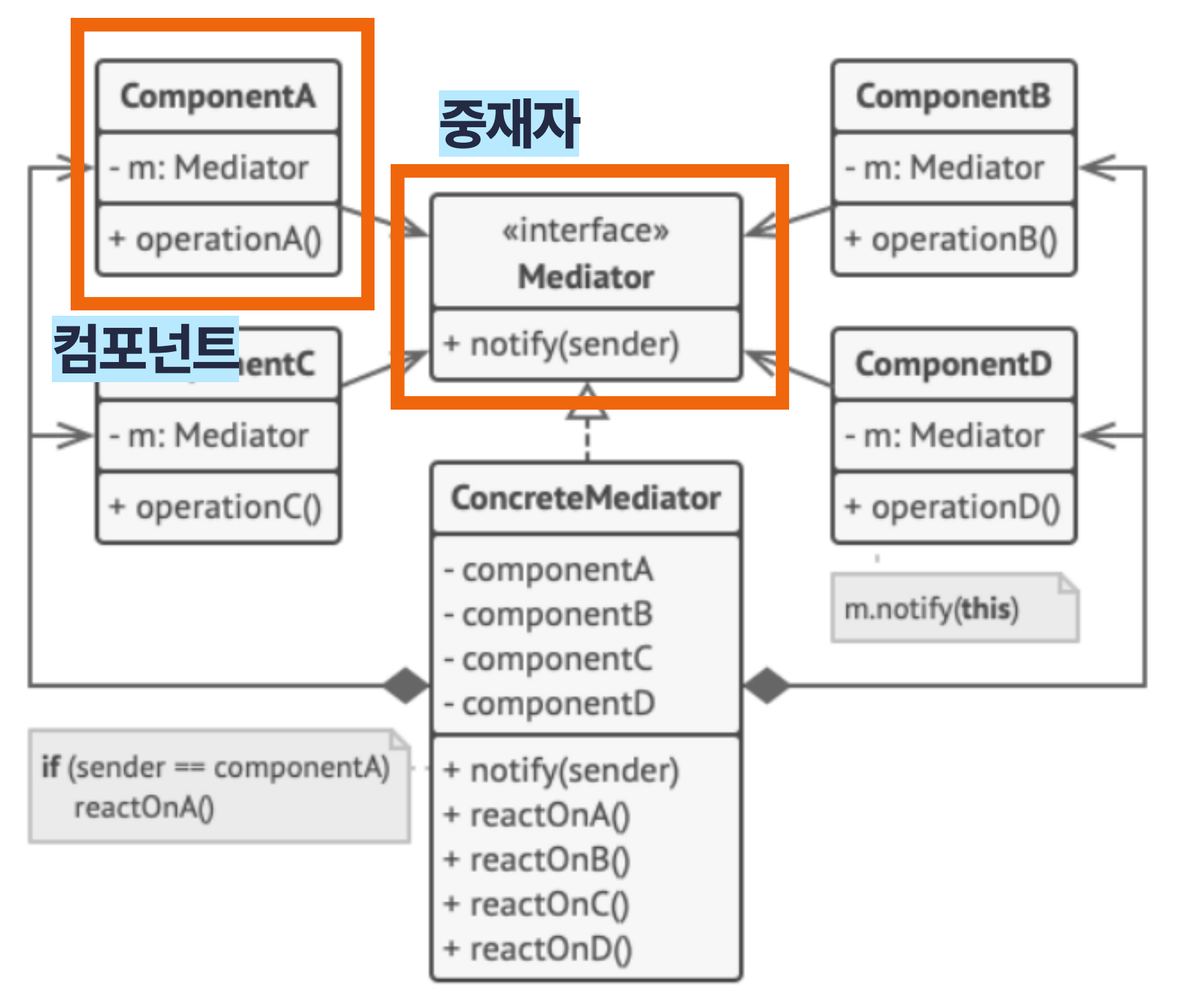


상호작용을 하기 위한
복잡한 의존 관계를 단순화

호출들을 적절한 컴포넌트들로 리다이렉션하는
특수 중재자 객체를 호출하여 간접적으로 협력

https://effectiveprogramming.tistory.com/entry/Mediator-%ED%8C%A8%ED%84%B4

구조



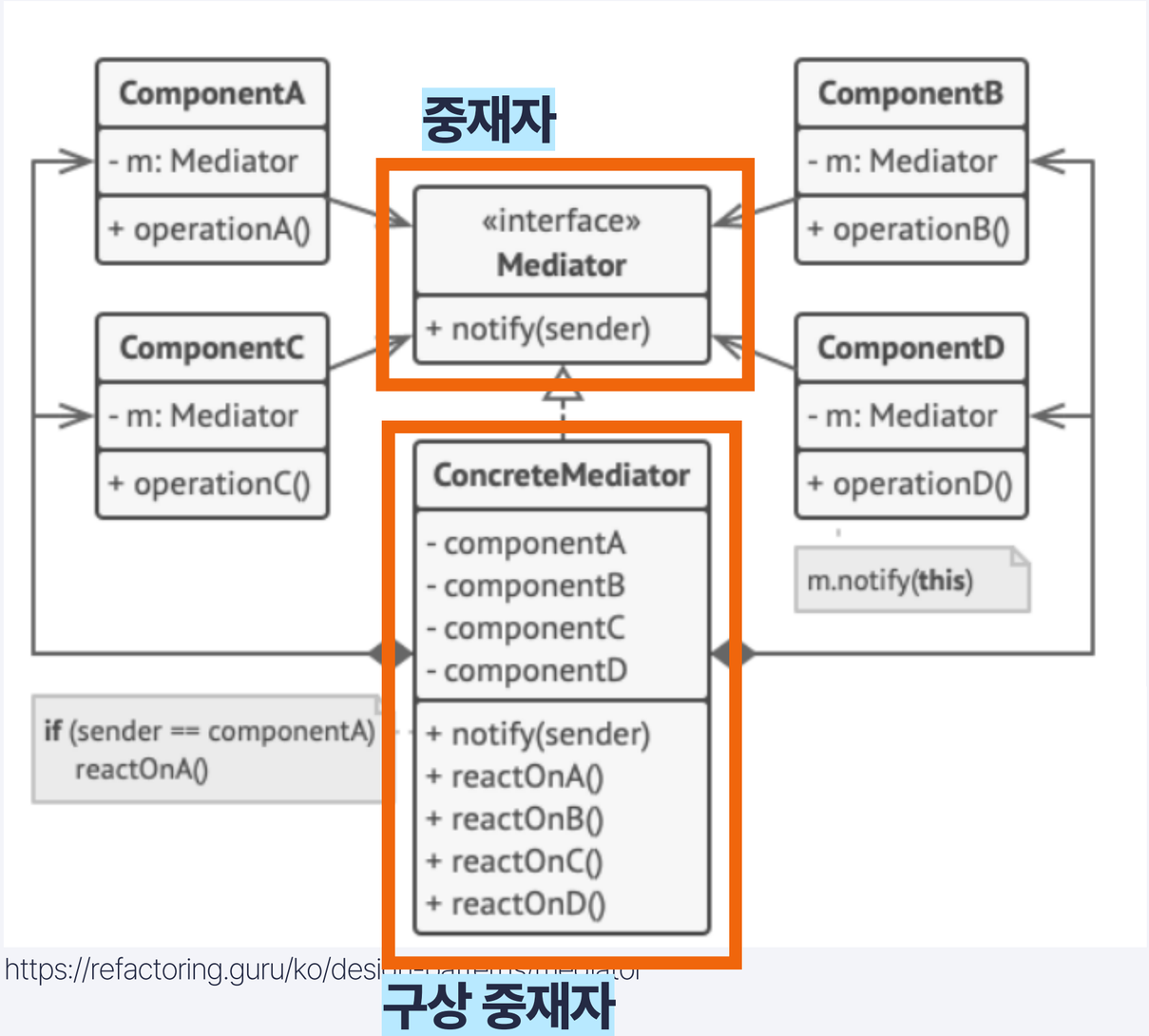
https://refactoring.guru/ko/design-patterns/mediator

컴포넌트
어떤 비즈니스 로직을 포함한 다양한 클래스들

각 컴포넌트에는 중재자에 대한 참조가 있다
중재자는 인터페이스의 유형으로 선언되어
컴포넌트를 다른 중재자에 연결하여 재사용할 수 있다

컴포넌트에 중요한 일이 발생하면
컴포넌트는 이를 중재자에게만 알려야 한다
(다른 컴포넌트들을 인식하면 X)

구조



https://refactoring.guru/ko/desi

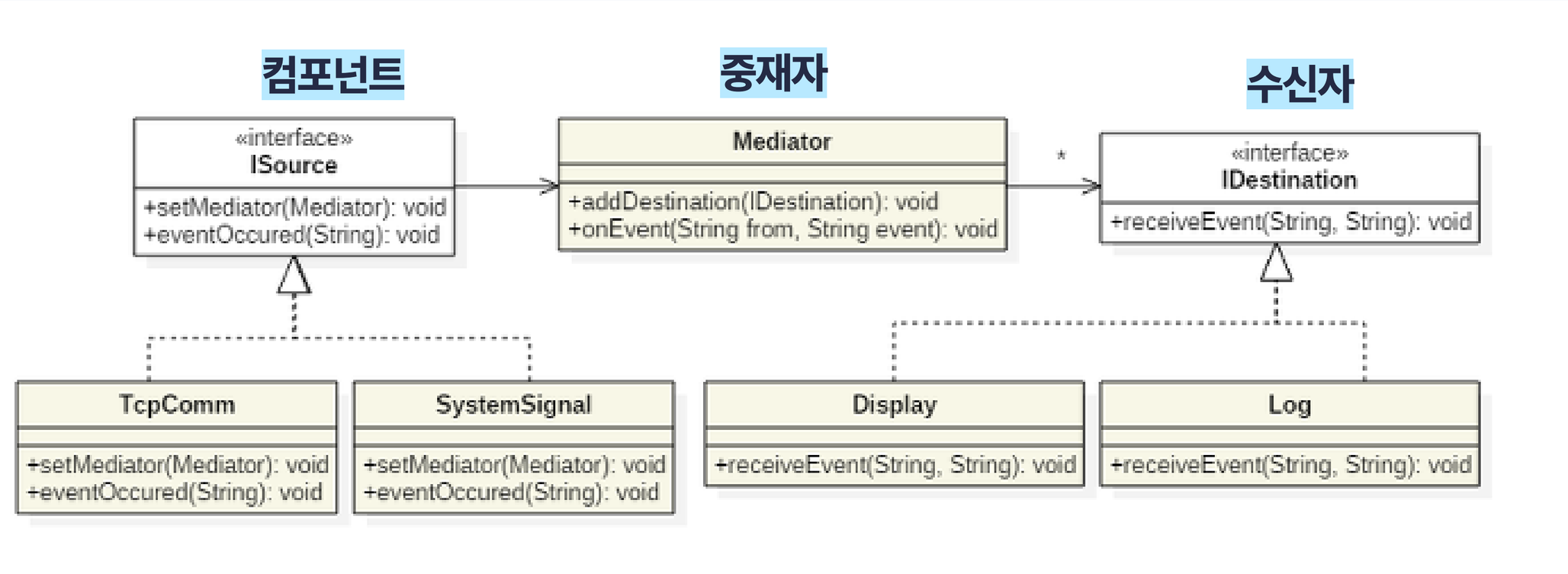
중재자 인터페이스

일반적으로 단일 알림 메서드만을 포함하는
컴포넌트들과의 통신 메서드들을 선언한다

구상 중재자

다양한 컴포넌트 간의 관계를 캡슐화한다
자신이 관리하는 모든 컴포넌트에 대한 참조를
유지하고 때로는 수명 주기를 관리하기도 한다

예제


컴포넌트
중재자
수신자
«interface»
ISource
+setMediator(Mediator): void
+eventOccured(String): void
Mediator
+addDestination(IDestination): void
+onEvent(String from, String event): void
*
«interface»
IDestination
+receiveEvent(String, String): void
TcpComm
+setMediator(Mediator): void
+eventOccured(String): void
SystemSignal
+setMediator(Mediator): void
+eventOccured(String): void
Display
+receiveEvent(String, String): void
Log
+receiveEvent(String, String): void

예제

```java
public interface ISource {
    2 usages  2 implementations
    void setMediator(Mediator mediator);
    2 usages  2 implementations
    void eventOccured(String event);
}
```

setMediator()
외부로부터 Mediator 객체를 주입

eventOccured()
Mediator 객체의 onEvent() 메소드를 호출해 자신에게 발생한 이벤트를 전달

구현

```java
public class TcpComm implements ISource {
    2 usages
    Mediator mediator;

    2 usages
    @Override //중재자 설정
    public void setMediator(Mediator mediator){
        this.mediator = mediator;
    }
    2 usages
    @Override //이벤트 전달
    public void eventOccured(String event){
        mediator.onEvent( from: "TCP 가 전달합니다.", event);
    }
}
```

```java
public class SystemSignal implements ISource {
    2 usages
    Mediator mediator;
    2 usages
    @Override //중재자 설정
    public void setMediator(Mediator mediator) {
        this.mediator = mediator;
    }
    2 usages
    @Override //이벤트 전달
    public void eventOccured(String event) {
        mediator.onEvent( from: "SYSTEM 이 전달합니다.", event);
    }
}
```

예제

```java
public interface IDestination {
    1 usage   2 implementations
    void receiveEvent(String from, String event);
}
```

receiveEvent()
이벤트를 전달받아 출력

수신자

구현

```java
public class Display implements IDestination {
    1 usage
    @Override
    public void receiveEvent(String from, String event){
        System.out.println("[Display] FROM : " + from + " EVENT : " + event);
    }
}
```

```java
public class Log implements IDestination{
    1 usage
    @Override
    public void receiveEvent(String from, String event){
        System.out.println("[Log] FROM : " + from + " EVENT : " + event);
    }
}
```

예제

```
public class Mediator {  Complexity is 3 Everything is cool!
    2 usages
    List<IDestination> list = new ArrayList<>();
    2 usages
    public void addDestination(IDestination destination) { list.add(destination); }

    2 usages
    public void onEvent(String from, String event){
        for(IDestination each : list){
            each.receiveEvent(from, event);
        }
    }
}
```

addDestination()
이벤트 발생 시 이벤트를 전달 받을 수신자들을 추가

onEvent()
호출 받으면 해당하는 메시지를 수신자의 receiveEvent()를 호출해 전달

예제

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

    Mediator mediatorA = new Mediator();
    ISource tcp = new TcpComm();
    tcp.setMediator(mediatorA);

    Mediator mediatorB = new Mediator();
    ISource system = new SystemSignal();
    system.setMediator(mediatorB);

    mediatorA.addDestination(new Display());
    mediatorB.addDestination(new Log());

    tcp.eventOccured("tcp 이벤트 발생");
    system.eventOccured("system 이벤트 발생");

}
```

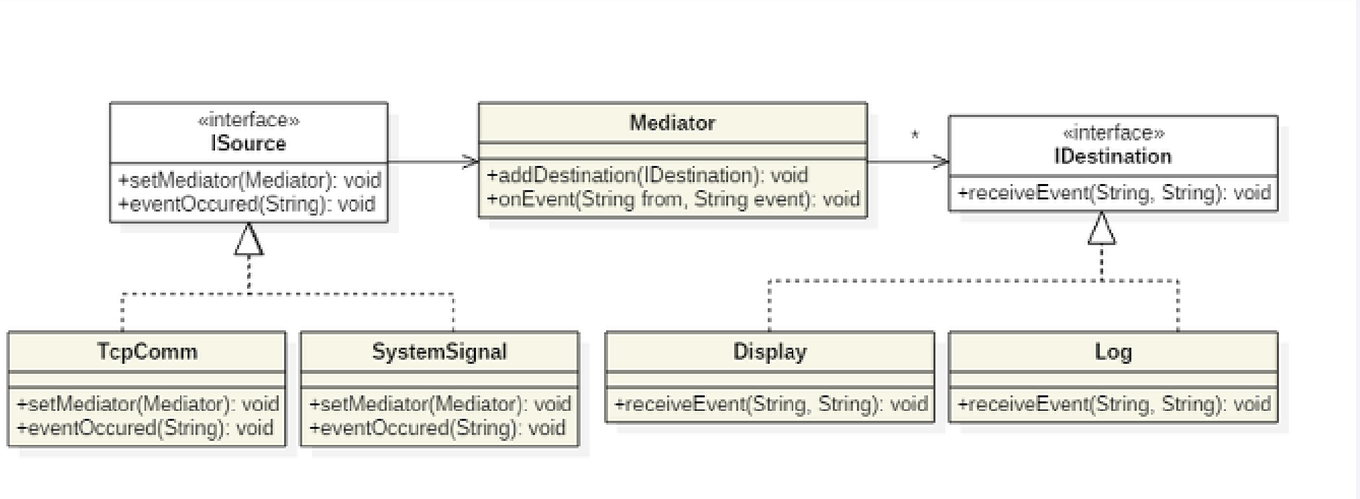
컴포넌트에 중재자 A 세팅

컴포넌트에 중재자 B 세팅

중재자 A 에 수신자 Display세팅

중재자 B 에 수신자 Log 세팅

각 컴포넌트에서 이벤트 발생




```
> Task :MediatorMain.main()
[Display] FROM : TCP 가 전달합니다. EVENT : tcp 이벤트 발생
[Log] FROM : SYSTEM 이 전달합니다. EVENT : system 이벤트 발생
```

중재자에 추가된 수신자를 통해 컴포턴트 이벤트 출력

언제 사용하면 좋을까?

일부 클래스들이 다른 클래스들과
단단하게 결합하여 변경하기 어려울 때

컴포넌트 간의 결합도를 줄일 수 있다

타 컴포넌트들에 너무 의존해 다른 프로그램에서
컴포넌트를 재사용할 수 없을 때

쉽게 재사용 할 수 있다

컴포넌트 간의 통신을 한곳으로 추출해
유지보수하기 쉽게 만들 수 있다

디자인 패턴의 장점을 이해하고
적절하게 사용해보아요

감사합니다

참고 :

https://refactoring.guru/ko/design-patterns/mediator ,https://www.crocus.co.kr/1542 ,
https://effectiveprogramming.tistory.com/entry/Mediator-%ED%8C%A8%ED%84%B4