# Mediator Pattern

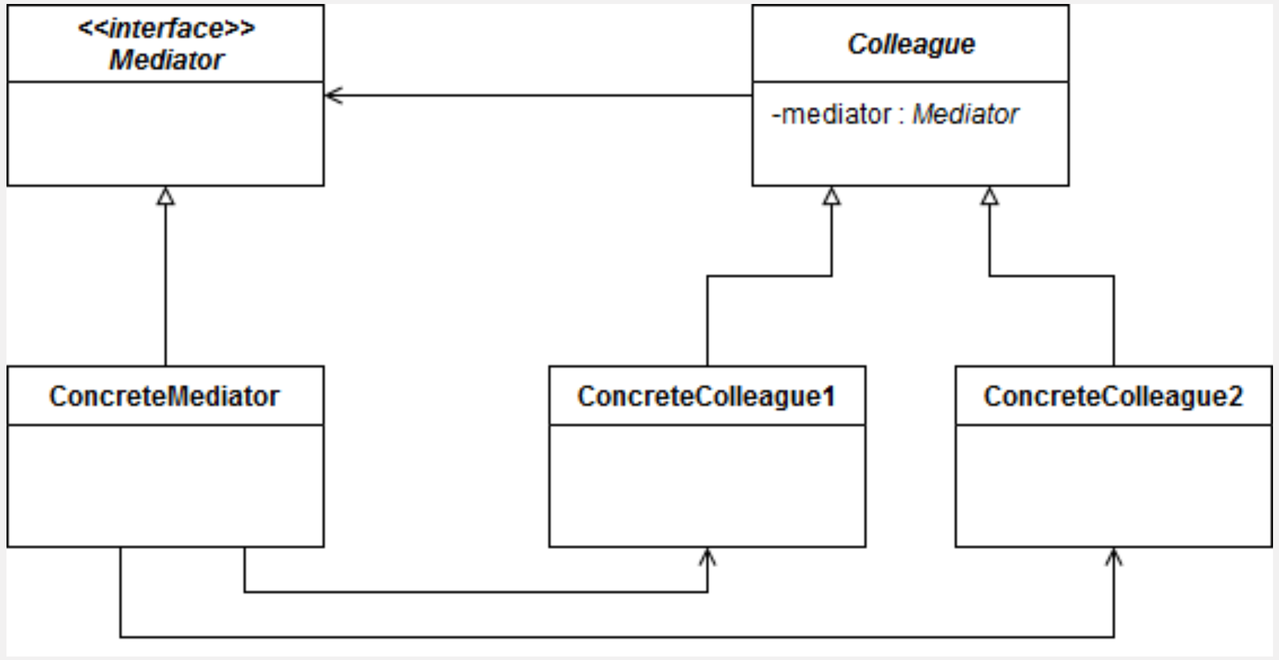
Lecture-9

## Mediator Pattern

- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to "full object status".

## Class Diagram

- Mediator defines the interface the Colleague objects use to communicate
- Colleague defines the abstract class holding a single reference to the Mediator
- ConcreteMediator encapsulates the interaction logic between Colleague objects
- ConcreteColleague1 and ConcreteColleague2 communicate only through the Mediator



## Example Scenario

Imagine we're building a simple cooling system that consists of a fan, a power supply, and a button. Pressing the button will either turn on or turn off the fan. Before we turn the fan on, we need to turn on the power. Similarly, we have to turn off the power right after the fan is turned off. Let's now take a look at the example implementation:

```java
public class Button {
    private Fan fan;

    // constructor, getters and setters

    public void press(){
        if(fan.isOn()){
            fan.turnOff();
        } else {
            fan.turnOn();
        }
    }
}
```

```java
public class Fan {
    private Button button;
    private PowerSupplier powerSupplier;
    private boolean isOn = false;

    // constructor, getters and setters

    public void turnOn() {
        powerSupplier.turnOn();
        isOn = true;
    }

    public void turnOff() {
        isOn = false;
        powerSupplier.turnOff();
    }
}
```

```java
public class PowerSupplier {
    public void turnOn() {
        // implementation
    }

    public void turnOff() {
        // implementation
    }
}
```

## Implementation With Mediator

```java
public class Mediator {
    private Button button;
    private Fan fan;
    private PowerSupplier powerSupplier;

    // constructor, getters and setters

    public void press() {
        if (fan.isOn()) {
            fan.turnOff();
        } else {
            fan.turnOn();
        }
    }

    public void start() {
        powerSupplier.turnOn();
    }

    public void stop() {
        powerSupplier.turnOff();
    }
}
```

```java
public class Button {
    private Mediator mediator;

    // constructor, getters and setters

    public void press() {
        mediator.press();
    }
}
```

```java
public class Fan {
    private Mediator mediator;
    private boolean isOn = false;

    // constructor, getters and setters

    public void turnOn() {
        mediator.start();
        isOn = true;
    }

    public void turnOff() {
        isOn = false;
        mediator.stop();
    }
}
```

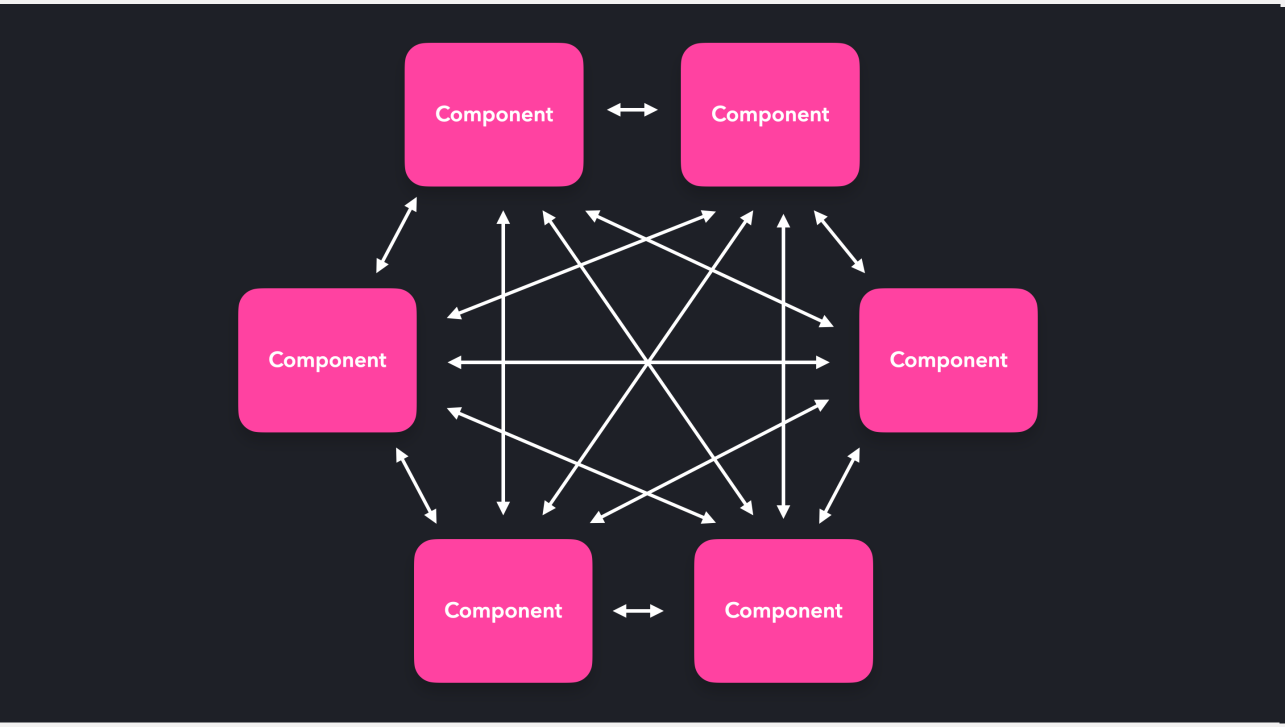
## Advantage

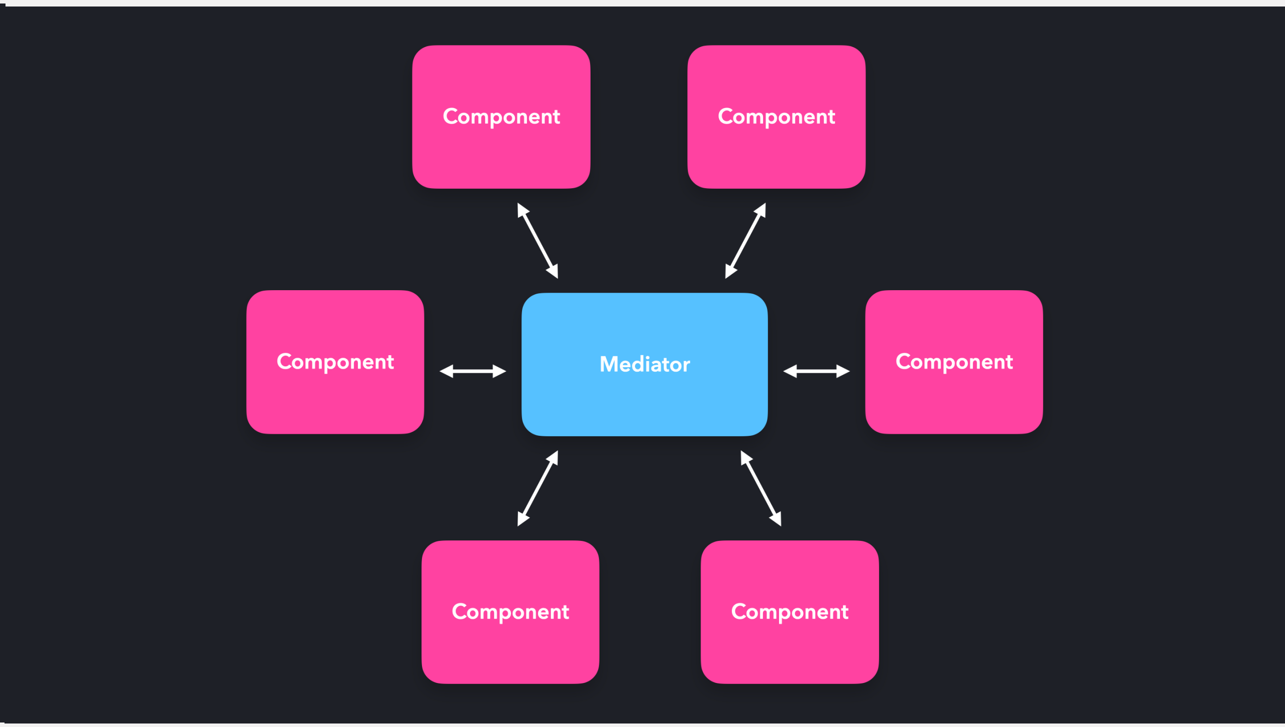
- It limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behaviour requires subclassing Mediator only, Colleague classes can be reused as is.

## Disadvantage

- It centralizes control. The mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that’s hard to maintain

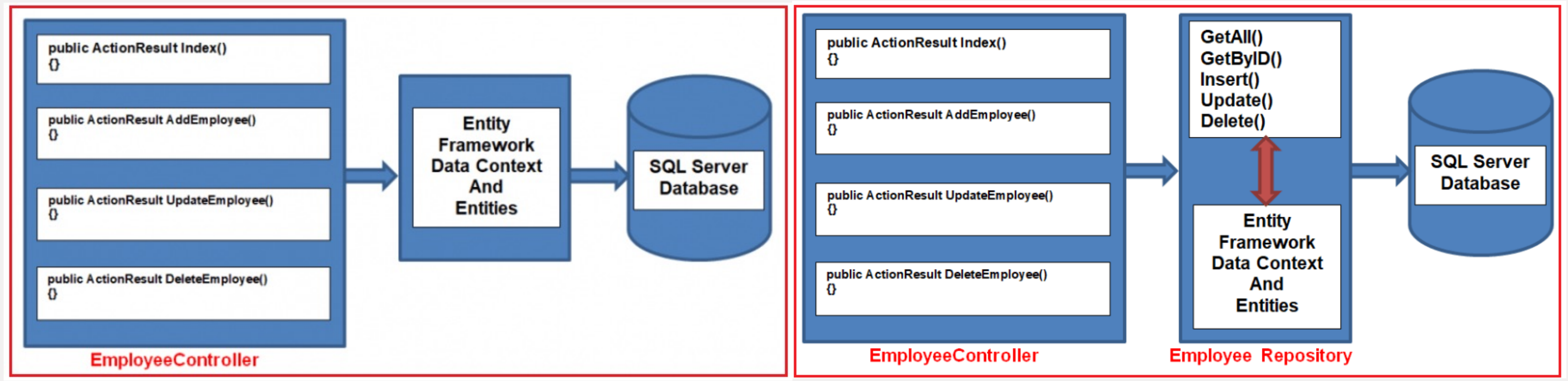
# Mediator/Middleware





## Repository Pattern

The repository pattern is extremely popular. In its modern interpretation, it abstracts the data store and enables your business logic to define read and write operations on a logical level. It does that by providing a set of methods to read, persist, update and remove an entity from the underlying data store.



public ActionResult Index()
{}
public ActionResult AddEmployee()
{}
public ActionResult UpdateEmployee()
{}
public ActionResult DeleteEmployee()
{}
EmployeeController
Entity Framework Data Context And Entities
SQL Server Database
public ActionResult Index()
{}
public ActionResult AddEmployee()
{}
public ActionResult UpdateEmployee()
{}
public ActionResult DeleteEmployee()
{}
EmployeeController
GetAll()
GetByID()
Insert()
Update()
Delete()
Entity Framework Data Context And Entities
Employee Repository
SQL Server Database