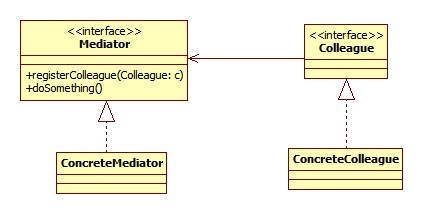
1. **Problem**

When colleague (peer) objects interact with each others, this will introduce high coupling between objects. The problem will become bigger when the complexity of the object increases. This will lead to hard to read and maintain because when you change something in one class, it will affect the others. Therefore, we should find a way to decouple those objects.

1. **Description**

Mediator will try to encapsulate the communication between objects into a Mediator object. Mediator object produces an abstraction layer among colleague objects so that any change in one object wont affect the others.

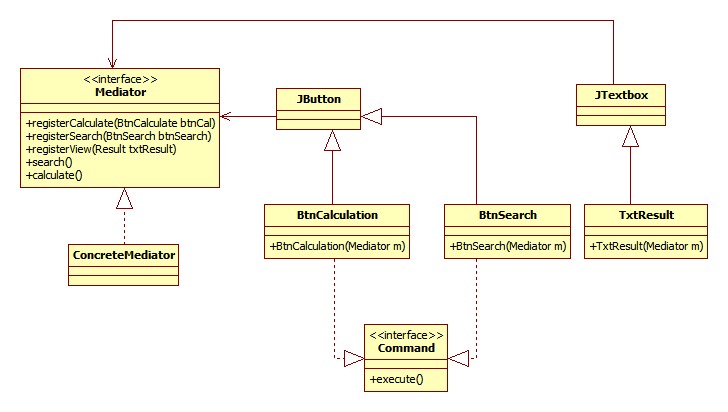
1. **Diagram**

****

1. **Example Scenario**

When developing GUI components for a desktop application, we will see the communication between components (JButton, JLabel) becomes very complicated. For example, in Management System at super market, users of this system need to have a lot of information and controls on the screen and these components need to be interact with each others to limit and reduce confusion for user. There is the case when an action listener is trigger, one button will call the label to edit its text and disable another button, or if there is no products then menu will be enabled and disable it when products are added into the system. At this point, you will see the high coupling between components, changing in one component could lead to others need to be edited. Furthermore, because the logic to control interaction is spread across components, maintenance would be extremely painful and wait of time. Therefore, mediator pattern will help you to centralize the interaction of components.

**Mediator Pattern Illustration:**



In the diagram, you can see, the interaction between of objects (btnCalculate and btnSearch) now moved to Mediator. You just need to register a component when it is constructed and call appropriate method in mediator. The mediator will control interactions among Buttons and Textbox.

**Sample Code:**

Public class Main extends JFrame implements ActionListener{

Public Main(){

Mediator mediator=new Mediator();

Add(new BtnCalculation(mediator));

Add(new BtnSearch(mediator));

Add(new TxtResult(mediator));

}

public void actionPerformed(ActionEvent e){

Command cmd=(Command) e.getSource();

cmd.execute();

}

public static void main(String[] args){

new Main();

}

}

public class BtnCalculation extends JButton implements Command{

public BtnCalculation(Mediator me){

me.registerCalculate(this);

}

public void execute(){

me.calculate();

}

}

1. **Advantages and Disadvantages**

* **Advantages:**
  + Decoupling objects supported by mediator
  + Reduce the complexity of calling each other within the peer objects
  + Simplify the maintenance process by centralizing control logic in mediator object
* **Disadvantages:**
  + The mediator itself will be complex because it will contain many other objects and control sending requests among them.
  + The multi-direction communication makes the design more complex compare to Observer pattern. Observer introduces the concept of observer and subject, hence there is one way of sending request.

1. **Alternative Pattern**

* Façade Pattern:

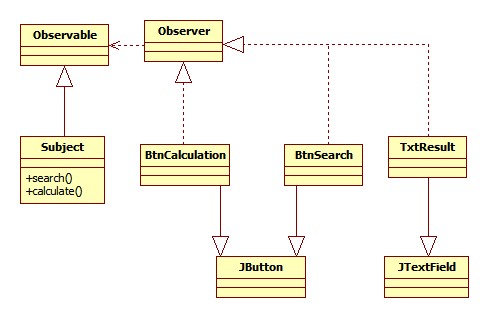
This pattern is quite similar to Mediator in term of centralizing the control logic of other objects and decoupling them. However, in some scenarios Façade pattern would outweigh the mediator. For example, when we have an external system communicate with a sub-system, we should use a façade instead of mediator to talk with sub-system to reduce the complexity. On the other hand, mediator just acts as the center point to interact with colleague objects. If we use the mediator object as an interface for external system, this would lead to mixing codes of internal and external system, and again, the object is not cohesive.

* Observer Pattern:

The pattern does the same thing as the mediator does. However, the observer pattern uses observer term for the object that is notified when another object state changes (the observable or subject). The interaction between observer and observable are in one direction whereas the mediator and peer objects have two references between them. In most of the case, using observer pattern is better because we usually need to be notified when state of an object changes. In addition, because we usually have a subject to notify for the observers to update, reusability is easier in observer pattern whereas in mediator you will have to modify new mediator so that it can register new object.

1. **Alternative Methods to achieve the same goal**

We can use the observer pattern to implement above example by changing mediator to subject object and colleague objects are observers. New class diagram can be seen as the following:



You can see now the subject (like the mediator) is simplified by removing the register methods. So when a method in subject is called, it will notify all observers to pull new data and update themselves. This action will put the renderer UI code toward UI components e.g setText in TxtResult.

* Advantages:
  + Simplify the Subject object by removing the register methods and providing data source for observers
  + Separate data source from the UI renderer
  + Reusable the observable objects and observer objects
* Disadvantages:
  + Subject class cant extend another class