Mediator Design Pattern in C#

In this article, I am going to discuss the **Mediator Design Pattern in C#** with examples. Please read our previous article where we discussed the **[Interpreter Design Pattern in C#](https://dotnettutorials.net/lesson/interpreter-design-pattern/)** with examples. The Mediator Design Pattern falls under the category of **[Behavioural Design Pattern](https://dotnettutorials.net/lesson/behavioral-design-pattern/)**. As part of this article, we are going to discuss the following pointers in detail.

1. **What is the Mediator Design Pattern?**
2. **Why do we need the Mediator Design Pattern?**
3. **Understanding the Class Diagram of Mediator Design Pattern**
4. **Real-Time Example of Mediator Design Pattern – Facebook Group**
5. **Implementing the Mediator Pattern in C#**

**What is the Mediator Design Pattern?**

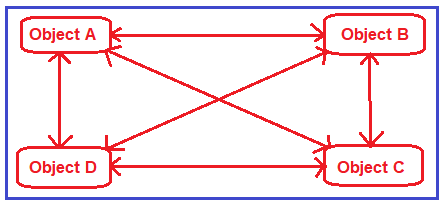
According to the Gang of Four’s definition, define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Let us first simplify the above GoF’s definitions: The Mediator Design Pattern is used to reduce the communication complexity between multiple objects. This design pattern provides a mediator object and that mediator object normally handles all the communication complexities between different objects.

The Mediator object acts as the communication center for all objects. That means when an object needs to communicate to another object, then it does not call the other object directly, instead, it calls the mediator object and it is the responsibility of the mediator object to route the message to the destination object.

**Why do we need the Mediator Pattern in C#?**

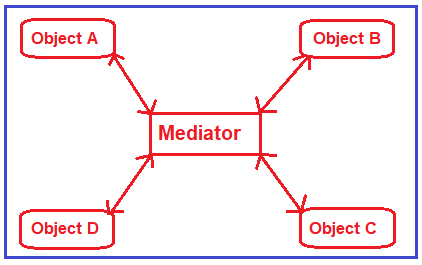
In order to understand this, please have a look at the following diagram. As you can see in the below image, we have four objects (Object A, Object B, Object C, and Object D). And these four objects want to communicate with each other. Suppose, Object A wants to communicate with Object B, then Object A should know the reference of Object B and using that reference Object A can call the method of Object B. Similarly, if Object B wants to send some message to Object C, then it should know the reference of Object C and using that reference it will call the methods of Object C and sends the message.



The couplings between the objects are more i.e. tight coupling. A lot of object knows each other. Now in the above image four objects are there. In real-time you have thousands of objects and those thousands of objects want to communicate with each other. Then writing code and maintainability of code is very difficult.

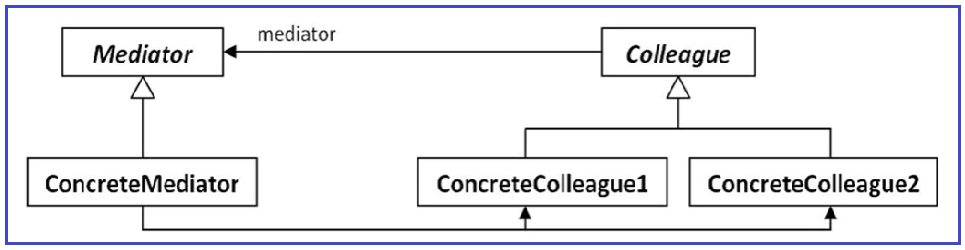
**How do we reduce the coupling between the objects or how to solve the above problem?**

Using Mediator Design Pattern we can solve the above problem. To understand this, please have a look at the following diagram. As you can see in the below image, here we can introduce the Mediator object. Each object has to send messages to the mediator object. What the mediator object do here is, it will receive the message from each object and route that message to the destination object. So, the mediator object will take care of handling the messages. So, in this type of scenarios, we can use the Mediator Design Pattern



**Understanding the Class Diagram of Mediator Design Pattern:**

Please have a look at the following image which shows the class diagram of the Mediator Design Pattern.



**Mediator:** It is an interface and it defines all possible interactions between colleagues.

**ConcreteMediator:** It is a class that implements the Mediator interface and coordinates communication between colleague objects.

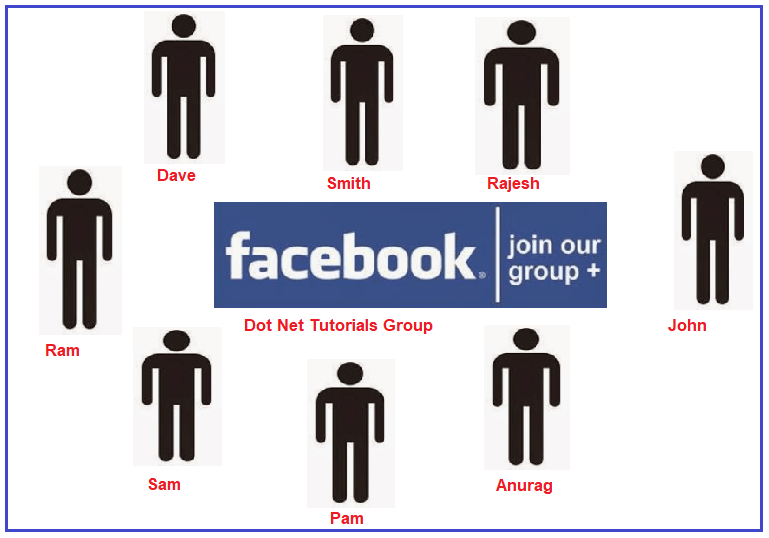
**Colleague:** It is an abstract class and this abstract class is going to be implemented by Concrete Colleague classes.

**ConcreteColleage1 / ConcreteColleage2:** These are classes and implements the Colleague interface. If a concrete colleague (let say ConcreteColleage1) wants to communicate with another concrete colleague (let say ConcreteColleage2), they will not communicate directly instead they will communicate via the ConcreteMediator.

If you are not understanding the above class diagram and the communication flow, then don’t worry we will try to understand this with an example.

**Real-Time Example of Mediator Design Pattern – Facebook Group**

Please have a look at the following diagram. Nowadays, everybody is aware of Facebook. On Facebook, we can create some specific groups and in that group, people can join and share their knowledge. Assume on Facebook there is a group called Dot Net Tutorials and in that group, many peoples (such as Ram, Sam, Pam, Dave, Steve, Anurag, etc.) are joined. Let say Ram sharing some dot net links in the Dot Net Tutorials group. Then what this Facebook Group (Dot Net Tutorials) will do is it will send that links to all the members who are joined in this group. So, here the Facebook group is acting as a Mediator.



**Implementing Mediator Design Pattern in C#:**

Let us implement the above Facebook example using the Mediator Design Pattern in C# step by step.

**Step1: Creating Mediator**

Create an interface with the name **FacebookGroupMediator** and then copy and paste the following code in it. This interface declares two abstract methods (i.e. SendMessage and RegisterUser).

namespace MediatorDesignPattern { public interface FacebookGroupMediator { void SendMessage(string msg, User user); void RegisterUser(User user); } }

**Step2: Creating ConcreteMediator**

Create a class file with the name **ConcreteFacebookGroupMediator.cs** and then copy and paste the following code in it. This class implements the **FacebookGroupMediator** interface and provides implementations for the SendMessage and RegisterUser abstract methods. The RegisterUser method will add the user to the particular Facebook group i.e. adding the user to the usersList variable. On the other hand, whenever any user shares any message in the group, then the SendMessage method will receive that message and it will send that message to all the users who are registered in the group except the user who shares the message.

using System.Collections.Generic; namespace MediatorDesignPattern { public class ConcreteFacebookGroupMediator : FacebookGroupMediator { private List<User> usersList = new List<User>(); public void RegisterUser(User user) { usersList.Add(user); } public void SendMessage(string message, User user) { foreach (var u in usersList) { // message should not be received by the user sending it if (u != user) { u.Receive(message); } } } } }

**Step3: Creating the Colleague**

Create an abstract class with the name **User** and then copy and paste the following code in it. This abstract class having two abstract methods i.e. Send and Receive.

**namespace** *MediatorDesignPattern*

**{**

**public** **abstract** **class** User

**{**

**protected** FacebookGroupMediator mediator;

**protected** string name;

**public** User**(**FacebookGroupMediator mediator, string name**)**

**{**

this.mediator = mediator;

this.name = name;

**}**

**public** **abstract** **void** Send**(**string message**)**;

**public** **abstract** **void** Receive**(**string message**)**;

**}**

**}**

**Step4: Creating ConcreteColleague**

Create a class file with the name **ConcreteUser** and then copy and paste the following code in it. This is a concrete class and this class implements the User abstract class and provides the implementation for the Send and Receive abstract methods. The Send method is basically used to send a message to a particular Facebook group. The Receive method is used to receive a message from a particular Facebook group.

**using** *System;*

**namespace** *MediatorDesignPattern*

**{**

**public** **class** ConcreteUser : User

**{**

**public** ConcreteUser**(**FacebookGroupMediator mediator, string name**)** : **base(**mediator, name**)**

**{**

**}**

**public** **override** **void** Receive**(**string message**)**

**{**

Console.WriteLine**(**this.name + ": Received Message:" + message**)**;

**}**

**public** **override** **void** Send**(**string message**)**

**{**

Console.WriteLine**(**this.name + ": Sending Message=" + message + "\n"**)**;

mediator.SendMessage**(**message, this**)**;

**}**

**}**

**}**

**Step5: Client**

Please modify the Main method as shown below. Here, first, we create the FacebookGroupMediator object, and then we are creating several Facebook users, and then we are registering all the users to the mediator. Then using the Dave and Rajesh user, we are sending a message to the Mediator.

using System; namespace MediatorDesignPattern { class Program { static void Main(string[] args) { FacebookGroupMediator facebookMediator = new ConcreteFacebookGroupMediator(); User Ram = new ConcreteUser(facebookMediator, "Ram"); User Dave = new ConcreteUser(facebookMediator, "Dave"); User Smith = new ConcreteUser(facebookMediator, "Smith"); User Rajesh = new ConcreteUser(facebookMediator, "Rajesh"); User Sam = new ConcreteUser(facebookMediator, "Sam"); User Pam = new ConcreteUser(facebookMediator, "Pam"); User Anurag = new ConcreteUser(facebookMediator, "Anurag"); User John = new ConcreteUser(facebookMediator, "John"); facebookMediator.RegisterUser(Ram); facebookMediator.RegisterUser(Dave); facebookMediator.RegisterUser(Smith); facebookMediator.RegisterUser(Rajesh); facebookMediator.RegisterUser(Sam); facebookMediator.RegisterUser(Pam); facebookMediator.RegisterUser(Anurag); facebookMediator.RegisterUser(John); Dave.Send("dotnettutorials.net - this website is very good to learn Design Pattern"); Console.WriteLine(); Rajesh.Send("What is Design Patterns? Please explain "); Console.Read(); } } }

**Output:**



**Real-time Example of Mediator Design Pattern – ATC**

ATC stands for Air Traffic Control. The ATC mediator is nothing but the Air Traffic Control Tower which is available at the Airport. Please have a look at the following image. Here, you can see different flights (such as Flight 101, Flight 202, Flight 707, and Flight 808).

Suppose Flight 101 wants to land at a particular terminal in the Airport. Then what the Flight Pilot will do is he will communicate with the **ATC Mediator** and saying he wants to land the **Flight 101** at the particular airport terminal. Then what the ATC Mediator will do is, he will check whether any flight is there at that particular terminal or not. If no flight is there, then what the ATC mediator will do is it will send a message to Pilots of other flights saying that Flight 101 is going to land and you should not land at that particular terminal. Then the ATC mediator sends a message to the **Flight 101 pilot** and saying you can land your flight at the particular airport terminal. Once the Flight 101 pilot got the confirmation message from the ATC Mediator then he will land the flight at that particular terminal.

Real-time Example of Mediator Design Pattern - ATC

So, here the ATC mediator will act as a central point and all flights should communicate to the ATC mediator. So, what the ATC mediator will do is, it will receive the message and route the message to the appropriate destinations. Here destinations are flight.

**Note:** The pilots of the planes are approaching or departing the terminal area communicates with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.