39. Mediator

Mediator is a design pattern where one object facilitates communication between multiple other objects. It's analogous to air traffic control, which acts as a mediator between different flights. Planes can talk to the air traffic control, which then delivers relevant information to other planes. But planes don't talk to each other directly.

The Mediator design pattern can be summarized as follows:

- There is a **Mediator** object, the role of which is to hold a list of **Participants** and facilitate communication between them.
- There are multiple **Participant** objects, which can be of completely different types.
- When a **Participant** wants to send a message to another **Participant**, it sends the message to **Mediator**.
- Based on some parameter that identifies the recipient, **Mediator** sends the message to an appropriate **Participant**.

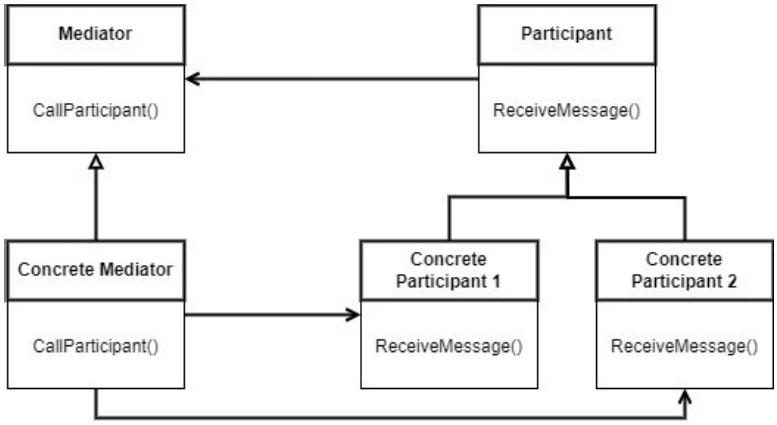

Figure 39.1 - Mediator UML diagram

We will now go through an example implementation of Mediator. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Mediator

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Mediator implementation example

As in all other examples, we will create a console application project. In this example, we will be building a system that represents a peer-to-peer computer network. Each device in the network

will be a **Participant**. The object representing the network will act as **Mediator**.

First, we will define the common interface that each **Participant** will implement. The interface will look as follows:

```
namespace Mediator_Demo;

internal interface IParticipant
{
    void SendCommand(
        string receiver, string command);
    void ReceiveCommand(
        string sender, string command);
}
```

Then, we will define the interface for the **Mediator**:

```
namespace Mediator_Demo;


internal interface IMediator
{
   void Register(
       string key,
       IParticipant participant);
   void SendCommand(
       string receiver,
       string sender,
       string command);
}
```

Because different **Participant** types will share some common functionality, we will also add the following abstract class that each **Participant** will inherit from:

```
namespace Mediator_Demo;

internal abstract class Participant(
    string key, IMediator mediator) :
        IParticipant
{
    protected string key = key;

    public virtual void SendCommand(
        string receiver,
        string command)
    {
        mediator.SendCommand(
            receiver, key, command);
    }

    public virtual void ReceiveCommand(
        string sender, string command)
    {
        Console.WriteLine(
            $"Executing command {command
            } issued by {sender}.");
    }
}
}
```

As you can see, when the `SendCommand` method is called, the call gets passed to the **Mediator**.

Now, we can add some concrete **Participant** implementations. We will first add the following class that represents a desktop computer:

```
namespace Mediator_Demo;

internal class DesktopComputer :
    Participant
{
        public DesktopComputer(
        string key,
        IMediator mediator) :
            base(key, mediator)
        {
        }

    public override void SendCommand(
        string receiver, string command)
    {
        Console.WriteLine(
            $"Sending {command
            } command to {receiver}.");
        base.SendCommand(
            receiver, command);
    }

    public override void ReceiveCommand(
        string sender, string command)
    {
        Console.Write(
            $"Desktop computer {key
            } received a command. ");
        base.ReceiveCommand(
            sender, command);
    }
}
```

Then, we will add the following class that represents a server:

```
namespace Mediator_Demo;

internal class Server(
    string key,
    IMediator mediator) :
        Participant(key, mediator)
{
    public override void SendCommand(
        string receiver,
        string command)
    {
        Console.WriteLine(
            $"Server has issued {command
            } command to {receiver}.");
        base.SendCommand(
            receiver, command);
    }

    public override void ReceiveCommand(
        string sender, string command)
    {
        Console.Write(
            $"Server {key
            } received a command. ");
        base.ReceiveCommand(
            sender, command);
    }
}
```

The logic in both of these **Participant** types is similar. But it's different enough to make them distinct.

Next, we will add the following **Mediator** implementation:

```
namespace Mediator_Demo;

internal class NetworkMediator :
    IMediator
{
        private Dictionary<string,
        IParticipant> participants;

        public NetworkMediator()
        {
                participants = [];
        }

        public void Register(
        string key,
        IParticipant participant)
    {
                participants[key] = participant;

        }

        public void SendCommand(
        string receiver,
        string sender,
        string command)
    {
                if (participants
            .ContainsKey(receiver))
        {
                        participants[receiver]
                .ReceiveCommand(
                    sender, command);

                }
        }
```

```
}
```

In here, we register all **Participant** instances in a dictionary. The dictionary key is the identifier that allows each **Participant** to tell the **Mediator** which **Participant** the message needs to be sent to.

Finally, we will bring it all together by replacing the content of the `Program.cs` file with the following:

```
using Mediator_Demo;

var networkMediator =
    new NetworkMediator();
var desktopComputer =
    new DesktopComputer(
        "computer-1", networkMediator);
var server = new Server(
    "server-1", networkMediator);

networkMediator.Register(
    "computer-1", desktopComputer);
networkMediator.Register(
    "server-1", server);

desktopComputer.SendCommand(
    "server-1", "reboot");
server.SendCommand(
    "computer-1", "trigger-updates");

Console.ReadKey();
```

And, as we can see from the screenshot below, we were able to register different **Participant** instances inside the **Mediator** object and get them to communicate with each other via the **Mediator**.

```
Sending reboot command to server-1.
Server server-1 received a command. Executing command reboot issued by computer-1.
Server has issued trigger-updates command to computer-1.
Desktop computer computer-1 received a command. Executing command trigger-updates issued by server-1.
```

Figure 39.2 - The results of using Mediator

Let's now summarize the main benefits of using the Mediator design pattern.

Benefits of using Mediator

The main benefits of using the Mediator design pattern are as follows:

- It significantly reduces coupling between objects, as none of the **Participants** call each other directly.
- Both the single responsibility principle and the open-closed principle are well enforced.

Perhaps there is only one caveat with the Mediator design pattern. And this is what we will have a look at next.

Caveats of using Mediator

The main danger with using the Mediator design pattern is that, if you have to deal with many different **Participant** types, your **Mediator** object can easily evolve into a God object - the antipattern that is opposite to the single responsibility principle. Therefore, while using Mediator, it's often worth applying other design patterns inside the **Mediator** object to ensure that each individual piece of logic is handled by its own component.