WIKIPEDIA

Mediator pattern

In software engineering, the **mediator pattern** defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

Usually a program is made up of a large number of classes. Logic and computation are distributed among these classes. However, as more classes are added to a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become more complex. This makes the program harder to read and maintain. Furthermore, it can become difficult to change the program, since any change may affect code in several other classes.

With the **mediator pattern**, communication between objects is encapsulated within a **mediator** object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

Contents

Overview

The Mediator [1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Mediator design pattern solve? [2]

- Tight coupling between a set of interacting objects should be avoided.
- It should be possible to change the interaction between a set of objects independently.

Defining a set of interacting objects by accessing and updating each other directly is inflexible because it tightly couples the objects to each other and makes it impossible to change the interaction independently from (without having to change) the objects. And it stops the objects from being reusable and makes them hard to test.
Tightly coupled objects are hard to implement, change, test, and reuse because they refer to and know about many different objects.

What solution does the Mediator design pattern describe?

- Define a separate (mediator) object that encapsulates the interaction between a set of objects.
- Objects delegate their interaction to a mediator object instead of interacting with each other directly.

The objects interact with each other indirectly through a mediator object that controls and coordinates the interaction.
This makes the objects *loosely coupled*. They only refer to and know about their mediator object and have no explicit knowledge of each other.

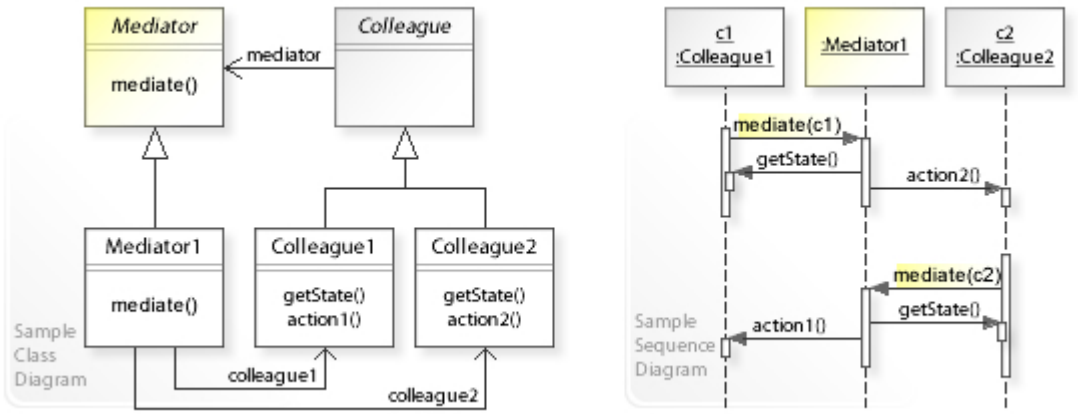
See also the UML class and sequence diagram below.

Definition

The essence of the Mediator Pattern is to "define an object that encapsulates how a set of objects interact". It promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to be varied independently.[3] Client classes can use the mediator to send messages to other clients, and can receive messages from other clients via an event on the mediator class.

Structure

UML class and sequence diagram

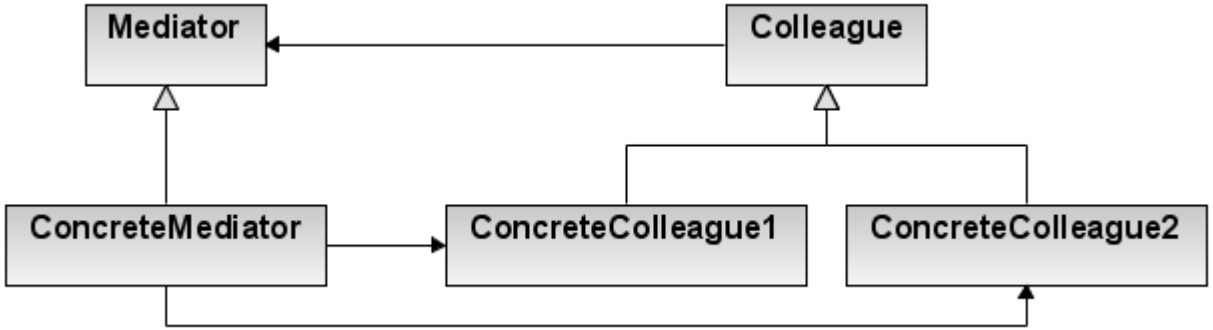


A sample UML class and sequence diagram for the Mediator design pattern. [4]

In the above UML class diagram, the `Colleague1` and `Colleague2` classes do not refer to (and update) each other directly. Instead, they refer to the common `Mediator` interface for controlling and coordinating interaction (`mediate()`), which makes them independent of how the interaction is carried out. The `Mediator1` class implements the interaction between `Colleague1` and `Colleague2`.

The UML sequence diagram shows the run-time interactions: In this example, a `Mediator1` object mediates (controls and coordinates) the interaction between `Colleague1` and `Colleague2` objects.
Assuming that `Colleague1` wants to interact with `Colleague2` (to update/synchronize its state, for example), `Colleague1` calls `mediate(this)` on the `Mediator1` object, which gets the changed data from `Colleague1` and performs an `action2()` on `Colleague2`.
Thereafter, `Colleague2` calls `mediate(this)` on the `Mediator1` object, which gets the changed data from `Colleague2` and performs an `action1()` on `Colleague1`.

Class diagram



The mediator behavioural design pattern

Participants

Mediator - defines the interface for communication between *Colleague* objects

ConcreteMediator - implements the Mediator interface and coordinates communication between *Colleague* objects. It is aware of all of the *Colleagues* and their purposes with regards to inter-communication.

Colleague - defines the interface for communication with other *Colleagues* through its *Mediator*

ConcreteColleague - implements the Colleague interface and communicates with other *Colleagues* through its *Mediator*

Example

C#

The Mediator pattern ensures that components are loosely coupled, such that they don't call each other explicitly, but instead do so through calls to a mediator. In the following example, the Mediator registers all Components and then calls their SetState methods.

```
public interface IComponent
{
    void SetState(object state);
}

public class Component1 : IComponent
{
    public void SetState(object state)
    {
        throw new NotImplementedException();
```

```
    }
}

public class Component2 : IComponent
{
    public void SetState(object state)
    {
        throw new NotImplementedException();
    }
}

// Mediates the common tasks
public class Mediator
{
    public IComponent Component1 { get; set; }
    public IComponent Component2 { get; set; }

    public void ChangeState(object state)
    {
        this.Component1.SetState(state);
        this.Component2.SetState(state);
    }
}
```

A chat room could use the Mediator pattern, or a system where many 'clients' each receive a message each time one of the other clients performs an action (for chat rooms, this would be when each person sends a message). In reality using the Mediator pattern for a chat room would only be practical when used with remoting. Using raw sockets wouldn't allow for the delegate callbacks (people subscribed to the Mediator class' MessageReceived event).

```
public delegate void MessageReceivedEventHandler(string message, string from);

public class Mediator
{
    public event MessageReceivedEventHandler MessageReceived;

    public void Send(string message, string from)
    {
        if (MessageReceived != null)
        {
            Console.WriteLine("Sending '{0}' from {1}", message, from);
            MessageReceived(message, from);
        }
    }
}

public class Person
{
    private Mediator _mediator;

    public string Name { get; set; }

    public Person(Mediator mediator, string name)
    {
        Name = name;
```

```
        _mediator = mediator;
        _mediator.MessageReceived += new MessageReceivedEventHandler(Receive);
    }

    private void Receive(string message, string from)
    {
        if (from != Name)
            Console.WriteLine("{0} received '{1}' from {2}", Name, message, from);
    }

    public void Send(string message)
    {
        _mediator.Send(message, Name);
    }
}
```

Java

In the following example, a `Mediator` object controls the values of several `Storage` objects forcing the user code to access to the stored values through the mediator. When a storage object want to emit an event indicating that its value has changed, it also go back to the mediator object (via the method `notifyObservers`) that controls the list of the observers (implemented using the observer pattern).

```
import java.util.HashMap;
import java.util.Optional;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.function.Consumer;

class Storage<T> {
  T value;

  T getValue() {
    return value;
  }
  void setValue(Mediator<T> mediator, String storageName, T value) {
    this.value = value;
    mediator.notifyObservers(storageName);
  }
}

class Mediator<T> {
  private final HashMap<String, Storage<T>> storageMap = new HashMap<>();
  private final CopyOnWriteArrayList<Consumer<String>> observers = new CopyOnWriteArrayList<>();

  public void setValue(String storageName, T value) {
    var storage = storageMap.computeIfAbsent(storageName, name -> new Storage<>());
    storage.setValue(this, storageName, value);
  }

  public Optional<T> getValue(String storageName) {
    return Optional.ofNullable(storageMap.get(storageName)).map(Storage::getValue);
  }
```

```
  public void addObserver(String storageName, Runnable observer) {
    observers.add(eventName -> {
      if (eventName.equals(storageName)) {
        observer.run();
      }
    });
  }

  void notifyObservers(String eventName) {
    observers.forEach(observer -> observer.accept(eventName));
  }
}

public class MediatorDemo {
  public static void main(String[] args) {
    var mediator = new Mediator<Integer>();
    mediator.setValue("bob", 20);
    mediator.setValue("alice", 24);
    mediator.getValue("alice").ifPresent(age -> System.out.println("age for alice: " + age));

    mediator.addObserver("bob", () -> {
      System.out.println("new age for bob: " + mediator.getValue("bob").orElseThrow());
    });
    mediator.setValue("bob", 21);
  }
}
```

See also

- Data mediation
- Design Patterns, the book which gave rise to the study of design patterns in computer science
- Design pattern (computer science), a standard solution to common problems in software design

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 273ff. ISBN 0-201-63361-2.
2. "The Mediator design pattern - Problem, Solution, and Applicability" (http://w3sdesign.com/?gr=b05&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-12.
3. Gang Of Four
4. "The Mediator design pattern - Structure and Collaboration" (http://w3sdesign.com/?gr=b05&ugr=struct). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- Mediator Design Pattern (http://sourcemaking.com/design_patterns/mediator)

- Is the use of the mediator pattern recommend? - Stack Overflow (https://stackoverflow.com/questions/12534338/is-the-use-of-the-mediator-pattern-recommend)
