search...

Home » Behavioral Patterns » Mediator

Bookmark this on Delicious

Main Menu

- Home
- Design Principles
  - Open Close Principle
  - Dependency Inversion Principle
  - Interface Segregation Principle
  - Single Responsibility Principle
  - Liskov's Substitution Principle
- Creational Patterns
  - Singleton
  - Factory
  - Factory Method
  - Abstract Factory
  - Builder
  - Prototype
  - Object Pool
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - Strategy
  - Template Method
  - Visitor
  - Null Object
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Flyweight
  - Proxy
- Design Pattern Books

Forum

- What Design Pattern To Choose?
- Design Principles and Patterns
- Enterprise Patterns
- Books

Mediator Pattern

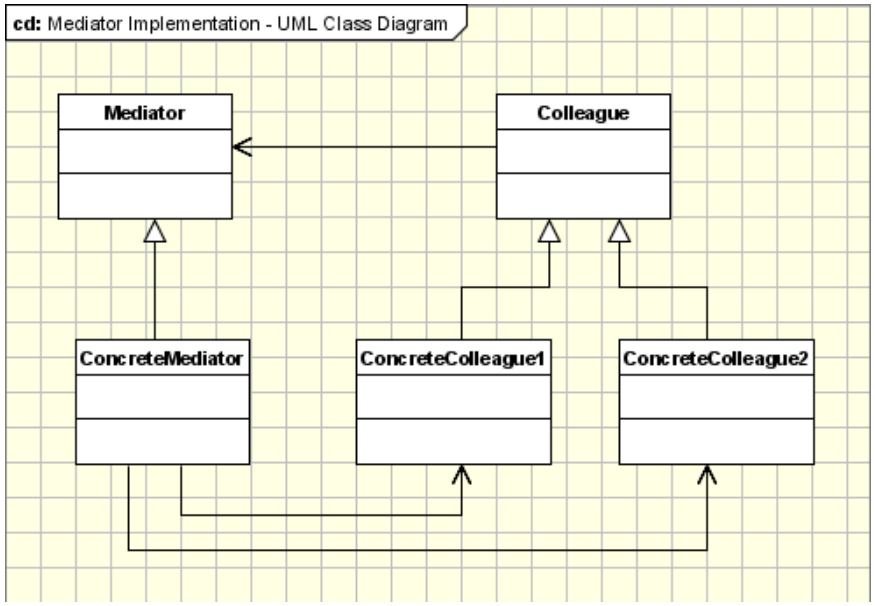
Motivation

In order to have a good object oriented design we have to create lots of classes interacting one with each other. If certain principles are not applied the final framework will end in a total mess where each object relies on many other objects in order to run. In order to avoid tight coupled frameworks, we need a mechanism to facilitate the interaction between objects in a manner in that objects are not aware of the existence of other objects.

Let's take the example of a screen. When we create it we add all sort of controls to the screen. This control need to interact with all the other control. For example when a button is pressed it must know if the data is valid in other controls. As you have seen if you created different applications using forms you don't have to modify each control class each time you add a new control to the form. All the operations between controls are managed by the form class itself. This class is called mediator.

Intent

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.



Implementation

Participants

The participants classes in this pattern are:
Mediator - defines an interface for communicating with Colleague objects.
ConcreteMediator - knows the colleague classes and keep a reference to the colleague objects.
- implements the communication and transfer the messages between the colleague classes
Colleague classes - keep a reference to its Mediator object
- communicates with the Mediator whenever it would have otherwise communicated with another Colleague.

Applicability

According to Gamma et al, the Mediator pattern should be used when:
a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
reusing an object is difficult because it refers to and communicates with many other objects.
a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Examples

Example 1 - GUI Libraries

The mediator example is one pattern that is already used in many applications. One of the examples is represented by the Dialog classes in GUI applications frameworks. A Dialog window is a collection of graphic and non-graphic controls. The Dialog class provides the mechanism to facilitate the interaction between controls. For example, when a new value is selected from a ComboBox object a Label has to display a new value. Both the ComboBox and the Label are not aware of each other structure and all the interaction is managed by the Dialog object. Each control is not aware of the existence of other controls.

Example 2 - Chat application

The chat application is another example of the mediator pattern. In a chat application we can have several participants. It's not a good idea to connect each participant to all the others because the number of connections would be really high, there would be technical problems due to proxies and firewalls, etc... . The most appropriate solution is to have a hub where all participants will connect; this hub is just the mediator class.

Participants:

Chatroom(Mediator) - Defines the interface for interacting with participants

ChatroomImpl (ConcreteMediator) - implements the operations defined by the Chatroom interface. The operations are managing the interactions between the objects: when one participant sends a message, the message is sent to the other participants.

Participant(Collegue) - defines an interface for the participants.

HumanParticipant, Bot (ConcreteCollegue) - implements participants; the participant can be a human or a bot, each one having a distinct implementation but implementing the same interface. Each participant will keep only a reference to the mediator.

Specific problems and implementation

Abstract Mediators

There is no need to create an Abstract Mediator class or an interface as long as the colleagues are going to use only one mediator. The definition of an abstract Mediator is required only if the colleagues needs to work with different mediators.

Communication between mediators and colleagues

There are different ways to realize the communication between the colleagues and its mediator:

One of the most used methods is to use the Observer pattern. The mediator can be also an observer and the Colleagues can be implement an observable object. Each time an change is made in the state of the observable object, the observer(mediator) gets notified and it notify all other colleagues objects.

Alternative methods can be used to send messages to the mediator. For example a simple delegation can be used and specialised methods can be exposed by the mediator

In more complex implementations asynchronous messages can be added to to a message queue, from where they can be picked up by the mediator object

Complexity of Mediator object

The mediator object handles all the interaction between the participants objects. One potential problem is complexity of the mediator when the number of participants is a high and the different participant classes is high. If you created custom dialogs for GUI applications you remember that after some time the dialogs classes become really complex because they had to manage a lot of operations.

Consequences

As with most design patterns, there are both advantages and disadvantages of using the Mediator Patern. The following section will briefly outline a few of these issues.

Advantages

Comprehension - The mediator encapsulate the logic of mediation between the colleagues. From this reason it' more easier to understand this logic since it is kept in only one class.
Decoupled Colleagues - The colleague classes are totally decoupled. Adding a new colleague class is very easy due to this decoupling level.
Simplified object protocols - The colleague objects need to communicate only with the mediator objects. Practically the mediator pattern reduce the required communication channels(protocols) from many to many to one to many and many to one.
Limits Subclassing - Because the entire communication logic is encapsulated by the mediator class, when this logic need to be extended only the mediator class need to be extended.

Disadvantages

Complexity - in practice the mediators tends to become more complex and complex. A good practice is to take care to make the mediator classes responsible only for the communication part. For example when implementing different screens the the screen class should not contain code which is not a part of the screen operations. It should be put in some other classes.

Related Patterns

There are a few design patterns that are closely related to the Mediator pattern and are often used in conjunction with it.
Facade Pattern - a simplified mediator becomes a facade pattern if the mediator is the only active class and the colleagues are passive classes. A facade pattern is just an implementation of the mediator pattern where mediator is the only object triggering and invoking actions on passive colleague classes. The Facade is being call by some external classes.
Adapter Pattern - the mediator patter just "mediate" the requests between the colleague classes. It is not supposed to change the messages it receives and sends; if it alters those messages then it is an Adapter pattern.
Observer Pattern - the observer and mediator are similar patterns, solving the same problem. The main difference between them is the problem they address. The observer pattern handles the communication between observers and subjects or subject. It's very probable to have new observable objects added. On the other side in the mediator pattern the mediator class is the the most likely class to be inherited.

Known Uses

In the following section, we'll discuss some real-world uses of the Mediator pattern. You'll find the Mediator in many situations where there are many components that must interact with one another in complex ways. User Interfaces

Maybe the mediator pattern is mostly used in the user interfaces. Almost any GUI framework is build around it. Like discussed before, the classes representing forms (Dialog, Form,...) represents the the mediator while each control represents a colleague. The form class provides the mechanism to facilitate the interaction between controls; an inherited class is created each time a new screen is created and the specific code is written in this class. This way, the controls communication is mediated by this form class. Java Message Service

The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. The JMS API supports 2 models. One of them is the publish-subscribe model. It is an implementation of the mediator pattern. The messages can be publisehd based on a particular topic. The publisher has to create a subscription to which different subscribers may subscribe. Zero or more subscribers may subscribe to receive messages on a particular message topic. The publisher and the subscriber doesn't know one about eachother, the subscriber can be even inactive. In this case the subscriber receives the messages when it will become active.

Conclusion

The mediator pattern is used to takes the role of a hub or router and facilitates the communication between many classes. A similarity can be made with the database systems. The mediator transform a hard to implement relation of many to many, where each calls has to communicate with each other class, in 2 relations, easy to implement, of many to one and one to many, where the communication is handled by the mediator class.

Next >

[Back]