GROUP 7

Trần Trí Thiện - 1551037 Nguyễn Trần Phước Thịnh - 1551038

Hồ Sỹ Nguyên - 1551023 Lâm Lê Thanh Thế - 1551034

CS202 SEMINAR REPORT

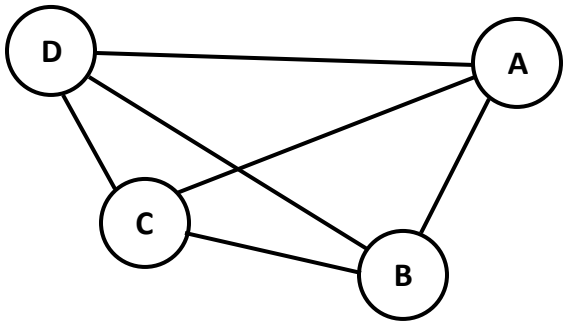
DESIGN PATTERN

MEDIATOR

1. Definition and Motivation

According to the Gang of Four, mediator is a behavioral pattern, intended to encapsulate some relationships or interactions between objects into a class. Hence, the mediator can be considered as a middle man who directs controls between other classes' instances.

The existence of mediators is given rise by complicated relationships between objects. As object-oriented programming has been becoming more popular, the projects to work on have turned huge, and complex. At those projects, one class (or particularly one object) is having lots of interaction with another, such as inheriting some base classes, observing (getting information from), or depending on the result data from other classes to function. This not only makes the relationship diagram sophisticated, but also the number of functions and objects inside one class that are related to other classes increases. Thus, programmers get more work to do.

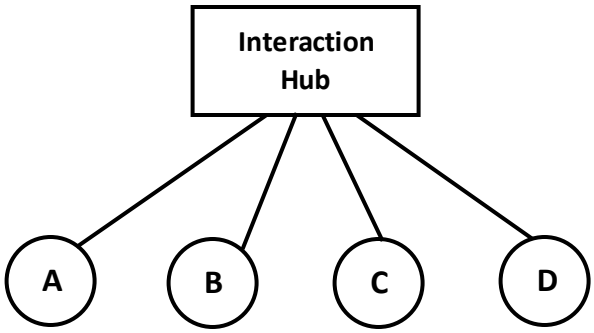


Large project or problem also requires the programmers to partition it into many classes and sub-problems. As the number of objects goes up, every object tends to have so many connections with the others.

For example, the figure to the left shows a highly dense direct connection mapping of four objects.

So, what is the actual problem of this? If our system is mapped like above, when a change needs to be make, it is definitely a hard task, not only time-consuming but also mind-puzzling. As an object depends on many other ones, when it changes, the others are affected. Therefore, it is really tough to maintain or modify the system's behaviors. To keep the consistence, when we need to change only one thing, we must change lots of other things.

What if, the connections go through a middle control center, not flowing directly?



The direct connection between a pair is loosened. It is easier to make changes to objects now. The only thing to be considered when an object is changed (e.g. disappears) is its connection to the control center.

Note:

Mediator controls the interactions of objects that are nearly of the same type, or say, at least their classes all inherit one base class.

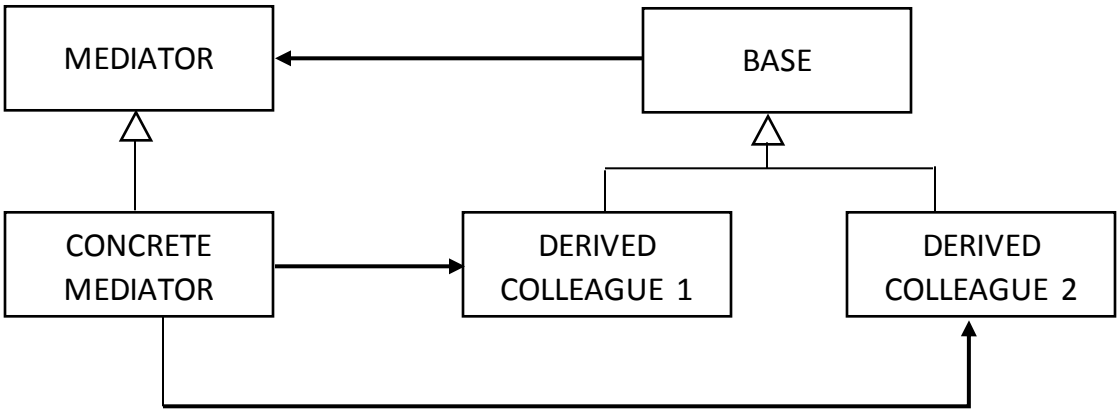
This should be keep in mind to distinguish between mediator and another design pattern: visitor, which will be discussed later.

2. Applicability

A mediator can be used in case:

- A set of objects are having complex dependencies on each other, which makes the design hard to understand.
- Reusing an object is difficult since it communicates or has many relationships with other objects.
- Many classes share the same behavior.

3. Design Structure



Mediator: Define the interface of the mediators used to communicate with objects.

Concrete Mediator: implement an actual mediator class, know and maintain a list of objects it needs to communicate with. This can be implemented using arrays, lists or vectors of pointers or references.

Colleague: The classes implementing the objects that having interactions with each other. Instead of having a direct connection between the two, they now communicate with each other through a concrete mediator.

Here, objects send and receive requests from the mediator, and the mediator decides the request routing among objects.

4. Example 1: Node deletion

When node objects interact directly with each other, recursion is required to operate, i.e. to print the list or remove a node. The linked list here is represented by a head pointer to type Node.

```
class Node
{
 public:
    Node(int v)
    {
        m_val = v;
        m_next = 0;
    }
    void add_node(Node *n)
    {
        if (m_next)
            m_next->add_node(n);
        else
            m_next = n;
    }
    void traverse()
    {
        cout << m_val << "  ";
        if (m_next)
            m_next->traverse();
        else
            cout << '\n';
    }
    void remove_node(int v)
    {
        if (m_next)
            if (m_next->m_val == v)
            {
                Node *tmp = m_next;
                m_next = m_next->m_next;
                delete tmp;
            }
            else
                m_next->remove_node(v);
```

```
    }
 private:
    int m_val;
    Node *m_next;
};
```

When using one more class to cover the nodes and manage them, we can avoid recursion (and avoid stack overflow). Also, the situation above doesn't allow us to delete the first node, as we are using the object itself to call the remove function. Now we can change the first node easily and use iteration to operate.

```
class Node
{
 public:
    Node(int v)
    {
        m_val = v;
    }
    int get_val()
    {
        return m_val;
    }
 private:
    int m_val;
};

class List
{
 public:
    void add_node(Node *n)
    {
        m_arr.push_back(n);
    }
    void traverse()
    {
        for (int i = 0; i < m_arr.size(); ++i)
            cout << m_arr[i]->get_val() << "  ";
        cout << '\n';
    }
    void remove_node(int v)
    {
        for (vector::iterator it = m_arr.begin(); it != m_arr.end(); ++it)
        if ((*it)->get_val() == v)
        {
            m_arr.erase(it);
            break;
        }
    }
 private:
    vector m_arr;
};
```

The class “List” is a center to manage connections among nodes. “List” acts not only as a mediator, but it is also sometimes referred as “a level of indirection”. All tasks on the nodes must go through the list first.

5. Example 2: Widget Management

Here only the prototypes are given; full code is available at: https://sourcemaking.com/design_patterns/mediator/cpp/1

```
class FileSelectionDialog;

class Widget
{
 public:
    Widget(FileSelectionDialog *mediator, char *name);
    virtual void changed();
    virtual void updateWidget() = 0;
    virtual void queryWidget() = 0;
  protected:
    char _name[20];
  private:
    FileSelectionDialog *_mediator;
};

class List: public Widget
{
 public:
    List(FileSelectionDialog *dir, char *name): Widget(dir, name){}
    void queryWidget();
    void updateWidget();
};

class Edit: public Widget
{
 public:
    Edit(FileSelectionDialog *dir, char *name): Widget(dir, name){}
    void queryWidget();
    void updateWidget();
};

class FileSelectionDialog
{
 public:
    enum Widgets
    {
        FilterEdit, DirList, FileList, SelectionEdit
    };
    FileSelectionDialog();
    virtual ~FileSelectionDialog();
    void handleEvent(int which);
```

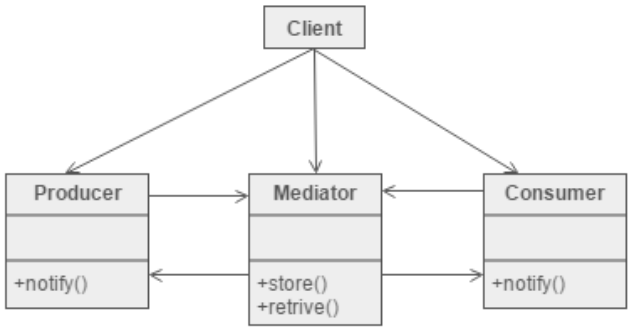
```
    virtual void widgetChanged(Widget *theChangedWidget);
 private:
    Widget *_components[4];
};
```

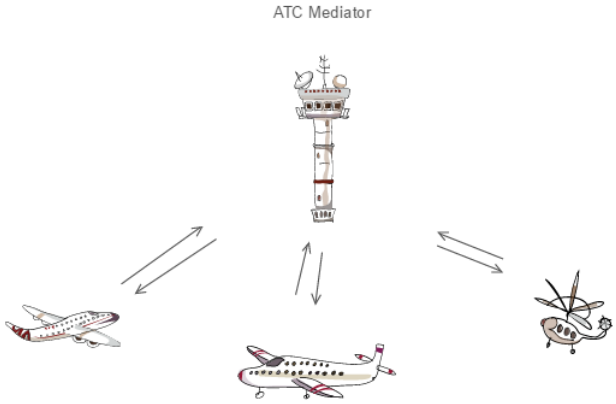
In this example, a dialog is the mediator – a hub of communication managing many widgets. The dialog holds a list of widgets it manages, and the child widgets do not care about their siblings. When a user makes changes to a widget through `Widget::changed()`, it only calls `mediator->widgetChanged(this)` and it is up to the dialog to decide all the things after that.

6. Other real-world examples and uses

Here are some models that apply mediators in real life:

Mediator directs the communication between a producer and a consumer.





Flight control station is also a kind of mediator.

Also mostly used in GUI design, mediator is applied in almost every user interfaces.

In communication API: The Java Message Service (JMS) API, a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients.

In coordinating complex updates: The Change Manager class, which mediates between subjects and observers to avoid redundant updates. When an object

changes, it notifies the Change Manager, which in turn coordinates the update by notifying the object's dependents.

In drawing frameworks: The Unidraw drawing framework, which uses a class called CSolver to enforce connectivity constraints between "connectors." Objects in graphical editors can appear to stick to one another in different ways. Connectors are useful in applications that maintain connectivity automatically, like diagram editors and circuit design systems. CSolver is a mediator between connectors. It solves the connectivity constraints and updates the connectors' positions to reflect them.

7. Pros and cons

Advantages:

- *Comprehension:* The mediator reduces a lot of direct connections among objects, which makes the logic via class diagrams or any models becomes easier to understand.
- *Pair-loose dependence:* The relationship between a pair of object is now less strict. This allows altering or reusing an object more easily and not affecting much the other objects.
- *Simplified object protocols:* An object now only needs to communicate with the mediator, and the rest of the job is taken care by the mediator itself, which the object will not care.
- *Sub-classing reduction:* When we need to update or extend the connections between objects, we do not have to deal with the classes of our objects. Instead, using inheritance and polymorphism with the mediators, simply create a new derived mediator and build up the new communication rules.

Drawback:

- *Complexity: This is not complex in terms of the relationship design, as mediators exist to solve that, but implementing a mediator requires some skills and effort. The important note is: keep the mediators mostly and only take care of the communication part.*

8. Related Patterns

Facade Pattern

A simplified kind of mediator that only itself can trigger actions or make requests on passive colleague classes. The Facade is being call by some external classes, and actions cannot be called by or requests cannot be sent from the colleague objects any more. The protocol is unidirectional.

Adapter Pattern

Another kind of "mediator" that can change the messages and requests it receives and sends from and to the colleagues, whereas a normal mediator will not do so.

Observer Pattern

Observers can be used to get information, requests or messages from the colleagues to the mediators.

However, the main using purposes are different: An observer is used to enable notifications of events in one object to a number of other objects; meanwhile, a mediator is used to centralize the communication between set of related objects.

A good explanation from stackoverflow.com, answered May 27 at 23:41 by Trix

Observer

1. Without

Client1: Hey **Subject**, when do you change?

Client2: When did you change **Subject**? I have not noticed!

Client3: I know that **Subject** has changed.

2. With

Clients are silent.

Some moments later …

Subject: Dear clients, I have changed!

Mediator

1. Without

Client1: Hey **Taxi1**, take me somewhere.

Client2: Hey **Taxi1**, take me somewhere.

Client1: Hey **Taxi2**, take me somewhere.

Client2: Hey **Taxi2**, take me somewhere.

2. With

Client1: Hey **Taxi Center**, please take me a Taxi.

Client2: Hey **Taxi Center**, please take me a Taxi.

Visitor Pattern

Visitors aim at separating the implementation of some operations from the object structure they work on, so that changing the operations will not cause the need to modify much the classes.

If a visitor is used among different types of objects, it can loosely be considered as a "mediator", though mediators mainly work on similar types of objects (e.g. a group of derived classes from a base class).

References

"Design Patterns: Elements of Reusable Object-Oriented Software" by The Gang of Four

http://www.oodesign.com/mediator-pattern.html

https://sourcemaking.com/design_patterns/mediator

https://en.wikipedia.org/wiki/Visitor_pattern