Chapter 17- Mediator Design Pattern

Contents

17.1 Introduction **2**

17.2 Mediator Design Pattern **2**
17.2.1 Identification 2
17.2.2 Structure 2
17.2.3 Participants 2
17.2.4 Problem 3

17.3 Mediator pattern explained **3**
17.3.1 Purpose 3
17.3.2 Improvements achieved 3
17.3.3 Implementation issues 4
17.3.4 Related patterns 4

17.4 Example **4**

References **6**

17.1 Introduction

The mediator design pattern extends the observer pattern. Where the observer registers observers that get updated whenever the subject changes, the mediator registers colleagues that get updated whenever one of the other colleagues notifies the mediator of an update.

17.2 Mediator Design Pattern

17.2.1 Identification

Name	Classification	Strategy
Mediator	Behavioural	Delegation
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. ([1]:273)		

17.2.2 Structure

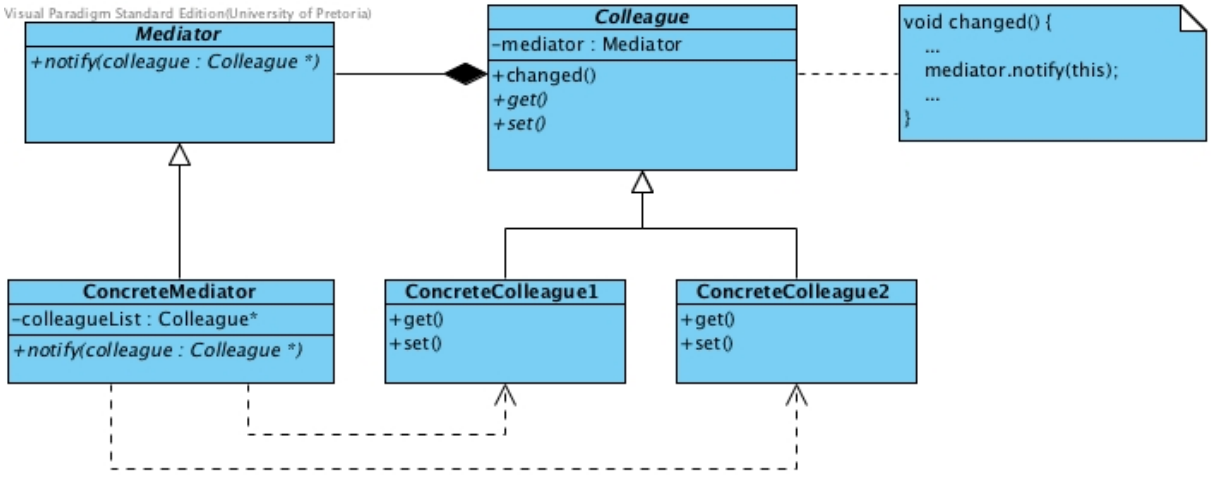

Figure 1: The structure of the Mediator Design Pattern

17.2.3 Participants

Mediator

- defines an interface for communicating with Colleague objects.

ConcreteMediator

- implements cooperative behavior by coordinating Colleague objects.
- knows and maintains its colleagues.

Colleague

- each Colleague class knows its Mediator object.
- each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

17.2.4 Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the spaghetti code phenomenon. When one wants to reuse only one or a few of the classes in a group of classes, it is virtually impossible to isolate them because they are to interconnected with one another. Trying to scoop a single serving results in an *all or nothing clump* [2].

17.3 Mediator pattern explained

17.3.1 Purpose

Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again. You can avoid this problem by capsulating the interconnections (i.e. the collective behavior) in a separate **mediator** object.

A mediator is responsible for controlling and coordinating the interactions of a group of objects.

17.3.2 Improvements achieved

Simplification of code updates
If the pattern is not applied and the behaviour of one of the classes in a group is changed it potentially necessitates the update of each class in the group to accommodate the changes made to this one element. The same applies when an element is added to the group or removed from the group. However, if the pattern is applied such changes will only require an update in the mediator class and none of the other classes in the group.

Increased reusability of code
The decoupling of the colleagues from one another increases their individual cohesiveness contributing to their reusability.

Simplification of object protocol
When refactoring into the mediator pattern a many-to-many relationship that exists between the elements in a group of objects is changed to a one-to-many relationship which is easier to understand and maintain.

17.3.3 Implementation issues

changed()

The `changed()` method is implemented in the colleague interface to allow each concrete colleague to call it. This method is used to notify the mediator of changes. It is the responsibility of each concrete colleague to call this method whenever it executes code that may impact on the other colleagues. Its implementation delegates to the mediator with a statement like:

```
mediator->notify(this);
```

notify()

The `notify()` method is called every time one of the concrete colleagues executes the `changed()` method. A pointer to the concrete colleague is sent as a parameter to allow the mediator to have knowledge about the originator of the notification. It is not desirable to send the content or nature of an update of a colleague to the mediator as a parameter of the `notify()` message. This is to insure that this interface is stable and generic enough to allow for different kinds of colleagues. It should get information about the nature and value of the changes that occurred and then propagate the change to all the concrete collegues. The following is pseudo code for the implementation of the `notify()` method:

```
Mediator::notify(originator: Collegue*)
{
    resultOfChange = originator->get();
    for (all colleagues)
    {
        set(resultOfChange);
    }
}
```

17.3.4 Related patterns

Facade

Facade differs from **Mediator** in that it abstracts a subsystem of objects to providea more convenient interface. Its protocol is unidirectional; that is, **Facade** objects make requests of the subsystem classes but not vice versa. In contrast, **Mediator** enables cooperative behaviour that colleague objects don't or can't provide, and the protocol is multidirectional.

Observer

Colleagues can communicate with the mediator using the **Observer** pattern.

17.4 Example

Figure 2 is a class diagram of a system illustrating the implementation of the Mediator design pattern. It is a simulation of the interaction between a number of widgets on a file dialog. It contains simple `cout` statements in the function bodies of most functions

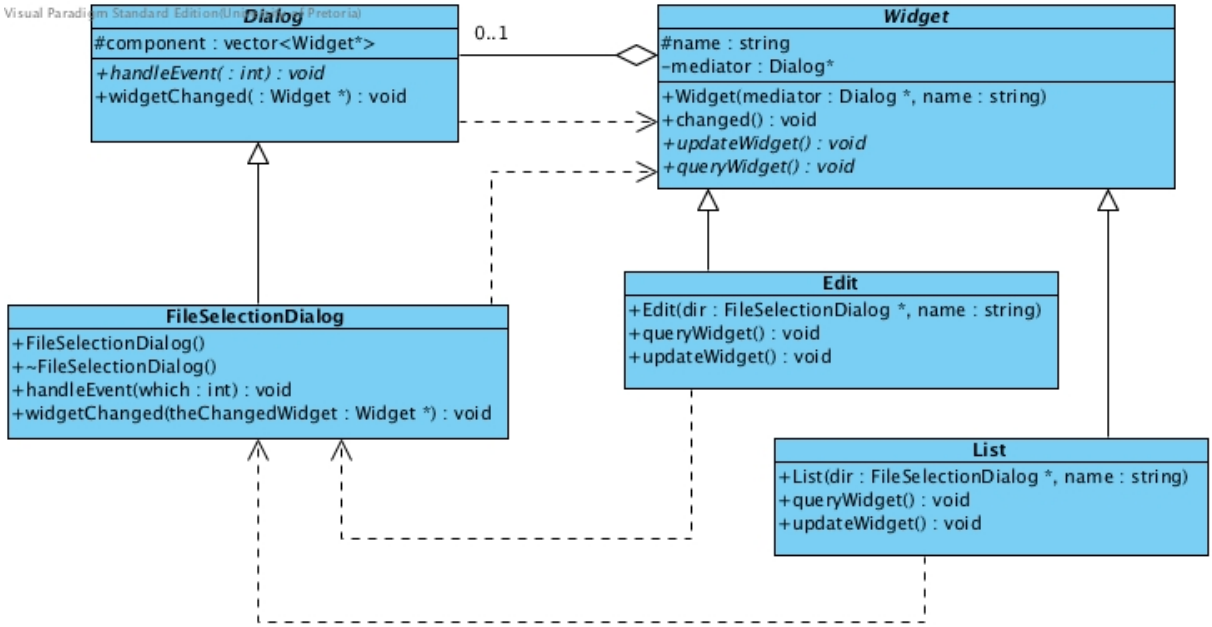

Figure 2: Class Diagram of a partial implementation of a file selection dialog

to be able to observe how the pattern operates. This example was adapted from [2]. The following table summarises how the implementation relates to the participants of this pattern:

Participant	**Entity in application**
Mediator	Dialog
Concrete Mediator	FileSelectionDialog
Colleague	Widget
Concrete Colleague	List, Edit
changed()	changed()
notify()	widgetChanged(: Widget)
get()	queryWidget()
set()	updateWidget()

- In this example, `FileSelectionDialog` is the designated the mediator for all the `Widget` siblings. The `FileSelectionDialog` does not know, or care, who the siblings of `Widget` are.

- Whenever a simulated user interaction occurs in a child of `Widget`, `Widget::changed()` signals it. This method does nothing except to *delegate* that event to the appropriate `Dialog` with the function call `mediator->widgetChanged(this)`. Note that the correct concrete mediator in the `Dialog` hierarchy will be called upon, because `mediator` is a variable which as been assigned the value of the appropriate mediator, when the child `Widget` was constructed. Also note that it passes a pointer to itself to the mediator so that the mediator can know where the change originated.

- `FileSelectionDialog::widgetChanged()` encapsulates all collective behaviour for the dialog box. It serves as the hub of communication. In this example it simply

queries the status form the `Widget` that signalled the change, and propagates the change to all its dependants.

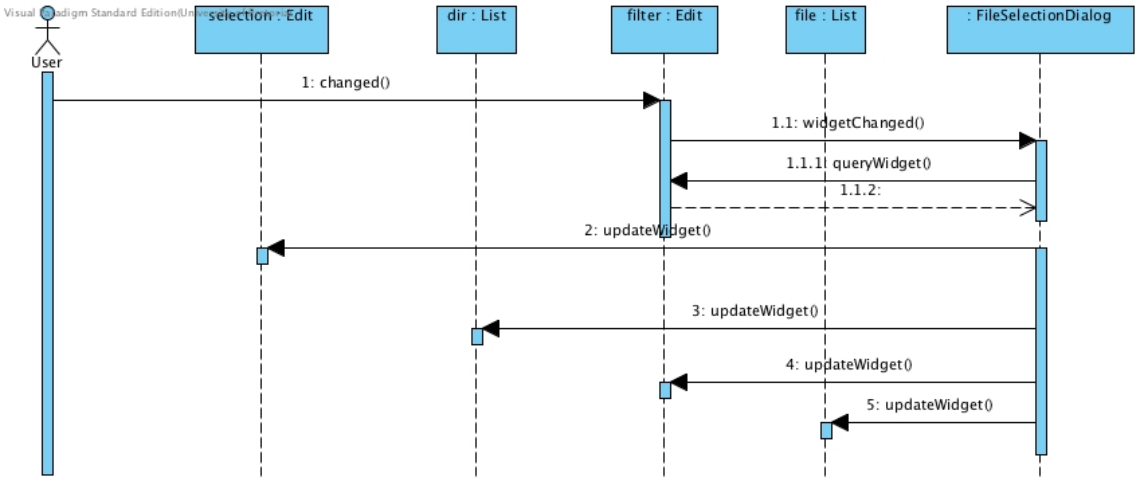

Figure 3: A Sequence Diagram to visualise the time ordering

Figure 3 illustrates how the mediator will govern the interaction between the widgets from the moment the user changes the content of `filter:Edit` until all the `Widget` objects are updated. Note that the sequence diagram shows only instantiated objects. Abstract classes such as `Dialog` and `Widget` are not part of a sequence diagram.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1994.

[2] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].