Refining the Observer Pattern: The Middle Observer Pattern

Pablo Iaría - iariap@sol.info.unlp.edu.ar
Ulises Chesini - uliche@sol.info.unlp.edu.ar

Abstract

The refinement presented in this paper incorporates the possibility of decoupling the common behavior from the observers avoiding data redundancy in such a way we can maintain consistency among them without the object that is being observed knows that feature.

Primary Contact: Pablo Iaría
50 y 115
LIFIA – Dto. de informática,
Facultad de Ciencias Exactas
La Plata (1900)
Phone +54 –21-228252-4

Refining the Observer Pattern: The Middle Observer Pattern

Pablo Iaría
Ulises Chesini

LIFIA – Dto. de informática,
Facultad de Ciencias Exactas, UNLP
[iariap,uliche]@sol.info.unlp.edu.ar

MIDDLE OBSERVER	Object Behavioral

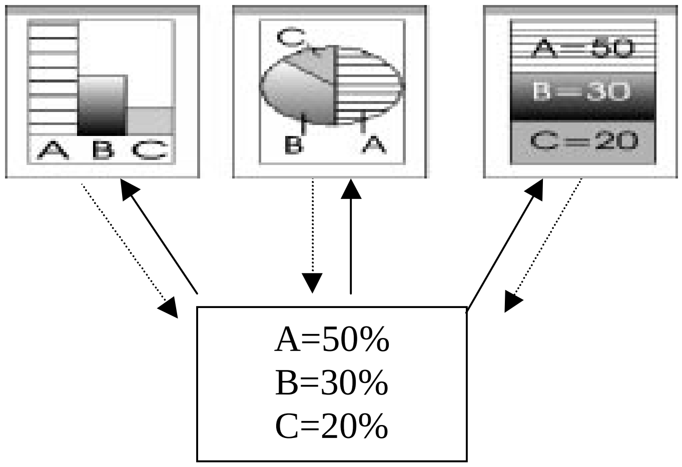
Intent

The same as its base pattern, the Observer Pattern, the Middle Observer pattern defines a dependency one to many between objects so that, when an object changes state, all its dependents are notified and updated automatically.

The refinement presented in this paper incorporates the possibility of decoupling the common behavior from the observers avoiding data redundancy in such a way we can maintain consistency among them without the object that is being observed knows that feature.

Motivation

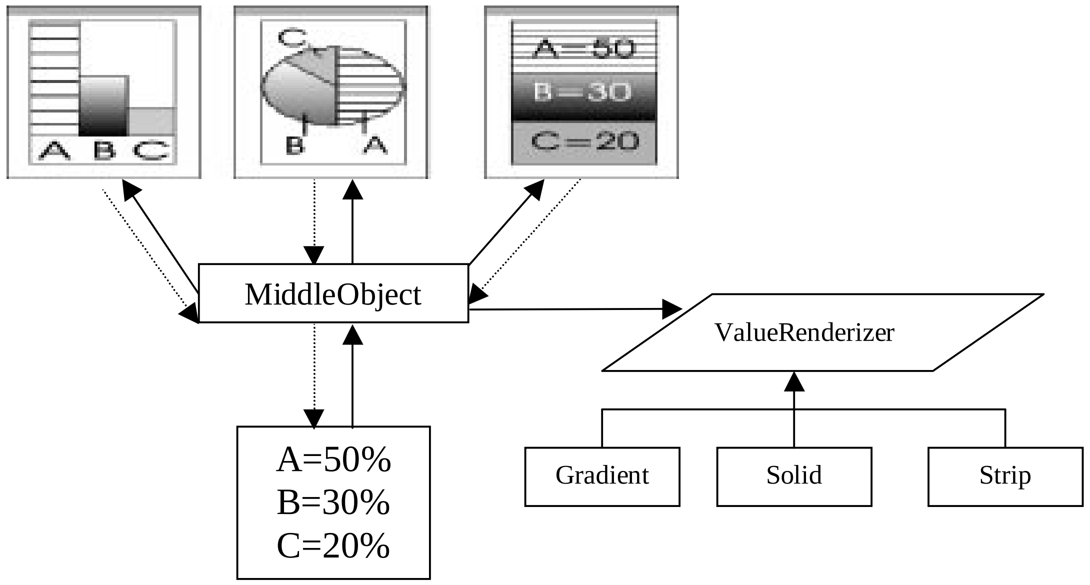
As an introduction we can consider the example presented for the Observer Pattern in [Gamma+95], where there are three different interfaces (observers) that show data of an object (subject) like a spreadsheet, bar and pie graphics. The different interfaces do not know each other, therefore each one can be used separately. Those interfaces, however**,** behave as if they were known**.** When the user changes the information in one of the interfaces, the others reflect the change instantly. This behavior implies that the changes produced in the model should be reflected in the interfaces.



Let us suppose now that, in addition to the information of the object, we wish to maintain another kind of information. For example, if we wish to have the values represented graphically in the interfaces with different kinds of backgrounds (solid, gradient, stripped, etc.), the same background for each value in all the interfaces and, furthermore, if we wish they could be configured from anyone of them, then, who is responsible for maintaining the information that assigns a background to a determined representation of each value? The observers or the subject?

If we put this information into the observers, the most harmful consequence would be a data redundancy that would grow in proportional way to the quantity of additional information (backgrounds) and the quantity of interfaces. On the other hand, if this information was maintained into the object of the application, we would be provoking a non desirable coupling between the interface and the model, since it is not responsibility of the model to keep referring information to the interfaces.

The solution proposed in this case consists of adding an object between the observers and the subject putting this new object as observer of the model and as subject of the interfaces. In this way, we have an object with the responsibility of maintaining all the information that it does not have to be neither in the interfaces nor in the model. In the previous example, this object has the responsibility of maintaining the information of the kind of background that corresponds to each value.

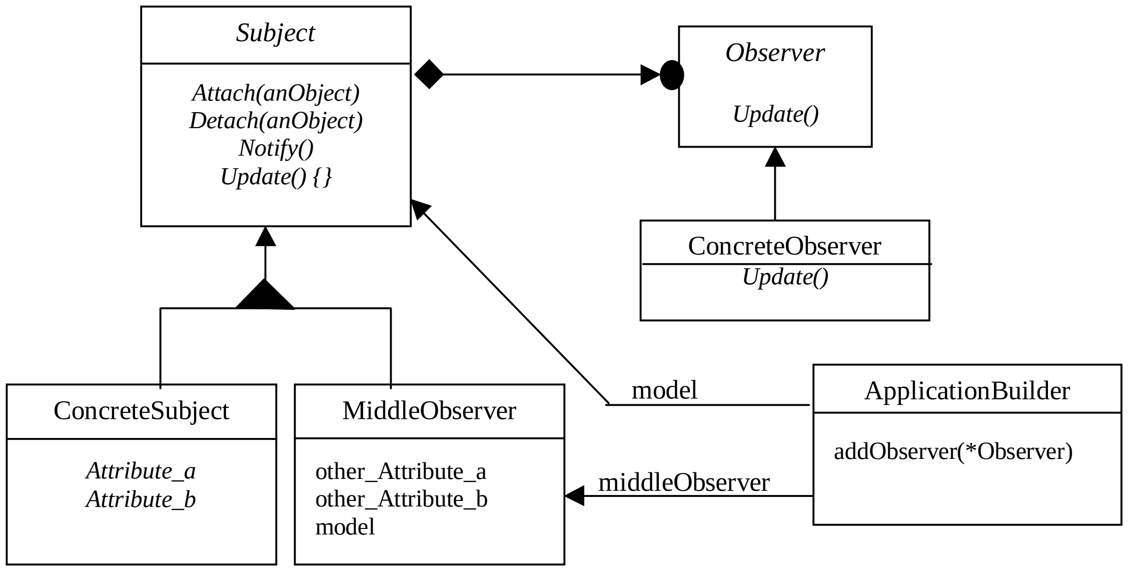


Applicability

Use the Middle Observer pattern when:

- You have many subjects with plenty of redundant information.
- You need to maintain consistency among the observers with information that belongs to them but not to the subject.
- You have information or behavior referring to the subject, but it does not have to be maintained either by the subject (because isn't its responsibility) or the observers (because it produce information/behavior redundancy).
- It also can be used in the same situations as its base pattern, the Observer Pattern.

Structure



Participants

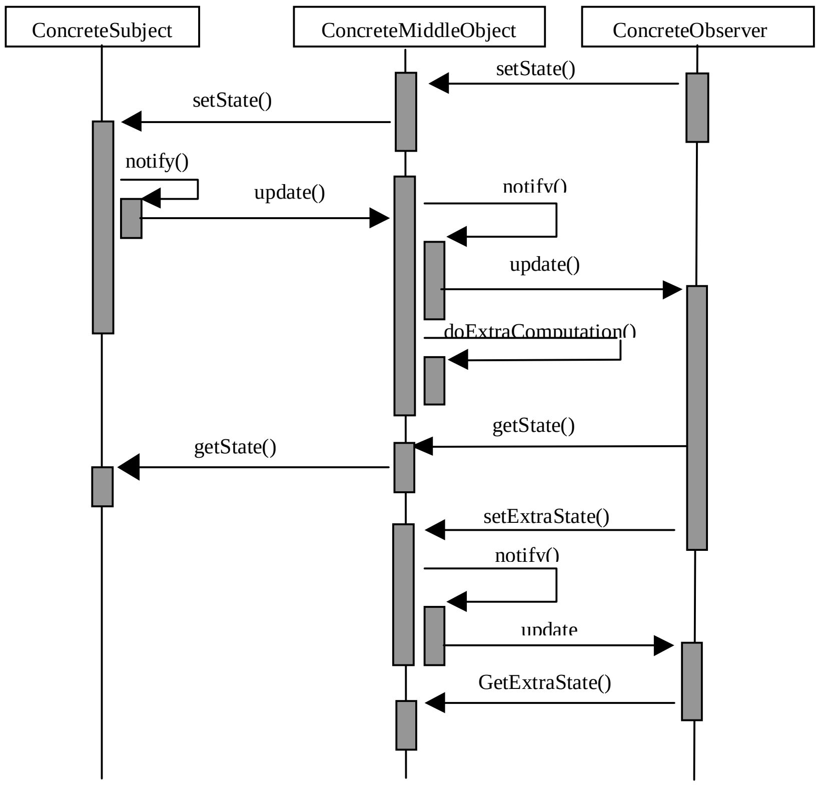
- Subject: Knows it's MiddleObservers and notifies the changes.
- ConcreteSuject: Broadcasts the notification request to MiddleObserver objects.
- MiddleObserver: Keeps responsibilities that neither belong to the Subject nor the Observer (Area filling algorithm for the values)
- Observer: reflects the subject changes that MiddleObserver objects request.
- ConcreteObserver: Implements the notify method that is sent by MiddleObserver objects.
- ApplicationBuilder: builds the relationships between the model, the middleObject and the observers. This means that sets the observer's model to middleObject and the middleObject's model to the concrete subject.

Collaborations

When an observer changes a middleObserver's aspect, two things can happen:

- The aspect being changed is a subject's aspect. In this case the middleObserver delegates the message to the subject, and the subject is who broadcasts the #changed request. Furthermore the middleObserver can intercept others requests from the subjects to do some internal updates.
- The aspect being changed is a middleObserver's aspect. Here, the middleObserver is who triggers the update request and the subject does not perceive this situation.

The following picture shows the interaction beetwen the ConcreteSubject, ConcreteMiddleObserver and ConcreteObservers.



Consequences

The advantages of the Middle Observer Pattern are:

1. It allows data and behavior consistency among the observers without the model intervention. When a subject causes a middleObserver's aspect changes state, then is the middleObserver who triggers the update request, and the model does not perceive it.

2. It simplifies the observer's knowledge relationship and avoids maintaining observer-related information to be explicit in the subject.

3. You may develop the objects separately, to minimize coupling (what they know about each other) and increases reuse (using one without the other).

4. It keeps the same advantages of the Observer Pattern.

The Middle Observer Pattern's main disadvantage is:

It must be implemented the entire subject's messages in the MiddleObserver class; this produce plenty of forwarding messages that just delegate the same request to the model and few of them could be useful for a MiddleObserver object.
Languages like Smalltalk have the '#doesNotUnderstood:' message that allows us to implement only the methods that are necessary for the MiddleObserver, and to delegate the rest of them to the subjects directly.

Implementation

The Observer-MiddleObserver and MiddleObserver-Subject relationships are maintained using the Observer Pattern twice. Firstly, we have used the Observer Pattern to create a dependency one-to-many between the Observers and the MiddleObserver. In this way, when a middleObserver's aspect changes state, the Observers are notified and they can make the MiddleObserver necessary request, as if it was their model.
The second use is between the Subject and the MiddleObserver. In this case, when a Subject's aspect changes state, it notifies to all the Observers with the well-defined protocol notify-update careless of any MiddleObserver object presence.

We could use the MiddleObserver as the Facade Pattern [Gamma+95] incorporating the possibility of having more than one subject without the observer have to know them. In this way, the observers make the entire request to the MiddleObserver object.

The ChangeManager (implementation section of the Observer Pattern, item 8) only provides updating functionality, on the other hand, the MiddleObserver pattern, provides updating and behavioral functionality (redundancy data of the subjects).

Sample Code

```
class Observer
{
public:
       virtual ~Observer();
       virtual void Update(Subject* theChangedSubject) = 0;
protected:
       Observer();
}

class Subject
{
public:
       virtual ~Subject();
       virtual void Attach(Observer*);
       virtual void Detach(Observer*);
       virtual void Notify();
protected:
       Subject():
Private:
       List<Observer*> *_observers;
}

class MiddleObserver::public Subject
{
  public:
  virtual void Update() {this.Notiy()}
}

class ConcreteMiddleObserver :: public MiddleObserver
{ public:
       virtual void Update();
       someType extraAttribute;
}

void ConcreteMiddleObserver :: update()
{
  this->doSomethingElse(); // do the extra tasks
  this->Notify();
}

void Subject :: Attach(Observer* obj)
{ _observers->Append(obj); }


void Subject :: Detach(Observer* obj)
{ _observers->Remove(obj); }

void Subject::Notify()
{ ListIterator<Observer*> i(_observers);
  for(i.first();!i.isDone();i.next())
  { i.currentItem()->Update(this) }}
```

Related Patterns

1. Decorator Pattern: in the way we had presented this pattern, we could see the middleObserver as a subject's decorator, which has the responsibility of maintain consistency and extra data among the observers by a notify-request protocol.

2. Proxy Pattern: The MiddleObserver can act as a proxy, because the observers do not interact directly with the subject and it can hide the way the subject is acceded.

3. Mediator Pattern: The idea of having an object in the middle of an application that intercepts all the update requests and broadcasts them to other objects doing some extra tasks where is needed and encapsulates how they interacts, make us think in a Mediator Pattern.

4. Singleton Pattern: Since there are only one ApplicationBuilder and MiddleObserver instance associated with the same model, this can be implemented with a Singleton Pattern.

5. Chain of responsibility: the MiddleObserver objects delegate to the subject all the responsibility that no belongs to them. In addition, the middleObserver has the possibility of handle some requests from the observers.

Known uses

Some examples of use are:

- [00-Navigator+96]
  "A Framework for Extending Object-Oriented Applications with Hypermedia Functionality", by Alejandra Garrido and Gustavo Rossi.
  The new review of Hypermedia and Multimedia. Applications and research. Taylor Graham. Volume 2, 1996.
  This paper explains the whole idea and architecture of the object-oriented framework providing hypermedia functionality. When an application's object (Subject) changes state, it broadcast a message to all its nodes (ConcreteMiddleObserver with the responsibility that is common to all the views but is not to be known by the Subject) and updates the views (Observers). Each view shows some aspects of a node and therefore a node may have different views.
  The framework global structure is the same of the MiddleObserver Pattern.

References

[GHJV95] Erich Gamma, Richard Helm Ralph Johnson and John Vlisedes.
Design Patterns elements of Reusable Object-Oriented Software. 1995

OO-Framework Navigator -Alejandra Garrido's degree thesis directed by Dr.Gustavo H. Rossi.