Observer Pattern

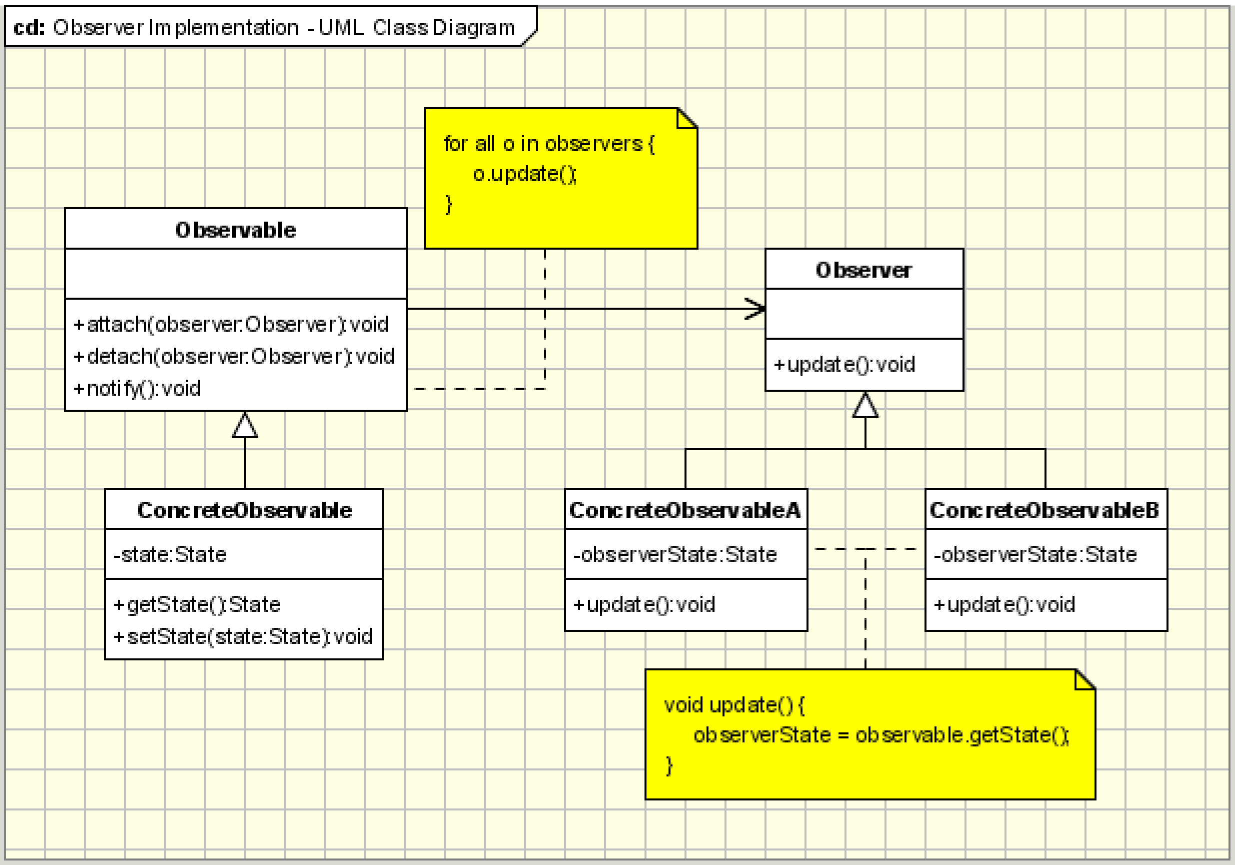
Motivation

We can not talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.

Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stocks server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server.

Intent

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Implementation

The participants classes in this pattern are:

- **Observable** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as **Subject**.
- **ConcreteObservable** - concrete Observable class. It maintain the state of the object and when a change in the state occurs it notifies the attached**Observers**.
- **Observer** - interface or abstract class defining the operations to be used to notify this object.
- **ConcreteObserverA, ConcreteObserver2** - concrete **Observer** implementations.

The flow is simple: the main framework instantiate the ConcreteObservable object. Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface. Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface. When a new Observer is added to the application, all we need to do is to instantiate it in the main framework and to add attach it to the Observable object. The classes already created will remain unchanged.

Applicability & Examples

The observer pattern is used when:

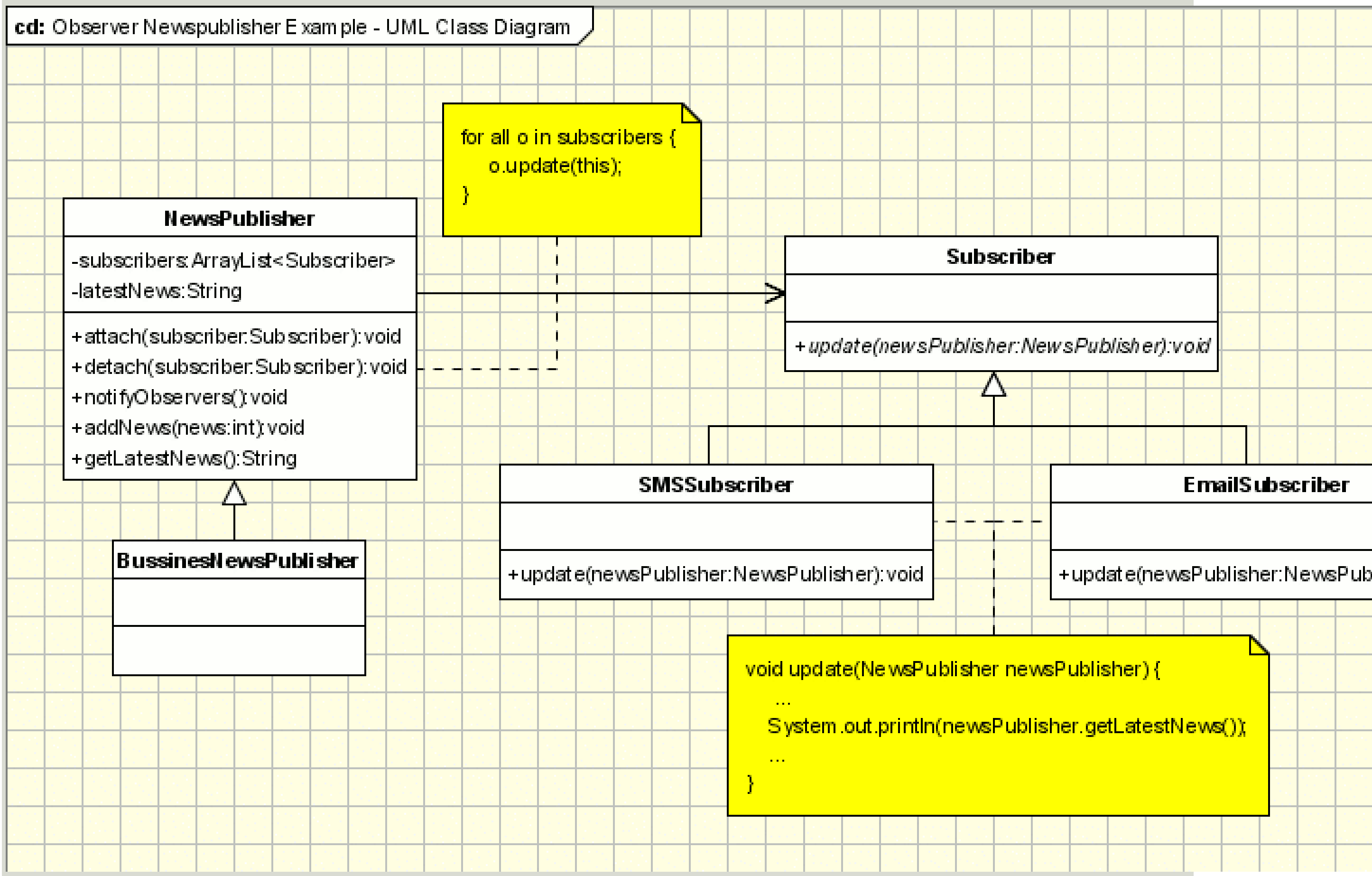
- the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- the framework we are writing needs to be enhanced in future with new observers with minimal changes.

Some Classical Examples:

- **Model View Controller Pattern** - The observer pattern is used in the model view controller (MVC) architectural pattern. In MVC the this pattern is used to decouple the model from the view. View represents the **Observer** and the model is the **Observable** object.
- **Event management** - This is one of the domains where the Observer patterns is extensively used. Swing and .Net are extensively using the Observer pattern for implementing the events mechanism.

Example - News Agency

Lets' take the example of a news agency. A news agency gather news news and publish them to different subscribers. We need to create a framework for and agency to be able to inform immediately, when event occurs, its subscribers about the event. The subscribers can receive the news in different ways: Emails, SMS, ... The solution need to be extensively enough to support new types of subscribers(maybe new communication technologies will appear).


cd: Observer Newspublisher Example - UML Class Diagram
NewsPublisher
-subscribers:ArrayList<Subscriber>
-latestNews:String
+attach(subscriber:Subscriber):void
+detach(subscriber:Subscriber):void
+notifyObservers():void
+addNews(news:int):void
+getLatestNews():String
BussinesNewsPublisher
for all o in subscribers { o.update(this); }
Subscriber
+update(newsPublisher:NewsPublisher):void
SMSSubscriber
+update(newsPublisher:NewsPublisher):void
EmailSubscriber
+update(newsPublisher:NewsPub
void update(NewsPublisher newsPublisher) { ... System.out.println(newsPublisher.getLatestNews()); ... }

Obviously, the agency is represented by an Observable(Subject) class named **NewsPublisher**. This one is created as an abstract class because the agency want to create several types of Observable objects: in the beginning only for business news, but after some time sport and political new will be published. The concrete class is **BusinessNewsPublisher**.

The observer logic is implemented in **NewsPublisher**. It keeps a list of all it subscribers and it informs them about the latest news. The subscribers are represented by some observers (SMSSubscriber, EmailSubscriber). Both the observers mentioned above are inherited from the Subscriber. The subscriber is the abstract class which is known to the publisher. The publisher doesn't know about concrete observers, it knows only about their abstraction.

In the main class a publisher(Observable) is built and a few subscribers(Observers). The subscribers are subscribed to the publisher and they can be unsubscribed. In this architecture new types of subscribers can be easily added(instant messaging, ...) and new types of publishers(Weather News, Sport News, ...).

Specific Implementation Problems

Many subjects to Many observers

It's not a common situation but there are cases when a there are many observers that need to observe more than one subject. In this case the observer need to be notified not only about the change, but also which is the subject with the state changed. This can be realized very simple by adding to the subjects reference in the update notification method. The subject will pass a reference to itself(this) to the when notify the observer.

Who triggers the update?

The communication between the subject and its observers is done through the notify method declared in observer interface. But who it cat be triggered from either subject or observer object. Usually the notify method is triggered by the subject when it's state is changed. But sometimes when the updates are frequent the consecutive changes in the subject will determine many unnecessary refresh operations in the observer. In order to make this process more efficient the observer can be made responsible for starting the notify operation when it consider necessary.

Making sure Subject state is self-consistent before notification

The subject state should be consistent when the notify operation is triggered. If changes are made in the subject state after the observer is notified, it will will be refreshed with an old state. This seems hard to achieve but in practice this can be easily done when Subject subclass operations call inherited operations. In the following example, the observer is notified when the subject is in an inconsistent state:

```
class Observable{
        ...
        int state = 0;
        int additionalState = 0;
        public updateState(int increment)
        {
                state = state + increment;
                notifyObservers();
        }
        ...
}

class ConcreteObservable extends Observable{
        ...
        public updateState(int increment){
                super.updateState(increment); // the observers are notified
                additionalState = additionalState + increment; // the state is changed afte
        }
        ...
}
```

This pitfall can be avoided using template methods in the abstract subject superclass for calling the notify operations. Then subject subclass will implement the operations(s) of the template:

```
class Observable{
        ...
        int state = 0;
        int additionalState = 0;
        public void final updateState(int increment)
        {
                doUpdateState(increment);
                notifyObservers();
        }
        public void doUpdateState(int increment)
        {
                state = state + increment;
        }
        ...
}

class ConcreteObservable extends Observable{
        ...
        public doUpdateState(int increment){
                super.doUpdateState(increment); // the observers are notified
                additionalState = additionalState + increment; // the state is
                                        changed after the notifiers are updated
        }
        ...
}
```

The Operations defined in the subject base class which triggers notify operation should be documented.

Push and pull communication methods

There are 2 methods of passing the data from the subject to the observer when the state is being changed in the subject side:

- **Push model** - The subjects send detailed information about the change to the observer whether it uses it or not. Because the subject needs to send the detailed information to the observer this might be inefficient when a large amount of data needs to be sent and it is not used. Another aproach would be to send only the information required by the observer. In this case the subject should be able to distinguish between different types of observers and to know the required data of each of them, meaning that the subject layer is more coupled to observer layer.
- **Pull model** - The subject just notifies the observers when a change in his state appears and it's the responsibility of each observer to pull the required data from the subject. This can be inefficient because the communication is done in 2 steps and problems might appear in

multithreading environments.

Specifying points of interests

The efficiency can be improved by specifying which are the events on which each observer is interested. This can be realized by adding a new class defining an aspect. When an observer is registering it will provide the aspects in which it is interested:

```
class Subject{
        ...
        void attach(Observer observer, Aspect interest);
        ...
}
```

Encapsulating complex update semantics

When we have several subjects and observers the relations between them we'll become more complex. First of all are have a many to many relation, more difficult to manage directly. Second of all the relation between subjects and observers can contain some logic. Maybe we want to have an observer notified only when all the subjects will change their states. In this case we should introduce another object responsible (called ChangeManager) for the following actions:

- to maintain the many to many relations between the subjects and their observers.
- to encapsulate the logic of notify the observers.
- to receive the notifications from subjects and delegate them to the observers(based on the logic it encapsulate)

Basically the Change Manager is an observer because if gets notified of the changes of the subject and in the same time is an subject because it notify the observers. The ChangeManager is an implemenation of the Mediator pattern.

The Observer pattern is usually used in combination with other design patterns:

- **Factory pattern** - It's very likely to use the factory pattern to create the Observers so no changes will be required even in the main framework. The new observers can be added directly in the configuration files.
- **Template Method** - The observer pattern can be used in conjunction with the Template Method Pattern to make sure that Subject state is self-consistent before notification
- **Mediator Pattern** - The mediator pattern can be used when we have cases of complex cases of many subjects an many observers