**Observer Pattern**

**Motivation**

It’s not possible to talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction.

Cases when certain objects need to be informed about the changes occurred 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.

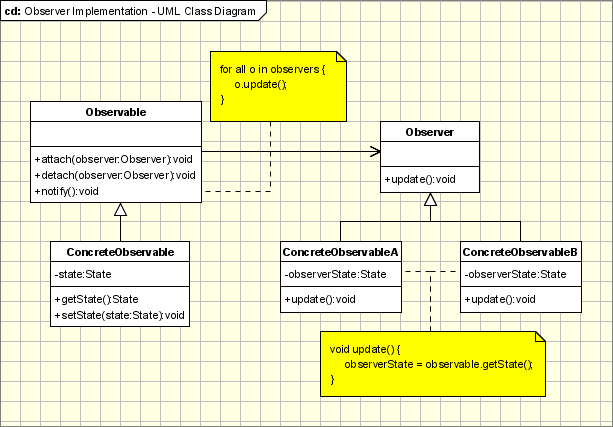
Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with SMS alerts.

Now it's simple to see purpose of the observer pattern: it separates the subject(stocks server) from its 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 of this pattern are:

* **Observable** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. This class/interface is also known as **Subject**.
* **ConcreteObservable** - concrete Observable class. It maintains 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 instantiates and attaches the concrete observers to it using the methods defined in the **Observable** interface.

Each time the state of the **subject** changes, 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 being created 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 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 pattern is extensively used. **Swing** and .**Net** are extensively using the Observer pattern for implementing the events mechanism.

<?php

**interface** IObserver

{

**function** onChanged( $sender, $args );

}

**interface** IObservable

{

**function** addObserver( $observer );

}

**class** UserList **implements** IObservable

{

**private** $\_observers = **array**();

**public function** addCustomer( $name )

{

**foreach**( $this->\_observers **as** $obs )

$obs->onChanged( $this, $name );

}

**public function** addObserver( $observer )

{

$this->\_observers []= $observer;

}

}

**class** UserListLogger **implements** IObserver

{

**public function** onChanged( $sender, $args )

{

**echo**( "'$args' added to user list\n" );

}

}

$ul = **new** UserList();

$ul->addObserver( **new** UserListLogger() );

$ul->addCustomer( "Jack" );

$ul->addCustomer( "Ben" );

?>

**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 needs to be notified not only about the change, but also which is the subject with the state changed.

This can be realized very simply by adding to the subject’s 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 how it can be triggered from either subject or observer object. Usually the notify method is triggered by the subject when its 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 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.

* **Push and Pull communication methods**

**Pull Model**

The observer is notified that a change has occurred and must find out itself what changes have occurred.

**Push Model**

The subject sends observers detailed information about the change that has occurred (in the simplest case, the entire new state itself).

The Pull model is simple, but leads to further requests from the observer to the subject.

The dumb Push model is simplest, but may be inefficient; trying to make it smarter on a per-observer basis increases subject observer coupling.

Extending the subject registration interface is also possible.

* **Encapsulating complex update semantics**

When there are several subjects and observers the relations between them will become more complex.

First of all having a many to many relation, makes it more difficult to manage directly.

Secondly 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 **ChangeManager** is an observer because it gets notified of the changes of the subject and at the same time it is a subject because it notifies the observers.

The **ChangeManager** is an implementation 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 that 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 and many observers.

**Research Work**

***Investigate about the Mediator Pattern and explore how it can be used in conjunction with the Observer pattern***