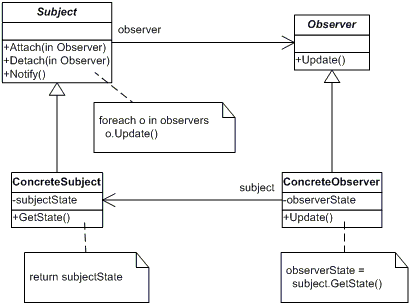
**Section18 Observer Design Pattern**

**Notes: -**

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



**Example: -**

**In this example we apply Observer Design Pattern by make ConcreteSubject which contains the IEnumerable of Concrete Observer to register it and then call them only on one direction means that it will notify the all observers not target one**

**namespace ObserverPatternSol.Observers{**

**abstract class Observer{**

**public abstract void Update();}}**

**using ObserverPatternSol.Observables;**

**using System;**

**namespace ObserverPatternSol.Observers{**

**//the concrete observer contains the ConcreteSubject which is the same instnace the register it //from ConcreteSubject Attach**

**class ConcreteObserver : Observer{**

**private string \_name;**

**private string \_observerState;**

**private ConcreteSubject \_subject;**

**public ConcreteObserver(ConcreteSubject subject, string name){**

**this.\_subject = subject;**

**this.\_name = name;}**

**//it will called through the observable instance which is ConcreteSubject**

**public override void Update(){**

**\_observerState = \_subject.SubjectState;**

**Console.WriteLine("Observer {0}'s new state is {1}", \_name, \_observerState);}**

**// Gets or sets subject**

**public ConcreteSubject Subject{get { return \_subject; }set { \_subject = value; }}}}**

**using ObserverPatternSol.Observers;**

**using System.Collections.Generic;**

**namespace ObserverPatternSol.Observables{**

**//this Subject class will register all the observers instance that each one refer to the same Subject instance**

**abstract class Subject{**

**private List<Observer> \_observers = new List<Observer>();**

**public void Attach(Observer observer){\_observers.Add(observer);}**

**public void Detach(Observer observer){\_observers.Remove(observer);}**

**//it will notify the Observer that something coming and not return as Mediator design pattern**

**public void Notify(){**

**foreach (Observer o in \_observers){o.Update();}}}}**

**namespace ObserverPatternSol.Observables{**

**class ConcreteSubject : Subject{**

**private string \_subjectState;**

**// Gets or sets subject state**

**public string SubjectState{get { return \_subjectState; }set { \_subjectState = value; }}}}**

**//on the Main we declare Subject instance and register multiple observers and then call the notify //which call all the observer assign to**

**using ObserverPatternSol.Observers;**

**using ObserverPatternSol.Observables;**

**using System;**

**namespace ObserverPatternSol{**

**class Program{**

**static void Main(string[] args){**

**// Configure Observer pattern**

**ConcreteSubject s = new ConcreteSubject();**

**//register observer instances on the same ConcreteSubject (many to one relationship)**

**s.Attach(new ConcreteObserver(s, "X"));**

**s.Attach(new ConcreteObserver(s, "Y"));**

**s.Attach(new ConcreteObserver(s, "Z"));**

**// Change subject and notify observers**

**s.SubjectState = "ABC";**

**s.Notify();**

**Console.ReadKey();}}}**

**Example: -**

**//the Stock class contains the List<IInvestor> which is the observer that used to call as //broadcast when register list of observers as below**

**using DoFactoryObserverPattern2Pro.Observers;**

**using System;**

**using System.Collections.Generic;**

**namespace DoFactoryObserverPattern2Pro.Observables{**

**abstract class Stock{**

**private string \_symbol;**

**private double \_price;**

**private List<IInvestor> \_investors = new List<IInvestor>();**

**public Stock(string symbol, double price){**

**this.\_symbol = symbol;**

**this.\_price = price;}**

**public void Attach(IInvestor investor){\_investors.Add(investor);}**

**public void Detach(IInvestor investor){\_investors.Remove(investor);}**

**public void Notify(){**

**foreach (IInvestor investor in \_investors){investor.Update(this);}**

**Console.WriteLine("");}**

**public double Price{**

**get { return \_price; }**

**set{**

**if (\_price != value){\_price = value;**

**Notify();}}}**

**public string Symbol{get { return \_symbol; }}}}**

**namespace DoFactoryObserverPattern2Pro.Observables{**

**class IBM : Stock{**

**public IBM(string symbol, double price) : base(symbol, price) {}}}**

**using DoFactoryObserverPattern2Pro.Observables;**

**namespace DoFactoryObserverPattern2Pro.Observers{**

**interface IInvestor{void Update(Stock stock);}}**

**using DoFactoryObserverPattern2Pro.Observables;**

**using System;**

**namespace DoFactoryObserverPattern2Pro.Observers{**

**class Investor : IInvestor{**

**private string \_name;**

**private Stock \_stock;**

**public Investor(string name){this.\_name = name;}**

**public void Update(Stock stock){**

**Console.WriteLine("Notified {0} of {1}'s " +**

**"change to {2:C}", \_name, stock.Symbol, stock.Price);}**

**// Gets or sets the stock**

**public Stock Stock{get { return \_stock; }set { \_stock = value; }}}}**

**//Main Page**

**using DoFactoryObserverPattern2Pro.Observables;**

**using DoFactoryObserverPattern2Pro.Observers;**

**using System;**

**namespace DoFactoryObserverPattern2Pro{**

**class Program{**

**static void Main(string[] args){**

**// Create IBM stock and attach investors**

**IBM ibm = new IBM("IBM", 120.00);**

**ibm.Attach(new Investor("Sorros"));**

**ibm.Attach(new Investor("Berkshire"));**

**// Fluctuating prices will notify investors broadcasting to all observers that created so it will show 8 message on console => 4 \* 2 observer = 8 messages**

**ibm.Price = 120.10;**

**ibm.Price = 121.00;**

**ibm.Price = 120.50;**

**ibm.Price = 120.75;**

**Console.ReadKey();}}}**