(/)

Login (/login.aspx) Join Now (/join.aspx)

(/net/design-patterns) back to .NET Design Patterns (/net/design-patterns)

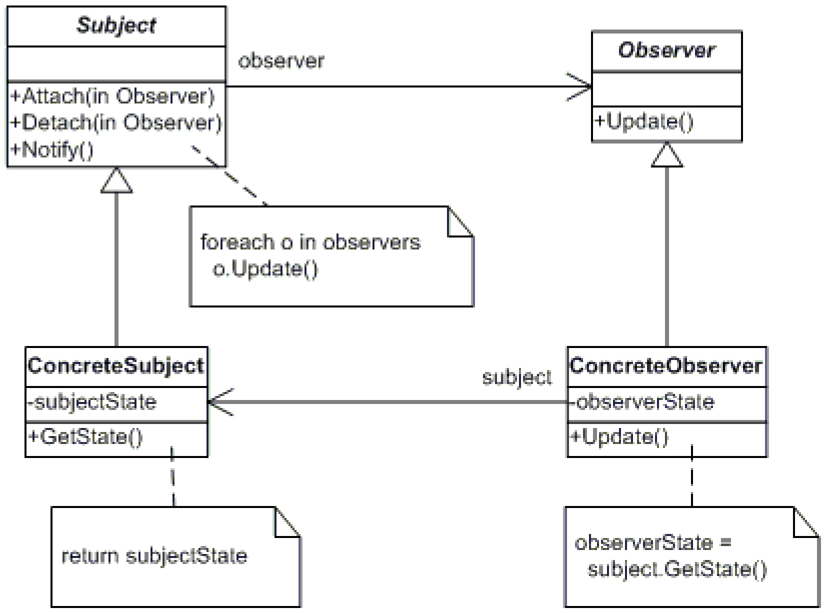
# Observer

## Definition

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

Frequency of use: 1 2 3 4 5 🟥🟥🟥🟥🟥 High

## UML class diagram



## Participants

The classes and objects participating in this pattern are:

- **Subject (Stock)**
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
- **ConcreteSubject (IBM)**
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
- **Observer (IInvestor)**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteObserver (Investor)**
  - maintains a reference to a ConcreteSubject object

- stores state that should stay consistent with the subject's
- implements the Observer updating interface to keep its state consistent with the subject's

## Structural code in C#

This structural code demonstrates the Observer pattern in which registered objects are notified of and updated with a state change.

```


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Observer.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Observer Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Configure Observer pattern
      ConcreteSubject s = new ConcreteSubject();

      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();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Subject' abstract class
  /// </summary>
  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);
      }

      public void Notify()
      {
        foreach (Observer o in _observers)
        {
          o.Update();
        }
      }
    }

    /// <summary>
    /// The 'ConcreteSubject' class
    /// </summary>
    class ConcreteSubject : Subject
    {
      private string _subjectState;

      // Gets or sets subject state
      public string SubjectState
      {
        get { return _subjectState; }
        set { _subjectState = value; }
      }
    }

    /// <summary>
    /// The 'Observer' abstract class
    /// </summary>
    abstract class Observer
    {
      public abstract void Update();
    }

    /// <summary>
    /// The 'ConcreteObserver' class
    /// </summary>
    class ConcreteObserver : Observer
    {
      private string _name;
      private string _observerState;
      private ConcreteSubject _subject;

      // Constructor
      public ConcreteObserver(
        ConcreteSubject subject, string name)
      {
        this._subject = subject;
        this._name = name;
      }

      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; }
      }
    }
  }




```

Output

```
Observer X's new state is ABC
Observer Y's new state is ABC
Observer Z's new state is ABC
```

## Real-world code in C#

This real-world code demonstrates the Observer pattern in which registered investors are notified every time a stock changes value.

```


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Observer.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Observer Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // 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
      ibm.Price = 120.10;
      ibm.Price = 121.00;
      ibm.Price = 120.50;
      ibm.Price = 120.75;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Subject' abstract class
  /// </summary>
  abstract class Stock
  {
    private string _symbol;
    private double _price;
    private List<IInvestor> _investors = new List<IInvestor>();

    // Constructor
    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("");
    }

    // Gets or sets the price
    public double Price
    {
      get { return _price; }
      set
      {
        if (_price != value)
        {
          _price = value;
          Notify();
        }
      }
    }

    // Gets the symbol
    public string Symbol
    {
      get { return _symbol; }
    }
  }

  /// <summary>
  /// The 'ConcreteSubject' class
  /// </summary>
  class IBM : Stock
  {
    // Constructor
    public IBM(string symbol, double price)
      : base(symbol, price)
    {
    }
  }

  /// <summary>
  /// The 'Observer' interface
  /// </summary>
  interface IInvestor
  {
    void Update(Stock stock);
  }

  /// <summary>
  /// The 'ConcreteObserver' class
  /// </summary>
  class Investor : IInvestor
  {
    private string _name;
    private Stock _stock;

    // Constructor
    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; }
    }
  }
}



```

Output

```
Notified Sorros of IBM's change to $120.10
Notified Berkshire of IBM's change to $120.10

Notified Sorros of IBM's change to $121.00
Notified Berkshire of IBM's change to $121.00

Notified Sorros of IBM's change to $120.50
Notified Berkshire of IBM's change to $120.50

Notified Sorros of IBM's change to $120.75
Notified Berkshire of IBM's change to $120.75
```

## .NET Optimized code in C#

The .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc. You can find an example on our Singleton (/net/singleton-design-pattern#net) pattern page.

All other patterns (and much more) are available in our **.NET Design Pattern Framework 4.5.**

Not only does the **.NET Design Pattern Framework 4.5** cover GOF and Enterprise patterns, it also includes .NET pattern architectures that reduce the code you need to write by up to 75%. This unique package will change your .NET lifestyle -- for only $79.  Here's what is included:

(/products/net-design-pattern-framework)

**Two editions: C# and VB.NET** (/products/net-design-pattern-framework)

Learn More (/products/net-design-pattern-framework)

- 69 gang-of-four pattern projects
- 46 head-first pattern projects
- Fowler's enterprise patterns
- Multi-tier patterns
- Convention over configuration

- Active Record and CQRS patterns
- Repository and Unit-of-Work patterns
- MVC, MVP, & MVVM patterns
- REST patterns with Web API
- Spark$^{TM}$ Rapid App Dev (RAD) platform!
- Art Shop MVC Reference Application
- 100% pure source code

**Company**

- About Us (/about)
- Our Story (/story)
- Services (/services)
- Training (/training)
- Contact Us (/contact)

- Privacy (/privacy)
- End User License (/eula)
- Terms (/terms)
- Licensing (/licensing)

**Customers**

- Our Customers (/customers)
- Customer Stories (/customers/stories)

## Community

- Questions (/topic/search.aspx)
- Explore (/topic/topics.aspx)
- Tags (/tag/tags.aspx)

## Reference Guides

- .NET Design Patterns (/net/design-patterns)
- JavaScript Design Patterns (/javascript/design-patterns)

- JavaScript Tutorial (/tutorial/javascript)
- SQL Tutorial (/sql/tutorial)

- Connection Strings (/reference/connection-strings)
- Visual Studio Shortcuts (/reference/visual-studio-shortcuts)
- C# Coding Standards (/reference/csharp-coding-standards)
- HTML Colors (/reference/html-color-codes)

## Our Products

- .NET Design Pattern Framework (/products/net-design-pattern-framework) TM
- PRO .NET Design Pattern Framework (/products/pro-net-design-pattern-framework) TM

- JavaScript + jQuery Pattern Framework (/products/javascript-jquery-design-pattern-framework) TM
- SQL + Database Pattern Framework (/products/sql-database-design-pattern-framework) TM

- Products and Pricing (/products)