Decorator Design Pattern

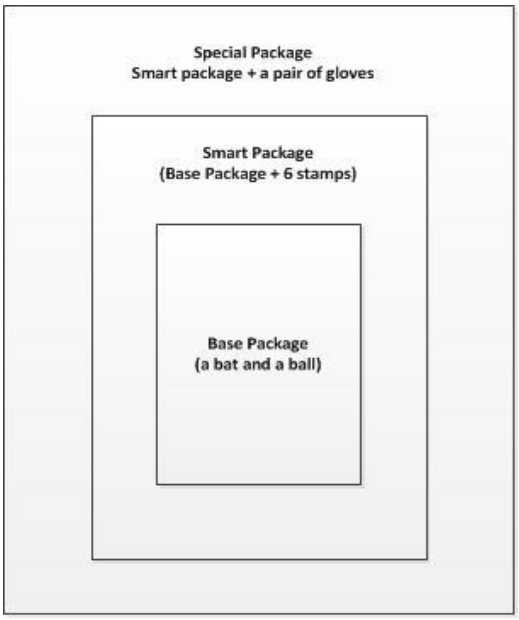
By Md. Mahedee Hasan, Software Architect, Leadsoft Bangladesh Ltd.
Source: http://mahedee.net/decorator-design-pattern/

You who work on design pattern must familiar with Gang of Four (GoF). **Design Patterns: Elements of Reusable Object-Oriented Software** is a Software Engineering book. The authors of this book are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The authors of this book are often refers to as Gang of Four (GoF). It's a parody relating to Mao's Gang of Four. This book describes the recurring solution of common problem in software design. The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral. Factory Pattern, Abstract Factory Pattern, Singleton Pattern, Builder e.t.c are creational design pattern. Decorator, Adapter, Bridge, Façade, Proxy, Composite e.t.c are structural design pattern. Command, interpreter, strategy, iterator e.t.c are behavioral design pattern.

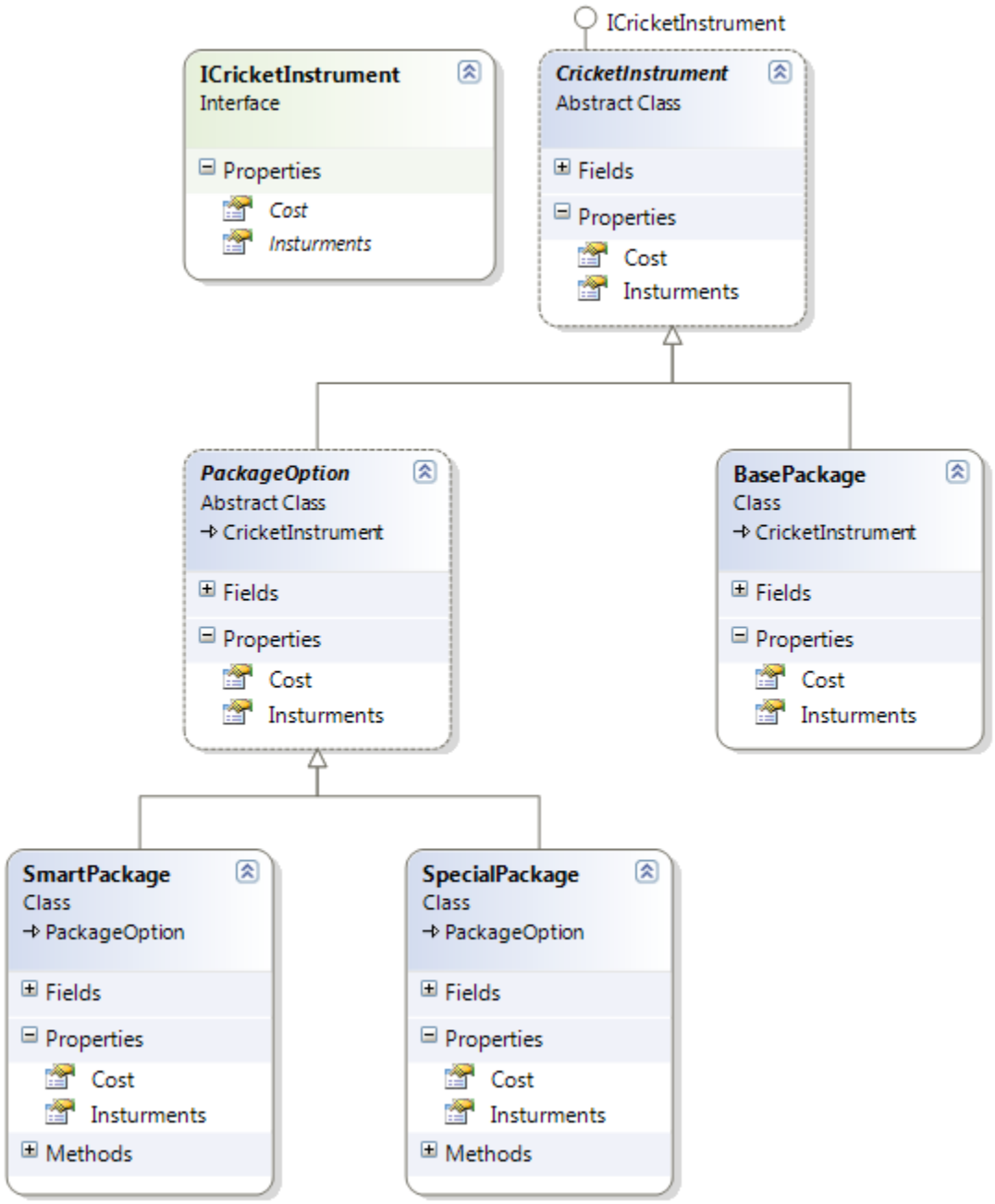
Decorator Design Pattern

Decorator Design Pattern is a structural design pattern. It is also known as wrapper. It is used to add additional functionality to a particular object during run time without affecting other objects. Responsibility can be adding or removing at run time. Critics says that it uses lot of little object of similar type.

Let's consider a scenario in which I am going to implement Decorated Design Pattern. We, Bangladeshi are very much cricket loving. A cricket equipment shop declared some package on the occasion of Cricket World Cup 2011. The base package is with a bat and a ball and its cost is 1500 Taka (Bangladeshi Currency). The smart package is with a bat, a ball and 6 stamps i.e base package plus 6 stamps and its cost is 1500 + 600 = 2100 Taka. The special package is with a bat, a ball, 6 stamps and a pair of gloves i.e smart package plus a pair of gloves. It's cost is 2100 + 500 = 2600 Taka. It looks like the following figure. Actually, the top one wrapped the inner packages.



Before going to implement the scenario by decorated design pattern, I would like to show you the class diagram. The class diagram is given below.



Implementation: Let's implement decorated design pattern by C#.

Step 1: Create interface ICricketInstrument for all package of cricket instrument.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace DecoratorPattern
{
    /// <summary>
    /// Interface for Cricket Instrument
    /// Developed by: Mahedee
    /// </summary>
    public interface ICricketInstrument
    {
        double Cost { get; }
        string Insturments { get; }
    }
}
```

Step 2: Create a base type for Concrete Cricket Instrument Package and Instrument Package Option

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DecoratorPattern
{
    /// <summary>
    /// Base type for concrete cricket instrument package and decorator(option).
    /// Developed by: Mahedee
    /// </summary>
    public abstract class CricketInstrument : ICricketInstrument
    {
        private double cost = 00;
        private string instrument = "Cricket Instruments: ";

        #region ICricketInstrument Members

        public virtual double Cost
        {
            get { return cost; }
        }

        public virtual string Insturments
        {
            get { return instrument; }
        }

        #endregion
    }
}
```

Step 3: Create a base type for Concrete Cricket Instrument Package Option. It is actually decorator – to decorate Instrument package.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace DecoratorPattern
{
    /// <summary>
    /// Decorator for concrete package option
    /// </summary>
    public abstract class PackageOption : CricketInstrument
    {
        double cost = 00;
        string instruments = "Abstract Package Option";

        public override double Cost
        {
            get { return cost; }
        }


        public override string Insturments
        {
            get { return instruments; }
        }

    }
}
```

Step 4: Create base package of Cricket Instrument.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DecoratorPattern
{
    /// <summary>
    /// Base Cricket Insturment Package
    /// </summary>
    public class BasePackage : CricketInstrument
    {
        double cost = 1500;
        string instruments = "Ball and Bat";

        public override double Cost
        {
            get { return cost; }
        }

        public override string Insturments
        {
            get { return base.Insturments + instruments; }
        }

    }
}
```

Step 5: Decorate Smart package for Cricket Instrument

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DecoratorPattern
{
    /// <summary>
    /// Smart Package = Base Package (Bat, Ball) + 6 Stamps
    /// Developed by: Mahedee
    /// </summary>
    public class SmartPackage : PackageOption
    {
        double cost = 600;
        string instruments = "6 Stamps";
        CricketInstrument objCricketInstrument;

        public SmartPackage(CricketInstrument objPCricketInstrument)
        {
            objCricketInstrument = objPCricketInstrument;
        }

        public override double Cost
        {
            get { return objCricketInstrument.Cost + cost; }
        }

        public override string Insturments
        {
            get { return objCricketInstrument.Insturments + ", " + instruments; }
        }
    }
}
```

Step 6: Decorate Special Package for Cricket Instrument.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DecoratorPattern
{
    /// <summary>
    /// Special Package = Base Package (Bat, Ball) + 6 Stamps + 1 Pair - Gloves
    /// Developed by: Mahedee
    /// </summary>
    public class SpecialPackage : PackageOption
    {
        double cost = 500;
        string instruments = "Gloves - 1 Pair";
        CricketInstrument objCricketInstrument;

        public SpecialPackage(CricketInstrument objPCricketInstrument)
```

```
        {
            objCricketInstrument = objPCricketInstrument;
        }

        public override double Cost
        {
            get { return objCricketInstrument.Cost + cost; }
        }

        public override string Insturments
        {
            get { return objCricketInstrument.Insturments + ", " + instruments; }
        }

    }
}
```

Step 7: Program class to create base package and decorate other package with option.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DecoratorPattern
{
    public class Program
    {
        static void Main(string[] args)
        {
            //Base Insturment class

            CricketInstrument objCricketInstruments = new BasePackage();
            Console.WriteLine("::Base Package::");
            Console.WriteLine(objCricketInstruments.Insturments);
            Console.WriteLine("Instrument's Cost: " + objCricketInstruments.Cost);

            Console.WriteLine();

            //Smart Package
            objCricketInstruments = new SmartPackage(objCricketInstruments);
            Console.WriteLine("::Smart Package::");
            Console.WriteLine(objCricketInstruments.Insturments);
            Console.WriteLine("Instrument's Cost: " + objCricketInstruments.Cost);

            Console.WriteLine();

            //Special Package
            objCricketInstruments = new SpecialPackage(objCricketInstruments);
            Console.WriteLine("::Special Package::");
            Console.WriteLine(objCricketInstruments.Insturments);
            Console.WriteLine("Instrument's Cost: " + objCricketInstruments.Cost);

            Console.ReadLine();
```

```
            }
        }
    }
```