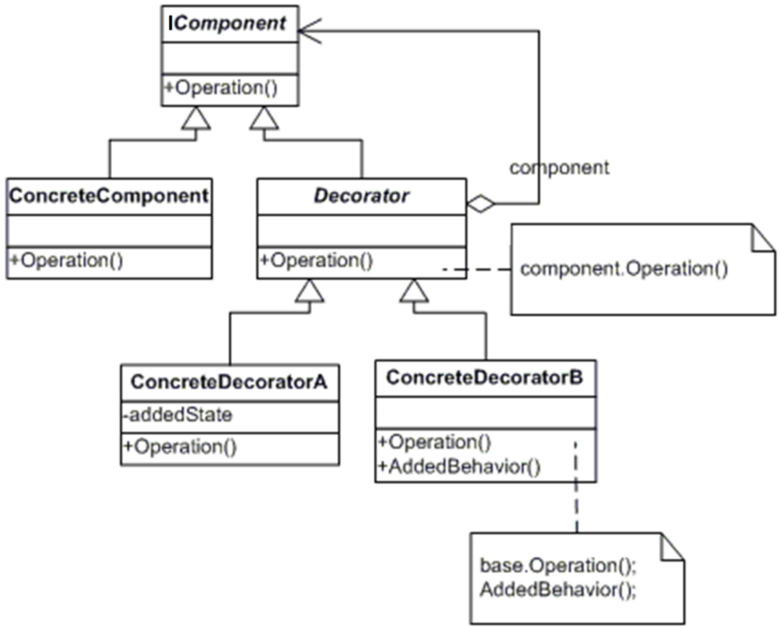
CPSC 501 – Assignment #7
Decorator and Proxy Patterns

Decorator Pattern: Decorator pattern allows providing enhanced behavior to an existing class's methods. It is a flexible alternative to sub classing (i.e., deriving from a base class). It is especially useful when we wanted to provide more than one enhancement (decoration) to a method and be able to select any order and combination of these enhancements. It also does not require to have the code for original class be available that we want to provide enhanced behavior for (referred to as component in the design pattern terminology).
The key idea in the decorator pattern is to define an interface for the class that needs decoration, and then the decorator inherits from this interface as well as contains an object of it (aggregation). The following UML depicts the general implementation of decorator. As you can see from the following diagram, a decorator can add extra behavior (another method) or state to the inherited interface.



Create a windows forms application project called "DecoratorPattern". Add an interface called IComponent to it with the following code.

```
interface IComponent
{
    string Welcome();
    string Welcome(string name);
}
```

Add a class called Component that implements the above interface. The component class is our original i.e., non decorated class.

```
class Component : IComponent
{

    public virtual string Welcome()
    {
        return "Welcome to our group";
    }

    public virtual string Welcome(string name)
    {
        return name + " Welcome to our group";
    }
}
```

Add a base class for decorators called Decorator with the following code in it.

```
abstract class Decorator : IComponent
{
    protected IComponent icmp = null;
    public Decorator(IComponent cmp)
    {
        icmp = cmp;
    }
    abstract public string Welcome();
    abstract public string Welcome(string name);
}
```

Now add a concrete decorator class called DecoratorTime with the following code.

```
class DecoratorTime : Decorator
{
    public DecoratorTime(IComponent cmp)
        : base(cmp)
    {
    }

    public override string Welcome()
    {
        return icmp.Welcome() +"\nTime = " + DateTime.Now.ToString();
    }

    public override string Welcome(string name)
    {
        return icmp.Welcome(name) + "\nTime = " + DateTime.Now.ToString();
    }
}
```

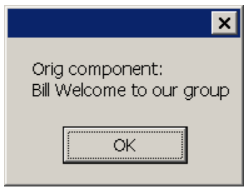
As you can see from the above code, it adds additional information of time by calling on the welcome methods of IComponent object aggregated by it.
Add another decorator class called DecoratorBday with the following code in it. This class adds the "Happy Birthday" message to the welcome methods of the IComponent object it aggregates.

```
class DecoratorBday : Decorator
{
    public DecoratorBday(IComponent cmp):base(cmp)
    {
    }

    public override string Welcome()
    {
        return icmp.Welcome() + "\nHappy Birthday ";
    }

    public override string Welcome(string name)
    {
        return icmp.Welcome(name) + "\nHappy Birthday";
    }
}
```

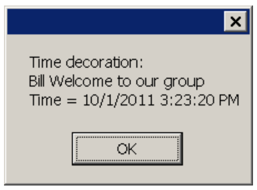
To test these decorators, add a button to the form called btnDecoratorSimple with the following code in its button handler.

```
        private void btnDecoratorSimple_Click(object sender, EventArgs e)
          {
              IComponent cmp = new Component();
              // base system object - undecorated
              MessageBox.Show("Orig component:\n" + cmp.Welcome("Bill"));

              // after decoration by DecoratorTime
              DecoratorTime dect = new DecoratorTime(cmp);
              // decorates cmp with time info
              MessageBox.Show("Time decoration:\n" + dect.Welcome("Bill"));

              // base component after decoration by DecoratorBday only
              DecoratorBday decb = new DecoratorBday(cmp);
              MessageBox.Show("Bday decoration:\n" + decb.Welcome("Bill"));

              // after decoration by both time and bday
              DecoratorBday decTimeBday = new DecoratorBday(dect);
              MessageBox.Show("Time,Bday decoration:\n" +
decTimeBday.Welcome("Bill"));
          }
```

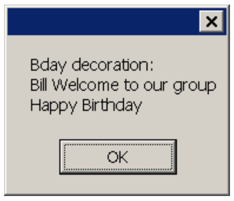
If you run the program and click on the above button, you will see the following output.



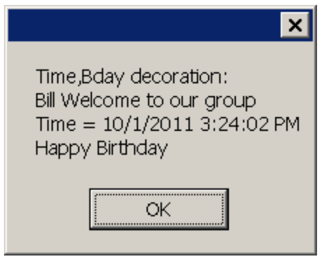
```
Orig component:
Bill Welcome to our group
```



```
Time decoration:
Bill Welcome to our group
Time = 10/1/2011 3:23:20 PM
```



```
Bday decoration:
Bill Welcome to our group
Happy Birthday
```



```
Time,Bday decoration:
Bill Welcome to our group
Time = 10/1/2011 3:24:02 PM
Happy Birthday
```

Question: What are the tradeoffs of using inheritance instead of the decorator pattern to provide enhanced behavior to an existing class?

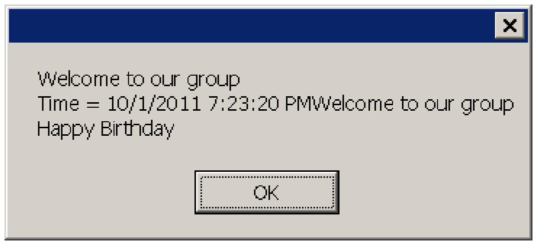
Answer: The existing class may not allow inheritance i.e., may be sealed or may not have virtual methods in it.
If we wanted more than one decorator effect such that the decorator effects can be chosen individually or their effects need to be combined, with strict inheritance approach, we will need many more classes. As an example, you can try creating two classes called DecoratorDerivedTime (corresponding to our first example in this chapter), and DecoratorDerivedBday

```
    class ComponentDerivedTime : Component
    {
        public override string Welcome()
        {
            return base.Welcome() + "\nTime = " +
                DateTime.Now;
        }
    }

    class ComponentDerivedBday : Component
    {
        public override string Welcome()
        {
            return base.Welcome() + "\nHappy Birthday";
        }
    }
```

Add a button to the form with a name of btnSubClassing with the following test code.

```
private void btnSubClassing_Click(object sender, EventArgs e)
        {
            ComponentDerivedTime cdt = new ComponentDerivedTime();
            MessageBox.Show(cdt.Welcome());
            ComponentDerivedBday cdb = new ComponentDerivedBday();
            MessageBox.Show(cdb.Welcome());

            // what if we wanted both time and bday decoration
            // This will require us to create another derived class
            // what if we wanted to print out time first, then bday
            // or vice versa. This will require us to create many
            // subclasses. Following does not produce desired output.
            MessageBox.Show(cdt.Welcome() + cdb.Welcome());

        }
```

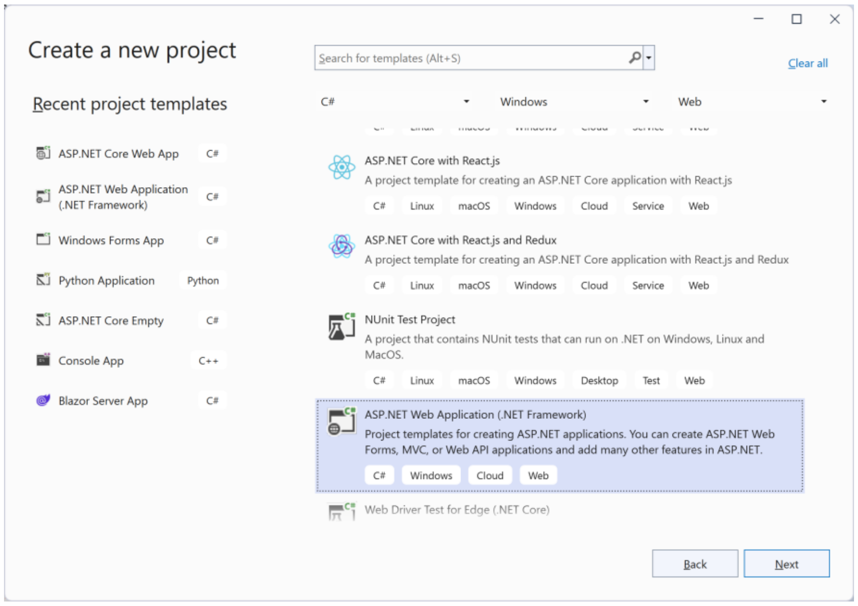
When we call the Welcome method in both ComponentDerivedTime and ComponentDerivedBday objects, the out contains duplicate information (not the desired effect of using multiple decorators).



```
Welcome to our group
Time = 10/1/2011 7:23:20 PMWelcome to our group
Happy Birthday
```

To present a slightly more practical example of the decorator pattern, we will develop a small web application with a simple page to display the shopping cart. Then, we will develop different decorators to provide a discount to the customer, if their purchase total is greater than $500, or if they are a first time customer, or if we have a holiday sale in progress.

Create a new Visual Studio Web application project called "ShoppingCartDecoratorPattern" as shown below.



Create a new project

Search for templates (Alt+S) — Clear all

Recent project templates

ASP.NET Core Web App	C#
ASP.NET Web Application (.NET Framework)	C#
Windows Forms App	C#
Python Application	Python
ASP.NET Core Empty	C#
Console App	C++
Blazor Server App	C#

C# — Windows — Web

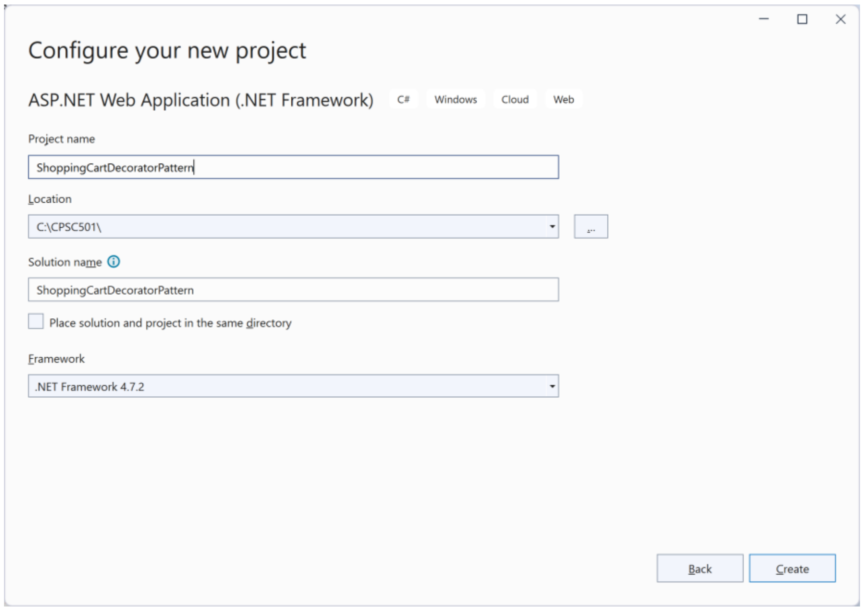
ASP.NET Core with React.js
A project template for creating an ASP.NET Core application with React.js
C# Linux macOS Windows Cloud Service Web

ASP.NET Core with React.js and Redux
A project template for creating an ASP.NET Core application with React.js and Redux
C# Linux macOS Windows Cloud Service Web

NUnit Test Project
A project that contains NUnit tests that can run on .NET on Windows, Linux and MacOS.
C# Linux macOS Windows Desktop Test Web

ASP.NET Web Application (.NET Framework)
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
C# Windows Cloud Web

Web Driver Test for Edge (.NET Core)



Configure your new project

ASP.NET Web Application (.NET Framework) C# Windows Cloud Web

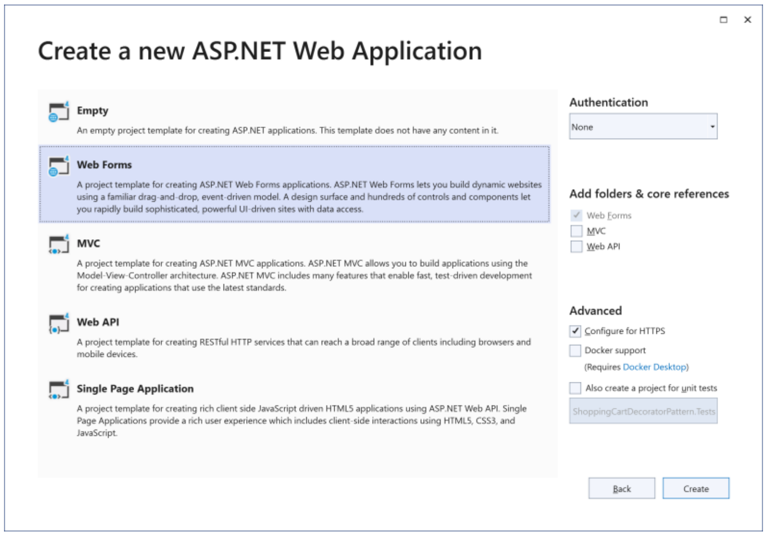
Project name: ShoppingCartDecoratorPattern

Location: C:\CPSC501\

Solution name: ShoppingCartDecoratorPattern

☐ Place solution and project in the same directory

Framework: .NET Framework 4.7.2



Create a new ASP.NET Web Application

Empty
An empty project template for creating ASP.NET applications. This template does not have any content in it.

Web Forms
A project template for creating ASP.NET Web Forms applications. ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

MVC
A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Web API
A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Single Page Application
A project template for creating rich client side JavaScript driven HTML5 applications using ASP.NET Web API. Single Page Applications provide a rich user experience which includes client-side interactions using HTML5, CSS3, and JavaScript.

Authentication: None

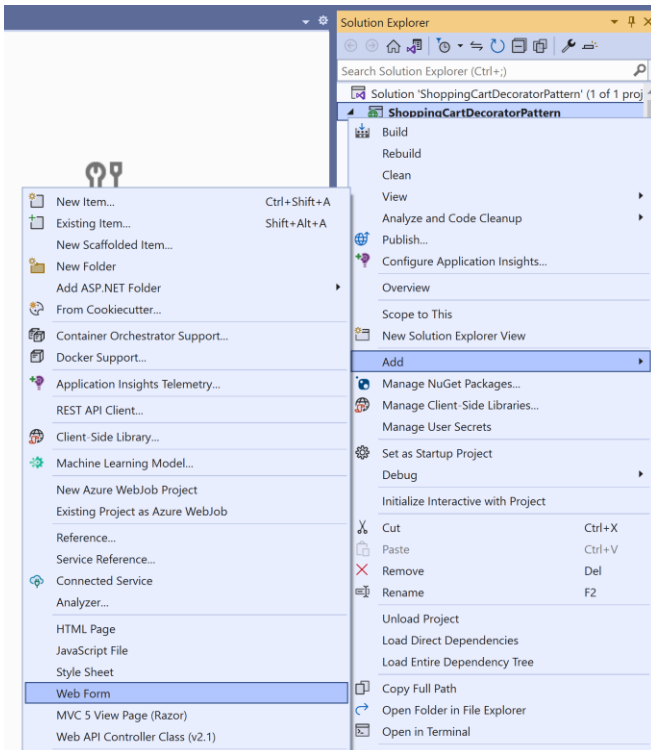
Add folders & core references
- [✓] Web Forms
- [ ] MVC
- [ ] Web API

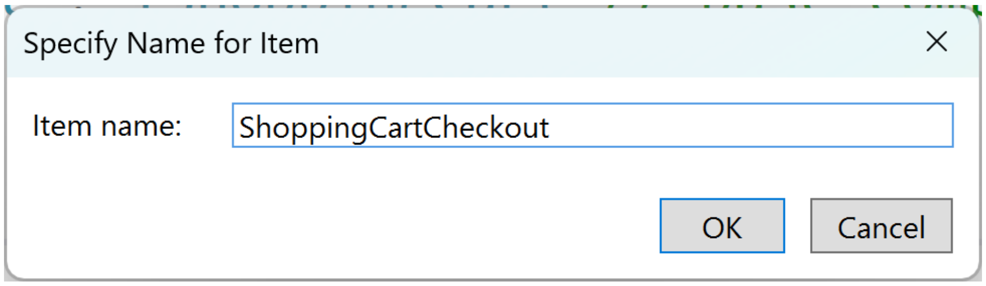
Advanced
- [✓] Configure for HTTPS
- [ ] Docker support (Requires Docker Desktop)
- [ ] Also create a project for unit tests

ShoppingCartDecoratorPattern.Tests

Back | Create

Add a webform to the project (webform is a web page in this technology) as shown below. Right click on the project name and choose Add-> WebForm.



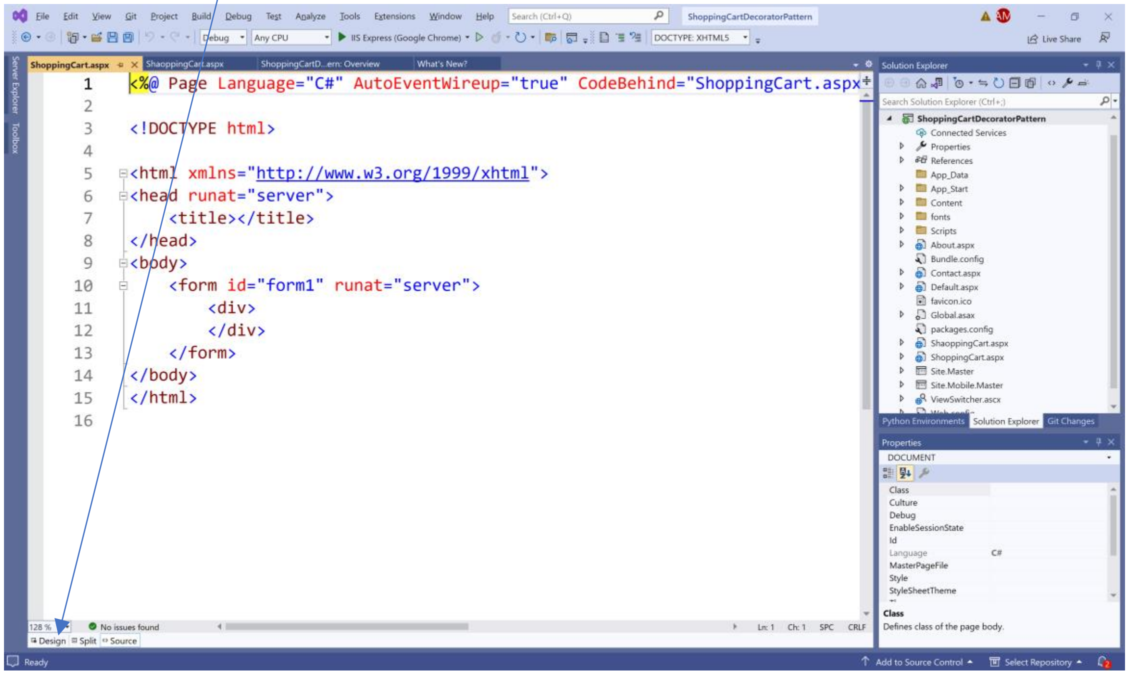


Specify Name for Item

Item name: ShoppingCartCheckout

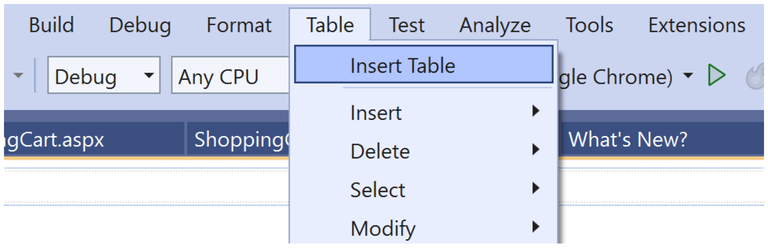
OK | Cancel

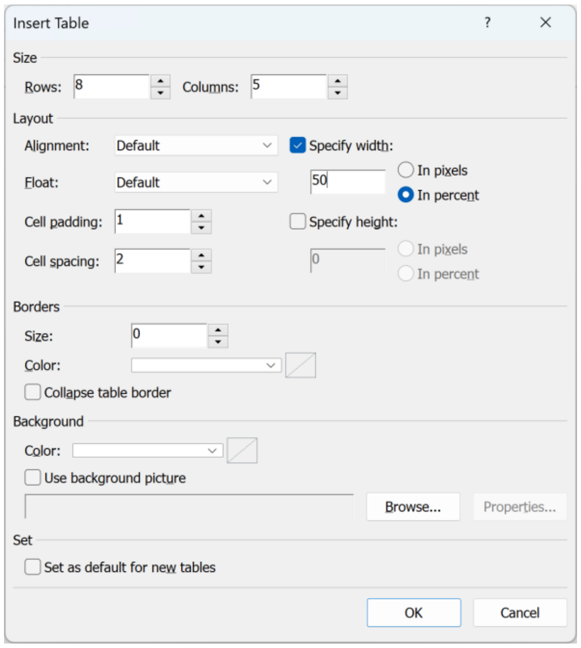
Switch to the Design view when ShoppingCartCheckout.aspx page is being displayed.



```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ShoppingCart.aspx

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        </div>
    </form>
</body>
</html>
```

Then insert table with 8 rows and 5 columns as shown below





Insert Table

Size

Rows: 8 Columns: 5

Layout

Alignment: Default

[✓ Specify width:]

Float: Default

50

In pixels [✓ In percent]

Cell padding: 1

Specify height:

Cell spacing: 2

0

In pixels In percent

Borders

Size: 0

Color:

Collapse table border

Background

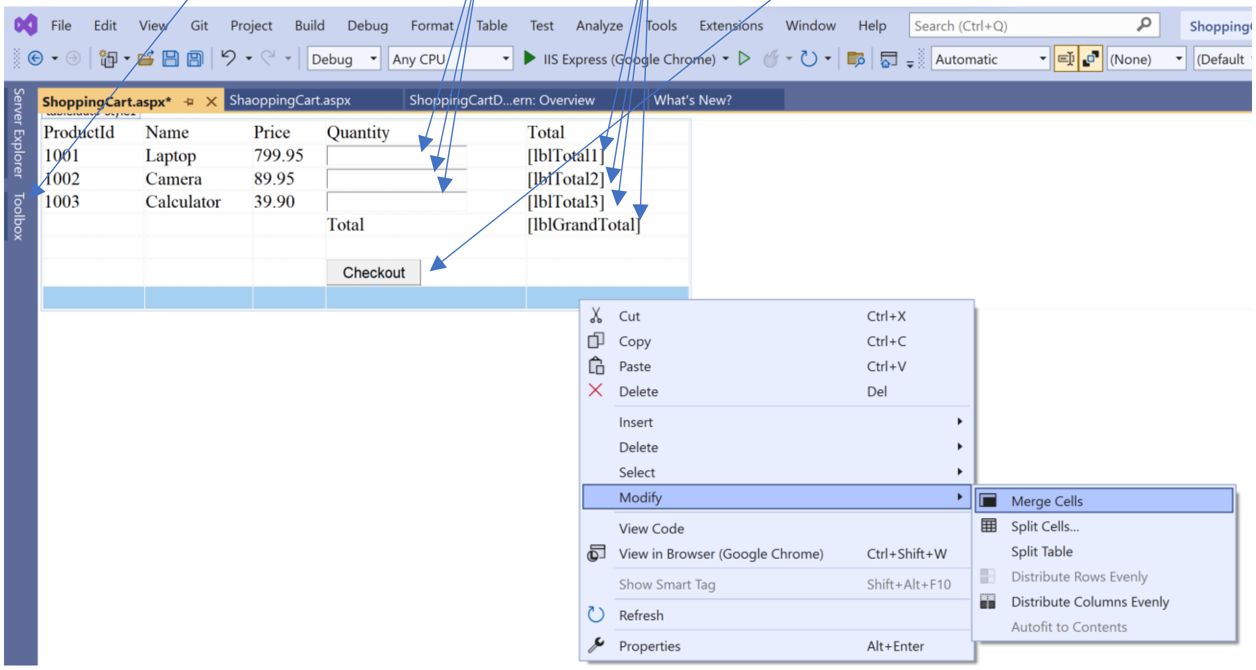
Color:

Use background picture

Set

Set as default for new tables

Then from the toolbox, put three text boxes, four labels, and one button in the cells of the table as shown below.



ProductId	Name	Price	Quantity	Total
1001	Laptop	799.95		[lblTotal1]
1002	Camera	89.95		[lblTotal2]
1003	Calculator	39.90		[lblTotal3]
			Total	[lblGrandTotal]
			Checkout	

Give IDs of txtQt1, txtQt2, txtQt3 to the three text boxes respectively, btnCheckout for the button, and lblTotal1, lblTotal2, lblTotal3 and lblGrandTotal to the labels. Clear the text property of lables, and assign a text property of Checkout to the button. Type the text in the remaining cells as shown above.

Highlight all cells in the last row, the right click on the last row, and choose Modify-> Merge cells as shown above. From the toolbox, put a label in the last row with an ID of lblMessage, and clear its text property.

Add a folder to the project called Models.

Add a class to the Models folder called CartRow with the following code in it. This represents one row in the shopping cart.

```
public class CartRow
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
    public decimal PriceTotal { get; set;}
}
```

Then add an interface called IShoppingCart to the Models folder with the following code in it.

```
public interface IShoppingCart // base interface
{
    List<CartRow> CartList { get; set; }
    decimal ComputeTotal();
}
```

Add a class called ShoppingCart to the Models folder with the following code in it.

```
public class ShoppingCart : IShoppingCart // base component
{
    public List<CartRow> CartList { get; set; } // list of items in cart

    public decimal ComputeTotal()
    {
        decimal grandTotal = 0;
        foreach (CartRow row in CartList)
        {
            row.PriceTotal = row.Price * row.Quantity;
            grandTotal+= row.PriceTotal;
        }
        return grandTotal;
    }
}
```

Add a class called ShoppingCartBaseDecorator to the Models folder with the following code in it.

```
public abstract class ShoppingCartBaseDecorator : IShoppingCart
{
    public List<CartRow> CartList { get; set; }
    public decimal OriginalGrandTotal { get; set; } // before discounts

    protected IShoppingCart _shoppingCart;
    public ShoppingCartBaseDecorator(IShoppingCart shoppingCart)
    {
        _shoppingCart = shoppingCart;
    }

    public abstract decimal ComputeTotal();
}
```

The above will act as the base class for all decorators.

Add a class called ShoppingCartGT500Decorator to the Models folder with the following code in it.

```
public class ShoppingCartGT500Decorator : ShoppingCartBaseDecorator
{
    public ShoppingCartGT500Decorator(IShoppingCart shoppingCart)
        : base(shoppingCart) // call base class construct[or
    {
    }
    public override decimal ComputeTotal()
    {
        decimal grandTotal = 0;
        grandTotal = _shoppingCart.ComputeTotal();
        if (grandTotal > 500)
        {
            grandTotal = grandTotal - (grandTotal * (decimal)(5 / 100.0));
        }
        return grandTotal;
    }
}
```

The above decorator provides a discount of 5% if the purchase total is greater than 500 dollars.

Add a class called ShoppingCartFirstTimeCustomerDecorator with the following code in it.

```
    public class ShoppingCartFirstTimeCustomerDecorator : ShoppingCartBaseDecorator
    {
        public ShoppingCartFirstTimeCustomerDecorator(IShoppingCart shoppingCart)
            : base(shoppingCart) // call base class construct[or
        {
        }

        public override decimal ComputeTotal()
        {
            decimal grandTotal = 0;
            grandTotal = _shoppingCart.ComputeTotal();

            // check database to see if first time customer
            bool firstTimeCustomer = true;
            if (firstTimeCustomer) // apply further 7.5% discount
            {
                grandTotal = grandTotal - grandTotal * (7m / 100.0m);
            }
            return grandTotal;
        }
    }
```

Add a class to the Models folder called ShoppingCartHolidaySaleDecorator with the following code in it.

```
    public class ShoppingCartHolidaySaleDecorator : ShoppingCartBaseDecorator
    {
        public ShoppingCartHolidaySaleDecorator(IShoppingCart shoppingCart)
            : base(shoppingCart) // call base class construct[or
        {
        }

        public override decimal ComputeTotal()
        { // 10% holiday sale
            decimal grandTotal = 0;
            grandTotal = _shoppingCart.ComputeTotal();
            grandTotal = grandTotal - (grandTotal * (decimal)(10 / 100.0));
            return grandTotal;
        }
    }
```

Our three concrete decorators are ready and they can be used individually, or in any combination and order to provide discounts to the shopping cart.

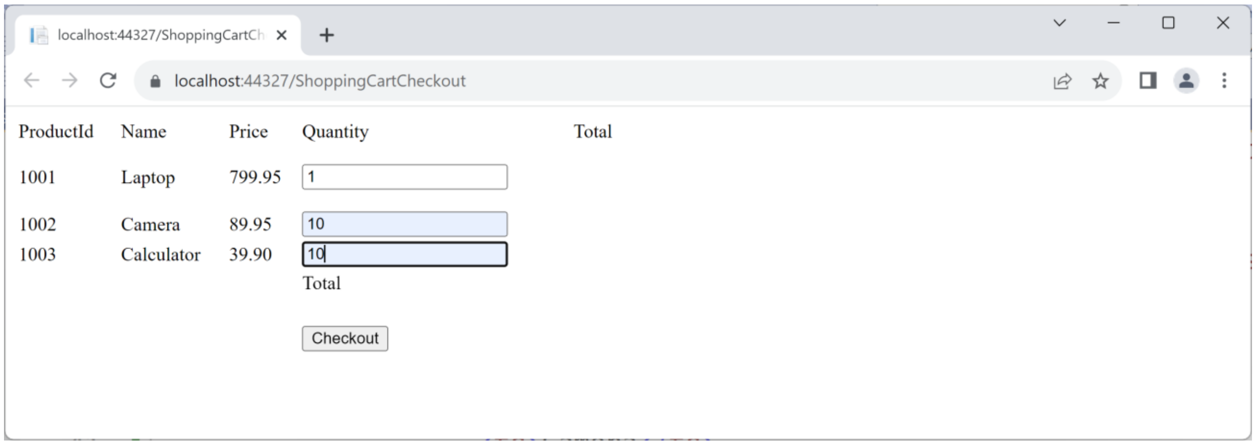
Modify the code in ShoppingCartCheckout.aspx.cs file to appear as:

```
using ShoppingCartDecoratorPattern.Models;
```

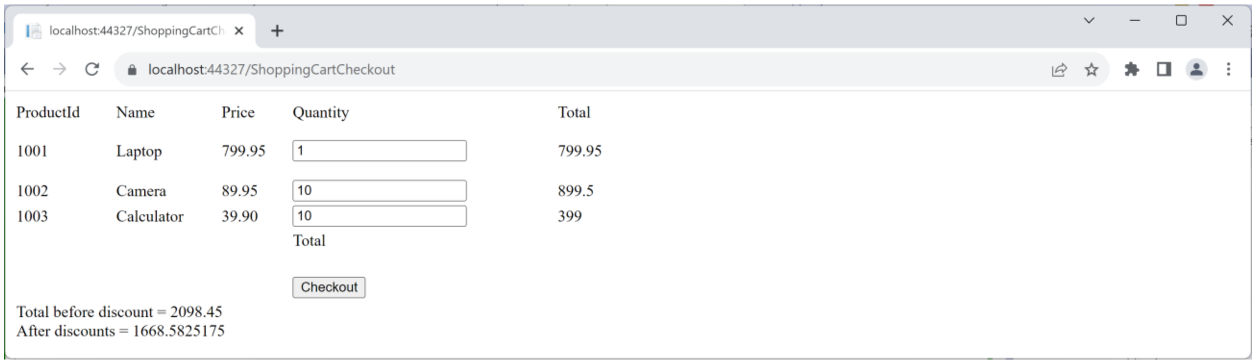
….

```
    public partial class ShoppingCartCheckout : System.Web.UI.Page
    {
        List<CartRow> CartList = new List<CartRow> ();
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void btnChecout_Click(object sender, EventArgs e)
        {
            CartRow row1 = new CartRow()
            {
                ProductId = 1001,
                ProductName = "Laptop",
                Price = 799.95m,
                Quantity = int.Parse(txtQt1.Text)
            };
            CartList.Add(row1);
            CartRow row2 = new CartRow()
            {
                ProductId = 1002,
                ProductName = "Camera",
                Price = 89.95m,
                Quantity = int.Parse(txtQt2.Text)
            };
            CartList.Add(row2);
            CartRow row3 = new CartRow()
            {
                ProductId = 1003,
                ProductName = "Calculator",
                Price = 39.90m,
                Quantity = int.Parse(txtQt3.Text)
            };
            CartList.Add(row3);
            IShoppingCart cart = new ShoppingCart();
            cart.CartList = CartList;
            lblTotal1.Text = (int.Parse(txtQt1.Text) * 799.95).ToString();
            lblTotal2.Text = (int.Parse(txtQt2.Text) * 89.95).ToString();
            lblTotal3.Text = (int.Parse(txtQt3.Text) * 39.90).ToString();

            var decorator = new ShoppingCartHolidaySaleDecorator(
                new ShoppingCartFirstTimeCustomerDecorator(
                new ShoppingCartGT500Decorator(cart)));
            var gtotal = decorator.ComputeTotal();
            lblMessage.Text = "Total before discount = " +
cart.ComputeTotal().ToString() + "<br/>" +
                "After discounts = " + gtotal.ToString();
        }
    }
```

To keep the example simple, the above code hard codes a few products that are added to the shopping cart. The user can change the quantity for any item via the web page. Right click on the ShoppingCartCheckout.aspx and set it as a start page. Then run the project. Your output will appear as (if you choose quantities of 1, 10 and 10). The page applies discounts if the total is greater than 500, then another discount is applied if the user is a first time customer, and further another discount if the holiday sale is on. The shopping cart object is passed to a chain of decorators (via constructor injection).


localhost:44327/ShoppingCartCh
localhost:44327/ShoppingCartCheckout
Checkout

ProductId	Name	Price	Quantity	Total
1001	Laptop	799.95	1	
1002	Camera	89.95	10	
1003	Calculator	39.90	10	
			Total	


localhost:44327/ShoppingCartCh
localhost:44327/ShoppingCartCheckout
Checkout

ProductId	Name	Price	Quantity	Total
1001	Laptop	799.95	1	799.95
1002	Camera	89.95	10	899.5
1003	Calculator	39.90	10	399
			Total	

```
Total before discount = 2098.45
After discounts = 1668.5825175
```

Study the code in ShoppingCartCheckout.aspx.cs to see how the different decorators are chained to provide the three discounts. You can choose to not apply a particular decorator. For example, if it is not the holiday season, you can skip the injection of the ShoppingCartHolidaySaleDecorator.

In short, the decorator pattern, allows us a clean object-oriented approach to providing individual enhancements to be applied in a flexible manner such that we can choose any number of these enhancements (or decorations) and in any order.

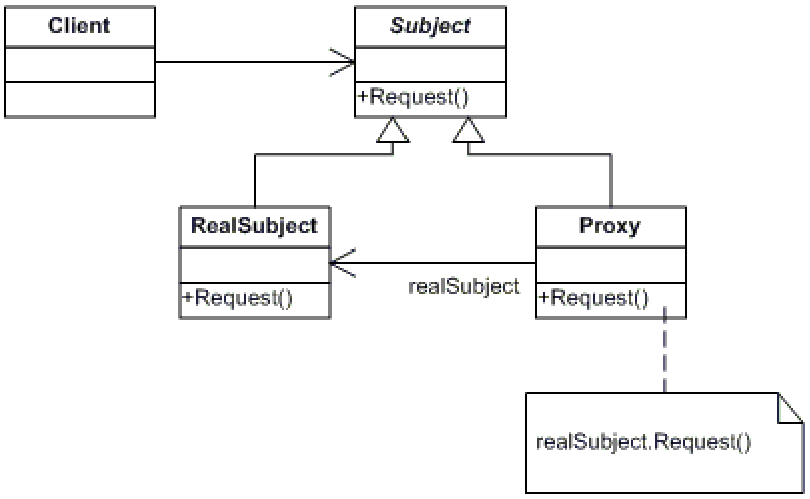
Proxy Pattern

Proxy pattern is used to provide an indirect access to an object via a surrogate object. The surrogate or the proxy object routes the calls to the methods of the actual object and may inject extra code such as security authentication as in the case of security proxy.

Proxy pattern is very popular in distributed computing where the access to a remote object is provided by a local proxy object which has the same interface as the remote object. The client calls methods of the local proxy object which routes the request to the remote object, collects the results and then passes it back to the caller. This way, the client is shielded from all the details of the TCP/IP communication in order to call methods of a remote object. When a proxy is used in a distributed computing environment, it is referred to as a remote proxy.

Another category of the proxy pattern is where the data is cached about a real object and given to the client. This is useful also in distributed computing environments where once a result is brought back from a remote method, the parameters and the result are cached locally. If the client makes another call later where the parameters are the same, the result is retrieved from the local cache and given to the client without having to make a remote call. Another variation is referred to as a virtual proxy where we can delay the creation and initialization of expensive objects until needed. For example, creating the real subject object only when some method is invoked on it.

The UML diagram for the proxy pattern is shown below.



The wrapper class (proxy) holds a reference to the real class and implements the same interface as the real class. The real class reference can be initialized at construction, or on its first use. .Each wrapper method in the proxy delegates the call to the real class object.

Virtual Proxy:.

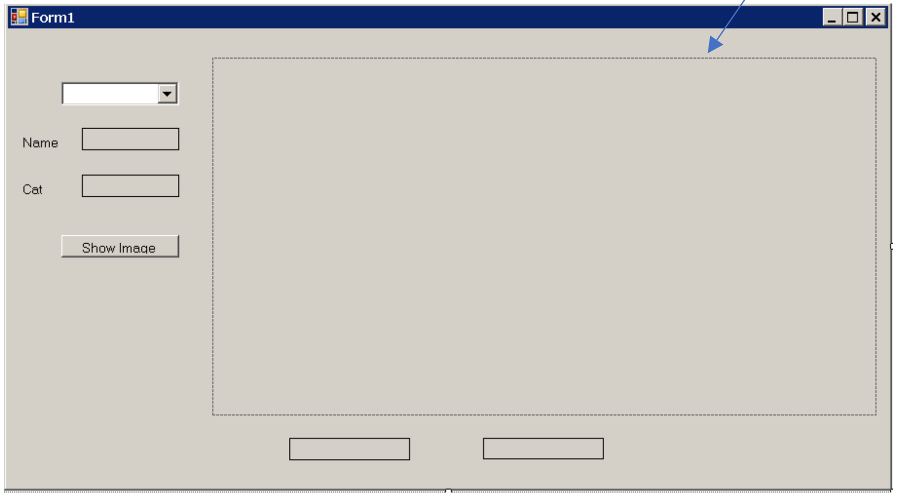
Virtual proxy is useful when the real subject is doing some expensive operation and we can defer executing this operation until it is needed. To demonstrate this, create a windows forms application project called “ProxyPattern”. Add an interface called ISubjectMyImage with the following code in it.

```
interface ISubjectMyImage
{
    string ShortName { get; }
    string Category { get; }
    string GetFileName();
    Bitmap GetBitmap();
    Size GetImageSize();
}
```

Add a class to the project called “SubjectMyImage” that implements the above interface with the code as shown below.

```
class SubjectMyImage : ISubjectMyImage
{
    Bitmap bmp = null;
    Size sz ;
    string shortName = "";
    string fileName;
    string category = "";

    public SubjectMyImage(string filename, string sName, string cat)
    {
        fileName = filename;
        category = cat;
        shortName = sName;
        bmp = new Bitmap(Image.FromFile(filename));
        sz = new Size(bmp.Width, bmp.Height);
    }


    public string ShortName
    {
        get { return shortName; }
    }

    public string GetFileName()
    {
        return fileName;
    }

    public string Category
    {
        get { return category; }
    }

    public Bitmap GetBitmap()
    {
        return bmp;
    }

    public Size GetImageSize()
    {
        return sz;
    }
}
```

To test the above image class, create the following GUI in the form. The combo box has a name of “cmbPictures”, the two labels below it (with border style of fixed single and auto size property set to false) have names of “lblShortName” and “lblCategory”. The button has a name of “btnShowImage”. The two labels at the bottom have names of “lblWidth” and “lblHeight”. The picture box (big rectangle) has a name of “pic1”.



Declare a list to store images at the top of the Form1 class as shown below.

```
public partial class Form1 : Form
{
    List<SubjectMyImage> PList = new List<SubjectMyImage>();
```

Create a folder called MyImages on the C drive. Then create two sub folders under it called Images and Images2. Then collect a few images of flowers and put them in the Images folder. Collect a few images of mountains and put them in the Images2 folder.

By double clicking on the form where there is no control, add the form load event handler with the following code in it. Form load event is used to provide an initialization code as the application starts.

```
        private void Form1_Load(object sender, EventArgs e)
        {
            cmbPictures.Items.Clear();
            string picfolder = @"c:\MyImages\Images";
            DirectoryInfo di = new DirectoryInfo(picfolder);
            foreach (FileInfo fi in di.GetFiles())
            {
                SubjectMyImage smi = new SubjectMyImage(fi.FullName,fi.Name,
"Flowers");
                PList.Add(smi);
                cmbPictures.Items.Add(smi.ShortName);
            }

            picfolder = @"c:\MyImages\Images2";
            di = new DirectoryInfo(picfolder);
            foreach (FileInfo fi in di.GetFiles())
            {
                SubjectMyImage smi = new
SubjectMyImage(fi.FullName,fi.Name,"Mountains");
                PList.Add(smi);
                cmbPictures.Items.Add(smi.ShortName);
            }
        }
```

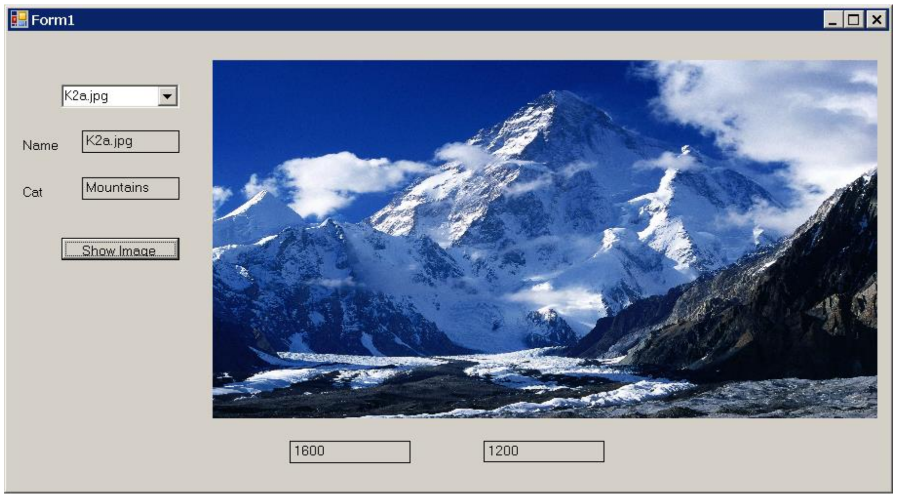
As you can see from the above code, it populates the PList by reading image files from two folders. PList contains objects of SubjectMyImage which further has fields for the image name, its size and the bitmap belonging to the image.
Double click on the combo box and write the following code in its selected index changed event handler (by double clicking on the combo box).

```
private void cmbPictures_SelectedIndexChanged(object sender, EventArgs e)
{
    btnShowImage.Enabled = true;
    string sname = cmbPictures.Text;
    foreach (SubjectMyImage smi in PList)
    {
        if (sname == smi.ShortName)
        {
            lblCategory.Text = smi.Category;
            lblShortName.Text = smi.ShortName;
        }
    }
}
```

Double click on the btnShowImage to write its click event handler. Type the following code in it.

```
private void btnShowImage_Click(object sender, EventArgs e)
{
    string sname = cmbPictures.Text;
    foreach (SubjectMyImage smi in PList)
    {
        if (sname == smi.ShortName)
        {
            pic1.Image = smi.GetBitmap();
            lblWidth.Text = smi.GetImageSize().Width.ToString();
            lblHeight.Text = smi.GetImageSize().Height.ToString();
        }
    }
}
```

Build and test the application.



K2a.jpg
Name K2a.jpg
Cat Mountains
1600
1200

Even though the above programs works correctly, it is not efficient in terms of memory usage. Since some image files are large, initializing all data for all images in the beginning of the program is inefficient especially, if the number of images is large.

We can overcome this problem by creating a proxy for the SubjectMyImage class that does not create the bitmap until an image needs to be displayed. The other fields related to the image such as category and short name are made available immediately to the client. To create the proxy properly, it should mimic the functionality of the real subject so that the real subject object in the code can be replaced by the proxy object without causing any additional code change.

Add a class to the project called ProxyMyImage with the following code in it.

```
class ProxyMyImage : ISubjectMyImage // real subject should be replacable by proxy

{
    ISubjectMyImage ism = null;
    string fileName = "";
    string shortName = "";
    string category = "";

    public ProxyMyImage(string fname, string sname, string cat)
    {
        fileName = fname;
        category = cat;
        shortName = sname;
    }

    void CreateSubject()
    {
        if (ism == null)
            ism = new SubjectMyImage(fileName, shortName, category);
    }

    public string ShortName
    {
        get
        {
            return shortName; // no need to create real subject
        }
    }

    public string Category
    {
        get
        {
            return category; // no need to create real subject
        }
    }

    public string GetFileName()
    {
        return fileName;     // no need to create real subject
    }

    public System.Drawing.Bitmap GetBitmap()
    {
        CreateSubject(); // create subject since bitmap is needed
        return ism.GetBitmap();
    }

    public System.Drawing.Size GetImageSize()
    {
        CreateSubject(); // create subject since it is needed
        return ism.GetImageSize();
    }
}
```

As you can see from the above code, the proxy has the same properties, constructor and the methods as the real subject. It contains a reference to the real subject but does not create it until an expensive operation is needed (e.g., GetBitmap). For simpler operations such as GetFileName, the ShortName, or the Category, the proxy returns its own data. To test it, replace every occurance of SubjectMyImage with the ProxyMyImage in the form code and build and test the program. It should function same as before. However, it is not allocating memory for the images until some one clicks on the "Show Image" button.

ProtectionProxy: Protection or security proxy is useful in those cases where the access to a class's methods require some authentication. To show an example, add a DataAccessFacade class to the project with the following code (replace the connection string with name of your database server).

```
    class DataAccessFacade
    {
        public string CONNSTR = "server=pico2\\SQLExpress;integrated
security=true;database=ProductsDB";

        public object GetSingleAnswer(string sql)
        {
            SqlConnection conn = new SqlConnection(CONNSTR);
            object obj = null;
            try
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                obj = cmd.ExecuteScalar();
            }
            catch (Exception)
            {
                throw;
            }
            finally
            {
                conn.Close();
            }
            return obj;
        }

        public int UpdateInsertDelete(string sql)
        {
            SqlConnection conn = new SqlConnection(CONNSTR);
            int rowsModified = 0;
            try
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                rowsModified = cmd.ExecuteNonQuery();
            }
            catch (Exception)
            {
                throw;
            }
            finally
            {
                conn.Close();
            }
            return rowsModified;
        }
        public DataTable GetManyRowsCols(string sql)
        {
            SqlConnection conn = new SqlConnection(CONNSTR);
            DataTable dt = new DataTable();
            try
            {
                conn.Open();
                SqlDataAdapter da = new SqlDataAdapter(sql, conn);
                da.Fill(dt);
                conn.Close();
            }
            catch (Exception)
            {
                throw;
            }
            finally
            {
                conn.Close();
            }
            return dt;
        }
    }
```

If we wanted password protection in calling the methods of the DataAccessFacade class, we can create a protection proxy. Add a class called ProtectionProxy with the following code in it. It is important that it contains all the existing methods of the DataAccessFacade class. We are going to add an extra authenticate method to the proxy though.

```
    class ProtectionProxy
    {
        DataAccessFacade daf = null;

        public bool Authenticate(string password)
        {
            bool res = false;
            if (password == "secret")
            {
                daf = new DataAccessFacade();
                res = true;
            }
            return res;
        }

        public object GetSingleAnswer(string sql)
        {
            object obj = null;
            if (daf != null)
                obj = daf.GetSingleAnswer(sql);
            else
                throw new Exception("requires authentication..");
            return obj;
        }

        public int UpdateInsertDelete(string sql)
        {
            int ret = 0;
            if (daf != null)
                ret = daf.UpdateInsertDelete(sql);
            else
                throw new Exception("requires authentication..");
            return ret;
        }
        public DataTable GetManyRowsCols(string sql)
        {
            DataTable dt = null;
            if (daf != null)
                dt = daf.GetManyRowsCols(sql);
            else
                throw new Exception("requires authentication..");
            return dt;
        }
    }
```

To test the protection proxy, add a button to the form called btnProtectionProxy with the following code in its click event handler.

```
      private void btnProtectionProxy_Click(object sender, EventArgs e)
        {
            try
            {
                ProtectionProxy prp = new ProtectionProxy();
                bool res = prp.Authenticate("secret");
                string sql = "select ProductName from Products where " +
                   "ProductId=1001";
                object obj = prp.GetSingleAnswer(sql);
                if (obj != null)
                    MessageBox.Show(obj.ToString());
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

Build and test by clicking on the above button handler. You will see the correct data from the database being displayed in a message box (if you have a product with productid of 1001, otherwise change the SQL to match an existing product in your database).

If you do not include the Authenticate call in the above code (i.e., comment this line), it will report a requires authentication message. Note that authenticate call needs to be made only once, then the proxy object for DataAccess can call the methods as many times without having to authenticate again.