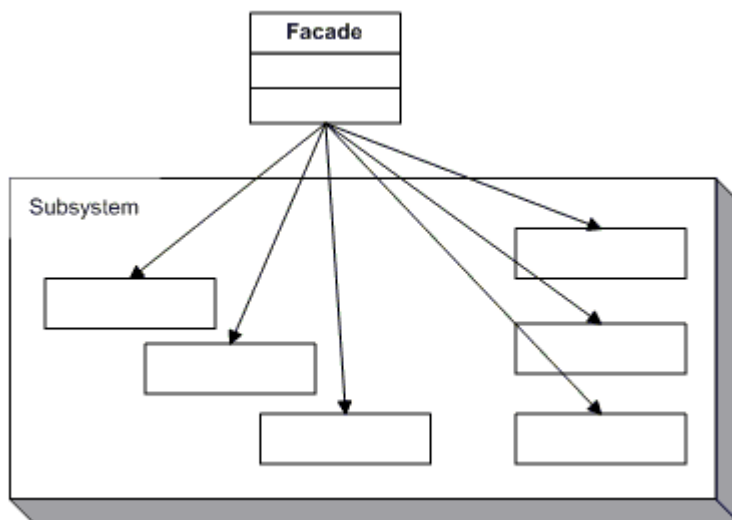


CPSC 501 – Assignment #5

Design Patterns – Façade, Singleton, Factory Patterns

Façade Pattern

The purpose of Façade design pattern is to provide a unified interface to a set of interfaces in the sub systems, so that the overall system is easier to use.



Example: Mortgage Application.

Create a new windows forms application project called FacadePattern. Add a class called Bank with the following code in it. To simulate the effect of different sub systems involved in a mortgage application's approval process, we will create Bank, Credit and Loan classes to check the financial health of the mortgage applicant. In real life, data will be obtained from a database. Here we have hard coded it.

```

class Bank
{
    public double GetCurrentCheckingBalance(int accountNum)
    {
        if (accountNum == 12345)
            return 1250;
        else if (accountNum == 12346)
            return 230;
        else
            return 0;
    }

    public double GetSavingBalance(int accountNum)
    {
        if (accountNum == 12345)
            return 35870;
        else if (accountNum == 12346)
            return 230;
        else
            return 0;
    }
}

```

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

```
enum CreditRating : int
{
    BAD,
    MEDIUM,
    GOOD,
    EXCELLENT
}

class Credit
{
    public CreditRating CheckCredit(string ssn)
    {
        CreditRating creditRating = CreditRating.BAD;
        if (ssn == "111-22-3333")
            creditRating = CreditRating.GOOD;
        else if (ssn == "111-22-3334")
            creditRating = CreditRating.MEDIUM;
        return creditRating;
    }
}
```

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

```
class Loan
{
    public double GetOutstandingLoans(int acctNum)
    {
        double res = 0;
        if (acctNum == 12345)
            res = 2100;
        else if (acctNum == 12346)
            res = 7880;
        return res;
    }
}
```

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

```
class MortgageApplicant
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BankAccountNum { get; set; }
    public int LoanAccountNum { get; set; }
    public string SSNum { get; set; }
    public double LoanAmountAsked { get; set; }
}
```

For the mortgage approval, we need several checks before approving a loan. For this purpose, add a button to the form with a name of btnMortgageApproval with a text property of Mortgage Approval. Type the following code in its button handler by double clicking on the button.

```

private void btnMortgageApproval_Click(object sender, EventArgs e)
{
    bool approved = false;
    string reason = "";
    MortgageApplicant mapp = new MortgageApplicant
    {
        FirstName = "Bill",
        LastName = "Baker",
        BankAccountNum = 12345,
        LoanAccountNum = 12345,
        SSNum = "111-22-3333",
        LoanAmountAsked = 150000
    };

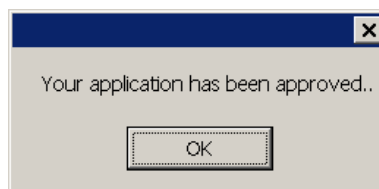
    Bank bk = new Bank();
    double bal = bk.GetCurrentCheckingBalance(mapp.BankAccountNum);
    bal = bal + bk.GetSavingBalance(mapp.BankAccountNum);

    Loan ln = new Loan();
    bal = bal - ln.GetOutstandingLoans(mapp.LoanAccountNum);
    if (bal < 0)
    {
        approved = false;
        reason = "negative balances ";
    }
    else
    {
        if (bal < 0.2 * mapp.LoanAmountAsked)
        {
            approved = false;
            reason = "Not enough balances";
        }
        else
        {
            Credit cr = new Credit();
            if (cr.CheckCredit(mapp.SSNum) < CreditRating.GOOD)
            {
                approved = false;
                reason = "not good credit rating..";
            }
            else
            {
                approved = true;
            }
        }
    }

    if (approved == true)
        MessageBox.Show("Your application has been approved..");
    else
        MessageBox.Show("Your application has been denied..\n" +
            reason);
}

```

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



Even though the above code works, it is not a clean solution. Since the mortgage approval process interacts with several classes (or sub systems) such as Bank, Credit, Loan, we can put all the logic of interacting with these subsystems in a separate class called Façade which will take the information about the mortgage applicant and give us an answer via a single method call.

To implement the Façade pattern, add a MortgageFacade class to the project and move the button handler code that makes the mortgage approval decision to a method of this class.

```
class MortgageFacade
{
    Bank bk = new Bank();
    Loan ln = new Loan();
    Credit cr = new Credit();

    public bool MortgageApproval(MortgageApplicant mapp, ref string
reason)
    {
        bool approved = false;
        double bal = bk.GetCurrentCheckingBalance(mapp.BankAccountNum);
        bal = bal + bk.GetSavingBalance(mapp.BankAccountNum);

        bal = bal - ln.GetOutstandingLoans(mapp.LoanAccountNum);
        if (bal < 0)
        {
            approved = false;
            reason = "negative balances ";
        }
        else
        {
            if (bal < 0.2 * mapp.LoanAmountAsked)
            {
                approved = false;
                reason = "Not enough balances";
            }
            else
            {
                if (cr.CheckCredit(mapp.SSNum) < CreditRating.GOOD)
                {
                    approved = false;
                    reason = "not good credit rating..";
                }
                else
                {
                    approved = true;
                }
            }
        }
        return approved;
    }
}
```

Now the button handler code can be reduced to as shown below.

```
private void btnMortgageApproval_Click(object sender, EventArgs e)
{
    string reason = "";
    MortgageApplicant mapp = new MortgageApplicant
    {
        FirstName = "Bill",
        LastName = "Baker",
        BankAccountNum = 12345,
        LoanAccountNum = 12345,
```

```

        SSNum = "111-22-3333",
        LoanAmountAsked = 150000
    };

    MortgageFacade mf = new MortgageFacade();
    bool approved = mf.MortgageApproval(mapp, ref reason);

    if (approved == true)
        MessageBox.Show("Your application has been approved..");
    else
        MessageBox.Show("Your application has been denied..\n" +
            reason);
}

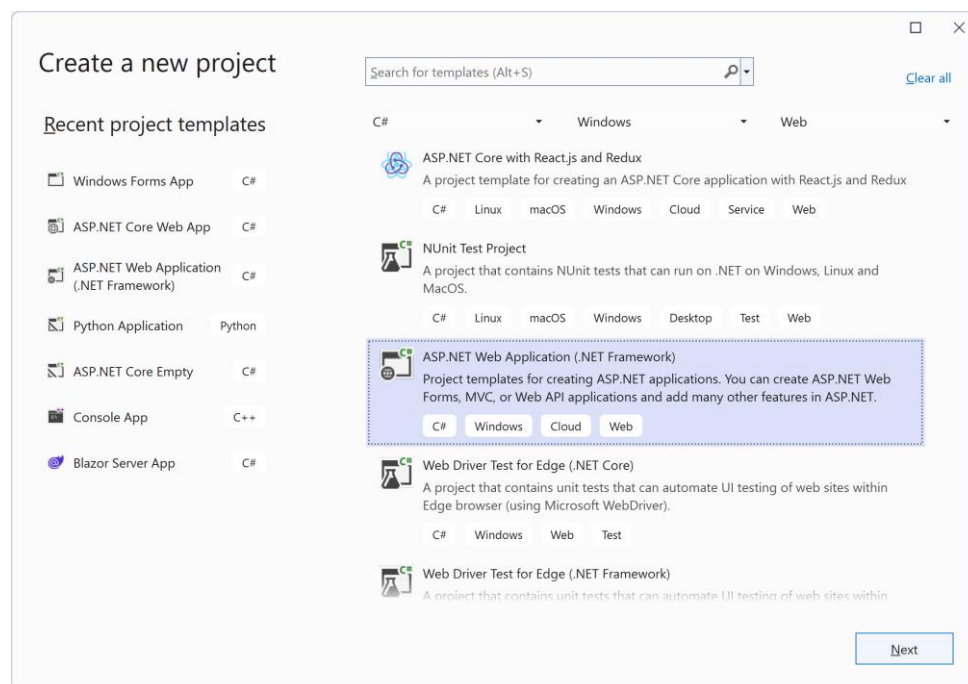
```

As you can see from the above code, checking the mortgage approval for a client is reduced to simply calling a single method in the MortgageFacade class.

Example : Session Façade in a Web Application:

Web applications extensively use the Session object for storing temporary information. The Session object in ASP.Net web forms technology allows you to store any thing in it e.g., integer, double, string, object of a class etc.. However, to recover the data stored in the Session object, you must type cast it to the same data type that was used to store the data. The developer can use any string as the name (i.e., key) of a session variable. Even though it is very flexible, this can cause quite a few problems ranging from mistyping the name of the session key in another page, to using the same Session key for storing a different data type. Session Façade can be created to overcome above problems and to make sure that there is no ambiguity between the names of session keys and their data types.

To demonstrate this, create an ASP.Net web application called SessionManagement.



Configure your new project

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

Project name
SessionManagement

Location
C:\CPSC555\...

Solution
Create new solution

Solution name
SessionManagement

☐ Place solution and project in the same directory

Framework
.NET Framework 4.7.2

Back Create

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
SessionManagement.Tests

Back Create

Add a web form called Page1.aspx (by right clicking on the project name and choosing “add web form”) with the following code in the Page_Load event in the Page1.aspx.cs file.

```
protected void Page_Load(object sender, EventArgs e)
{
    session["Userdata"] = 25;
    Response.Write("User data set to " + ((int)
(Session["Userdata"])).ToString());
}
```

Add another page i.e., web form called page ReadUserData.aspx. Put a label called lblStatus in the ReadUserData.aspx (by dragging and dropping a label from the toolbox in between the two div tags. The code in the ReadUserData.aspx will appear as:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ReadUserData.aspx.cs"
Inherits="SessionManagement.ReadUserData" %>

<!DOCTYPE html>

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

    <form id="form1" runat="server">
        <div>
            <asp:Label ID="lblUserData" runat="server" Text="Label"></asp:Label>
        </div>
    </form>
</body>
</html>
```

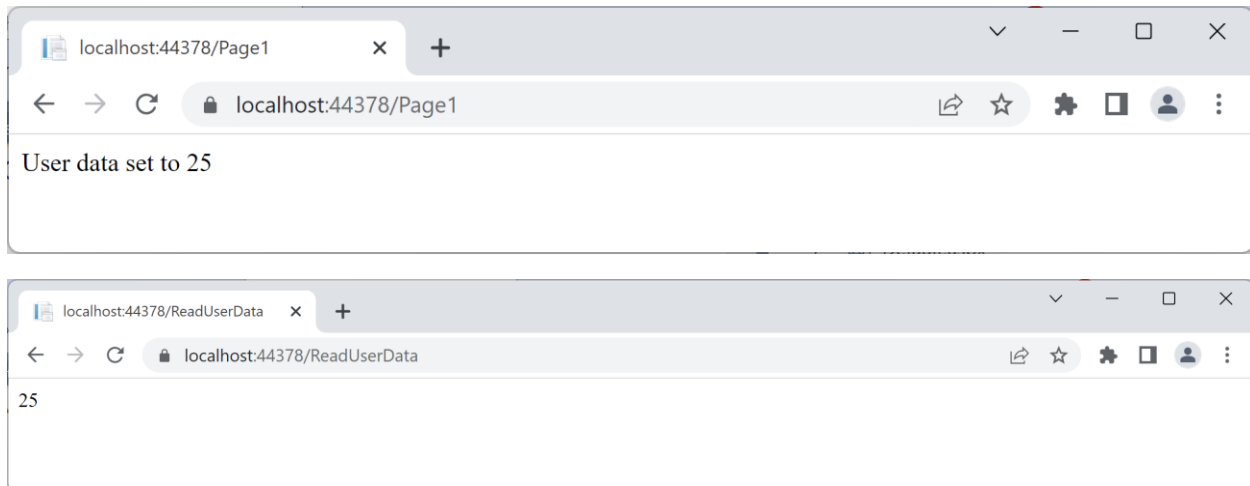
Type the following code in the Page_Load event of the ReadUserData.aspx.cs file.

```
protected void Page_Load(object sender, EventArgs e)
{
    lblUserData.Text = ((int)Session["UserData"]).ToString();
}
```

Add links to the Page1 and ReadUserData in the Site.Master file as (lines added are shown in bold).

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a runat="server" href="/">Home</a></li>
        <li><a runat="server" href="/About">About</a></li>
        <li><a runat="server" href="/Contact">Contact</a></li>
        <li><a runat="server" href="/Page1">Page1</a></li>
        <li><a runat="server" href="/ReadUserData">User Data</a></li>
    </ul>
</div>
```

If you run the program (Debug-> Start without Debugging) and click on Page1 menu, followed by User Data (hit the back button to see the menu), it will correctly report the value contained in Session["Userdata"] as:



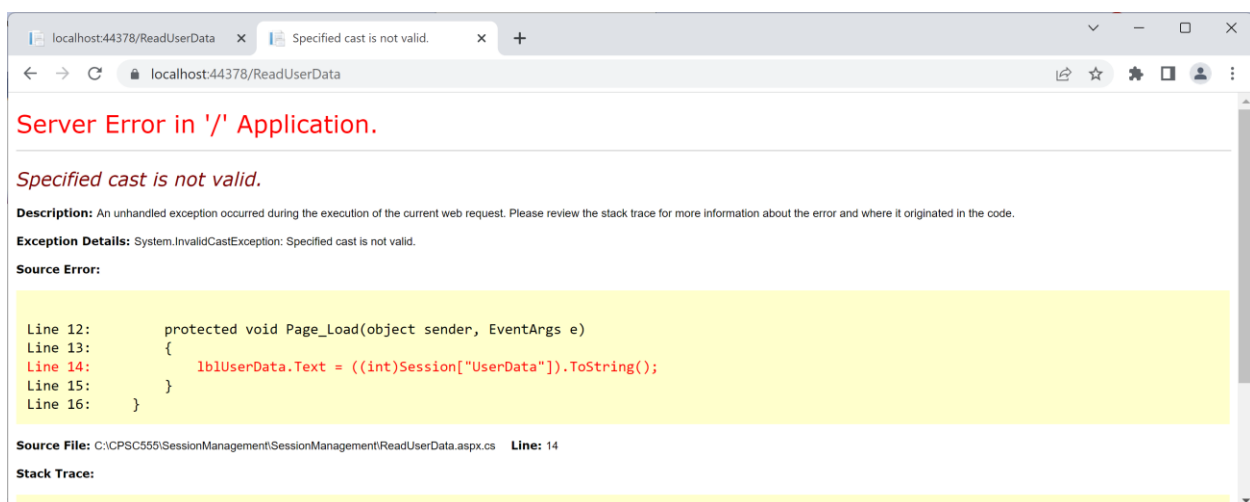
However, if some one adds a page called Page2.aspx with the following code in the Page_Load event:

```
protected void Page_Load(object sender, EventArgs e)
{
    Session["Userdata"] = true;
}
```

Add a link to Page 2 in Site.Master,

```
<li><a runat="server" href="/Page2">Page2</a></li>
```

Now if you trigger Page1.aspx followed by Page2.aspx, followed by PageReadUserData.aspx, you will get the following run time error.



This is because when we trigger Page2, the data type of the UserData session variable is changed to Boolean. The ReadUserData is type casting the UserData to an int and thus the error.

Proper solution to the storage and access of Session object is to create a SessionFacade so that data types of session variables and the key names are encapsulated as properties in the Façade class resulting in error free development. Add a folder called Utils to the project. Then add a class called SessionFacade to the Utils folder of your project with the following code in it.

```
public class SessionFacade
{
    SessionFacade()
    {
    }

    // declare session constants for the key names
    const string USERNAME = "USERNAME";
    const string ACCOUNTNUM = "ACCOUNTNUM"; // integer storage
    const string LASTLOGIN = "LASTLOGIN"; // datetime

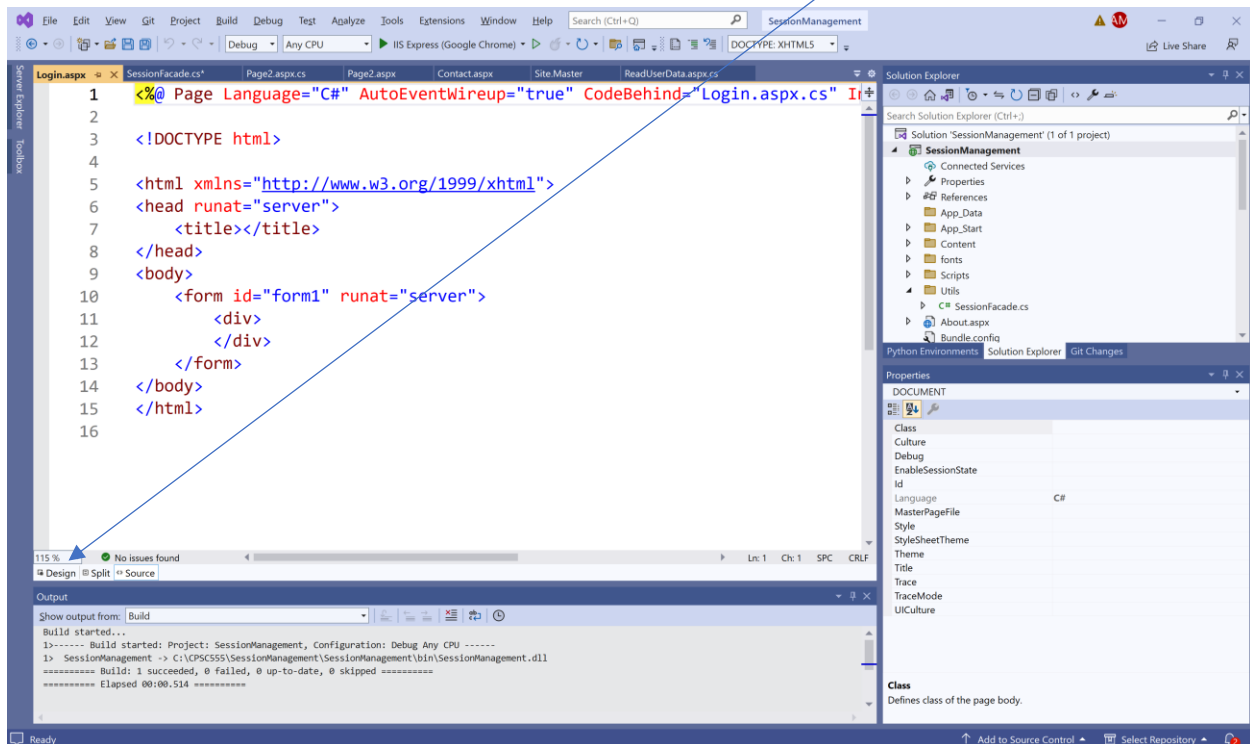
    public static string UserName
    {
        get
        {
            if (HttpContext.Current.Session[USERNAME] != null)
                return (string)HttpContext.Current.Session[USERNAME];
            else
                return null;
        }
        set
        {
            HttpContext.Current.Session[USERNAME] = value;
        }
    }

    public static int? Account
    {
        get
        {
            if (HttpContext.Current.Session[ACCOUNTNUM] != null)
                return (int)HttpContext.Current.Session[ACCOUNTNUM];
            else
                return null;
        }
        set
        {
            HttpContext.Current.Session[ACCOUNTNUM] = value;
        }
    }

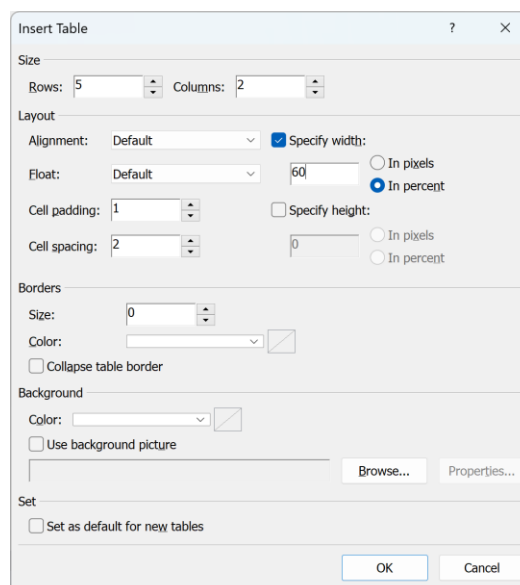
    public static DateTime LastLogin
    {
        get
        {
            if (HttpContext.Current.Session[LASTLOGIN] != null)
                return (DateTime)HttpContext.Current.Session[LASTLOGIN];
            else
                return DateTime.MinValue; // or use nullable
        }
        set
        {
            HttpContext.Current.Session[LASTLOGIN] = value;
        }
    }
}
```

As you can see from the above code, it demonstrates use of three session variables USERNAME, ACCOUNTNUM, LASTLOGIN with storage data types of string, int and DateTime respectively. The code in the properties does proper type checking before type casting it to the data type. The end result is that now we can use these session variables in any page in a safe manner. As an example of use of these session variables, add a web form to the project called Login.aspx with the following user interface and code in it.

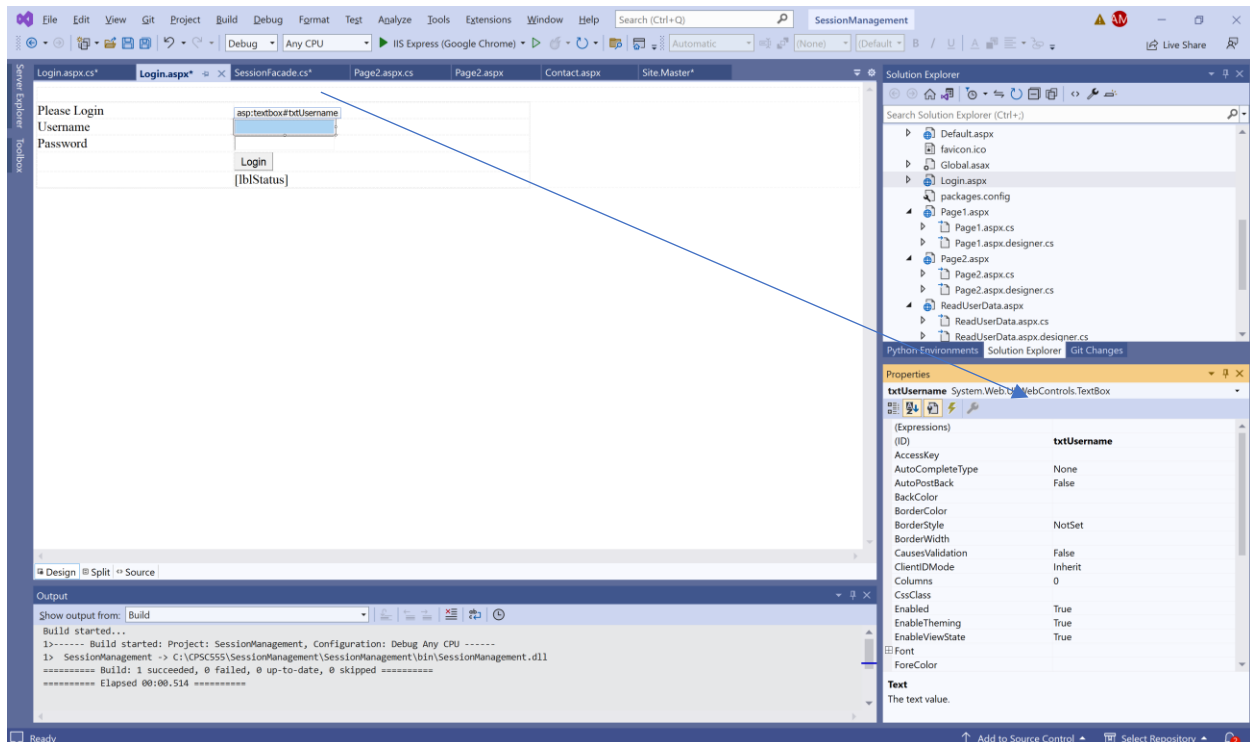
To create the User interface for the Login.aspx page, switch to the design view.



Then from the Table menu, choose Table->Insert Table, and add a table with 5 rows and 2 columns as shown below.



Then drag and drop from the tool box, two text boxes, a button, and a label as shown below. Give an ID of txtUsername to the first text box, an ID of txtPassword to the second text box, btnLogin to the button and lblStatus to the label. The button has a text property of Login. Clear the text property for the label.



Double click on the Login button and type the following code in its button handler.

```
protected void btnLogin_Click(object sender, EventArgs e)
{
    if ((txtUsername.Text == "bill") &&
        (txtPassword.Text == "bill100"))
    {
        lblStatus.Text = "welcome " + txtUsername.Text + "..";
        SessionFacade.LastLogin = DateTime.Now;
        SessionFacade.UserName = txtUsername.Text;
        SessionFacade.Account = 1234; // check DB and populate it
    }
}
```

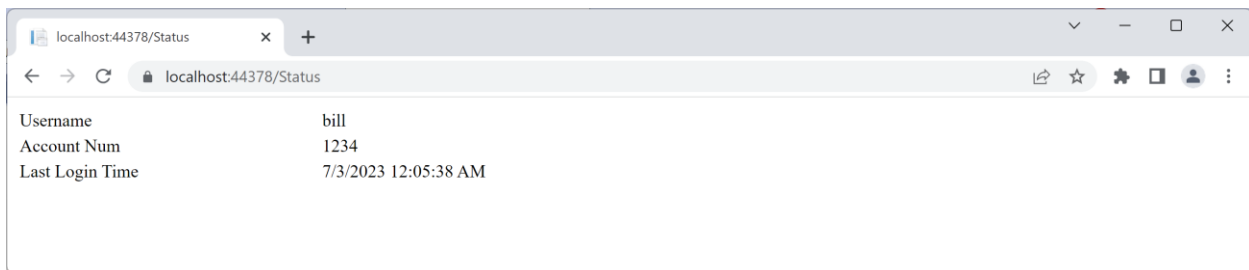
Add a web form called Status.aspx to the project. Put four labels in the Status.aspx with IDs of lblUsername, lblAccountNum, lblLastLogin, and lblStatus. The labels are put in a table with four rows and two columns. Clear the text property of the four labels.

Username	[lblUsername]
Account Num	[lblAccountNum]
Last Login Time	[lblLastLogin]
	[lblStatus]

Type the following code in the Page_Load event of Status.aspx.cs page.

```
protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        lblUsername.Text = SessionFacade.UserName;
        lblAccountNum.Text = SessionFacade.Account.ToString();
        lblLastLogin.Text = SessionFacade.LastLogin.ToString();
    }
    catch (Exception ex)
    {
        lblStatus.Text = ex.Message;
    }
}
```

Add a link to the Login page and the Status page in the Site.Master file. You can test the use of the SessionFacade by asking for Login.aspx page and providing a username, password of bill, bill100 respectively, and then asking for Status page. The output will appear as:



The screenshot shows a web browser window with the address bar displaying 'localhost:44378/Status'. The page content is a table with three rows and two columns, showing the status of a user named 'bill'.

Username	bill
Account Num	1234
Last Login Time	7/3/2023 12:05:38 AM

Singleton Pattern

The purpose of the Singleton pattern is to guarantee that there is only a single instance of the class available, and to provide an access to this single instance. It is easily achieved by declaring the constructor private, by including an instance of itself, by providing a GetObject method, or Instance property.

```
class MySingleton
{
    static MySingleton instance = null;

    MySingleton(){} // private constructor

    public static MySingleton Instance
    {
        get
        {
            if (instance == null)
                instance = new MySingleton();
            return instance;
        }
    }
}
```

The best implementation in C# as shown by the following MySingleton3 class.

```
class MySingleton3 // best approach in C#
{
    static readonly MySingleton3 instance = new MySingleton3();
    // static members are lazily initialized.
    // readonly allows thread-safety

    MySingleton3() { }

    public static MySingleton3 Instance
    {
        get
        {
            return instance;
        }
    }
}
```

Note that the Singleton class can contain other methods and properties.

For example, Facade is often implemented as Singleton because only one Facade object is needed.

Exercise: Implement MortgageFacade as a Singleton.

Solution: Add a class called MortgageFacadeSingleton to the FacadePattern project with the following code in it:

```

internal class MortgageFacadeSingleton
{
    Bank bk = new Bank();
    Loan ln = new Loan();
    Credit cr = new Credit();

    static readonly MortgageFacadeSingleton instance = new
MortgageFacadeSingleton();
    public static MortgageFacadeSingleton Instance
    {
        get { return instance; }
    }
    MortgageFacadeSingleton() { } // private constructor
    public bool MortgageApproval(MortgageApplicant mapp, ref string reason)
    {
        bool approved = false;
        double bal = bk.GetCurrentCheckingBalance(mapp.BankAccountNum);
        bal = bal + bk.GetSavingBalance(mapp.BankAccountNum);

        bal = bal - ln.GetOutstandingLoans(mapp.LoanAccountNum);
        if (bal < 0)
        {
            approved = false;
            reason = "negative balances ";
        }
        else
        {
            if (bal < 0.2 * mapp.LoanAmountAsked)
            {
                approved = false;
                reason = "Not enough balances";
            }
            else
            {
                if (cr.CheckCredit(mapp.SSNum) < CreditRating.GOOD)
                {
                    approved = false;
                    reason = "not good credit rating..";
                }
                else
                {
                    approved = true;
                }
            }
        }
        return approved;
    }
}

```

Modify the button handler code in the form class to appear as:

```

private void btnMortgageApproval_Click(object sender, EventArgs e)
{
    string reason = "";
    MortgageApplicant mapp = new MortgageApplicant
    {
        FirstName = "Bill",
        LastName = "Baker",
        BankAccountNum = 12345,

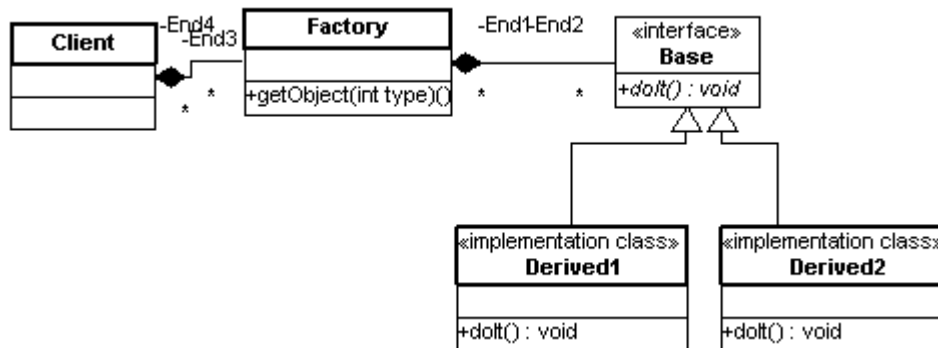
```

```
        LoanAccountNum = 12345,  
        SSNum = "111-22-3333",  
        LoanAmountAsked = 150000  
    };  
  
    //MortgageFacade mf = new MortgageFacade();  
    //bool approved = mf.MortgageApproval(mapp, ref reason);  
    MortgageFacadeSingleton mf = MortgageFacadeSingleton.Instance;  
    bool approved = mf.MortgageApproval(mapp, ref reason);  
    if (approved == true)  
        MessageBox.Show("Your application has been approved..");  
    else  
        MessageBox.Show("Your application has been denied..\n" +  
            reason);  
}
```

Run the project. It should work same as before.

Factory Pattern

The purpose of the factory pattern is to provide creation of a class object from a possible set of classes that implement a particular interface, or inherit from a particular abstract base class. The factory class typically provides a GetObject method (or an equivalent of it) where the caller can specify a class type whose object needs to be created. When we call the methods of the class object, we do not know which derived class is servicing our call. The object creation in factory pattern occurs in a single place i.e., the Factory class. A Factory class can contain more than one Factory methods which can be either static or non-static. The UML diagram for the factory pattern is shown below.



Main idea in the Factory method pattern is to isolate the client from the details of which appropriate class object should be created from a set of classes that implement a particular interface.

To demonstrate the Factory method pattern, create a new windows form application project called "FactoryPattern". Add an interface to it called ICheckWeather with the following code in it.

```

interface ICheckWeather
{
    double GetCurrentTemperature(string zipcode);
    string CheckNextDayForecast(string city, string state);
}
  
```

Add a class called WeatherNOAA which implements the above interface. The code for the class is shown below.

```

class WeatherNOAA : ICheckWeather
{
    public double GetCurrentTemperature(string zipcode)
    {
        // contact the web service to find current Temperature
        double temp = 0;
        switch (zipcode)
        {
            case "99164": temp = 67;
                break;
            case "06484": temp = 57;
                break;
            case "06601": temp = 73;
                break;
            case "99352": temp = 81;
                break;
        }
    }
}
  
```



```

        default: temp = 50;
            break;
    }
    return temp;
}

public string CheckNextDayForecast(string city, string state)
{
    string res = "";
    if ((city.ToUpper() == "AUSTIN") && (state.ToUpper() == "TX"))
    {
        res = "partly cloudy with a high of 78";
    }
    else if ((city.ToUpper() == "Topeka") && (state.ToUpper() ==
"KS"))
    {
        res = "thunder storms in the afternoon with a high of 71";
    }
    return res;
}
}

```

Add another class called WeatherYahoo which also implements the ICheckWeather interface with the following code. Assume Yahoo is providing a more detailed and accurate weather information.

```

class WeatherYahoo : ICheckWeather
{
    public double GetCurrentTemperature(string zipcode)
    {
        // contact the web service to find current Temperature
        double temp = 0;
        switch (zipcode)
        {
            case "99164": temp = 66.3;
                break;
            case "06484": temp = 58.5;
                break;
            case "06601": temp = 72.73;
                break;
            case "99352": temp = 87.9;
                break;
            default: temp = 50;
                break;
        }
        return temp;
    }

    public string CheckNextDayForecast(string city, string state)
    {
        string res = "";
        if ((city.ToUpper() == "AUSTIN") && (state.ToUpper() == "TX"))
        {
            res = "mix of sun and clouds with a high of 78, winds 1015
mph";
        }
        else if ((city.ToUpper() == "Topeka") && (state.ToUpper() ==
"KS"))
        {
            res = "thunder showers early with a high of 71, winds 5-10
mph";
        }
    }
}

```

```

    }
    return res;
}
}

```

Now add a WeatherFactory class with the following code. The create method in the factory either creates an object of WeatherNOAA or WeatherYahoo depending upon if an accurate or an approximate weather information is needed.

```

enum ForecastAccuracy : int
{
    GOOD,
    FAIR
}

class WeatherFactory
{
    public static ICheckWeather CreateWeatherInfo(ForecastAccuracy wtype)
    {
        ICheckWeather icw = null;
        if (wtype == ForecastAccuracy.FAIR)
        {
            icw = new WeatherNOAA();
        }
        if (wtype == ForecastAccuracy.GOOD)
        {
            icw = new WeatherYahoo();
        }
        return icw;
    }
}

```

To test the above Factory method pattern, add a button to the form called btnWeatherFactory with the following code in its button handler.

```

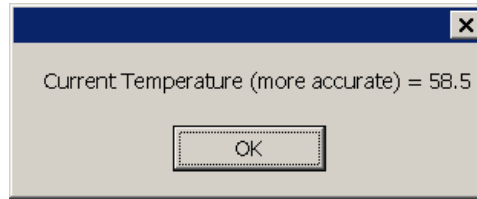
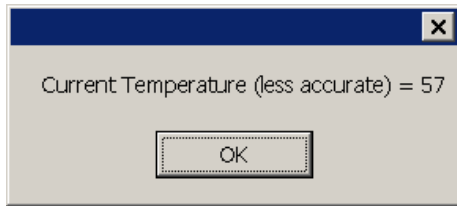
private void btnWeatherFactory_Click(object sender, EventArgs e)
{
    ICheckWeather icw =
WeatherFactory.CreateWeatherCheck(ForecastAccuracy.FAIR);
    double temp = icw.GetCurrentTemperature("06484");
    MessageBox.Show("Current Temperature (less accurate) = " +
        temp.ToString());

    ICheckWeather icw2 =
WeatherFactory.CreateWeatherCheck(ForecastAccuracy.GOOD);
    double temp2 = icw2.GetCurrentTemperature("06484");
    MessageBox.Show("Current Temperature (more accurate) = " +
        temp2.ToString());
}

```

As you can see from the above test code, the client is isolated from the details of which class is actually providing the weather information.

The output by running the above code appears as:



Factory pattern is also useful for maintaining multiple versions of a class. To demonstrate this, suppose you had created a class called `MyCompute` that implements an `ICompute` interface as shown below.

```
interface ICompute
{
    double ComputeCylindervolume(float radius, float height);
    double ComputeSphereVolume(float radius);
}

class MyCompute : ICompute // older version
{
    public double ComputeCylindervolume(float radius, float height)
    {
        return 2 * 3.1415 * radius * height ;
    }

    public double ComputeSphereVolume(float radius)
    {
        return 4/3.0*3.1415*radius*radius*radius;
    }
}
```

Now suppose after some time, we developed a more accurate version of the above `MyCompute` class and called it `Computev2` with the code as shown below.

```
class MyComputev2 : ICompute // newer version
{
    public double ComputeCylindervolume(float radius, float height)
    {
        return 2 * Math.PI * radius * height;
    }

    public double ComputeSphereVolume(float radius)
    {
        return 4 / 3.0 * Math.PI * radius * radius * radius;
    }
}
```

To make sure that existing code continues to use the `MyCompute` class while the new code uses `MyComputev2` class, we can create a `ComputeFactory` class and an enumeration with the following code.

```
enum CVersion : int
{
    v1_1,
    v2_0
}

class ComputeFactory
{

```

```

public static ICompute CreateComputeObject(CVersion ver)
{
    if (ver == CVersion.v1_1)
        return new MyCompute();
    else if (ver == CVersion.v2_0)
        return new MyComputev2();
    else
        throw new Exception("Unsupported version");
}
}

```

To test the above factory, add a button to the form with a name of btnComputeFactory with the following code in its handler.

```

private void btnComputeFactory_Click(object sender, EventArgs e)
{
    ICompute ic = ComputeFactory.CreateComputeObject(CVersion.v1_1);
    double res1 = ic.ComputeCylinderVolume(4.5f, 3.7f);
    MessageBox.Show("Result from old version = " +
        res1.ToString());

    ICompute ic2 = ComputeFactory.CreateComputeObject(CVersion.v2_0);
    double res2 = ic2.ComputeCylinderVolume(4.5f, 3.7f);
    MessageBox.Show("Result from new version = " +
        res2.ToString());
}

```

Running the program and clicking on the above button produces the following output.

