CPSC 501 – Assignment #6

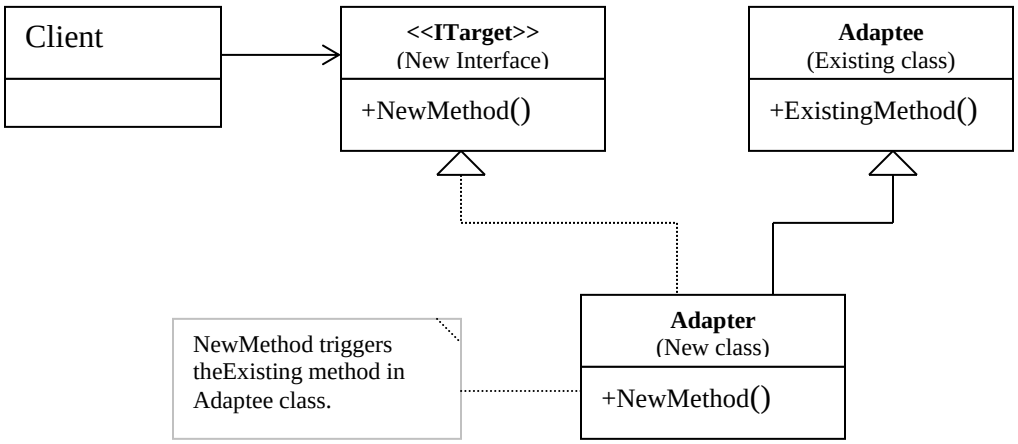
Adapter, Bridge Patterns, MySql and Sql Server Databases

Adapter Pattern:

The goal of adapter pattern is to provide a new interface to an existing class such that the caller can call on the existing methods of the class via the new interface. There are different variations of the Adapter patter, such as interface adapter, object adapter and two-way adapter.

Interface Adapter:

One application of adapter pattern is known as "*interface adapter*" where we are able to call methods in an existing class via a newly required interface without changing the existing class. In this type of pattern, the approach is to define the new interface, and then create the adapter class that inherits from the existing class (adaptee) and also implements the new interface. This is indicated by the following UML diagram.



Create a windows forms type of application using Visual Studio called AdapterPattern. Add a class called ExistingAdaptee with the following code in it. This class represents existing code that needs to be made available via new interface without having to modify or rewrite the existing code. Suppose this class was written many years ago and the methods take a fixed number of inputs as floats. We will like to change the way these methods are called.

```
class ExistingMathAdaptee
{
    public double ComputeAvg(float a, float b, float c)
    {
        double sum = (a + b + c) ;
        return sum / 3.0;
    }

    public double ComputeAvg(float a, float b, float c, float d)
    {
        double sum = (a + b + c + d);
        return sum / 4.0;
    }

    public double FindMin(float a, float b, float c)
```

```
        {
            double min = a;
            if (b < min)
                min = b;
            if (c < min)
                min = c;
            return min;
        }

        public double FindMin(float a, float b, float c, float d)
        {
            double min = a;
            if (b < min)
                min = b;
            if (c < min)
                min = c;
            if (d < min)
                min = d;
            return min;
        }
```

Add an interface to the project called INewMath. This is the new interface often referred to as ITarget. In our new interface, methods take inputs as array of doubles.

```
interface INewMath // ITarget, new interface for existing code
    {
        double ComputeAvgNew(double[] Arr);
        double FindMinNew(double[] Arr);
    }
```

As you can see this interface provides somewhat similar methods to the ExistingMathAdaptee class with the difference that the methods in this interface take an array of doubles as a parameter as opposed to individual float parameters for the ComputeAvg and the FindMin methods.

Add a class called AdapterMath that will act as the wrapper class for the existing class (adaptee). Note that the AdapterMath class implements the INewMath and also inherits from the ExistingMathAdaptee class. The inheritance is done so that this class has access to the existing code. Recall that the goal of adapter is to use the existing code, but provide a new interface to it.

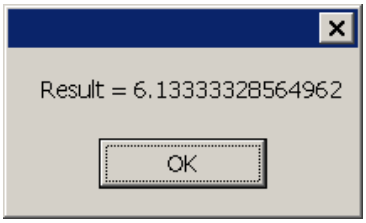
```
class AdapterMath : ExistingMathAdaptee, INewMath
    {

        public double ComputeAvgNew(double[] Arr)
        {
            if (Arr.Length == 3)
                return ComputeAvg((float)Arr[0], (float)Arr[1],
(float)Arr[2]);
            else if (Arr.Length == 4)
                return ComputeAvg((float)Arr[0], (float)Arr[1],
(float)Arr[2], (float)Arr[3]);
            else
                throw new Exception("Array size is not currently supported
for ComputeAvg");
        }
```

```
        public double FindMinNew(double[] Arr)
        {
            // exercise - try writing this code yourself
            throw new NotImplementedException();
        }
    }
```

Add a button to the form with a name of btnAdapterMath. Double click on the button and type the following code in the button handler.

```
    private void btnAdapterMath_Click(object sender, EventArgs e)
        {
            try
            {
                INewMath am = new AdapterMath();
                double[] A = { 3.7, 8.4, 6.3 };

                double res = am.ComputeAvgNew(A);
                MessageBox.Show("Result = " + res.ToString());
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

Build and test the code. Once you click on the above button, you should see an output as:



Note: An adapter can add extra behavior between the ITarget interface and the Adaptee. For example, what if ITarget needed to add another method called ComputeAvg that returns an rounded integer for the result.
Modify the INewMath interface to appear as:

```
interface INewMath // ITarget, new interface for existing code
    {
        double ComputeAvgNew(double[] Arr);
        int ComputeAvgRound(double[] Arr);
        double FindMinNew(double[] Arr);
    }
```

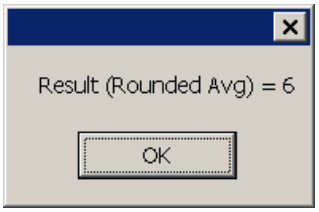
Modify the AdapterMath class to provide include the code for the ComputeAvgRound method.

```
        public int INewMath.ComputeAvgRound(double[] Arr)
        {
            if (Arr.Length == 3)
                return (int)Math.Round(ComputeAvg((float)Arr[0],
(float)Arr[1], (float)Arr[2]));
            else if (Arr.Length == 4)
                return (int)Math.Round(ComputeAvg((float)Arr[0],
(float)Arr[1], (float)Arr[2], (float)Arr[3]));
            else
                throw new Exception("Array size is not currently supported
for ComputeAvg");
        }
```

Add another button to the form called btnAdapterMath2 with the following code in the button handler.

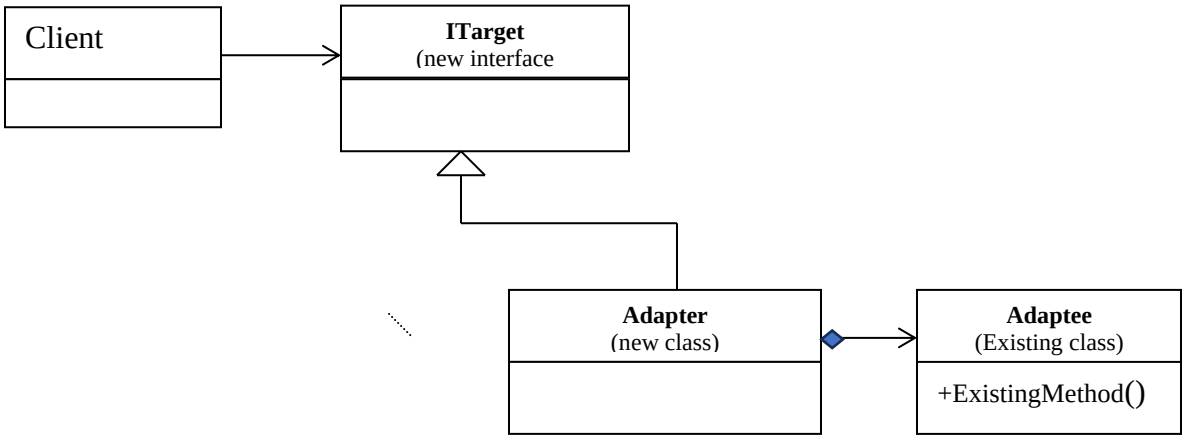
```
        private void btnAdapterMath2_Click(object sender, EventArgs e)
        {
            try
            {
                INewMath am = new AdapterMath();
                double[] A = { 3.7, 8.4, 6.3 };

                int res = am.ComputeAvgRound(A);
                MessageBox.Show("Result (Rounded Avg) = " + res.ToString());
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

Run the program and test by clicking on the above button. You will see the following output.



Object Adapter:

Instead of inheriting from the existing class, we can use composition (i.e., create an object of the Adaptee class) as shown below. This design is often referred to as "Object Adapter".



To show an implementation example of the object adapter, add an interface called "IMathTarget" to the project with the following code:

```
interface MathTarget
{
    double ComputeAvgNew(double[] Arr);
    int ComputeAvgRound(double[] Arr);
    double FindMinNew(double[] Arr);
}
```

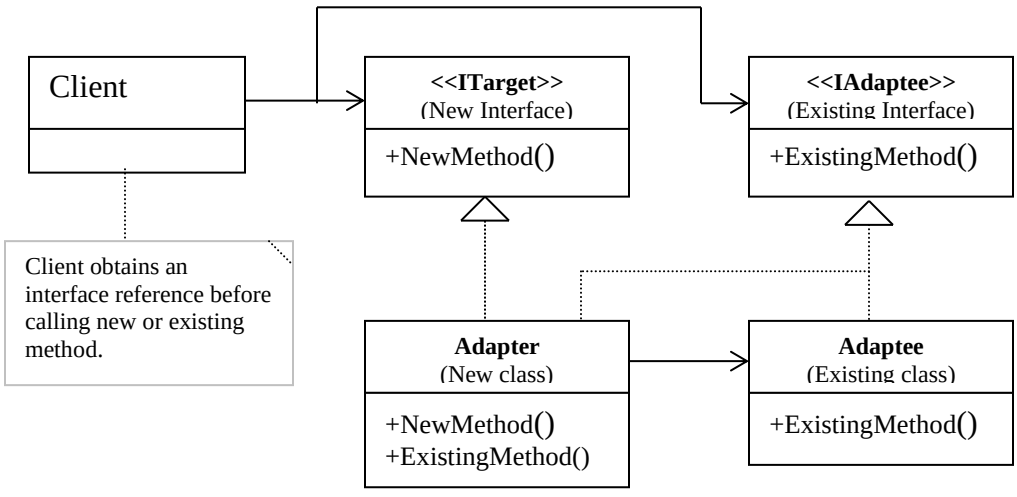
Add a class to the project called MathAdapter2 with the following code.

```
class AdapterMath2 : IMathTarget
{
    ExistingMathAdaptee ema = null;

    public AdapterMath2()
    {
        ema = new ExistingMathAdaptee();
    }

    public double ComputeAvgNew(double[] Arr)
    {
        if (Arr.Length == 3)
            return ema.ComputeAvg((float)Arr[0], (float)Arr[1],
(float)Arr[2]);
        else if (Arr.Length == 4)
            return ema.ComputeAvg((float)Arr[0], (float)Arr[1],
(float)Arr[2], (float)Arr[3]);
        else
            throw new Exception("Array size is not currently supported
for ComputeAvg");

    }
```

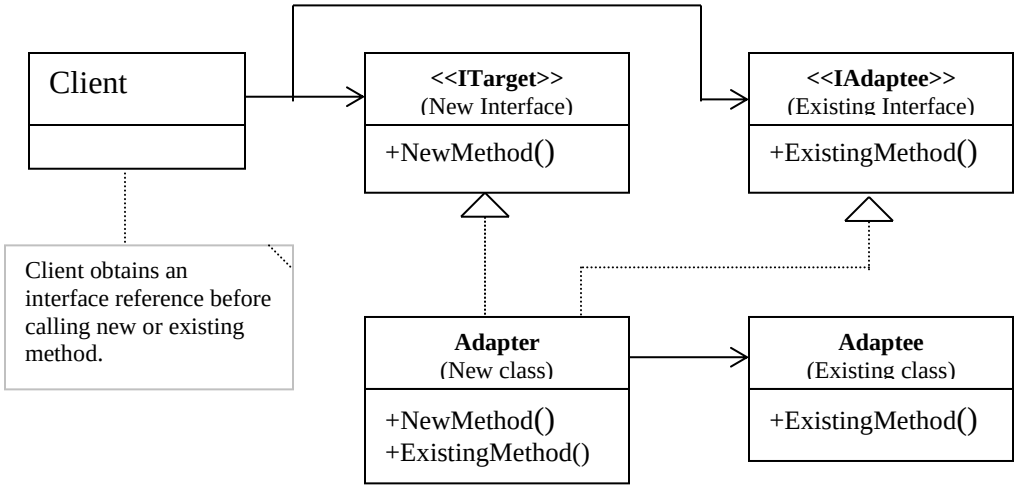
```
        public int ComputeAvgRound(double[] Arr)
        {
            if (Arr.Length == 3)
                return (int)Math.Round(ema.ComputeAvg((float)Arr[0],
(float)Arr[1], (float)Arr[2]));
            else if (Arr.Length == 4)
                return (int)Math.Round(ema.ComputeAvg((float)Arr[0],
(float)Arr[1], (float)Arr[2], (float)Arr[3]));
            else
                throw new Exception("Array size is not currently supported
for ComputeAvg");
        }

        public override double FindMinNew(double[] Arr)
        {
            // to do - your implementation
            throw new NotImplementedException();
        }

    }
```

As you can see from the above MathAdapter2 class code, it contains an object of the ExistingMathAdaptee class, and implements the IMathTarget interface. When one of the methods in the IMathTarget is invoked, the MathAdpater2 class delegates the call to the appropriate method in the contained ExistingMathAdaptee object.

Two-way Adapter: Two way adapters implement interfaces of both the target and the adaptee. This way, such an adapter can be used in new systems or keeping the compatibility with older systems, as existing interface is also maintained.



Recall that Adapter pattern is mostly used when we have existing class (Adaptee) that needs to be adapted to new requirements. If the existing class does not implement an interface, then the above UML changes to:



Before creating the adapter, we will create the IAdaptee and ITarget interfaces, and the adapter class will delegate the calls to the adaptee class via the contained object.

Add an interface called IExistingMath matching to the signatures of the methods in the ExistingMathAdaptee class, as shown below.

```
interface IExistingMath
{
    double ComputeAvg(float a, float b, float c);
    double ComputeAvg(float a, float b, float c, float d);
    double FindMin(float a, float b, float c);
    double FindMin(float a, float b, float c, float d);
}
```

Add a class called ExistingMathAdapteeDerived with the following code in it:

```
class ExistingMathAdapteeDerived : ExistingMathAdaptee
    {
        public float FindMax(float a, float b, float c)
        {
            float max = a;
            if (b > max)
                max = b;
            if (c > max)
                max = c;
            return max;
        }
    }
```

Add a class called MathAdapterTwoWay with the following code in it.
As you can see, this class implements both the existing and the new Math interfaces, and contains objects of the ExistingMathAdaptee and the ExistingMathAdapteeDerived class.

```
    class MathAdapterTwoWay : INewMath, IExistingMath
    {
        ExistingMathAdaptee ema = new ExistingMathAdaptee();
        ExistingMathAdapteeDerived emad = new ExistingMathAdapteeDerived();
```

```
        public double ComputeAvg(float a, float b, float c)
        {
            return ema.ComputeAvg(a,b,c);
        }

        public double ComputeAvg(float a, float b, float c, float d)
        {
            return ema.FindMin(a,b,c,d);
        }

        public double FindMin(float a, float b, float c)
        {
            return ema.FindMin(a,b,c);
        }

        public double FindMin(float a, float b, float c, float d)
        {
            return ema.FindMin(a,b,c,d);
        }

        public double ComputeAvgNew(double[] Arr)
        {
            if (Arr.Length == 3)
                return ema.ComputeAvg((float)Arr[0], (float)Arr[1], (float)Arr[2]);
            else if (Arr.Length == 4)
                return ema.ComputeAvg((float)Arr[0], (float)Arr[1],
(float)Arr[2], (float)Arr[3]);
            else
                throw new Exception("Array size is not currently supported
for ComputeAvg");
        }

        public int ComputeAvgRound(double[] Arr)
        {
            if (Arr.Length == 3)
                return (int)Math.Round(ema.ComputeAvg((float)Arr[0],
(float)Arr[1], (float)Arr[2]));
            else if (Arr.Length == 4)
                return (int)Math.Round(ema.ComputeAvg((float)Arr[0],
(float)Arr[1], (float)Arr[2], (float)Arr[3]));
            else
                throw new Exception("Array size is not currently supported
for ComputeAvg");

        }

        public double FindMinNew(double[] Arr)
        {
            // to do - you will implement this
            throw new NotImplementedException();
        }

        public double FindMax(double[] Arr)
        {
            if (Arr.Length == 3)
                return emad.FindMax((float)Arr[0], (float)Arr[1], (float)Arr[2]);
            else
                throw new Exception("Array size > 3 not supported");
        }
    }
```

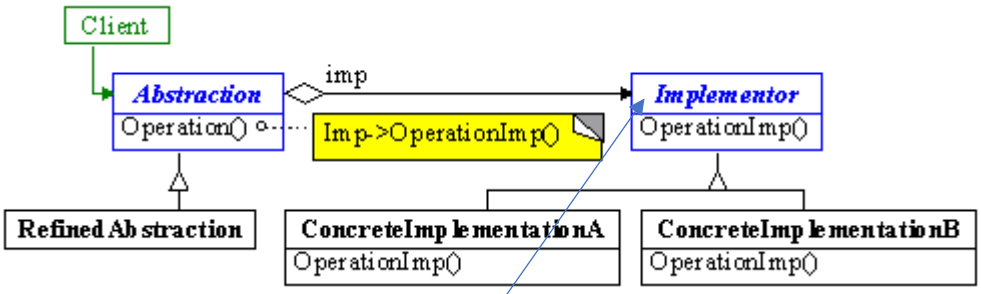
To test the above two way adapter, add a button to the form with the name of btnAdapter2Way. Write the following code in the button handler.

```
private void btnAdapter2Way_Click(object sender, EventArgs e)
        {
            IExistingMath im = new MathAdapterTwoWay();
            double res = im.ComputeAvg(7.3f,6.4f,8.5f);
            MessageBox.Show("Avg from legacy ComputeAvg = " + res.ToString());

            INewMath inm = new MathAdapterTwoWay();
            double []Arr = {7.3f,6.4f,8.5f};
            double res2 = inm.ComputeAvgNew(Arr);
            MessageBox.Show("Result from new ComputeAvg = " + res2.ToString());
        }
```

As you can see from the above test code, our two way adapter class "MathAdapterTwoWay" can be easily used in existing clients as well as in the new environment.

Bridge Pattern:

The bridge pattern is useful in situations where an existing code needs to be invoked with a particular version, without requiring the calling code to change.
The following UML diagram indicates the concepts of the bridge pattern.



The client interacts via the abstraction which may have similar methods as the Implementor interface, Additional methods can be added on the client's view. Since the Bridge does aggregation on the Implementation interface, when we call methods in the Bridge, the call is delegated to the Implementation that is plugged into the Bridge.

To demonstrate the bridge pattern, create a windows forms project called "BridgePattern". Add an interface called IBridge to the project with the following code.

```
interface IBridge  // implementor interface
{
    string OperationImp();
    string AnotherOperationImp(string msg);
}
```

Add a class to the project called *ImplementationA* that implements the *IBridge* with the following code.

```
class ImplementationA : IBridge
```

```
{

    public string OperationImp()
    {
        return "result from ImplementationA operation";
    }

    public string AnotherOperationImp(string msg)
    {
        return "Greetings " + msg;
    }
}
```

Add another class to the project called *ImplementationB* that implements the *IBridge* with the following code.

```
class ImplementationB : IBridge
{

    public string OperationImp()
    {
        return "result from ImplementationB operation"; ;
    }

    public string AnotherOperationImp(string msg)
    {
        return "GreetingsB and Welcome " + msg;
    }
}
```

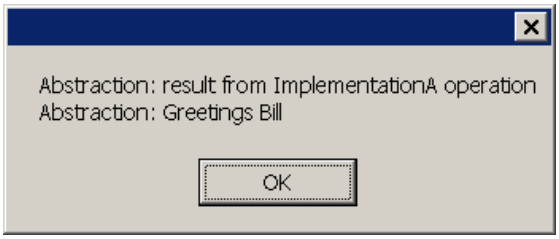
Add a class called Abstraction (client's view) with the following code:

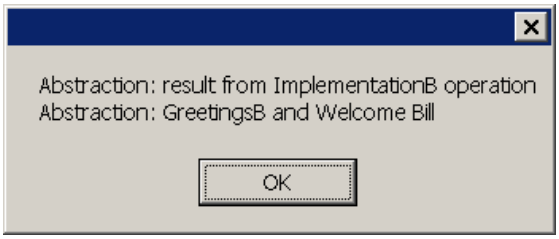
```
class Abstraction
{
    protected IBridge bridge;
    public Abstraction(IBridge br) // client's view
    {                               // uses aggregation
        bridge = br;
    }

    public string Operation()
    {
        return "Abstraction: " + bridge.OperationImp();
    }

    public string OperationGreet(string nm)
    {
        return "Abstraction: " + bridge.AnotherOperationImp(nm);
    }
}
```

Add a button to the form with a name of "btnBridgeBasic" with the following code in its handler for testing the Bridge pattern.

```
private void btnBridgeBasic_Click(object sender, EventArgs e)
        {
            Abstraction abs = new Abstraction(new ImplementationA());
            string res1 = abs.Operation();
            string res2 = abs.OperationGreet("Bill");
            MessageBox.Show(res1 + "\n" + res2);

            // switch to a different implementation
            abs = new Abstraction(new ImplementationB());
            res1 = abs.Operation();
            res2 = abs.OperationGreet("Bill");
            MessageBox.Show(res1 + "\n" + res2);
        }
```

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





Adding more methods on the client side (Abstraction) without affecting the Implementation:

Since the purpose of the Bridge pattern is to decouple the abstraction from its implementation, we can add more methods to the abstraction (via inheritance from the abstraction class) and invoking the IBridge methods via the contained bridge reference in the Abstraction. To demonstrate this, add a class to the project called RefinedAbstraction with the following code.

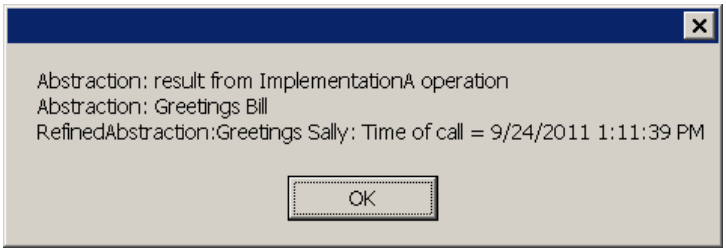
```
    class RefinedAbstraction : Abstraction
    {
        public RefinedAbstraction(IBridge br)
            : base(br)
        {
        }

        public string OperationGreetTime(string nm)
        {
            return "RefinedAbstraction:" + bridge.AnotherOperationImp(nm) +
":" +
                " Time of call = " + DateTime.Now.ToString();
        }
    }
```

As you can see from the above code, it adds a method called OperationGreetTime to the already inherited methods from the Abstraction class (Operation and OperationGreet). The OperationGreetTime method further triggers the existing method in the implemntation via the contained bridge reference in the base class.

To test the added method in the RefinedAbstraction, add another button to the project called "btnRefinedAbstraction" with the following code in its handler.

```
private void btnRefinedAbstraction_Click(object sender, EventArgs e)
        {
            RefinedAbstraction rabs = new RefinedAbstraction(new ImplementationA());
            string res1 = rabs.Operation();
            string res2 = rabs.OperationGreet("Bill");
            string res3 = rabs.OperationGreetTime("Sally");
            MessageBox.Show(res1 + "\n" + res2 + "\n" + res3);
        }
```

If you run and test the above code, the output will appear as:



Bridge pattern provides support for switching between different implementations because of the aggregation being used in the Bridge pattern in the abstraction, allowing for easy switching between different implementations (the created implementation is passed to the constructor of the abstraction).

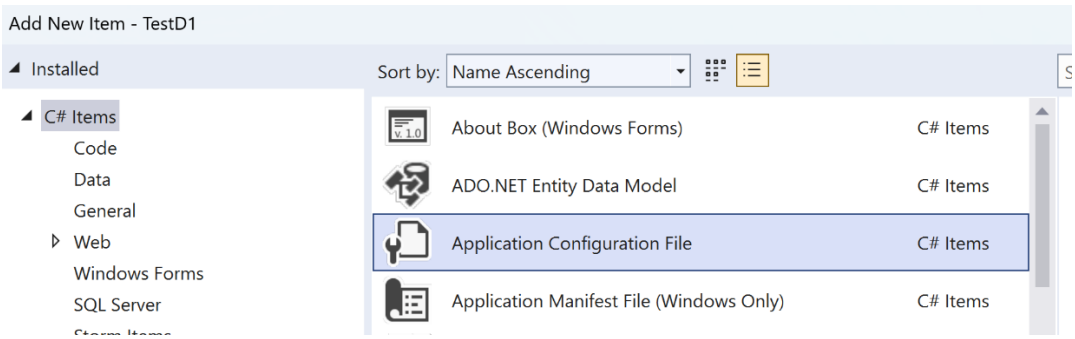
Practical Example of Bridge Pattern: Suppose you will be designing a database driven application that will need to interact with different databases such as MySql or Sql Server. If we develop the Bridge pattern properly, our application can switch to one database or the other by simply changing one line of code (which database implementation is plugged into the Abstraction).

Create a windows forms application called TestD1. Add a folder called DataLayer to the project. Add an interface to the project called IDataAccess to the DataLayer folder with the following code in it.

```
internal interface IDataAccess
{
    object GetSingleAnswer(string sql);
    DataTable GetManyRowsCols(string sql);
    int InsertUpdateDelete(string sql);
}
```

You need to create two databases: one called ProductsDB in SQL server, and another called productsdb in MySql. Refer to the handout on "Creating Databases in SQL Server" and "Creating Databases in MySql".

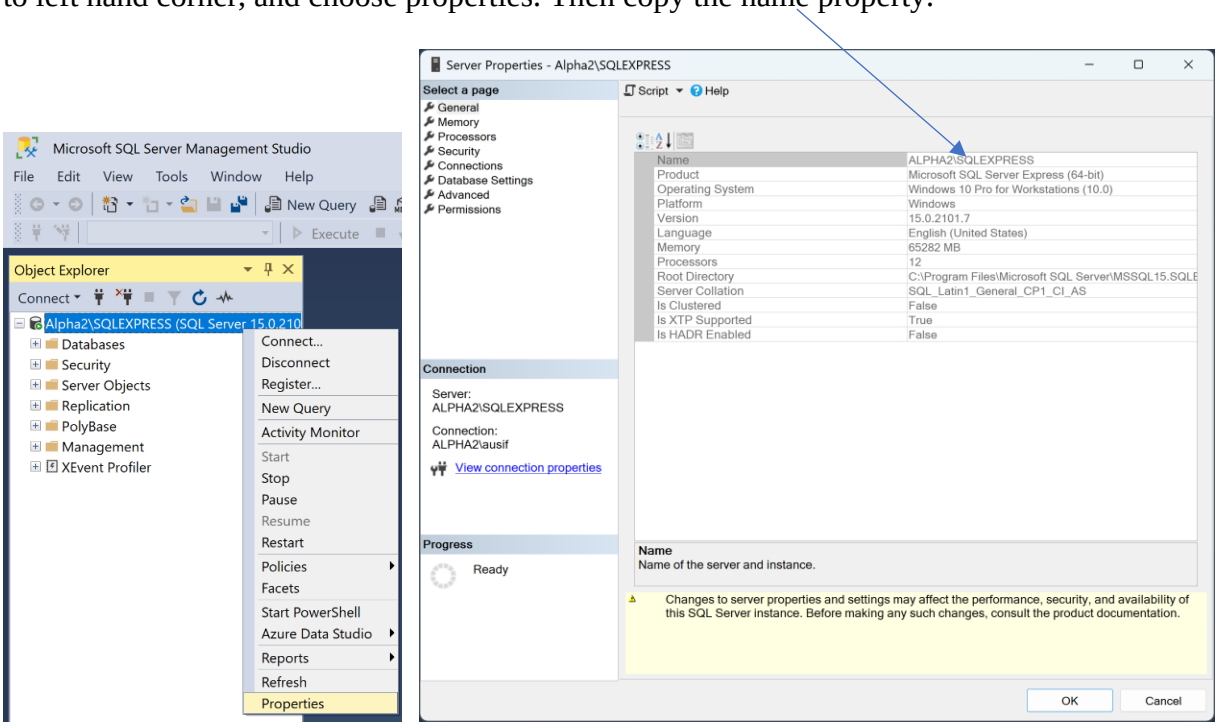
Add an application configuration file called App.config the project by right clicking on the project name and choosing "Add New Item", then choose Application Configuration.



Type the connection string information in the App.config file.

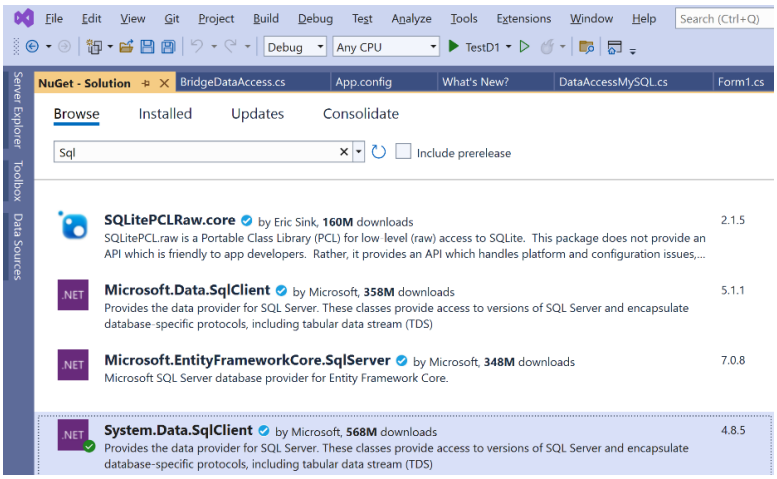
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
      <connectionStrings>
            <add name="MYDBCONN"
connectionString="server=ALPHA2\SQLEXPRESS;integrated
security=true;database=ProductsDB;"/>
            <add name="MYDBCONNMYSQL"
connectionString="server=localhost;UserID=root;Password=duportail;Database=ProductsDB;"/>
      </connectionStrings>
</configuration>
```

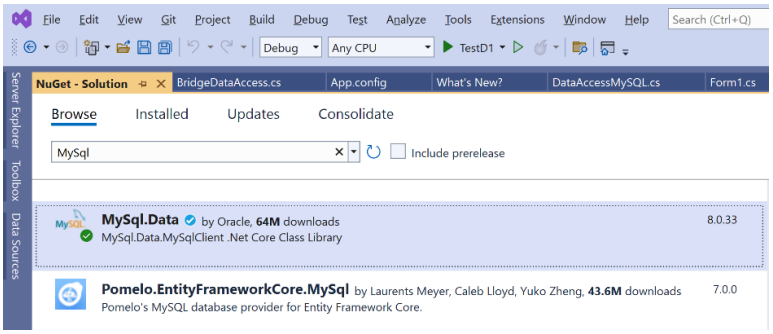
Replace the “ALPHA2\SQLExpress” with the name of your SQL server. You can find this by launching the SQL Server Management Studio, then right click on the SQL server instance in the to left hand corner, and choose properties. Then copy the name property.



From the Tools menu, choose Package Manager->Manage NuGet Packages for Solution, then search for SQL and install the System.Data.SqlClient into the project. This library contains the classes for interacting with SQL Server.

Also search for MySql.Data in the Nuget package manager, and install this package in the project. This library contains classes for interacting with the MySql database.





Add a class called DataAccess to the DataLayer with the following code in it. This class contains the code for interacting with the SQL server database.

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestD1.DataLayer
{
    internal class DataAccess : IDataAccess
    {
        string connStr =
ConfigurationManager.ConnectionStrings["MYDBCONN"].ConnectionString;
        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);
            }
            catch { throw; }
            finally { conn.Close(); }
            return dt;
        }

        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 { throw; }
            finally { conn.Close(); }
            return obj;
        }

        public int InsertUpdateDelete(string sql)
        {
            SqlConnection conn = new SqlConnection(connStr);
            int rows = 0;
            try
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                rows = cmd.ExecuteNonQuery();
            }
            catch { throw; }
            finally { conn.Close(); }
            return rows;
        }
    }
}
```

Add a class to the DataLayer folder called DataAccessMySql with the following code in it. This class has the functions to communicate with the MySql database.

```
using System;
using System.Collections.Generic;
using MySql.Data.MySqlClient;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Configuration;
using System.Data.SqlClient;

namespace TestD1.DataLayer
```

```
{
    internal class DataAccessMySQL : IDataAccess
    {
        string connStr =
ConfigurationManager.ConnectionStrings["MYDBCONNMYSQL"].ConnectionString;

        public DataTable GetManyRowsCols(string sql)
        {
            MySqlConnection conn = new MySqlConnection(connStr);
            DataTable dt = new DataTable();
            try
            {
                conn.Open();
                MySqlDataAdapter da = new MySqlDataAdapter(sql, conn);
                da.Fill(dt);
            }
            catch { throw; }
            finally { conn.Close(); }
            return dt;
        }

        public object GetSingleAnswer(string sql)
        {
            MySqlConnection conn = new MySqlConnection(connStr);
            object obj = null;
            try
            {
                conn.Open();
                MySqlCommand cmd = new MySqlCommand(sql, conn);
                obj = cmd.ExecuteScalar();
            }
            catch { throw; }
            finally { conn.Close(); }
            return obj;
        }

        public int InsertUpdateDelete(string sql)
        {
            MySqlConnection conn = new MySqlConnection(connStr);
            int rows = 0;
            try
            {
                conn.Open();
                MySqlCommand cmd = new MySqlCommand(sql, conn);
                rows = cmd.ExecuteNonQuery();
            }
            catch { throw; }
            finally { conn.Close(); }
            return rows;
        }
    }
}
```

So that we can switch between the Sql Server and the MySql databases in a seamless manner, add a class called BridgeDataAccess with the following code in it.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestD1.DataLayer
{
    internal class BridgeDataAccess : IDataAccess
    {
        IDataAccess _idac = null;
        public BridgeDataAccess(IDataAccess idac) { _idac = idac; }

        public DataTable GetManyRowsCols(string sql)
        {
            return _idac.GetManyRowsCols(sql);
        }

        public object GetSingleAnswer(string sql)
        {
            return _idac.GetSingleAnswer(sql);
        }

        public int InsertUpdateDelete(string sql)
        {
            return (_idac.InsertUpdateDelete(sql));
        }
    }
}
```

As you can see, the above class does an aggregation on the IDataAccess and delegates the call to the plugged in IDataAccess type of class i.e., DataAccess or DataAccessMySql.

Double click on the Form1 in the solution explorer and drag and drop a combo box and a DataGridView from the tool box as shown below.

Name the combo box, cmbCategories, and the Data Grid View as dg1. Then double click on the combo box and type the following code in the Form1.cs.

```
private void cmbCategories_SelectedIndexChanged(object sender, EventArgs e)
        {
            try
            {
                var cat = cmbCategories.SelectedValue;
                if (cat.GetType().Name != "Int32")
                    return;
                string catid = cmbCategories.SelectedValue.ToString();
                string sql = "select * from Products where CategoryId=" + catid;
                DataTable dt = idac.GetManyRowsCols(sql);
                dg1.DataSource = dt;
                dg1.Refresh();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

Double click on Form1 in the area where there is no control i.e., away from the combo box and the data grid view,

This will end up giving you the Form load event handler which is triggered as soon as the program is run. Type the following code in Form1_load event handler.

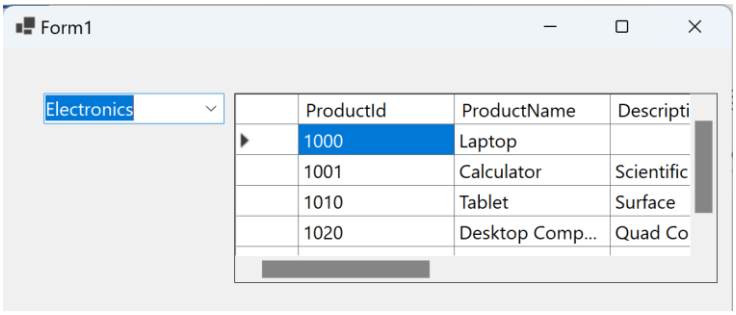
```
private void Form1_Load(object sender, EventArgs e)
        {
            try
            {
                string sql = "select * from categories";
                DataTable dt = idac.GetManyRowsCols(sql);
                cmbCategories.DataSource = dt;
                cmbCategories.ValueMember= "CategoryID";
                cmbCategories.DisplayMember = "CategoryName";
```

```
            cmbCategories.Refresh();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

In the beginning of the Form1 class, type the following code:

```
public partial class Form1 : Form
{
    //IDataAccess idac = new BridgeDataAccess(new DataAccessMySQL());
    IDataAccess idac = new BridgeDataAccess(new DataAccess());
```

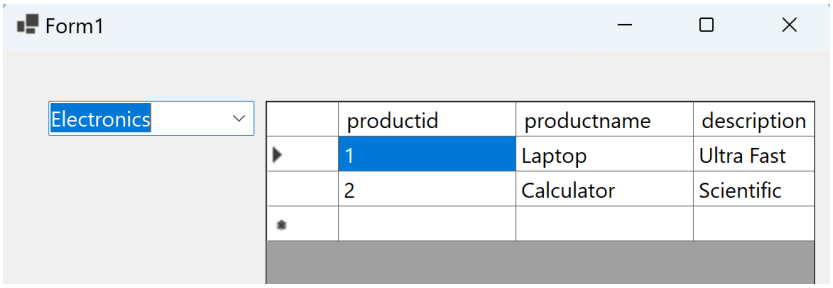
As you can see, the above code plugs in a particular implementation into the BridgeDataAccess. If you run the program, the form will display the data being obtained from the SQL server database.



If you change the beginning of the form class to:

```
public partial class Form1 : Form
    {
        IDataAccess idac = new BridgeDataAccess(new DataAccessMySQL());
        //IDataAccess idac = new BridgeDataAccess(new DataAccess());
```

Now the data will be obtained from the MySql database.



Thus the Bridge pattern allows us to switch an implementation by changing one line of code without requiring any changes in code in the rest of the application, no matter how many places, the database code is being triggered.