Example of Layered Architecture and Database Trasactions

As we have discussed in the class lectures, it is a good practice to implement Layered architecture. Typical layers include the Data Layer, the Business Layer and the Presentation or the UI layer. The layers should be loosely coupled, so that an easy injection of test data can be done and individual layers can be properly tested in isolation.

We will use Northwind database in this example. You can download the Northwind database from the following link and restore it in your SQL server database server.

https://northwinddatabase.codeplex.com/

We will also use another small database called "MorningBank" with two tables called "CheckingAccounts" and "SavingAccounts" in it as shown below.

![Database Table](image)

You can create the above database yourself. CheckingAccountNumber and SavingAccountNumber are of type BigInt. Balance is of type Money.

We will develop a small example of the above layered architecture concepts. Create a Windows Forms application called “ProductsApp”. By right clicking on the project name, add a folder called DataLayer. Then add an interface called IDataAccess with the following code in it.

```csharp
public interface IDataAccess
{
    // transaction capable methods, last three parameters of SqlConnection, SqlTransaction are optional
    object GetSingleAnswer(string sql, List<DbParameter> PList, SqlConnection conn = null, SqlTransaction sqtr = null, bool bTransaction = false);
    DataTable GetManyRowsCols(string sql, List<DbParameter> PList, SqlConnection conn = null, SqlTransaction sqtr = null, bool bTransaction = false);
    int InsertUpdateDelete(string sql, List<DbParameter> PList, SqlConnection conn = null, SqlTransaction sqtr = null, bool bTransaction = false);
}
```
Note that the above methods include transaction support so that they could be invoked as a part of transaction or without one. For this reason, the last three parameters in each interface are optional with default values of null.

Add a class called DataAccess to the DataLayer folder with the following code in it.

```csharp
public class DataAccess : IDataAccess
{
    public string CONNSTR = ConfigurationManager.ConnectionStrings["ProductsApp.Properties.Settings.NORTHWNDConnectionString"].ConnectionString;
    public DataAccess() { }
    public DataAccess(string connstr) // to be able to change the connectionstring
    {
        this.CONNSTR = connstr;
    }
    #region IDataAccess Members
    public object GetSingleAnswer(string sql, List<DbParameter> PList, SqlConnection conn = null, SqlTransaction sqtr = null, bool bTransaction = false)
    {
        object obj = null;
        if (bTransaction == false)
            conn = new SqlConnection(CONNSTR);
        try
        {
            if (bTransaction == false)
                conn.Open();
            SqlCommand cmd = new SqlCommand(sql, conn);
            if (bTransaction == true)
                cmd.Transaction = sqtr;
            if (PList != null)
            {
                foreach (DbParameter p in PList)
                    cmd.Parameters.Add(p);
            }
            obj = cmd.ExecuteScalar();
        }
        catch (Exception)
        {
            throw;
        }
        finally
        {
            if (bTransaction == false)
                conn.Close();
        }
        return obj;
    }
    public DataTable GetManyRowsCols(string sql, List<DbParameter> PList, SqlConnection conn = null, SqlTransaction sqtr = null, bool bTransaction = false)
    {
        DataTable dt = new DataTable();
    }
```
if (bTransaction == false)
    conn = new SqlConnection(CONNSTR);
try
{
    conn.Open();
    SqlDataAdapter da = new SqlDataAdapter();
    SqlCommand cmd = new SqlCommand(sql, conn);
    if (PList != null)
    {
        foreach (DbParameter p in PList)
            cmd.Parameters.Add(p);
    }
    if (bTransaction == true)
        cmd.Transaction = sqtr;
    da.SelectCommand = cmd;
    da.Fill(dt);
} catch (Exception)
{
    throw;
}
finally
{
    if (bTransaction == false)
        conn.Close();
    return dt;
}

public int InsertUpdateDelete(string sql, List<DbParameter> PList,
SqlConnection conn = null, SqlTransaction sqtr = null, bool bTransaction = false)
{
    int rows = 0;
    if (bTransaction == false)
        conn = new SqlConnection(CONNSTR);
try
{
    if (bTransaction == false)
        conn.Open();
    SqlCommand cmd = new SqlCommand(sql, conn);
    if (bTransaction == true)
        cmd.Transaction = sqtr;
    if (PList != null)
    {
        foreach (SqlParameter p in PList)
            cmd.Parameters.Add(p);
    }
    rows = cmd.ExecuteNonQuery();
} catch (Exception)
{
    throw;
}
finally
{
if (bTransaction == false)
    conn.Close();
}
return rows;
#endregion

Since we will be returning a List of strongly typed objects from some of the Repository methods, we will develop a reusable code based on generics and reflection to convert the data table returned by the DataAccess layer to particular type of list. To speed up the development, we can use an object to relational mapping tool that converts a database table to a corresponding class representing each row in the table.

On the left hand corner in your Visual Studio, expand on the Server Explorer (if it is not visible, then from the View menu, choose Server Explorer).

![Server Explorer](image)

Right click on the data connections and then choose “add connection”. Then create a connection pointing to the Northwind database as shown below. You will need to select your own database server instead of the “Alpha” that you see in the following screen shot.
Right click on name of the project and choose add “LINQ to SQL classes” as shown below. After the connection has been added, expand the Northwind connection in the server explorer to see if you can see all the tables. Now you will add LINQ to SQL classes in your visual studio project and drag and drop the different tables from the Northwind database so that an equivalent class can be created for each table in the database.

Add a folder called Models to the project.
Right click on the Models folder and add “LINQ To SQL classes” and give it a name of ProductsApp.dbml as shown below.
Once the designer surface for the INQ To SQL classes appears, drag and drop the different tables from the Northwind database onto the designer surface as shown below.

This will end up creating a class representing each table in the database.

Add an interface called IEntity to the DataLayer folder with the following code in it.
public interface IEntity
{
    void SetFields(DataRow dr);
}

This interface will be implemented by each class representing the database (created by the LINQ To SQL classes). The purpose of this interface is to allow each class object to populate itself from one row of the DataTable.

In the Models folder, add a class called EntityBase with the following code in it.

    public class EntityBase : IEntity
    {
        public void SetFields(DataRow dr)
        {
            // use reflection to set the fields from DataRow
            Type tp = this.GetType();
            foreach (PropertyInfo pi in tp.GetProperties())
            {
                if (null != pi && pi.CanWrite)
                {
                    string nm = pi.PropertyType.Name.ToUpper();
                    if (nm.IndexOf("ENTITY") >= 0) // In LINQ to SQL Classes, last properties are links to other tables
                        break;
                    if (pi.PropertyType.Name.ToUpper() != "BINARY")
                        pi.SetValue(this, dr[pi.Name], null);
                }
            }
        }
    }

The above class uses reflection to populate each field from the DataRow of a DataTable. All the column names are obtained via reflection.

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

    public partial class Category : EntityBase
    {
    }

Note that the above is a partial class, so the remaining code for the Category comes from the drag and drop of the Categories table using the LINQ To SQL classes that we did earlier. Inheriting from EntityBase, gives it the method SetFields.

Similarly, add a class called Product to the Models folder with the following code in it.

    partial class Product : EntityBase
    {
    }

Now add a class called DBList that provides a generic method to convert a DataTable to a strongly typed list to the DataLayer folder with the following code
public class DBList
{
    public static List<T> ToList<T>(DataTable dt)
        where T : IEntity, new()
    {
        List<T> TList = new List<T>();
        foreach (DataRow dr in dt.Rows)
        {
            T tp = new T();
            tp.SetFields(dr);
            TList.Add(tp);
        }
        return TList;
    }
}

Add an interface called IRepositoryProducts to the DataLayer folder with the following code.

    interface IRepositoryProducts
    {
        List<Category> GetAllCategories();
        List<Product> GetProductsByCat(int catId);
    }

Add another interface called IRepositoryCustomerOrders with the following code in it.

    interface IRepositoryCustomerOrders
    {
        // to do
    }

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

    class Repository : IRepositoryCustomerOrders, IRepositoryProducts
    {
        IDataAccess _idac = null;
        public Repository()
        {
            string CONNSTR =
            _idac = new DataAccess(CONNSTR);
        }
        public Repository(IDataAccess idac) { _idac = idac; }
        public List<Category> GetAllCategories()
        {
            List<Category> CList = null;
            try
            {
                string sql = "select * from Categories";
                DataTable dt = _idac.GetManyRowsCols(sql, null);
                CList = DBList.ToList<Category>(dt);
            }
            catch (Exception)
Add an interface called IRepositoryBanking to the DataLayer folder with the following code in it.

```csharp
interface IRepositoryBanking
{
    decimal GetCheckingBalance(long checkingAcctNum);
    decimal GetSavingBalance(long savingAcctNum);
    bool TransferCheckingToSaving(decimal amount, long checkingAccountNum, long savingAccountNum);
}
```

Add a class RepositoryBanking to the DataLayer folder with the following code in it. The TransferCheckingToSaving implements the transactons so study the code carefully.

```csharp
class RepositoryBanking : IRepositoryBanking
{
    IDataAccess _idac = null;
    public RepositoryBanking()
    {
        string CONNSTR = ConfigurationManager.ConnectionStrings["MORNINGBANK"].ConnectionString;
    }
}
_idac = new DataAccess(CONNSTR);
}
public RepositoryBanking(IDataAccess idac) { _idac = idac; }
public decimal GetCheckingBalance(long checkingAcctNum) {
    decimal balance = 0;
    try {
        string sql = "select balance from CheckingAccounts where CheckingAccountNumber=@CheckingAccountNumber";
        List<DbParameter> ParamList = new List<DbParameter>();
        SqlParameter p1 = new SqlParameter("@CheckingAccountNumber", SqlDbType.BigInt);
        p1.Value = checkingAcctNum;
        ParamList.Add(p1);
        object obj = _idac.GetSingleAnswer(sql, ParamList);
        if (obj != null)
            balance = decimal.Parse(obj.ToString());
    } catch (Exception)
    {
        throw;
    }
    return balance;
}

public decimal GetSavingBalance(long savingAcctNum) {
    decimal balance = 0;
    try {
        string sql = "select balance from SavingAccounts where SavingAccountNumber=@SavingAccountNumber";
        List<DbParameter> ParamList = new List<DbParameter>();
        SqlParameter p1 = new SqlParameter("@SavingAccountNumber", SqlDbType.BigInt);
        p1.Value = savingAcctNum;
        ParamList.Add(p1);
        object obj = _idac.GetSingleAnswer(sql, ParamList);
        if (obj != null)
            balance = decimal.Parse(obj.ToString());
    } catch (Exception)
    {
        throw;
    }
    return balance;
}

public bool TransferCheckingToSaving(decimal amount, long checkingAccountNum, long savingAccountNum) {
    // transfer checking to saving has to be done as a transaction
    // transactions are associated with a connection
    bool ret = false;
    string CONNSTR = ConfigurationManager.ConnectionStrings["MORNINGBANK"].ConnectionString;
    SqlConnection conn = new SqlConnection(CONNSTR);
SqlTransaction sqtr = null;
try {
    conn.Open();
    sqtr = conn.BeginTransaction();
    string sql1 = "Update CheckingAccounts set Balance=Balance-@Amount where CheckingAccountNumber=@CheckingAccountNumber";
    List<DbParameter> ParamList = new List<DbParameter>();
    SqlParameter p1 = new SqlParameter("@CheckingAccountNumber", SqlDbType.BigInt);
    p1.Value = checkingAccountNum;
    ParamList.Add(p1);
    SqlParameter p2 = new SqlParameter("@Amount", SqlDbType.Decimal);
    p2.Value = amount;
    ParamList.Add(p2);
    int rows = _idac.InsertUpdateDelete(sql1, ParamList, conn, sqtr, true);
    // part of transaction
    if (rows > 0)
    {
        string sql2 = "select Balance from CheckingAccounts where CheckingAccountNumber=@CheckingAccountNumber";
        List<DbParameter> ParamList2 = new List<DbParameter>();
        SqlParameter pa = new SqlParameter("@CheckingAccountNumber", SqlDbType.BigInt);
        pa.Value = checkingAccountNum;
        ParamList2.Add(pa);
        object obj = _idac.GetSingleAnswer(sql2, ParamList2, conn, sqtr, true);
        if (obj != null)
        {
            if (decimal.Parse(obj.ToString()) < 0) // exception causes transaction to be rolled back
                throw new Exception("Insufficient funds in Checking Account - rolling back transaction");
        }
        else
            throw new Exception("Problem in transferring from Checking Account..");
    }
    else
        throw new Exception("Problem in transferring from Checking Account..");
    string sql3 = "Update SavingAccounts set Balance=Balance+@Amount where SavingAccountNumber=@SavingAccountNumber";
    List<DbParameter> ParamList3 = new List<DbParameter>();
    SqlParameter pb = new SqlParameter("@SavingAccountNumber", SqlDbType.BigInt);
    pb.Value = savingAccountNum;
    ParamList3.Add(pb);
    SqlParameter pc = new SqlParameter("@Amount", SqlDbType.Decimal);
    pc.Value = amount;
    ParamList3.Add(pc);
    rows = _idac.InsertUpdateDelete(sql3, ParamList3, conn, sqtr, true); // part of transaction
    if (rows > 0)
    {
        sqtr.Commit();
        ret = true;
    }
    else
throw new Exception("Problem in transferring to Saving Account..");
}
catch (Exception)
{
    throw;
}
finally
{
    conn.Close();
}
return ret;
}

Add a folder called BusinessLayer to the project. Then add an interface called IBusinessProducts with the following code in it.

interface IBusinessProducts
{
    List<Category> GetAllCategories();
    List<Product> GetProductsByCat(int catId);
}

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

class Business : IBusinessProducts
{
    public IRepositoryProducts _irepProds = null;
    public IRepositoryCustomerOrders _irepCustomers = null;
    public Business()
    {
        _irepCustomers = new Repository() as IRepositoryCustomerOrders;
        _irepProds = new Repository() as IRepositoryProducts;
    }
    public Business(IRepositoryProducts irepProds, IRepositoryCustomerOrders irepCustOrders)
    {
        _irepProds = irepProds;
        _irepCustomers = irepCustOrders;
    }

    public List<Category> GetAllCategories()
    {
        return _irepProds.GetAllCategories();
    }

    public List<Product> GetProductsByCat(int catId)
    {
        return _irepProds.GetProductsByCat(catId);
    }
}
Add an interface IBusinessBanking to the BusinessLayer folder with the following code in it.

```csharp
interface IBusinessBanking
{
    decimal GetCheckingBalance(long checkingAcctNum);
    decimal GetSavingBalance(long savingAcctNum);
    bool TransferCheckingToSaving(decimal amount, long checkingAccountNum, long savingAccountNum);
}
```

Add a class called BusinessBanking to the BusinessLayer folder with the following code in it.

```csharp
class BusinessBanking : IBusinessBanking
{
    public IRepositoryBanking _irepBank = null;
    public BusinessBanking()
    {
        _irepBank = new RepositoryBanking() as IRepositoryBanking;
    }
    public BusinessBanking(IRepositoryBanking irepbank)
    {
        _irepBank = irepbank;
    }
    public decimal GetCheckingBalance(long checkingAcctNum)
    {
        return _irepBank.GetCheckingBalance(checkingAcctNum);
    }
    public decimal GetSavingBalance(long savingAcctNum)
    {
        return _irepBank.GetSavingBalance(savingAcctNum);
    }
    public bool TransferCheckingToSaving(decimal amount, long checkingAccountNum, long savingAccountNum)
    {
        return _irepBank.TransferCheckingToSaving(amount, checkingAccountNum, savingAccountNum);
    }
}
```

Now we are ready to create the user interface. Add a combo box and a datagridview to the form. The code for the different event handlers is shown below. There are a few labels and buttons as shown below.
The code for the form class is shown below.

```csharp
public partial class Form1 : Form
{
    IBusinessProducts _ibusinessProds = null;
    IBusinessBanking _ibusinessBanking = null;

    public Form1()
    {
        InitializeComponent();
        _ibusinessProds = new Business() as IBusinessProducts;
        _ibusinessBanking = new BusinessBanking() as IBusinessBanking;
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        try
        {
            List<Category> CList = _ibusinessProds.GetAllCategories();
            cmbCategories.DataSource = CList;
            cmbCategories.DisplayMember = "CategoryName";
            cmbCategories.ValueMember = "CategoryId";
            cmbCategories.Refresh();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }

    private void cmbCategories_SelectionChangeCommitted(object sender, EventArgs e)
    {
    }
```
try
{
    int catId = int.Parse(cmbCategories.SelectedValue.ToString());
    List<Product> PList = _ibusinessProds.GetProductsByCat(catId);
    dg1.DataSource = PList;
    cmbCategories.Refresh();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

private void btnGetBalances_Click(object sender, EventArgs e)
{
    try
    {
        long checkingAccountNum = 10000; // checking accountnum for user Bill, pretend this was obtained at the time of Login
        long savingAccountNum = 100000; // saving accountnum for user Bill, pretend this was obtained at the time of Login
        decimal checkingBalance =
            _ibusinessBanking.GetCheckingBalance(checkingAccountNum);
        decimal savingBalance =
            _ibusinessBanking.GetSavingBalance(savingAccountNum);
        lblCheckingBalance.Text = checkingBalance.ToString();
        lblSavingBalance.Text = savingBalance.ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

private void btnTransferCheckingToSaving_Click(object sender, EventArgs e)
{
    try
    {
        decimal amount = decimal.Parse(txtAmount.Text);
        long checkingAccountNum = 10000; // checking accountnum for user Bill, pretend this was obtained at the time of Login
        long savingAccountNum = 100000; // saving accountnum for user Bill, pretend this was obtained at the time of Login
        bool ret = _ibusinessBanking.TransferCheckingToSaving(amount, checkingAccountNum, savingAccountNum);
        if (ret == true)
            btnGetBalances.PerformClick();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

Build and test the application.