LINQ E-Commerce Store (Design Patterns)

Student Name	Student ID
Abhinav Satish Shah	6224423
Manthankumar Makwana	6210600
Mohnish Sethi	6380387
Niketh Jain	6405568
Shrey Desai	6462928
Date: April 16, 2013	

a) Façade Design Pattern (Abhinav Satish Shah)

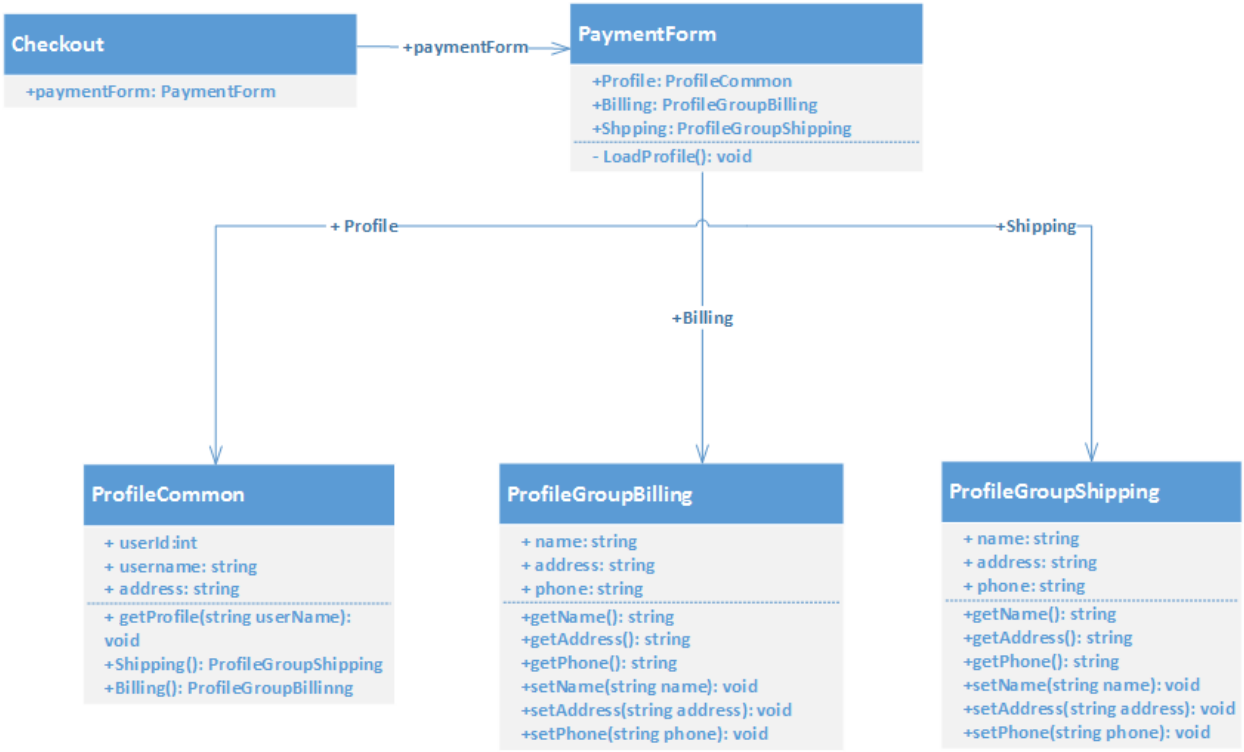

Figure 1: UML Class Model for Façade Pattern

Description of Façade Pattern can be found at the following link:
http://www.dofactory.com/Patterns/PatternFacade.aspx#_self1

The classes participating in the façade pattern are:

1) PaymentForm (Façade).
2) Profile Common, ProfileGroupBilling, ProfileGroupShipping (Subsystem classes).
3) Checkout (Client).

This pattern is in need because it makes the subsystems easier to use. For instance, when the LoadProfile() from PaymentForm is invoked, the objects of the subsystems are created to implement their respective functionality, handle the work assigned by the façade object and keep no reference to façade.

Once it receives the request, PaymentForm (Façade) delegates the responsibility to the subsystems to perform their individual work instead of handling all the functionalities in one single class. This also solves the problem of High Coupling among classes to some extent.

Related code for Façade Design Pattern:

The pseudo code demonstrates the use of façade pattern which provides a simplified and a uniform class to a large subsystem of classes.

```
        public partial class PaymentForm {
        {
           private void LoadProfile(){

               ProfileCommon Profile = new ProfileCommon();

               //Put all Billing Info into the form for billing

               BillNTB.Text = Profile.Billing.Name;
               BillPhoneTB.Text = Profile.Billing.Phone;
               BillAdTB.Text = Profile.Billing.Address;

               //Put all Shipping Info into the form for shipping
               ShipNTB.Text = Profile.Shipping.Name;
               ShipPhoneTB.Text = Profile.Shipping.Phone;
               ShipAdTB.Text = Profile.Shipping.Address;
                      .
                      .
                      .
        }
                      .
                      .
                      .
}
        public class ProfileCommon {
                      .
                      .
                      .
               public virtual ProfileGroupShipping Shipping {
                 get {
                     return ((ProfileGroupShipping)(this.GetProfileGroup("Shipping")));
                 }
            }

            public virtual ProfileGroupBilling Billing {
                get {
                     return ((ProfileGroupBilling)(this.GetProfileGroup("Billing")));
                }
            }
                      .
                      .
                      .
        }
```

b) **Adapter Design Pattern** (**Manthankumar Makwana**)

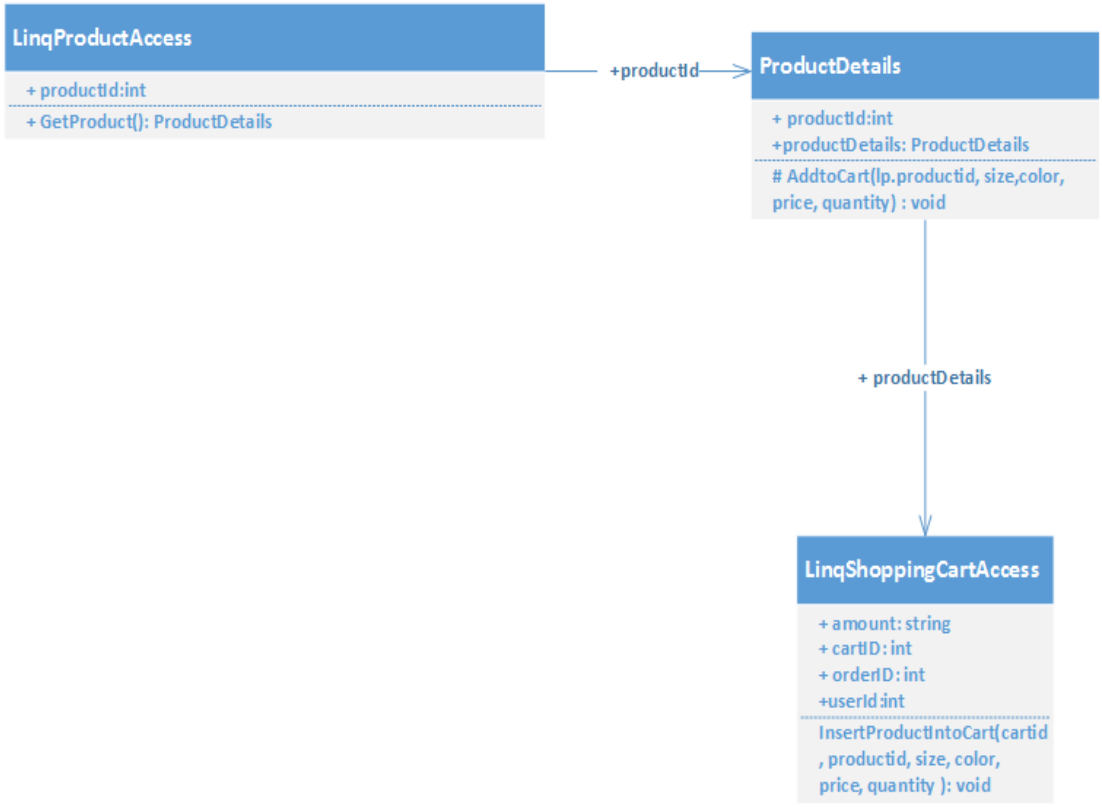

Figure 2: UML Class Model for Adapter Pattern

Here is the link to the description of the Adapter pattern:

http://www.oodesign.com/adapter-pattern.html

In this example, **ProductDetails** is a partial class that performs operation of adding product to a shopping cart. In doing so, it first it takes ProductID from **LinqProductAccess** class. Further, it passes ProductID, Product Size, Product Color, Product Price and Product Quantity to **LinqShippingCartAccess** class. Finally, **LinqShippingCartAccess** generates 32-bit unique CartID. If product is being insert to an existing Cart, it uses the current CartID else it creates unique 32-bit CartID.

Here, **ProductDetails** works as an adapter between **LinqProductAccess** and **LinqShippingCartAccess** (adaptee) classes.

Related code for Adapter Design Pattern:

```
public partial class ProductDetails
{
        protected void addToCartButton_Click(object sender, EventArgs e)
        {
                ls.InsertProductIntoCart(lp.GetProduct().Price, pSize, pColor,
                Convert.ToInt32(QuantityTextBox.Value));
                .
                .
                .
        }
}

public class LinqProductAccess
{
        .
        .
        .
public void GetUserShoppingCart()
        {
         LinqCommerceDataContext db = new LinqCommerceDataContext();
         var squery = from s in db.lc_ShoppingCarts
                    where s.CartID == LinqShoppingCartAccess.cartID
                    select s;

        }
}

public class LinqShoppingCartAccess
{
        public void InsertProductIntoCart(double Price, string Size, string
Color)
        {
          LinqCommerceDataContext db = new LinqCommerceDataContext();
          lc_ShoppingCart s = new lc_ShoppingCart

           CartID = LinqShoppingCartAccess.cartID,
           ProductID = LinqProductAccess.ProductID,
           DateAdded = System.DateTime.Today,
           Quantity = 1,
           Price = Price,
           SizeID = 1,
           ColorID = 1

           db.lc_ShoppingCarts.InsertOnSubmit(s);
           db.SubmitChanges();
        }
}
```

c) Strategy Design Pattern (**Mohnish Sethi**)

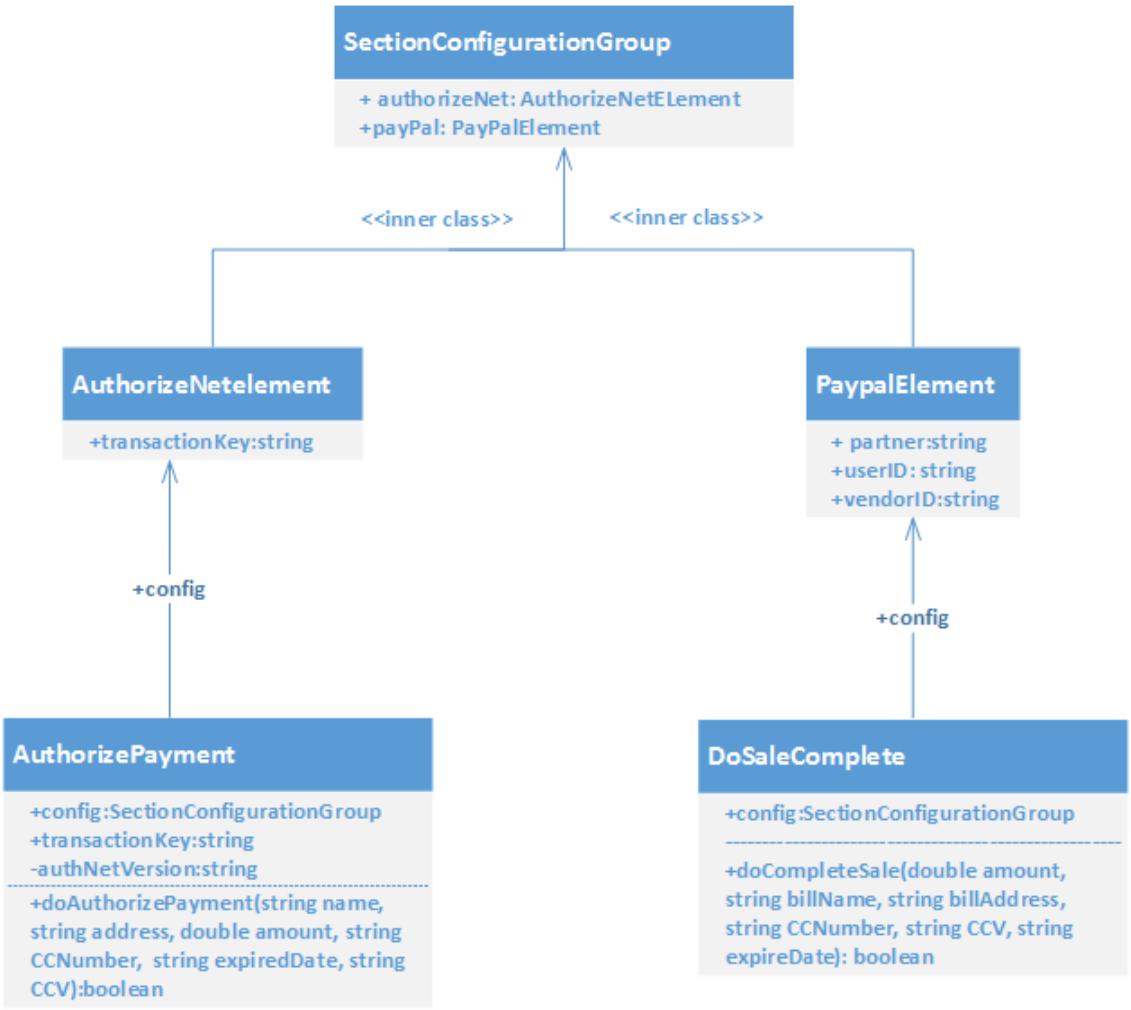

Figure 3: UML Class Model for Strategy Pattern

I have provided the following link for describing the Strategy pattern:

http://www.oodesign.com/strategy-pattern.html

Following are the classes participating in the pattern are as follows:

1. SectionConfiguratonGroup (Client)

2. AuthorizeNetElement (Strategy1)

3. PayPalElement (Strategy 2)

A Strategy defines a set of algorithms that can be used interchangeably. Modes of payment to buy any product would act like a strategy pattern. The SectionConfiguratonGroup class consist of two inner classes such as PayPalElement and AuthorizeNETElement. At the runtime object config of the

SectionConfiguration class will be created and by selecting the payment mode, it can either as PayPalElement and AuthorizeNETElement depending upon users need. So we can call that the pattern applied over here is strategy pattern because each payment mode has their respective way of payment. In this case AuthorizePayment class has its own strategy or algorithm written and DoSaleComplete class has its own strategy or algorithm.

Related code for Strategy Design Pattern:

```
public class SectionConfigurationGroup : ConfigurationSection
    {
       .
       .
       .
public class PayPalElement : ConfigurationElement
        {
       .
       .
       .
        }
public class AuthorizeNETElement : ConfigurationElement
        {
       .
       .
       .
        }
       }
public class AuthorizePayment
{


    public static bool DoAuthorizePayment(string name, string address,
    double amount, string CCNumber, string ExpiredDate, string CCV)
    {

        SectionConfigurationGroup config =
       (SectionConfigurationGroup)WebConfigurationManager.GetSection("LinqCommer
       ce/AuthorizeNETSettings");


.
.
.

    }

}
public class DOSaleCompleteCS
    {
public static bool doCompleteSale(double amount, string billName, string
billAddress,
    string CCNumber, string CCV, string ExpireDate)
    {

       SectionConfigurationGroup config =
       (SectionConfigurationGroup)WebConfigurationManager.GetSection("LinqCommer
       ce/Payment");

...
     }

}
```

d) **Strategy Design Pattern** (**Niketh Jain**)

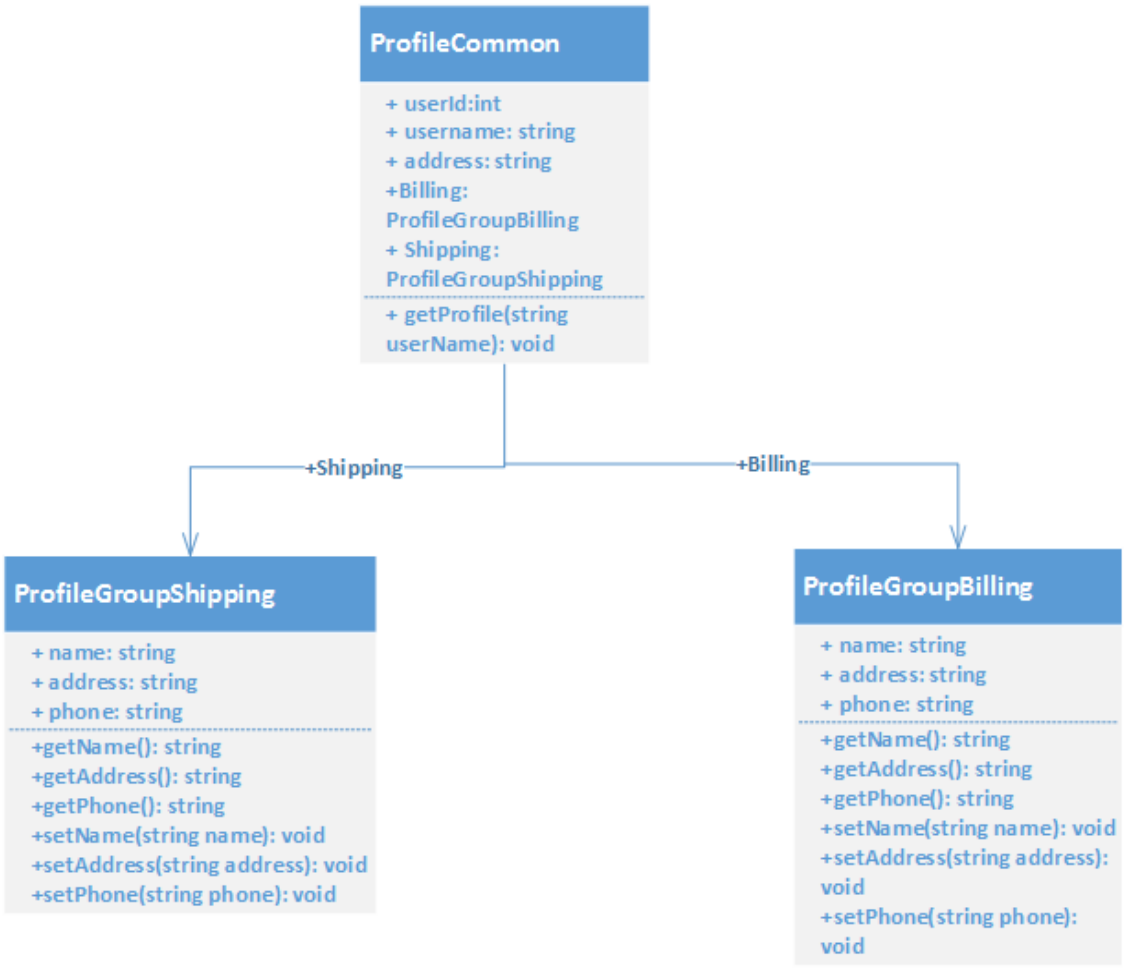

Figure 4: UML Class Model for Strategy Pattern

The link describing the strategy pattern is as follows:

http://www.vincehuston.org/dp/strategy.html

The following are the interacting classes for this strategy design pattern:

1) ProfileCommon (Client)
2) ProfileGroupShipping (Strategy 1)
3) ProfileGroupBilling (Strategy 2)

This strategy pattern uses directed association instead of inheritance. In this strategy pattern, behaviors are defined as separate classes. Here, in this pattern, ProfileGroupBilling and ProfileGroupShipping each have different behaviors as they have the similar methods each with a different body. This allows better decoupling between the behavior (ProfileCommon) and the classes (ProfileGroupShipping & ProfileGroupBilling) that uses the behavior. The behavior can be changed without breaking the classes that

use it, and the classes can switch between behaviors by calling the appropriate object at run time used without requiring any significant code changes.

Related code for Strategy Design Pattern:

```
public class ProfileCommon {
        .
        .
        .
       public virtual ProfileGroupShipping Shipping {
         get {
             return ((ProfileGroupShipping)(this.GetProfileGroup("Shipping")));
         }
    }

    public virtual ProfileGroupBilling Billing {
         get {
             return ((ProfileGroupBilling)(this.GetProfileGroup("Billing")));
         }
    }

}
public class ProfileGroupShipping
{
        .
        .
        .
}

public class ProfileGroupBilling
{
        .
        .
        .
}

public partial class PaymentForm
{

  private void LoadProfile()
    {
         //Put all Billing Info into the form for viewinguct
         BillNTB.Text = Profile.Billing.Name;
         BillPhoneTB.Text = Profile.Billing.Phone;
         BillAdTB.Text = Profile.Billing.Address;

         //Put all Shipping Info into the form for shipping
         ShipNTB.Text = Profile.Shipping.Name;
         ShipPhoneTB.Text = Profile.Shipping.Phone;
         ShipAdTB.Text = Profile.Shipping.Address;
   }
        .
        .
        .

}
```

e) **Factory Design Pattern** (**Shrey Desai**)

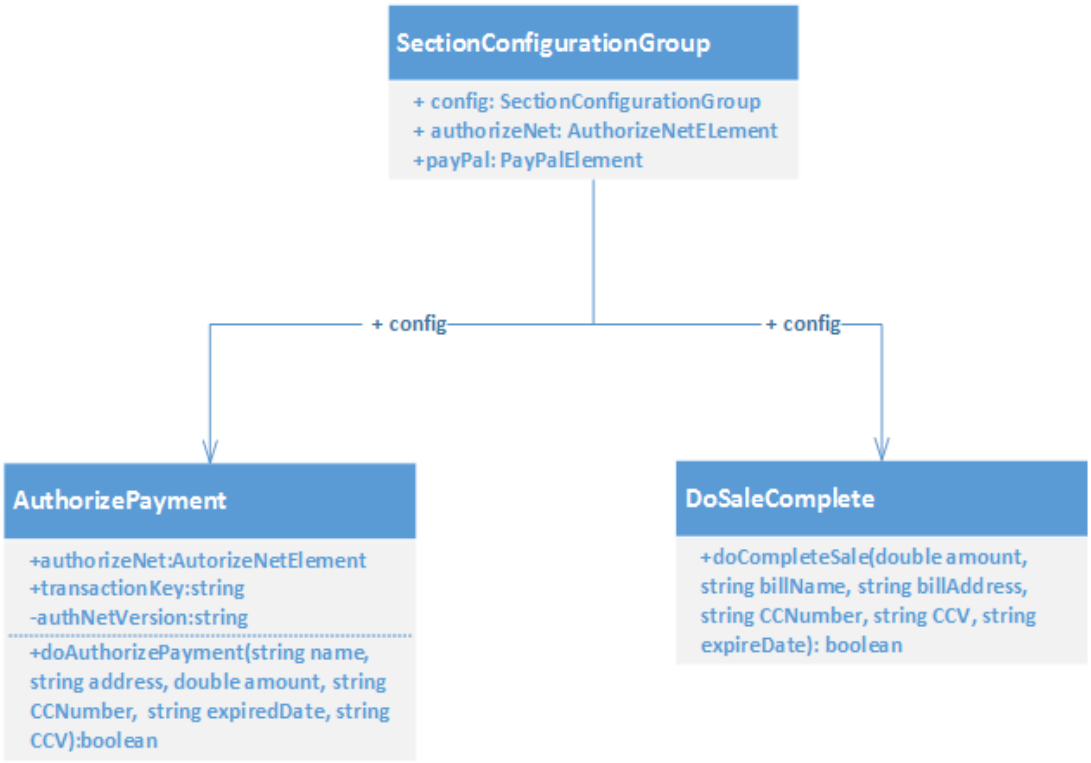

Figure 5: UML Class Model for Factory Pattern

The description of Factory Pattern is found at the following link:

http://sourcemaking.com/design_patterns/factory_method

The classes taking part in the Factory pattern are as under:

1) SectionConfigurationGroup (Factory)

2) AuthorizePayment (Subsystem classes)

3) DoSaleComplete (Subsystem classes)

The AuthorizePayment class consists of DoAuthorizePayment method which is concerned with the AuthorizeNET payment mode. The DoSaleComplete class contains doCompleteSale method which deals with the PayPal payment mode. The Factory pattern is implemented in this case because the SectionConfiguration class cannot anticipate which type of objects it must create. The object, 'config' of the SectionConfiguration class will be created at runtime and based on the user's input, it will call either AuthorizePayment class or DoSaleComplete class.

Related code for Factory Design Pattern:

```
public class SectionConfigurationGroup : ConfigurationSection
       {
          .
          .
          .
   public class PayPalElement : ConfigurationElement
            {
          .
          .
          .
             }
   public class AuthorizeNETElement : ConfigurationElement
            {
          .
          .
          .
          }
          }

   public class AuthorizePayment
   {


       public static bool DoAuthorizePayment(string name, string address,
       double amount, string CCNumber, string ExpiredDate, string CCV)
       {

           SectionConfigurationGroup config =
          (SectionConfigurationGroup)WebConfigurationManager.GetSection("LinqCommer
          ce/AuthorizeNETSettings");


   .
   .
   .


       }

   }
   public class DOSaleCompleteCS
       {
   public static bool doCompleteSale(double amount, string billName, string
   billAddress,
       string CCNumber, string CCV, string ExpireDate)
       {

          SectionConfigurationGroup config =
          (SectionConfigurationGroup)WebConfigurationManager.GetSection("LinqCommer
          ce/Payment");
   .
   .
       }
   }
```

Please find the patchset.zip file containing 3 patch files where the relevant description has been provided at the end of each file.