LINQ E-Commerce Store (Milestone 3)

Student Name	Student ID
Abhinav Satish Shah	6224423
Manthankumar Makwana	6210600
Mohnish Sethi	6380387
Niketh Jain	6405568
Shrey Desai	6462928
Date: Mar 25, 2013	

1. Class Diagram of Actual System

1.1 UML Domain Diagram

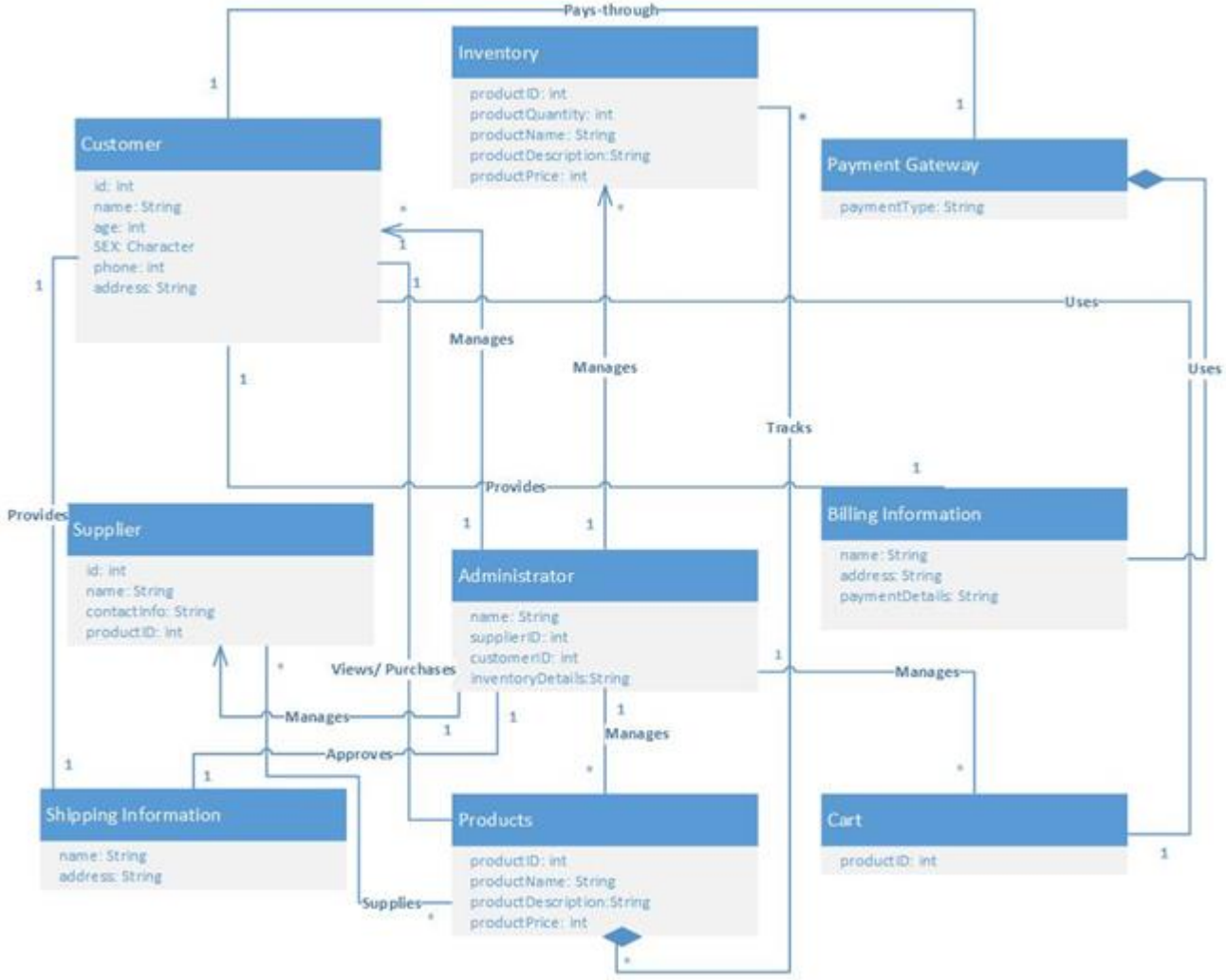

Figure 1: UML DOMAIN DIAGRAM

1.2 UML Class Diagram

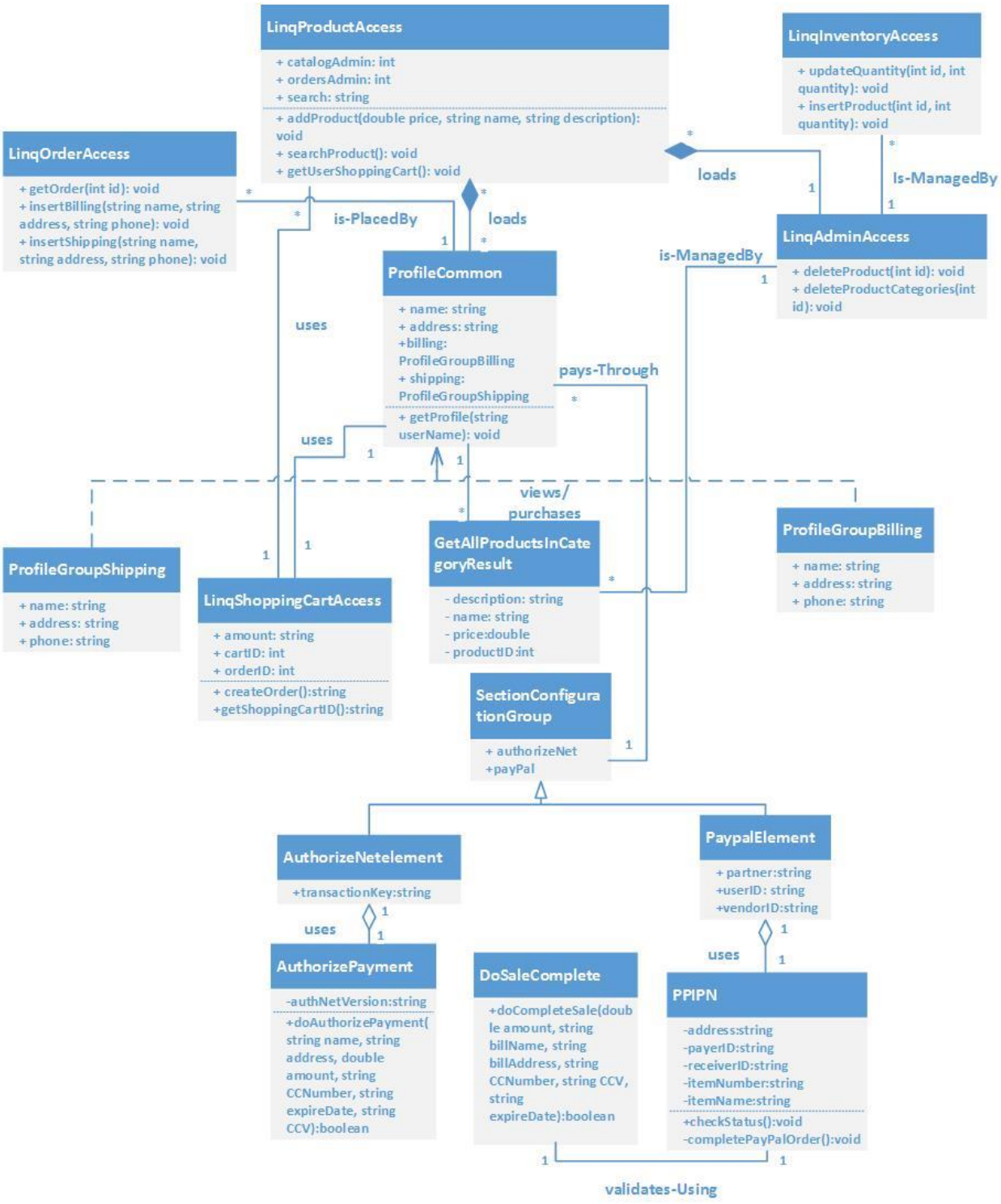

Figure 2: UML CLASS DIAGRAM

1.3 Class Diagram Elaboration

The class diagram (CD) of Linq E-Commerce Store resembles in some ways to what we had imagined in milestone 2 demonstrated via domain model (DM). CD: **ProfileCommon** is equivalent to DM: **Customer** where both the classes store the detailed information of the person involved in viewing / purchasing the products using the application. CD: **ProfileCommon** is directly linked to various other classes namely CD: **LinqOrderAccess**, CD: **ProfileGroupBilling**, CD: **ProfileGroupShipping** and CD: **SectionConfigurationGroup** which store the different orders the customers want to make, billing information, shipping information and payment details respectively. Most of these classes also exist in our domain model whose behaviour is identical to the real classes we have found in the system.

CD: **LinqAdminAccess** is the real class which gives the privileges to the administrator to manage the products available in the system and therefore uses CD: **LinqInventoryAccess** class to mainly insert a new product or update the quantity of the products. These classes are also identical to DM: **Products** and DM: **Inventory** conceptual classes found in our domain model.

However, there are certain differences between the real architecture and the domain model of the Linq E-Commerce Store. Some of the conceptual classes, for instance, DM: **Products** is not found in our class diagram. Instead, the CD: **AdminAccess** class manages what we had imagined the DM: **Products** would be doing – inserting, deleting and updating the products' name, price, description and its categories. Another subtle difference is in managing different orders. CD: **LinqShoppingCartAccess** class takes the customer information from CD: **ProfileCommon** and creates order(s) for the customer. These orders, however, are managed by the CD: **LinqOrderAccess** class which gets the orders based on ID and inserts the billing and shipping information for the orders directly from the user interface. We had initially thought that the orders are entirely managed by DM: **Cart** conceptual class.

We also found a real class CD: **GetAllProductsInCategoryResult** not found in DM, which gets different products of a category and stores the name, price, and description of each product based on the product ID.

One another difference lies in the fact to process payments for different kinds of payment systems – Authorize .Net and PayPal. The class diagram depicts that the 2 classes CD: **AuthorizeNetElement** and CD: **PaypalElement** are children of CD: **SectionConfigurationGroup** which in turn is analogous to DM: **PaymentGateway** conceptual class. The 2 child classes make use of CD: **AuthorizePayment** and CD: **PPIPN** classes respectively. CD: **AuthorizePayment** class has a method which shows a successful message if the payment using Authorize .Net is processed successfully. The CD: **DoSaleComplete** class is related to CD: **PPIPN** class that has a method which shows a successful message if the payment is processed successfully. CD: **DoSaleComplete** class is invoked if the customer selects to pay using the PayPal system. On the other hand, in a hindsight, we thought that the entire payment functionality is handled only by the DM: **PaymentGateway** conceptual class.

In a nutshell, we can say that the architecture of Linq E-Commerce Store satisfies "High Cohesion" GRASP principle as related functionalities is distributed in different classes of the system. This can be demonstrated from the fact that the entire payment functionality is handled by different classes instead of just one class. In addition, we can say that the different code smells (indicated in the following sections)

identified will help us in refactoring some of the classes using different refactoring techniques, thereby improving the architecture of the entire system.

All the class diagrams have been created by making use of the templates available for UML class diagrams in MS Visio 2013.

1.4 Relationship between Classes & Sample Code

The following code has been taken from **LinqProductDetails** class user control which shows relationship between **LinqProductAccess** and **LinqShoppingCartAccess** classes.

```
protected void addToCartButton_Click(object sender, EventArgs e)
 {
   LinqProductAccess lp = new LinqProductAccess();
   LinqShoppingCartAccess ls = new LinqShoppingCartAccess();

 // Check product inventory
if  (LinqProductAccess.GetProductInventory(LinqProductAccess.ProductID,  pSize,  pColor)
== true)
{
try
{//How many of that product are in stock?
 int  quantityInStock  =  LinqProductAccess.QuantityInStock(LinqProductAccess.ProductID,
 pSize, pColor);
//Are there enough available?
   if (QuantityTextBox.Value <= quantityInStock)
   {
ls.InsertProductIntoCart(lp.GetProduct().Price,pSize,pColor,Convert.ToInt32(QuantityTe
xtBox.Value));
    FeedbackLabel.Text = "Product added successfully!";
   }
 else
 {
  FeedbackLabel.Text = "<b>Quantity isn't available.</b>";
 }
} //Product exists in cart
catch(Exception ex)
 {
  FeedbackLabel.Text  =  "<b>This  product  has  already  been  added  to  the  shopping
 cart.</b>";
 }
}
   else
    {
      FeedbackLabel.Text = "<b>I'm sorry, but this product is out of stock.</b>";
          }
 }
```

"GetProduct" method from **LinqProductAccess** class returns product which is further used in "InsertProductIntoCart" method of **LinqShoppingCartAccess** class. This operation takes place when the user tries to add a particular product to the shopping cart. While adding the selected product to the shopping cart, the system checks whether the entered quantity is less than OR equal to the quantity available in the stock.

2. Code Smell and Possible Refactoring

2.1 Code Smells

Long Parameter List:
Long list of parameters are passed in "insertBilling" and "insertShipping" methods of **LinqOrderAccess** class. This violates object oriented programming style as in object oriented programming, parameter lists tend to be much smaller. This code smell can be addressed by using "Introduce Parameter Object" refactoring technique.

Duplicated Code/Large Class:
The "doAuthorizePayment" and the "doCompleteSale" methods of **AuthorizePayment** and **DoSaleComplete** classes respectively contain duplicate data and too many instance variables. These code smells are a sign of lazy programming style which results in higher maintenance costs later on. Firstly. "Extract Method" refactoring technique is used to create a new method incorporating the business logic for processing the payment functionality. Finally, "Extract Class" refactoring technique is used to address this code smell where duplicated data is extracted and depicted in the **ValidatePayment** class.

Data Class:
The **ProfileGroupBilling** and **ProfileGroupShipping** classes contain only accessors and mutators with the fields having public access rights. The result is violation in the object oriented principle of data encapsulation. This code smell is handled by "Encapsulate Field" refactoring technique by making all the fields in the **ProfileGroupBilling** and **ProfileGroupShipping** classes private.

2.2 UML Class Diagram after Refactoring

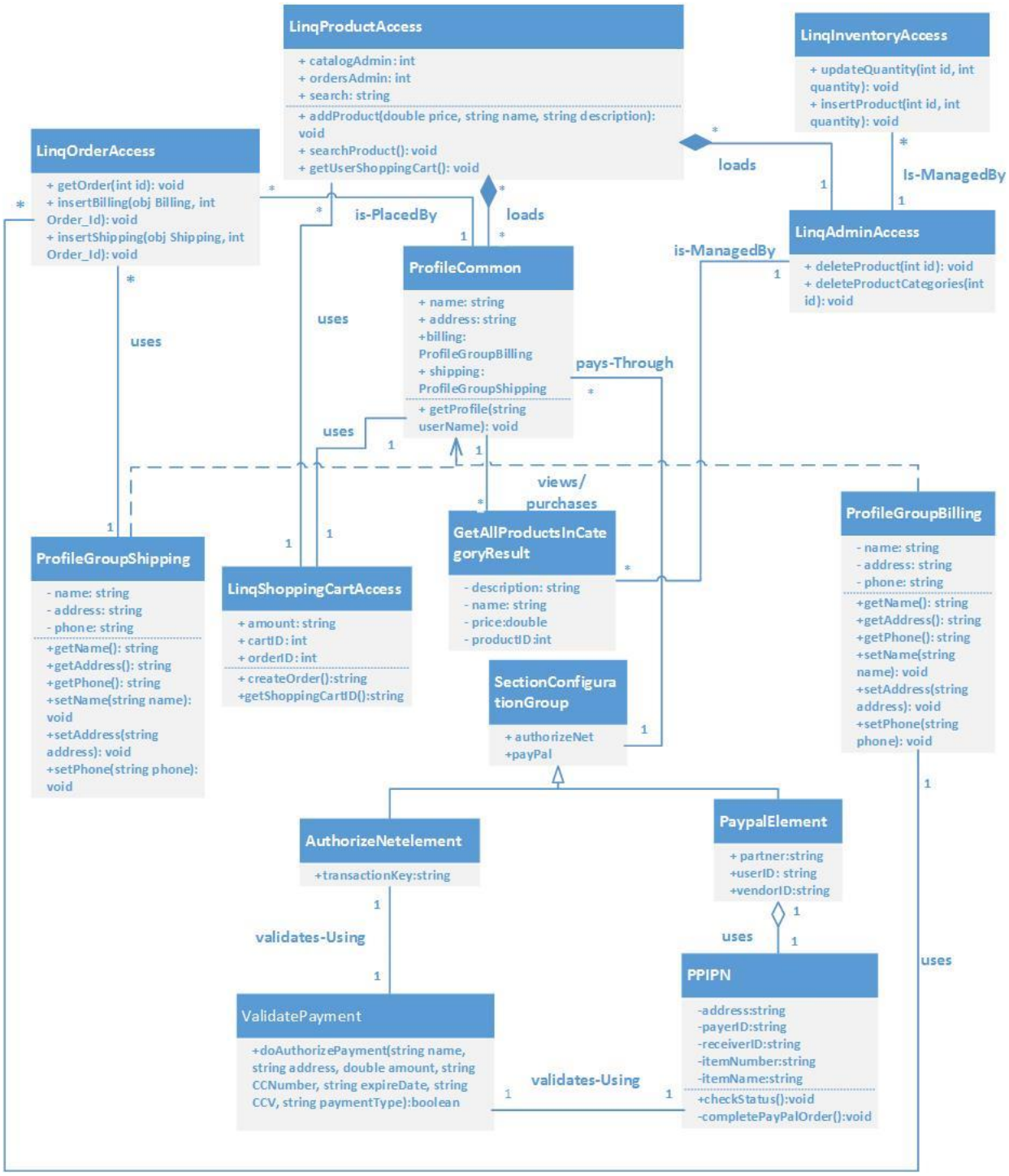

Figure 3: Refactored Class Diagram

2.3 Refactored Class Diagram Elaboration

To improve the architecture of Linq E-Commerce Store, we suggest the following classes and the relationships between the classes to be modified in some ways as detailed below:

LinqOrderAccess class takes the billing and shipping information of the customer directly from the user interface that violates the object oriented programming principles for passing parameters to methods. Therefore, we establish the relationship between **LinqOrderAccess**, **ProfileGroupShipping** and **ProfileGroupBilling** classes by using "Introduce Parameter Object" that passes the objects of **ProfileGroupBilling** and **ProfileGroupShipping** classes as parameters in the "insertBilling" and "insertShipping" methods of **LinqOrderAccess** class.

AuthorizePayment and **DoSaleComplete** classes each have a similar, but different methods to process payments for Authorize .Net Element and PayPal systems respectively. In other words both the methods take parameters, for instance, name, address, CC number, CCV, expiry date and amount. This violates the principle of polymorphism. Therefore, we first extract these methods into a common method. We then, extract a new class namely **ValidatePayment** and insert a method in this class which will process payments for both the systems based on payment type.

ProfileGroupBilling and **ProfileGroupShipping** classes currently have public attributes and contain accessors and mutators for these attributes. It does not make sense to access these attributes using get methods and modify those using set methods, thus violating the object oriented programming principle of Data Encapsulation. We change the access rights of the attributes in these classes and make them private, thereby implementing "Encapsulate Field refactoring technique".

2.4 Sample class to be refactored

Following is the code for **LinqOrderAccess** class which takes long parameters in "insertBilling" and "insertShipping" methods. We can therefore infer that this class has a code smell "Long Parameter Lists". We can address this code smell by using "Introduce Parameter Object" refactoring technique.

```
    Protected void btnSubmit_Click(object sender, EventArgs e)
    {
        string message = null;
        bool t = AuthorizePayment.DoAuthorizePayment(out message, BillFNTB.Text,
BillLNTB.Text, BillAdTB.Text, BillCityTB.Text, BillStateDropDown.Text,
BillZipCodeTB.Text, BillCountryDropDown.Text,
            Amount, true, CreditCardNumber, Month + "/" + Year, CVC);
         if (t)
        {
            lo.InsertBilling(BillFNTB.Text, BillLNTB.Text, BillMNTB.Text, "",
lc_OrderID,
                BillPhoneTB.Text, BillPrefixCombo.Text, BillStateDropDown.Text,
BillZipCodeTB.Text);
            lo.InsertShipping(ShipFNTB.Text, ShipLNTB.Text, ShipMNTB.Text, "",
lc_OrderID,
                ShipPhoneTB.Text, ShipPrefixCombo.Text, ShipStateDropDown.Text,
ShipZipCodeTB.Text);
        }
        Redirect(message);
    }
//LinqOrderAccess Class

public void InsertBilling(string FirstName, string LastName, string MiddleName, string
NickName, int OrderID, string Phone, string Prefix, string State, string Zip)
        {
            LinqCommerceDataContext db = new LinqCommerceDataContext();
             lc_BillingInfoTable bi = new lc_BillingInfoTable();
      …(unwanted code has been removed)
            db.lc_BillingInfoTables.InsertOnSubmit(bi);
            db.SubmitChanges();
        }
public void InsertShipping(string FirstName, string LastName, string MiddleName,
string NickName, int OrderID, string Phone, string Prefix, string State, string Zip)
        {
            LinqCommerceDataContext db = new LinqCommerceDataContext();
            lc_ShippingInfoTable bi = new lc_ShippingInfoTable();
       …(unwanted code has been removed)
            db.lc_ShippingInfoTables.InsertOnSubmit(bi);
            db.SubmitChanges();
        }
```

The GitHub link for the project milestone 3 and the source code is:

https://github.com/desaishrey90/SOEN6471