# LINQ E-Commerce Store (M1, M2 and M3)


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

# 1. MILESTONE - 1

## 1.1 General Overview and Motivation

LINQ E-Commerce Store is an open source application developed using LINQ having features such as product search and inventory tracking based on websites like Amazon. In addition, the application makes use of payment gateways such as PayPal integration, Authroized .NET, AIM Gateway and Google Checkout. This project uses Telerik's RadControls for ASP.NET and AJAX.

LINQ E-Commerce Store consists of two different modules, one for the client - who can view, order, add, and delete items to the cart which are available in the shop online and the other one for the administrator - who manages inventory to keep a track on product levels, sales & deliveries, and approves the orders. Portions of this open source project were adapted from Christian Darie's E-Commerce book, "Beginning ASP.NET 2.0 E-Commerce in C# 2005."

Our project follows object oriented programming principles which led to the realization that the project has a wide scope of implementing refactoring techniques. Also, most of the team members are familiar with C# programming & with an interest in learning LINQ and PayPal payment technology motivated us to select this project.

## 1.2 Maturity of the project

The project is stable with 20 contributors and the last commit to the project was done on May 9, 2011 as per http://linqcommerce.codeplex.com/team/view. Further, there are three versions developed for the application. Each of the versions has evolved over time. This information is extracted using the discussion forum (http://linqcommerce.codeplex.com/discussions).

## 1.3 Size of the Project

We used Microsoft Line of Code counter to count lines of code for each class.

The entire project consists of 10745 Lines of Code (LOC) among 73 classes. We see this as medium sized project which would be helpful in not only understanding the business logic but also will give us wide scope to refactor the source code by applying design patterns.

We found that there are 1377 Commented Lines of Code (CLOC), which is estimated to be about 13% of the total lines of code. This will also help us in understanding the business logic of the code.

Following chart displays Lines of Code that each module consists of:

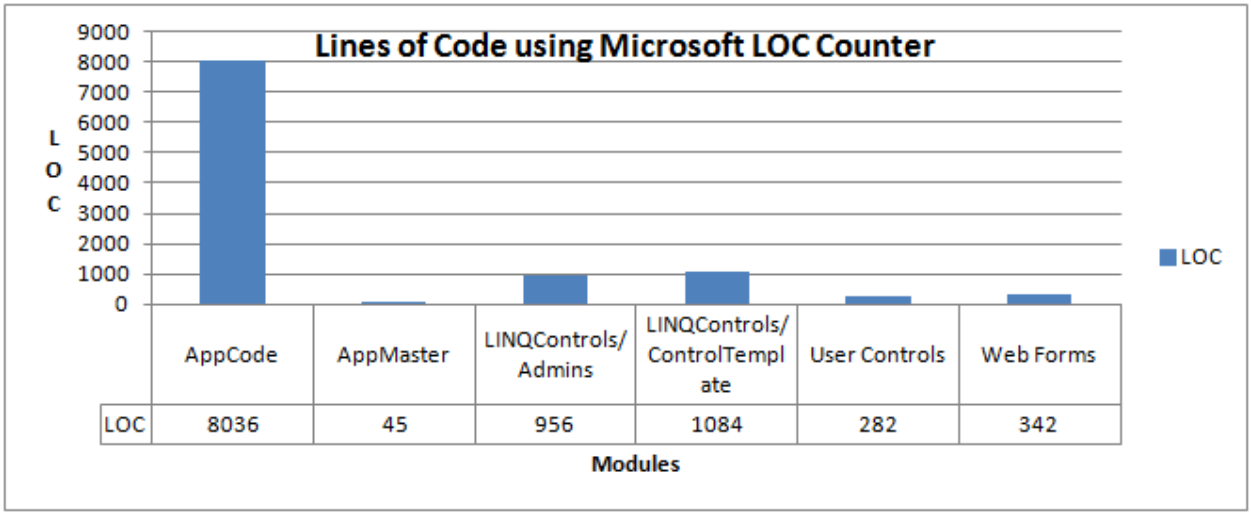


***Figure 1: Microsoft LOC Counter***

## 1.4 Group Members

**Abhinav Satish Shah:**

I was part of the quality assurance team performing code reviews for a project that was developed in the last term in C#. Although I do not have much experience in developing applications in C#, my relevant experience in working on similar object oriented programming languages including Java and Adobe Flex for over 2 years will help me in contributing to LINQ E-Commerce Store application to understand and revise the architecture by refactoring the code.

**Manthankumar Makwana:**

I have been working on .Net technology for more than 3 years out of which most of the time I have developed web based applications. Currently I am currently developing web based applications for Continuous Improvement department of Rolls-Royce Canada Limited for their internal use. In my career, I have got less opportunity to work on LINQ. This project will give me an opportunity to learn this technology along with Payment gateway integration. Further, I would learn to refactor the code by applying design patterns.

**Mohnish Sethi:**

Working on a .NET framework makes me feel comfortable. I have experience in working and developing web applications using C#. My recent project was built in C# with Microsoft SQL used as a back-end technology. This understanding of how C# projects will help me understand the coding standards which can lead to some refactoring tactics. Further, it can help me understand the use of LINQ.

**Niketh Jain:**

I have worked on Java web applications based on struts framework in the past which provides me the sound knowledge of object oriented programming. It will help me in contributing to understand the architecture for the current project. In addition to this, I have played a key role in being a part of quality assurance team for an academic project which was developed in .NET framework. With this relevant experience I believe this would not only help me in understanding the process but also would enhance my technical knowledge in refactoring the code. This, as a whole will benefit the team.

**Shrey Desai:**

I have worked in the .NET technology for almost a year. Apart from this, I have developed several academic projects in this domain like an E-Commerce application and an Inventory Management System to list a few. In my previous academic project, I implemented factory pattern, observer pattern and singleton pattern. I have not got much opportunity to work on LINQ, so this project will help me to learn the same. Also, I will get knowledge of applying refactoring techniques in this project.

# 2. MILESTONE - 2

## 2.1. Personas, Actors, and Stakeholders

### 2.1.1 Persona

John Smith is a 35 years old marketing manager who looks for different strategies to sell his products. He is always in search of the products which he can buy based on the people's reviews, thus helping him in escalating his business. He has a good knowledge of using computers as he often spends time for social networking, banking or just browsing on the web for the products. He seeks to buy products which are on discount so that he can sell them to his clients. He deals with his clients online and prefers to have the payment transactions done online. Therefore, he is in need of the system which can help him buy these discounted products online, and also helping him find new clients.

### 2.1.2 Stakeholders and Actors

**Customers:** Customers are the primary actors that use the Linq E-Commerce Store application to shop products online. They can register themselves online and login to the application to select the products they want to purchase. They can add these products to the cart and manipulate the quantity of these products on the fly. After the payment is processed, they can review the products, helping other customers to purchase products using this application.

**Administrator:** Administrator forms the heart of Linq E-Commerce Store. An administrator can manage the shopping cart, to empty those carts that the customers never checked out. He/she is able to manage product catalogs by adding discount coupons to different products, add new products and edit existing products in the system. He also has the authority to manage orders and inventory, where he can add / remove / edit suppliers' information and keep a track of different product levels.

**Payment Gateways:** Linq E-Commerce Store is integrated with PayPal Website Payments Standard and Google Checkout which are used to process payments. These two payment gateways send customers to a 3rd party site to accept payment, then notify the customers that payment has been received. Linq E-Commerce Store sends these payment systems, an order ID, an order name, a quantity of 1 (there is only 1 order), and a total for the order. Once the order is complete, both systems use a payment notification system to contact Linq E-Commerce Store. This system also supports other payment gateways like Authorize.NET AIM, and PayPal PayFlow Pro. These are the gateways that allow the customers to accept credit cards directly on the website.

**Suppliers:** Suppliers are secondary actors who supply the products they want to sell and publish on the site. These suppliers are managed by the administrator. They send the request to the administrator with their personal information and the detailed information related to the products that they want to sell. Administrator in turn, with the support of the suppliers will be able to add the suppliers' information and the products which will be accessible to the customers.

## 2.2 Informal Use Case

### 2.2.1 Queue Card Length Use Case

**Customer Goal Use-cases**

| **Customer Registration** |
|---|
| 1. Customer enters profile information to get registered as an authorized user. |
| 2. System validates the information entered by the customer and stores the information into the database. |
| 3. Customer gets the notification on UI that the profile has been successfully created. |

| **Manage Profile** |
|---|
| 1. Customer opens relevant form to update his profile. |
| 2. System provides the form filled with the customer details he wants to edit. |
| 3. Customer provides the information in his profile that need to be updated. |
| 4. System validates the information entered by the customer and stores the information into the database. |

| **Maintain Billing Information** |
|---|
| 1. Customer enters billing information to purchase the products. |
| 2. System checks whether the customer has entered all the mandatory fields and has entered all input data in valid format. |
| 3. Customer gets the notification on UI that the products were successfully purchased and receives an e-mail about the purchased products. |

| **Maintain Wish list** |
|---|
| 1. Customer requests to add/remove products based on his needs on the wish list. |
| 2. The system approves the customer's requests. |
| 3. The customer can also add the products to the shopping cart from the wish list. |

| **Search Products** |
|---|
| 1. Customer enters the product name he wants to view / purchase in the search bar. |
| **2.** The system filters the product from the list of products in the database and the result is shown to the customer. |

| **Buy Products** |
|---|
| 1. Once the customer adds the desired products to the cart, he shall request to checkout. |
| 2. The system acknowledges the request and redirects the customer to a secure payment gateway. |
| 3. The customer enters the payment details and the billing/shipping information. |
| 4. Once the payment is processed, the customer will be notified. |

**Administrator Goal Use-cases**

| **Manage Shopping Cart** |
|---|
| 1. The customer requests to add the desired products to the cart. |
| 2. The system acknowledges the request and adds to the shopping cart. |
| 3. The customer shall log out of the application without wishing to checkout products. |
| 4. The system deletes the shopping cart after certain point of time. |

| **Manage Product Catalog** |
|---|
| 1. The administrator requests to add discount coupons to the products in the inventory. |
| 2. The system acknowledges the request and adds the discount coupons. |
| 3. The administrator requests to add new products to the inventory. |
| 4. The system acknowledges the request and adds new product to the inventory. |
| 5. The administrator requests to edit the details of existing products in the inventory. |
| 6. The system acknowledges the request to edit the details |

| **Manage Orders** |
|---|
| 1. The administrator keeps track of the orders placed by different customers. |
| 2. The administrator approves the order. |
| 3. The administrator can see details of each order. |

| **Manage Inventory** |
|---|
| 1. The administrator keeps a track of product levels in the inventory. |
| 2. The administrator adds new product order or edits existing product order in the application. |
| 3. The administrator enters the details of the product. |
| 4. The administrator adds new suppliers or edits existing suppliers. |
| 5. The administrator enters the details of the suppliers. |

**Fully Dressed use case for "Buy Products"**

| |
|---|
| **Use Case UC6:** Buy Products |
| **Primary Actor:** Customer |
| **Stakeholder and Interest**: Customer, Admin and Supplier |
| **Preconditions**:<br>• Customer is identified and authenticated.<br>• Product is available in the inventory.<br>• Authorize.Net account exists, if payment needs to be done through Authorize.net payment gateway.<br>• PayPal account exists, if payment needs to be done through PayPal payment gateway. |
| **Post conditions:**<br>• Entered quantity related to the product is checked out.<br>• Inventory is updated.<br>• Receipt is generated.<br>• Payment authorization is updated. |
| **Main Success Scenario:**<br>1. Customer logs into Linq E-Commerce Store to checkout with products to purchase.<br>2. The system starts a new sale.<br>3. The system enters product identifier.<br>4. System records sale line item and presents products description, price, and running total.<br>*System repeats steps* 3-4 *until indicates done.*<br>5. System tells Customer the total, and asks for payment.<br>6. Customer selects payment gateway.<br>7. Customer enters billing and shipping information.<br>8. Customer pays and System handles payment.<br>9. System logs completed sale and sends sale and payment information to the external Accounting system.<br>10. System updates inventory.<br>11. System presents receipt. |
| **Extensions (Alternative Flow of Events):**<br>1a) Customer is not able to successfully log into the Linq E-Commerce Store.<br>8a) System fails to communicate with external systems for payment.<br>8b) System prompts the customer to select an alternate payment method. |

| |
|---|
| 10a) System fails to update the inventory. |

## 2.3. UML Domain Diagram

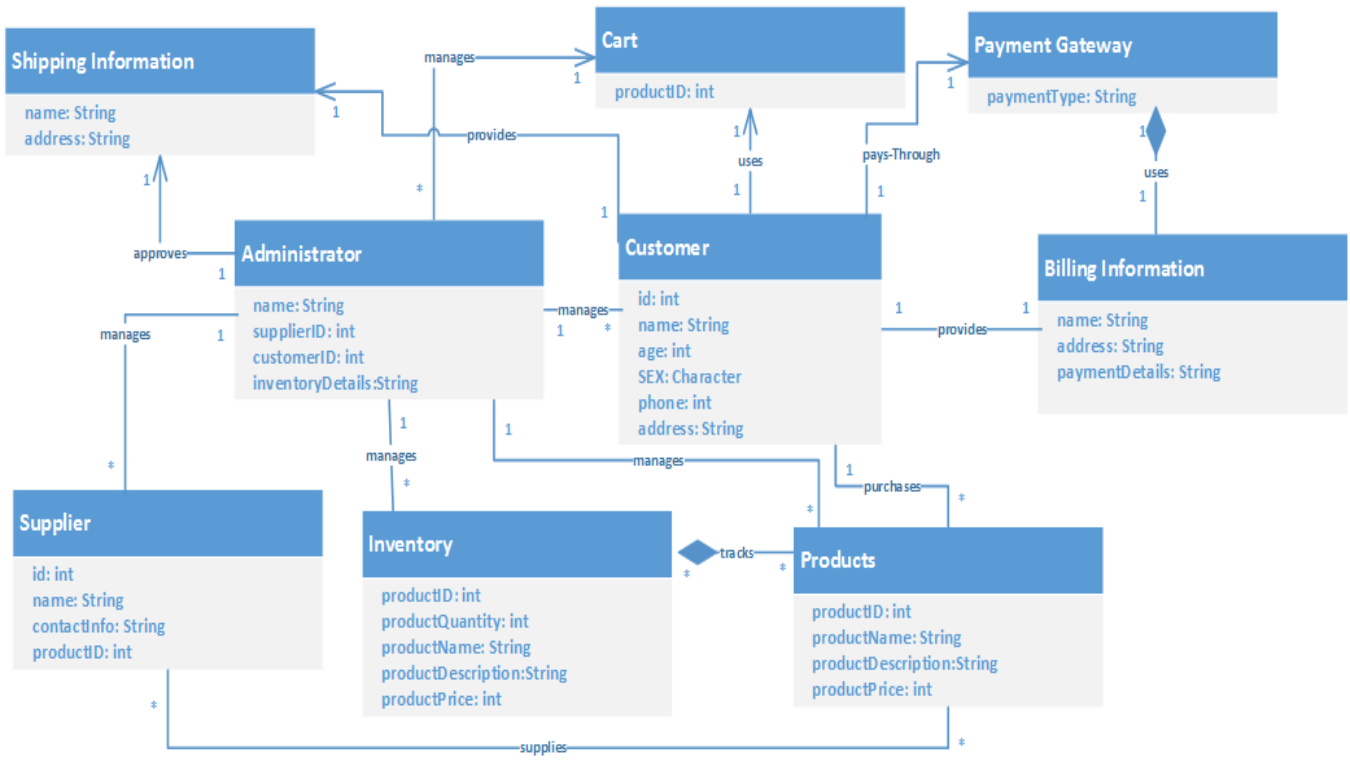


***Figure 2: UML Domain Diagram***

The conceptual class **CUSTOMER** holds the information about authorized users registered as a customer. Each customer provides the shipping and billing information which is depicted by the **SHIPPING INFORMATION** and **BILLING INFORAMTION** conceptual classes respectively. **CUSTOMER** looks for the products he wants to buy and the details of these products are held in the **PRODUCTS** conceptual class. Once he knows the products he wants to buy, he inserts the products into the cart depicted by **CART.** He finalizes the purchase by paying for the products and the details will be given by the **PAYMENT GATEWAY** conceptual class. **SUPPLIER** conceptual class contains the information about the authorized users who are responsible to supply the products they want to sell. **INVENTORY** conceptual class holds information about the availability of the products in the e-commerce store. **ADMINISTRATOR** holds the information about the authorized users who manage the customers, suppliers, products and inventory.

# 3. MILESTONE – 3

## 3.1. Class Diagram of Actual System

### 3.1.1 UML Domain Diagram

Please refer to *Figure 1: Domain Model*.

### 3.1.2 UML Class Diagram

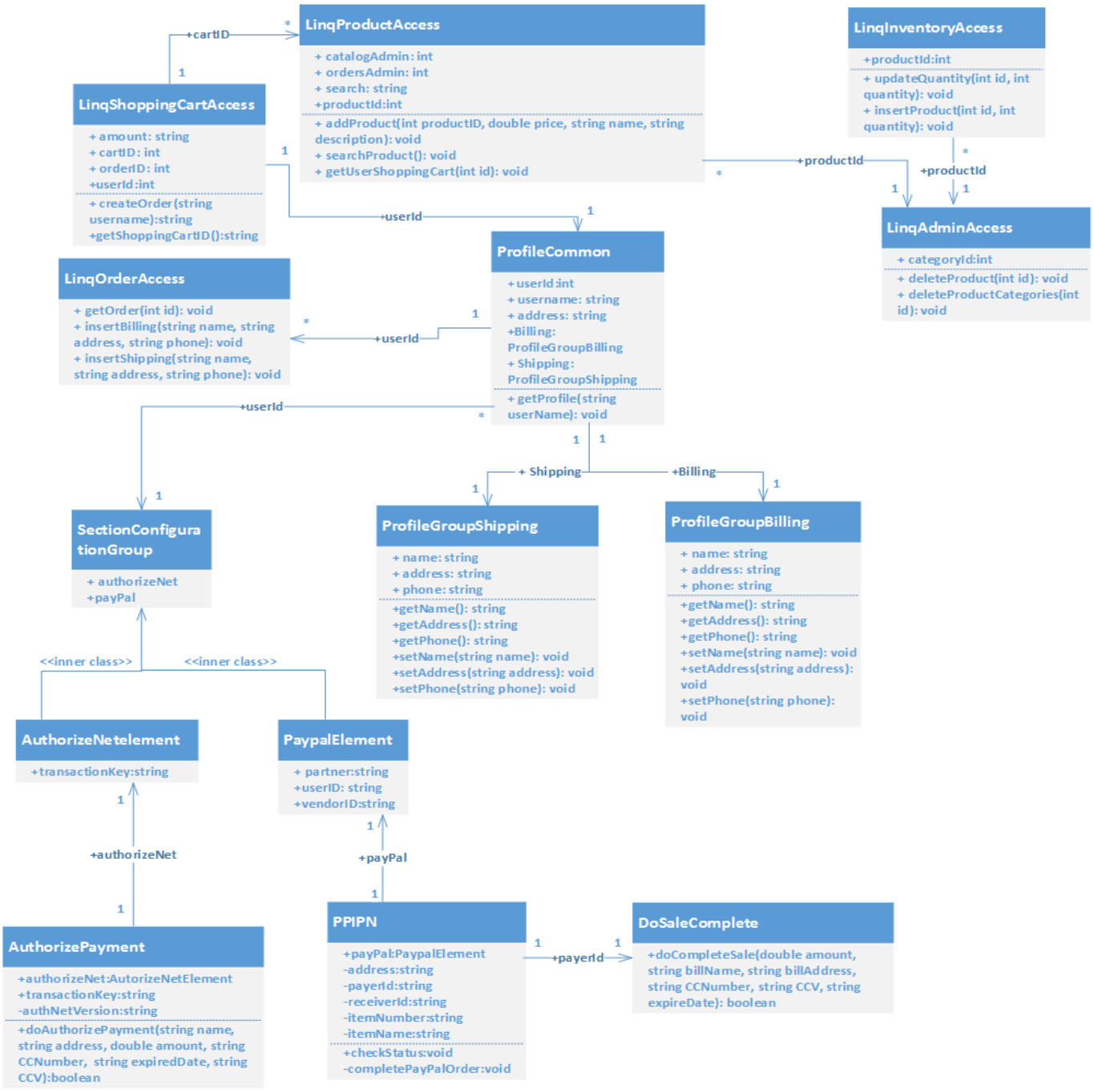


***Figure 3: UML Class Diagram***

### 3.1.3 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.

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 nested with CD: **SectionConfigurationGroup** which in turn is analogous to DM: **PaymentGateway** conceptual class. The 2 nested 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 hindsight, we thought that the entire payment functionality is handled only by the DM: **PaymentGateway** conceptual class.

The relationships between the classes seem to be obviously defined where the variables in a class are being used in the related class. For instance, CD: **AuthorizePayment** class is related to CD: **AuthorizeNetElement** because the object of CD: **AuthorizeNetElement** is used in CD: **AuthorizePayment** class.

In terms of architecture, 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. On the other

hand, we also observe that the different payment strategies use different classes and methods which have common functionality and can be extracted into a single class. This also leads to High Coupling which is not good in terms of the architecture of the system. 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.

### 3.1.4 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();

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.

## 3.2. Code Smell and Possible Refactoring

### 3.2.1 Code Smells

We have found the following code smells with respect to our project. However, we will be explaining in detail about the **Duplicated Code/Large Class** code smell in section 3.2.2.

**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.

**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.

**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.

### 3.2.2 Possible Refactoring

One of the major problems found in the architecture of the system was the duplicated code and large class. Currently, there are 2 payment functionalities implemented – Authorize .Net and PayPal. These 2 classes perform similar functionalities but have different methods leading to redundant data.

At a high level, we found that we can refactor the payment functionality which has duplicated code / large class code smell in Linq E-Commerce Store project in the following steps:

1) First, we will create a new class called **ValidatePayment**
2) Second, we create a new method called "doAuthorizePayment" in **ValidatePayment**
3) Thirdly, we use "Move Method" refactoring technique to move the "doSaleComplete" and "doAuthorizePayment" methods from **DoSaleComplete** and **AuthorizePayment** classes respectively and insert into the "doAuthorizePayment" method in **ValidatePayment**.
4) We will now delete the **AuthorizePayment** and **DoSaleComplete** classes to reduce code duplication.
5) Finally, we will now test this new class and the method to ensure that the behaviour is not changed.

We can demonstrate this refactoring by the following UML class diagram:

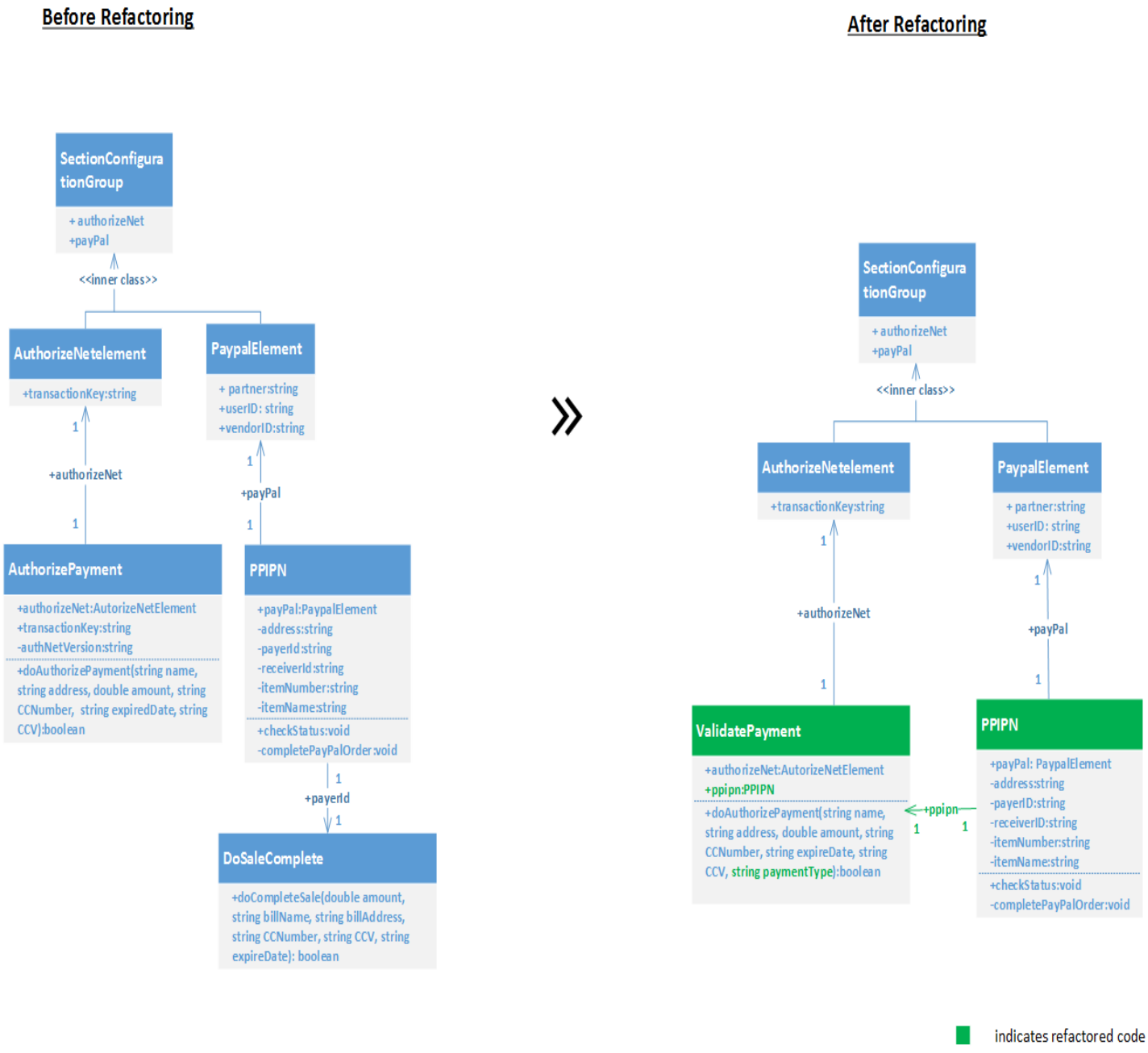


*Figure 4: UML Class Model before & after refactoring for Payment*

### 3.2.3 UML Class Diagram after all possible refactoring

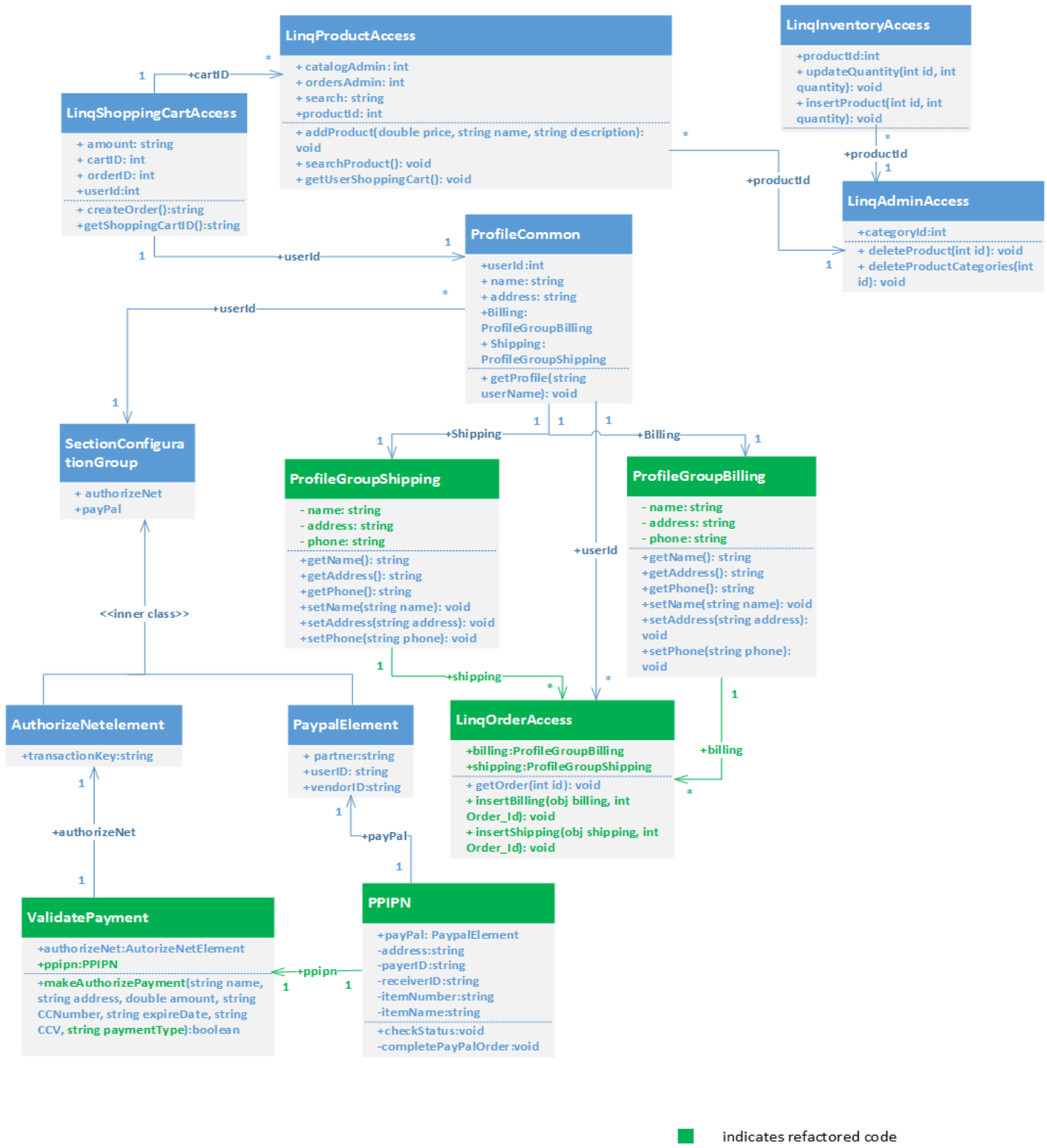


*Figure 5: Refactored Class Diagram*

### 3.2.4 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 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 create a new class namely **ValidatePayment** and insert a new method namely "makeAuthorizePayment". We then extract the "doSaleComplete" and "doAuthorizePayment" methods from **DoSaleComplete** and **AuthorizePayment** classes into "makeAuthorizePayment" which will in turn 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".

### 3.2.5 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();
        }
```