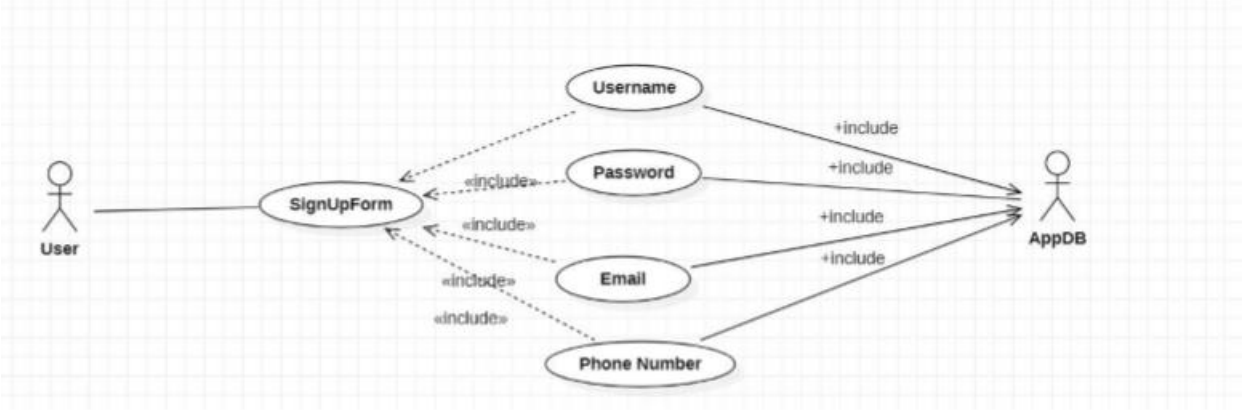
REPORT OF OBJECT-ORIENTED SOFTWARE DESIGN (DESIGN PATTERN)
- - - Group 09: Application for fast delivery - - -

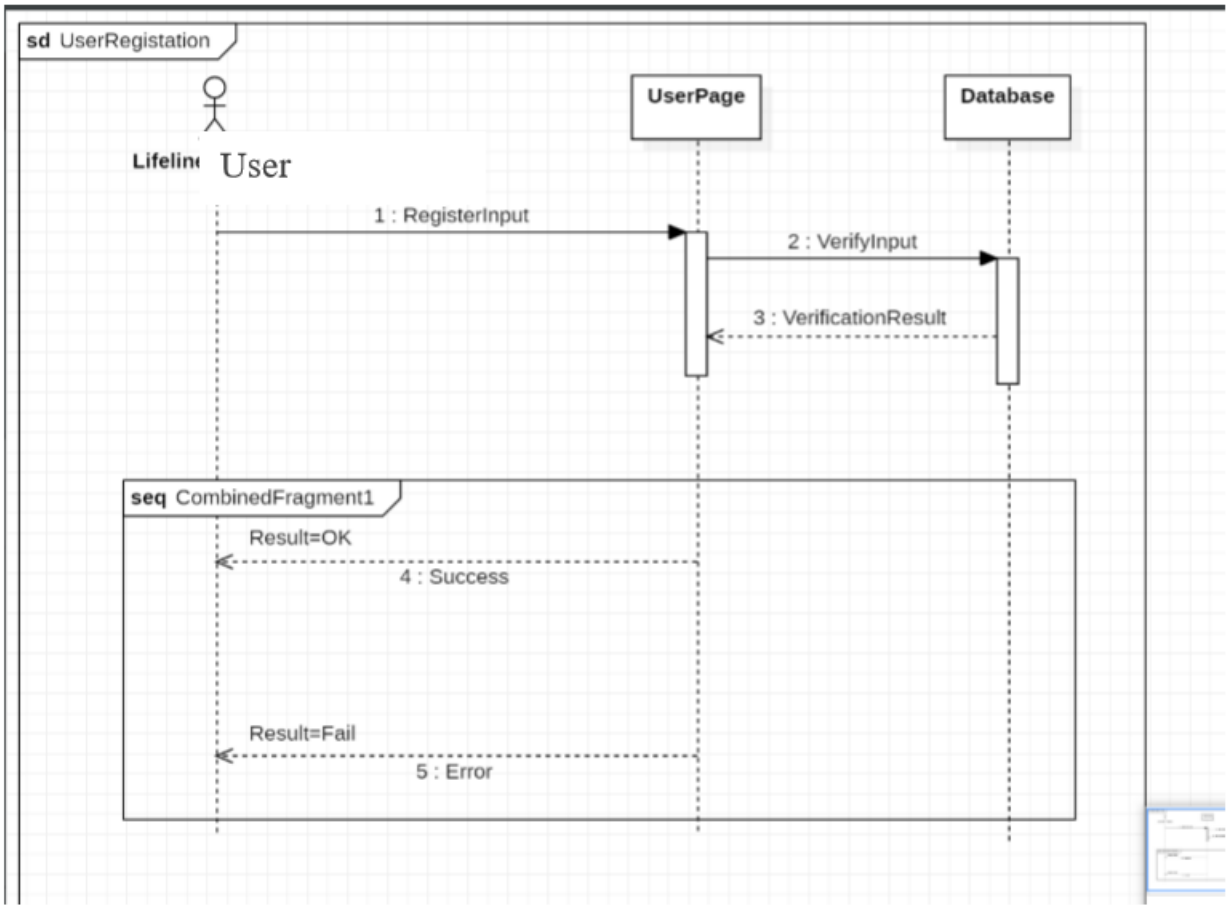
1. DESIGN PATTERN - BRIDGE:

- Implemented Function: Register of Customer, Shipper and Adminstrator.
- Name of Pattern implemented: Bridge
- Reason apply this design pattern:
    + This pattern has a simple architecture so it is easy to understand.
    + This pattern suitable to solve the problem in software.
    + Using this pattern it help us easy to develop our software, especially the register function of Customer, Shipper and Administator when necessary.
    + Code is simple to read and understand, easy to fix if has any problems.
    + It is too long to write regist function for 3 object (Customer, Shipper, Admin) so this method help us to write the code clearer and as minized as possible, avoid wasting time and make the code longer and complex.
- Pattern characteristic:
    + A type of Structural Pattern.
    + This pattern work seems like a bridge to connect abstract classes.
    + Devide the abstract parts and implemented parts.
    + We can devide functions into the smaller parts base on the abstract class and interface. After that with using this pattern we can connect to abstract class together. This help to develope many objects have a same properties and method. For example: We have Customer, Shipper and Admin at the abstraction part and CheckValid before SavingAccount at the implement part.

USE CASE DIAGRAM:



SEQUENCE DIAGRAM:



- C# CODE:

```
using System;

namespace Bridge
{
    class Program
    {
        static void Main(string[] args)
        {

            USER customer = new CUSTOMER(new CheckingAccount());
            customer.RegisterAccount();
        }
    }

    public interface Account
    {
        void RegisterAccount(string USERNAME, string PASSWORD);
    }
```

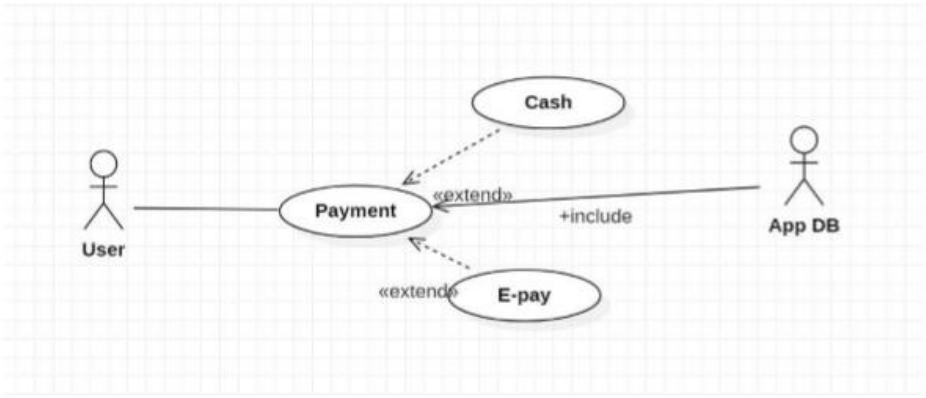
```
public class CheckingAccount : Account
{
    public void RegisterAccount(string USERNAME, string PASSWORD)
    {
        if(USERNAME != "Admin")
        {
            Console.WriteLine("CHECKED USERNAME");
        }
        else
        {
            Console.WriteLine("Invalid");
        }

    }
}
public class SavingAccount: Account
{
    public void RegisterAccount(string USERNAME, string PASSWORD)
    {
        Console.WriteLine("SAVING ACCOUNT");
    }
}

public abstract class USER
{

    protected Account account;

    public USER(Account account)
    {
        this.account = account;
    }

    public abstract void RegisterAccount();
}

public class CUSTOMER : USER
{
       public CUSTOMER(Account account) : base(account)
       {

       }
       public override void RegisterAccount()
       {
            string Username, Password;
            Console.WriteLine("Register: ");
            Console.Write("Input Username: ");
            Username = Console.ReadLine();
            Console.Write("Input Password: ");
            Password = Console.ReadLine();
            Console.WriteLine("REGISTED AS CUSTOMER");
            account.RegisterAccount(Username,Password);
        }
}
public class Shipper : USER
{
    public Shipper(Account account) : base(account)
    {
```

```
    }
    public override void RegisterAccount()
    {
        string Username, Password;
        Console.WriteLine("Register: ");
        Console.Write("Input Username: ");
        Username = Console.ReadLine();
        Console.Write("Input Password: ");
        Password = Console.ReadLine();
        Console.WriteLine("REGISTED AS SHIPPER");
        account.RegisterAccount(Username,Password);
    }
}

public class Admin : USER
{
    public Admin(Account account) : base(account)
    {

    }
    public override void RegisterAccount()
    {
        string Username, Password;
        Console.WriteLine("Register: ");
        Console.Write("Input Username: ");
        Username = Console.ReadLine();
        Console.Write("Input Password");
        Password = Console.ReadLine();
        Console.WriteLine("REGISTED AS ADMIN");
        account.RegisterAccount(Username,Password);
    }
}
   }
```

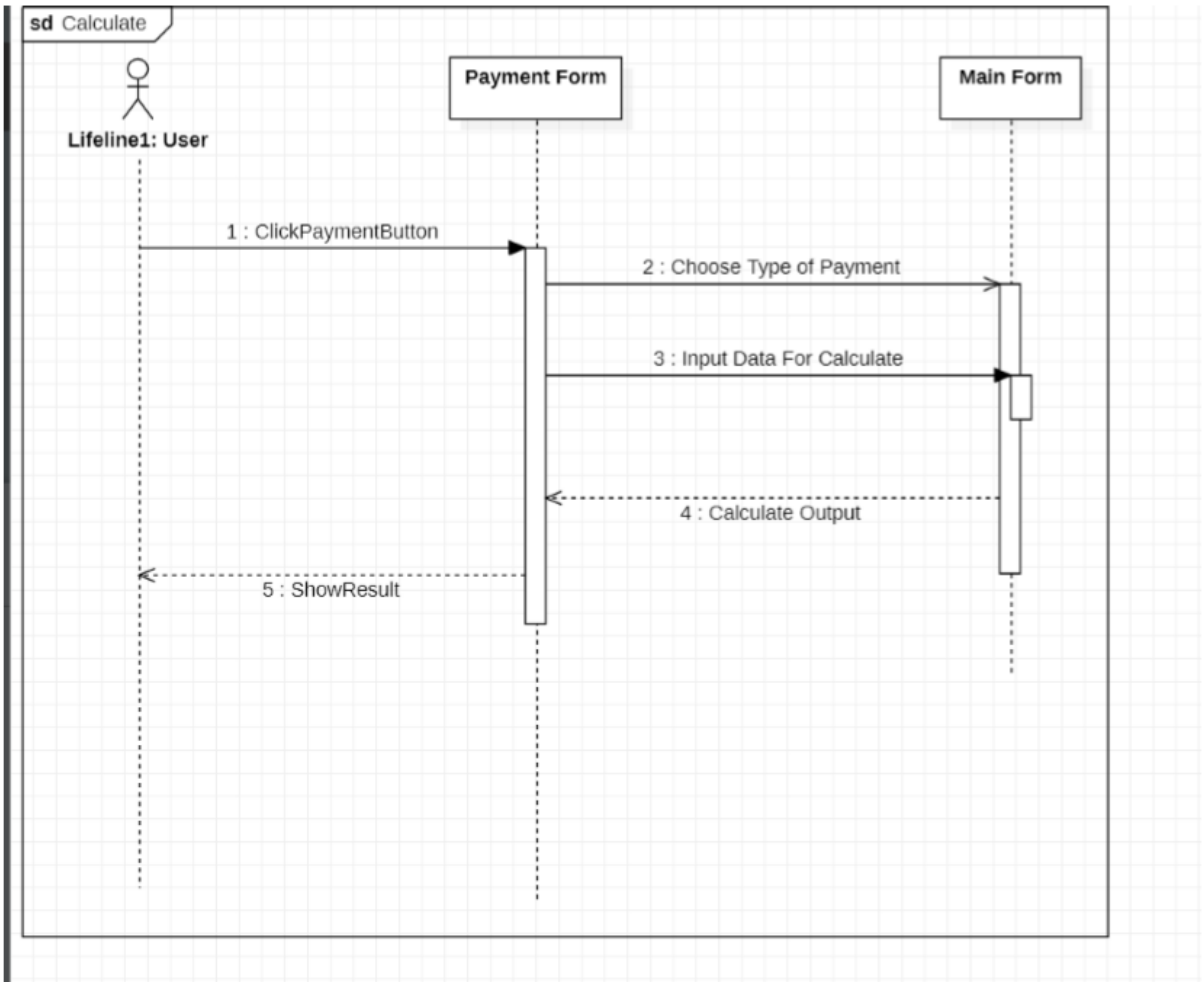
2. DESIGN PATTERN - FACADE:

- Implemented Function: Calculate Payment.
- Name of Pattern implemented: Facade
- Reason apply this design pattern:
    + This pattern has a simple architecture so it is easy to understand.
    + This pattern suitable to solve the problem in software.
    + Code is simple to read and understand, easy to fix if has any problems.
    + Allow system/software work thru the façade class, make the code not complex.
    + Easier to extend payment method and related functions.
- Pattern characteristic:
    + A type of Structural Pattern.
    + Facade Pattern provide a Facade class to interact with the subsystem.
    + Hide the complex methods of subsystem.

USE CASE DIAGRAM:



SEQUENCE DIAGRAM:



- C# CODE:

```
using System;
```

```
namespace Facade
{
    class Program
    {
        static void Main(string[] args)
        {
            ShippingPaymentFacade.getInstance().CalculateShipment();
        }
    }

    public class AccountService
    {

        public void getAccount(String email)
        {
            Console.WriteLine("Getting the account of " + email);
        }
    }

    public class PaymentService
    {

        public void paymentByPaypal()
        {
            Console.WriteLine("Payment by Paypal");
        }

        public void paymentByCreditCard()
        {
            Console.WriteLine("Payment by Credit Card");
        }

        public void paymentByEbankingAccount()
        {
            Console.WriteLine("Payment by E-banking account");
        }

        public void paymentByCash()
        {
            Console.WriteLine("Payment by Cash");
        }
    }

    public class ShippingService
    {

        public void freeShipping()
        {
            Console.WriteLine("Free Shipping");
        }

        public void standardShipping()
        {
            Console.WriteLine("Standard Shipping");
        }

        public void expressShipping()
        {
```

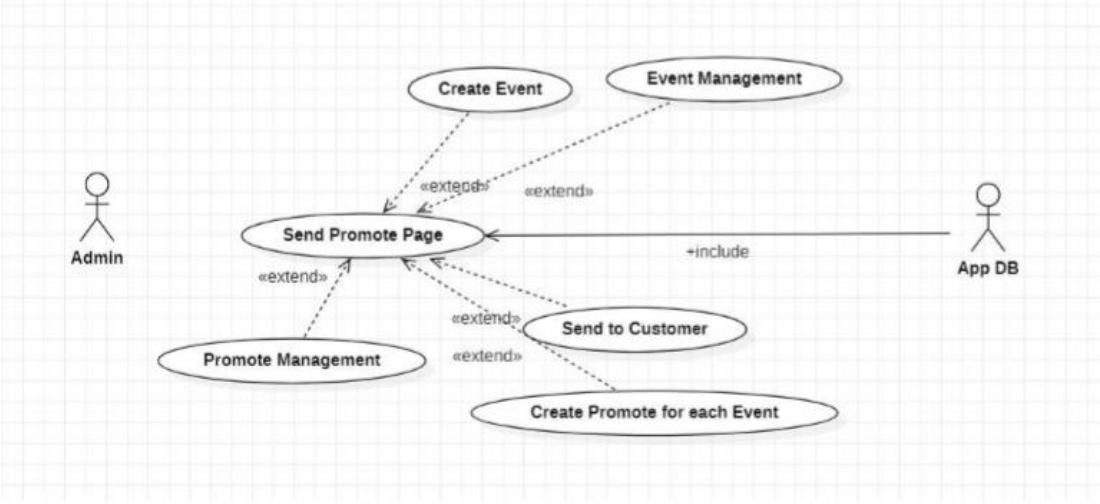
```
        Console.WriteLine("Express Shipping");
    }
}

public class ShippingPaymentFacade
{
    private AccountService accountService;
    private PaymentService paymentService;
    private ShippingService shippingService;
    private static ShippingPaymentFacade INSTANCE = new ShippingPaymentFacade();

    private ShippingPaymentFacade()
    {
        accountService = new AccountService();
        paymentService = new PaymentService();
        shippingService = new ShippingService();
    }

    public static ShippingPaymentFacade getInstance()
    {
        return INSTANCE;
    }

    public void CalculateShipment()
    {
        Double Total = 0;
        int choice1 = 0;
        int choice2 = 0;
        Console.Write("Input Email to Contact: ");
        String email = Console.ReadLine();
        Console.Write("Input Estimated Kilometer: ");
        Double Kilometer = Double.Parse(Console.ReadLine());
        Console.WriteLine("============================");
        Console.WriteLine("Choose 1: Use Paypal");
        Console.WriteLine("Choose 2: Use Credit Card");
        Console.WriteLine("Choose 3: Use E-banking account");
        Console.WriteLine("Choose 4: Use Cash");
        Console.Write("Please choose your way to pay:");
        choice1 = Int32.Parse(Console.ReadLine());
        Console.WriteLine("============================");
        Console.WriteLine("Choose 1: Free Shipping");
        Console.WriteLine("Choose 2: Standard Shipping");
        Console.WriteLine("Choose 3: Express Shipping");
        Console.Write("Please choose type of Delivery:");
        choice2 = Int32.Parse(Console.ReadLine());
        Console.Clear();
        accountService.getAccount(email);
        if(choice1 == 1){
            paymentService.paymentByPaypal();
        }else if (choice1 == 2){
            paymentService.paymentByCreditCard();
        }else if (choice1 == 3)
        {
            Total = Kilometer + 2;
            paymentService.paymentByEbankingAccount();
        }else{
            Total = Kilometer + 3;
            paymentService.paymentByCash();
```

```
            }

            if (choice2 == 1)
            {
                shippingService.freeShipping();
            }
            else if (choice2 == 2)
            {
                Total = Kilometer * 5;
                shippingService.standardShipping();
            }
            else
            {
                Total += Kilometer * 10;
                shippingService.expressShipping();
            }
            Console.WriteLine("Total Price: " + Total + "$");
            Console.WriteLine("Calculate Done");
        }
    }
}
```

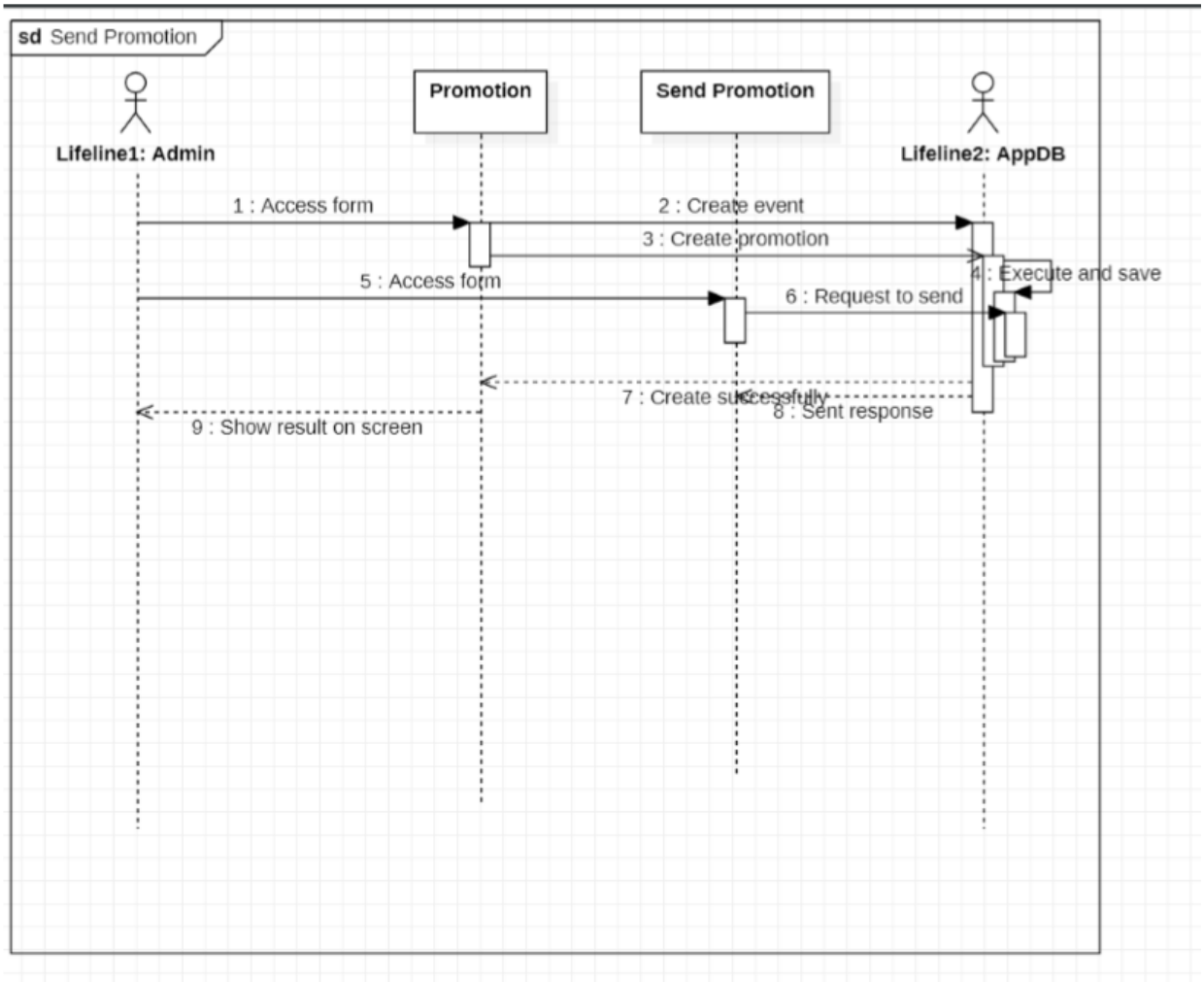
3. DESIGN PATTERN – FACTORY:

- Implemented Function: Send Promotion to customer.
- Name of Pattern implemented: Factory
- Reason apply this design pattern:
  - + This pattern has a simple architecture so it is easy to understand.
  - + This pattern suitable to solve the problem in software.
  - + Code is simple to read and understand, easy to fix if has any problems.
  - + Easier to create objects and life cycle of objects.
- Pattern characteristic:
  - + A type of Creational Design Pattern.
  - + Manage and return a suitable object this help to init an object more faster.
  - + Seems like a factory work to create a products, in this situation it is an object.
  - + Has many child classes base on the root one.

USE CASE DIAGRAMS:



SEQUENCE DIAGRAM:



- C# CODE:

```
using System;
using System.Net.Mail;
namespace FactoryMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            string  PromotionName = "";
            string  ExpiryDate = "";
            int     Quantity = 0;
            int i = 1;
            while (i != 0)
            {
                Console.WriteLine("Choose 0: Exit Programme");
                Console.WriteLine("Choose 1: Create Basic Promotion");
                Console.WriteLine("Choose 2: Send Email");
                Console.Write("Your Choice: ");
                i = Int32.Parse(Console.ReadLine());
```

```
                if (i == 1)
                {
                    Promotion promotion =
PromotionFactory.GetPromotion("BasicPromotion");
                    PromotionName = promotion.GetPromotionName();
                    ExpiryDate = promotion.GetExpireTime();
                    Quantity = promotion.GetQuantity();
                }else if( i== 2)
                {
                    String BodyMessage = "Promotion Name: " + PromotionName + "\n" +
"Expiry Date: " + ExpiryDate + "\n" + "Quantity: " + Quantity;
                    MailMessage message = new
MailMessage("laquoctoan30062000@gmail.com","laquoctoan3006@gmail.com","Promotion",
BodyMessage);
                    SmtpClient client = new SmtpClient("smtp.gmail.com",587);
                    client.EnableSsl = true;
                    client.Credentials = new
System.Net.NetworkCredential("laquoctoan30062000@gmail.com", "");
                    client.Send(message);
                }
                Console.WriteLine("\n\n================================\n\n");
            }
        }
    }
    public interface Promotion
    {
        String GetPromotionName();
        String GetExpireTime();
        int GetQuantity();
    }

    public class BasicPromotion : Promotion
    {
        public String GetPromotionName()
        {
            Console.Write("Input the Promotion Name: ");
            String PromotionName = Console.ReadLine();
            return PromotionName;
        }
        public String GetExpireTime()
        {
            Console.Write("Input the Promotion Expiry Date: ");
            String ExpiryDate = Console.ReadLine();
            return ExpiryDate;
        }
        public int GetQuantity()
        {
            Console.Write("Input the Promotion Quantity Turns: ");
            int Quantity = Int32.Parse(Console.ReadLine());
            return Quantity;
        }
    }

    public class LuxuryPromotion : Promotion
    {
        public String GetPromotionName()
        {
            Console.Write("Input the Promotion Name: ");
```

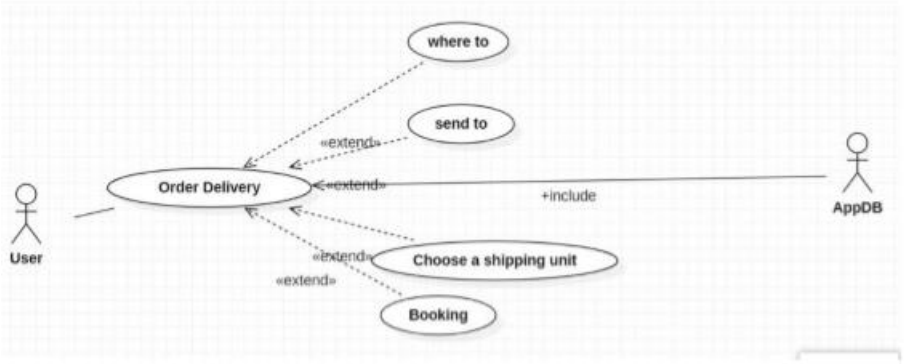
```
            String PromotionName = Console.ReadLine();
            return PromotionName;
        }
        public String GetExpireTime()
        {
            Console.Write("Input the Promotion Expiry Date: ");
            String ExpiryDate = Console.ReadLine();
            return ExpiryDate;
        }
        public int GetQuantity()
        {
            Console.Write("Input the Promotion Quantity Turns: ");
            int Quantity = Int32.Parse(Console.ReadLine());
            return Quantity;
        }
    }

    public class PromotionFactory
    {

        private PromotionFactory()
        {
        }

        public static Promotion GetPromotion(String PromotionType)
        {
            switch (PromotionType)
            {

                case "BasicPromotion":
                    return new BasicPromotion();
                case "LuxuryPromotion":
                    return new LuxuryPromotion();
                default:
                    return null;
            }
        }

    }

}
```

4. DESIGN PATTERN – ABSTRACT FACTORY:

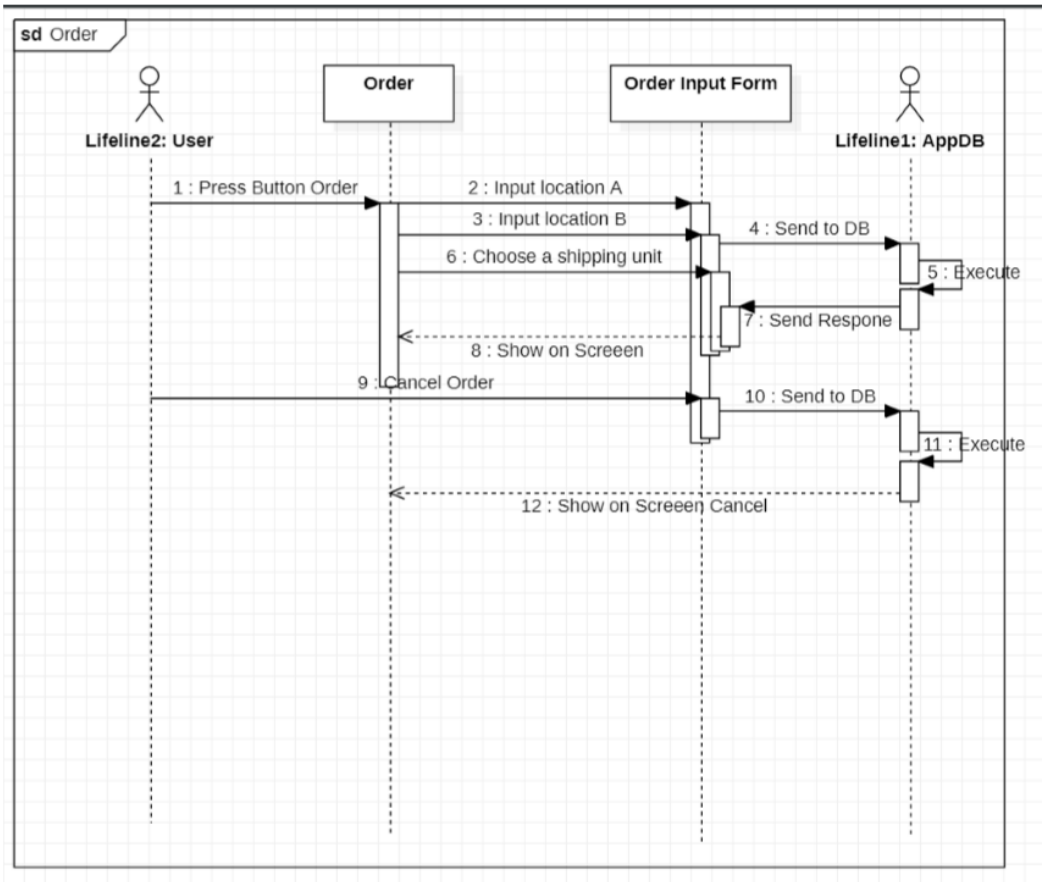
- Implemented Function: Order shipment.
- Name of Pattern implemented: Abstract Factory
- Reason apply this design pattern:
    + This pattern has a simple architecture so it is easy to understand.
    + Code is simple to read and understand, easy to fix if has any problems.
    + Try to not re-coding the similar code and this make the code shorter.
    + On the other hand, A shipment always include many properties and with each properties also have many things in there so this help developer easier to manage the code and create, manage many object relate to the class.
- Pattern characteristic:
    + A type Creational pattern.

+ This work seems like a super factory that include many factories in there.
+ Each factory created can also create objects by using factory pattern.

USE CASE DIAGRAM:



SEQUENCE DIAGRAM:



- C# CODE:

```
using System;

namespace AbstractFactory
{
    class Program
    {
        static void Main(string[] args)
        {
            int Choice = 0;
            Console.WriteLine("Choose 1: For Fast Shipping With Motorbikes");
            Console.WriteLine("Choose 2: For Fast Shipping With Truck");
            Console.Write("Your Choice: ");
            Choice = Int32.Parse(Console.ReadLine());
            if(Choice == 1)
            {
                FastShipment fastShipment = new FastShipment();
                fastShipment.Show();
            }
            else
            {
                TruckShipment truckShipment = new TruckShipment();
                truckShipment.Show();
            }
        }
    }

    public class FastShipment : OrderFactory
    {
        public override ShipmentLength GetDetails1()
        {
            return new Under10km();
        }

        public override ShipmentTime GetDetails2()
        {
            return new FastSpeed();
        }

        public override ShipmentVehicles GetDetails3()
        {
            return new Motorbikes();
        }
    }

    public class TruckShipment : OrderFactory
    {
        public override ShipmentLength GetDetails1()
        {
            return new Under10km();
        }

        public override ShipmentTime GetDetails2()
        {
            return new FastSpeed();
        }
```

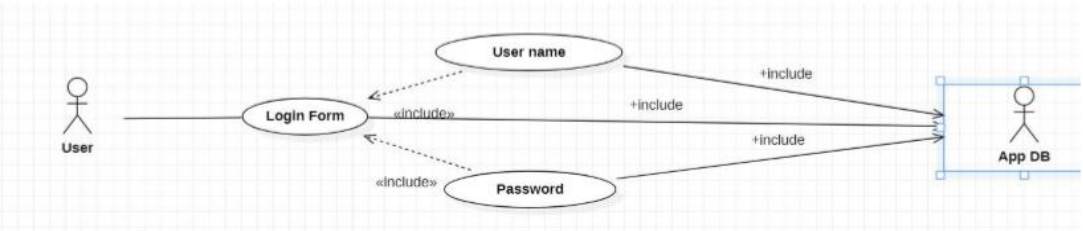
```
    public override ShipmentVehicles GetDetails3()
    {
        return new Trucks();
    }
}

public abstract class OrderFactory
{
    public abstract ShipmentLength GetDetails1();
    public abstract ShipmentTime GetDetails2();
    public abstract ShipmentVehicles GetDetails3();

    public void Show()
    {
        Console.WriteLine(this.GetDetails1());
        Console.WriteLine(this.GetDetails2());
        Console.WriteLine(this.GetDetails3());
    }
}

public interface ShipmentLength
{
    object Details();
}

public interface ShipmentTime
{
    object Details();
}

public interface ShipmentVehicles
{
    object Details();
}

public class Under10km : ShipmentLength
{
    public object Details()
    {
        return "The length is under 10km";
    }
}

public class Morethan10KM : ShipmentLength
{
    public object Details()
    {
        return "The length is more than 10km";
    }
}

public class NormalSpeed : ShipmentTime
{
    public object Details()
    {
        return "At most 5 hours to delivery";
    }
```

```
    }
    public class FastSpeed : ShipmentTime
    {
        public object Details()
        {
            return"At most 2 hours to delivery";
        }
    }

    public class Motorbikes : ShipmentVehicles
    {
        public object Details()
        {
           return "This item will be delivery by motobikes";
        }
    }
    public class Vans : ShipmentVehicles
    {
        public object Details()
        {
            return "This item will be delivery by vans";
        }
    }

    public class Trucks : ShipmentVehicles
    {
        public object Details()
        {
            return "This item will be delivery by trucks";
        }
    }

}
```

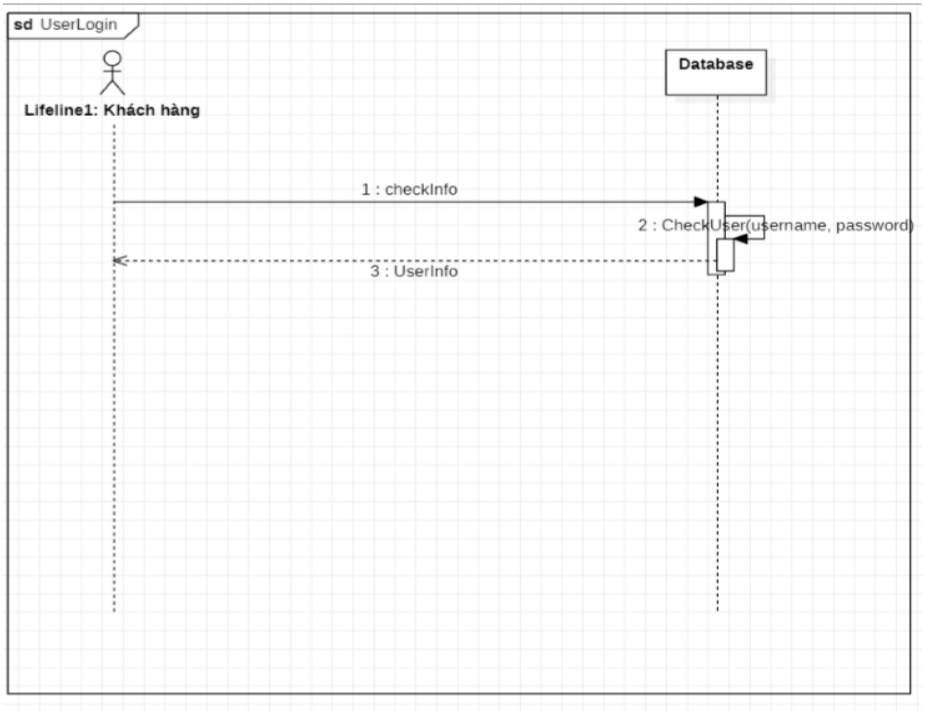
5. DESIGN PATTERN – SINGLETON:

- Implemented Function: Login of Customer, Shipper and Adminstrator.
- Name of Pattern implemented: Singleton
- Reason apply this design pattern:
    + This pattern has a simple architecture so it is easy to understand.
    + Code is simple to read and understand, easy to fix if has any problems.
    + This pattern provide the method so it is suitable to set up for a static value, in this software is login term.
- Pattern characteristic:
    + A type of Creational Design Pattern.
    + Make sure that just one instace is created and provide a method that use only one time in the software.

USE CASE DIAGRAM:



SEQUENCE DIAGRAM:



- C# CODE:

```
using System;

namespace SingletonLogin
{
    class Program
    {
        static void Main(string[] args)
        {
            LoginWithSingleTon Login = LoginWithSingleTon.GetInstance;
        }
    }
    public sealed class LoginWithSingleTon
    {
        private static String Username;
        private static String Password;
        private static LoginWithSingleTon Instance = null;
        public static LoginWithSingleTon GetInstance
        {
```

```
            get
            {
                if (Instance == null)
                {
                    Instance = new LoginWithSingleTon();
                }
                return Instance;
            }
        }
        private LoginWithSingleTon()
        {
            String name, pass;
            Console.Write("Please Input Username: ");
            name = Console.ReadLine().ToString();
            Console.Write("Please Input Password: ");
            pass = Console.ReadLine().ToString();
            LoginWithSingleTon.Username = name;
            LoginWithSingleTon.Password = pass;
            if(LoginWithSingleTon.Username == "Admin" && LoginWithSingleTon.Password ==
"1")
            {
                Console.WriteLine("Welcome " + LoginWithSingleTon.Username + ".
Sucessfully Login !!!");
            }
            else
            {
                Console.WriteLine("Invalid User");
            }

        }
    }
}
```