REPORT OF OBJECT-ORIENTED SOFTWARE DESIGN (DESIGN PATTERN)

- - - Group 09: Application for fast delivery - - -

1. DESIGN PATTERN - BRIDGE:

- Implemented Function: Login 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 login 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 login 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 for login at the implement part.

- C# Code:

```
using System;

namespace BridgePattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Login AsCustomer = new CustomerLogin(new CheckingValid());
            AsCustomer.CheckAccount();
            Login AsShipper = new ShipperLogin(new CheckingValid());
            AsShipper.CheckAccount();
            Login AsAdministrator = new AdminLogin(new CheckingValid());
            AsAdministrator.CheckAccount();
        }
    }

    public interface Account
    {
        void CheckAccount();
    }

    public class CheckingValid : Account
    {
        public void CheckAccount()
        {
                Console.WriteLine("Account was checked !!!");
            }
        }

        public abstract class Login
        {
            protected Account account;

            protected Login()
            {
            }

            public abstract void CheckAccount();
        }

        public class AdminLogin : Login
        {
            public AdminLogin(Account acc)
            {
                this.account = acc;
            }
            public override void CheckAccount()
            {
                Console.WriteLine("You loggin as Administrator: ");
                account.CheckAccount();
            }
        }

        public class ShipperLogin : Login
        {
            public ShipperLogin(Account acc)
            {
                this.account = acc;
            }
            public override void CheckAccount()
            {
                Console.WriteLine("You loggin as Shipper: ");
                account.CheckAccount();
            }
        }

        public class CustomerLogin : Login
        {
            public CustomerLogin(Account acc)
            {
                this.account = acc;
            }
            public override void CheckAccount()
            {
                Console.WriteLine("You loggin as Customer: ");
                account.CheckAccount();
            }
        }
    }
```

2. DESIGN PATTERN - FACADE:

- Implemented Function: 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.

- C# Code:

```
using System;


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

ShippingPaymentFacade.getInstance().buyProductByCashWithFreeShipping("helloworld@gmail.co
m");
        }
    }

    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 static ShippingPaymentFacade INSTANCE = new ShippingPaymentFacade();
        private PaymentService paymentService;

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

        public static ShippingPaymentFacade getInstance()
        {
            return INSTANCE;
        }

        public void buyProductByCashWithFreeShipping(String email)
        {
            accountService.getAccount(email);
            paymentService.paymentByCash();
            Console.WriteLine("Done");
        }
    }
}
```
- Result when run these code above:

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.
- **C# Code:**

```
using System;


namespace FactoryPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Promotion promotion = PromotionFactory.GetPromotion("Discount50Percentage");
            String A = promotion.getPromotionName();
            for (int i=1;i<=10;i++)
            {
                Console.WriteLine("Send " + A + " to Customer " + i);
            }
        }
    }
    public interface Promotion
    {
        String getPromotionName();
        String getTime();
        String CustomerStageForApply();
        int Quantity();
    }

    public class Discount10Percentage : Promotion
    {
        public String getPromotionName()
        {
            return "Discount 10%";
        }
        public String getTime()
        {
            return "1 month";
        }
        public String CustomerStageForApply()
        {
            return "Basic";
        }
        public int Quantity()
        {
            return 5;
        }
    }

    public class Discount50Percentage : Promotion
    {
        public String getPromotionName()
        {
            return "Discount 50%";
        }
        public String getTime()
        {
            return "15 days";
        }
        public String CustomerStageForApply()
        {
            return "Gold";
        }
        public int Quantity()
        {
            return 1;
        }
    }
    public class PromotionFactory
    {
        private PromotionFactory()
        {
        }
        public static Promotion GetPromotion(String promoteName)
        {
            switch (promoteName)
            {

                case "Discount10Percentage":
                    return new Discount10Percentage();

                case "Discount50Percentage":
                    return new Discount50Percentage();

                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.
- **C# Code:**

```
using System;

namespace AbstractFactory
{
    class Program
    {
        static void Main(string[] args)
        {
            FastShipment Delivery = new FastShipment();
            Delivery.Show();
            Console.WriteLine("==============================");
            TruckShipment TruckDelivery = new TruckShipment();
            TruckDelivery.Show();
        }
    }
    public abstract class ShipmentOrder
    {
        public abstract ShipmentLength GetDetails1();
        public abstract ShipmentMethod GetDetails2();
        public abstract ShipmentVehicles GetDetails3();
        public abstract ShipmentStatus GetDetails4();
        public abstract PricePerKilomet GetDetails5();

        public void Show()
        {
            Console.WriteLine(this.GetDetails1());
            Console.WriteLine(this.GetDetails2());
            Console.WriteLine(this.GetDetails3());
            Console.WriteLine(this.GetDetails4());
            Console.WriteLine(this.GetDetails5());
        }
    }

    public class FastShipment : ShipmentOrder
    {
        public override ShipmentLength GetDetails1()
        {
            return new Under10km();
        }
    public override ShipmentMethod GetDetails2()
    {
        return new FastSpeed();
    }

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

    public override ShipmentStatus GetDetails4()
    {
        return new Pending();
    }
    public override PricePerKilomet GetDetails5()
    {
        return new Price1();
    }
}

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

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

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

    public override ShipmentStatus GetDetails4()
    {
        return new Pending();
    }
    public override PricePerKilomet GetDetails5()
    {
        return new Price3();
    }
}

public interface ShipmentLength
{
    object Details();
}

public interface ShipmentMethod
{
    object Details();
}
```

```
public interface ShipmentVehicles
{
    object Details();
}

public interface ShipmentStatus
{
    object Details();
}

public interface PricePerKilomet
{
    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 : ShipmentMethod
{
    public object Details()
    {
        return "At most 5 hours to delivery";
    }
}
public class FastSpeed : ShipmentMethod
{
    public object Details()
    {
        return "At most 2 hours to delivery";
    }
}
public class Motobikes : 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";
        }
    }
    public class WaitingToGet : ShipmentStatus
    {
        public object Details()
        {
            return "This item is in waiting";
        }
    }

    public class Pending : ShipmentStatus
    {
        public object Details()
        {
            return "This order is delivering";
        }
    }

    public class Finish : ShipmentStatus
    {
        public object Details()
        {
            return "This order is finished";
        }
    }
    public class Price1 : PricePerKilomet
    {
        public object Details()
        {
            return "Price: 2$ per KM";
        }
    }

    public class Price2 : PricePerKilomet
    {
        public object Details()
        {
            return "Price: 3.5$ per KM";
        }
    }

    public class Price3 : PricePerKilomet
    {
        public object Details()
        {
            return "Price: 5$ per KM";
        }
    }
}
```

5. DESIGN PATTERN – SINGLETON:

- Implemented Function: Modify Price.
- 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 the price: The price need to be unique and only one in this situation so we apply this pattern for this function.
- 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.

- C# Code:

```
using System;

namespace Singleton
{
    class Program
    {
        static void Main(string[] args)
        {
                     ModifyPriceSingleTon PriceModify = ModifyPriceSingleTon.GetInstance;
        }
    }

       public sealed class ModifyPriceSingleTon
       {
              private static double PricePerKilometer;
        private static ModifyPriceSingleTon Instance = null;
              public static ModifyPriceSingleTon GetInstance
        {
            get
            {
                if(Instance == null)
                {
                    Instance = new ModifyPriceSingleTon();
                }
                return Instance;
            }
        }
        private ModifyPriceSingleTon()
              {
            Console.Write("Price change: ");
            ModifyPriceSingleTon.PricePerKilometer = Double.Parse(Console.ReadLine());
            Console.WriteLine("Price change to: " + PricePerKilometer);
              }
    }
}
```