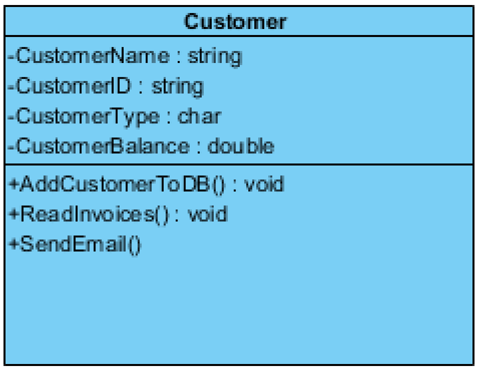
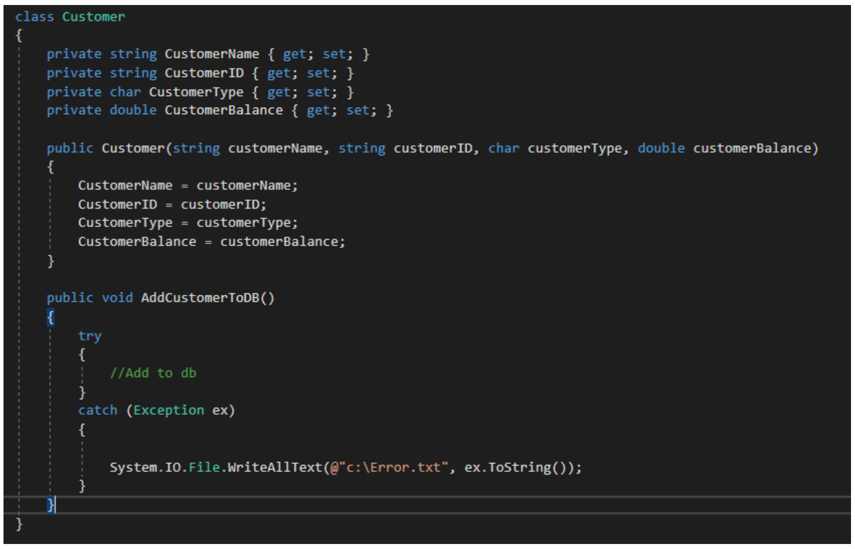
SOLID – assignment 3

We choose a customer object that was loaded with responsibilities.

Figure below before SOLID. (STUPID)





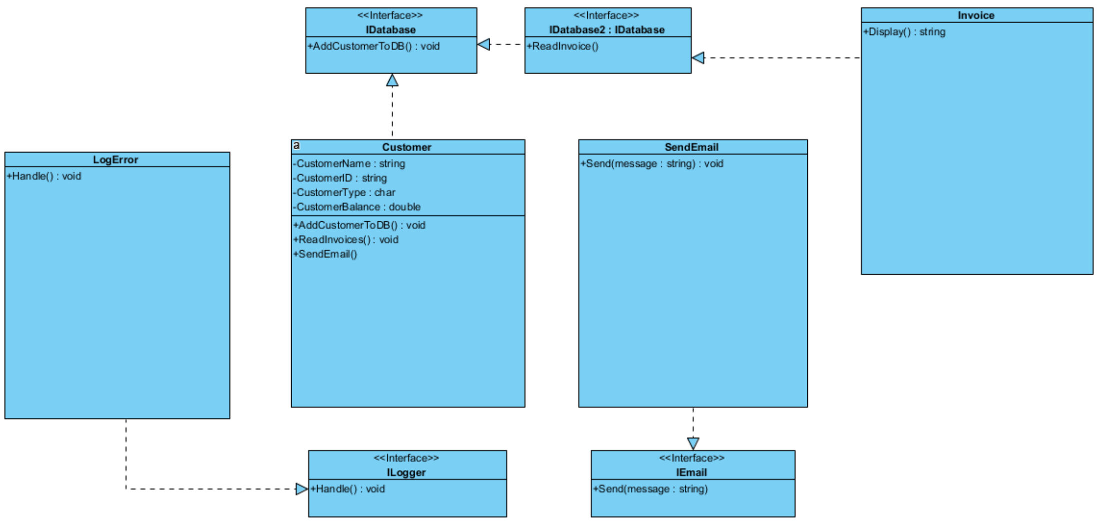
```
class Customer
{
    private string CustomerName { get; set; }
    private string CustomerID { get; set; }
    private char CustomerType { get; set; }
    private double CustomerBalance { get; set; }

    public Customer(string customerName, string customerID, char customerType, double customerBalance)
    {
        CustomerName = customerName;
        CustomerID = customerID;
        CustomerType = customerType;
        CustomerBalance = customerBalance;
    }

    public void AddCustomerToDB()
    {
        try
        {
            //Add to db
        }
        catch (Exception ex)
        {

            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }
}
```

Figure below SOLID



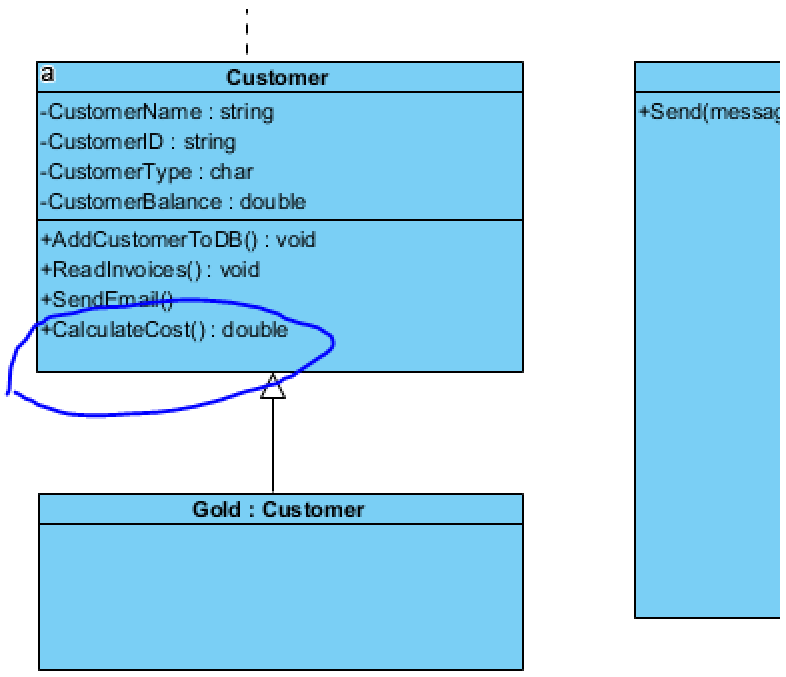
From this STUPID to SOLID, we've learnt that the Customer class had more responsibilities than it should had. Customer class was adding customer to database, making error logs, producing invoices, sending emails.

Instead the customer class should only be adding customers and nothing else.

- Log class to handle all error logs
- Customer class just to handle adding customer
- SendEmail class just to handle sending emails to customer
- Invoice class is just used to produce invoices

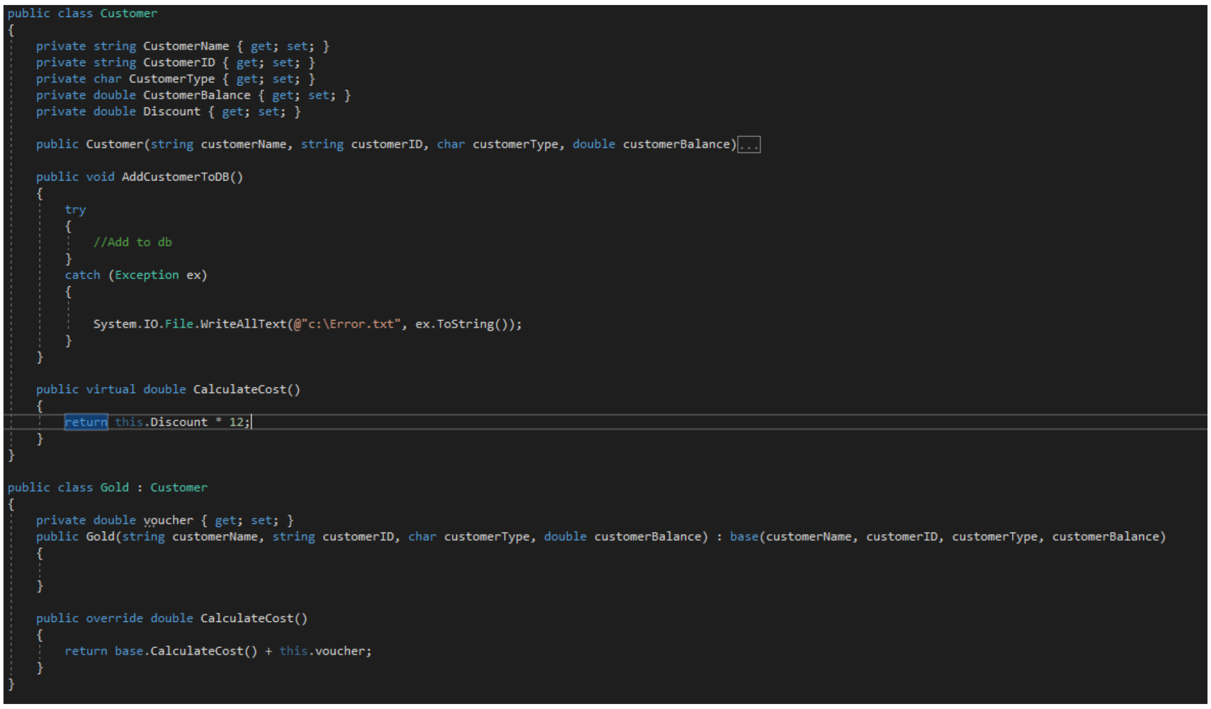
S.O.L.I.D

- Each class must have a single resposnsibility
- Application should be open close this means



Now the same class has a CalculateCost()

However, we've decided to add a customer type, Gold membership receives premium products and discounts.



```
public class Customer
{
    private string CustomerName { get; set; }
    private string CustomerID { get; set; }
    private char CustomerType { get; set; }
    private double CustomerBalance { get; set; }
    private double Discount { get; set; }

    public Customer(string customerName, string customerID, char customerType, double customerBalance)...

    public void AddCustomerToDB()
    {
        try
        {
            //Add to db
        }
        catch (Exception ex)
        {

            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }

    public virtual double CalculateCost()
    {
        return this.Discount * 12;
    }
}

public class Gold : Customer
{
    private double voucher { get; set; }
    public Gold(string customerName, string customerID, char customerType, double customerBalance) : base(customerName, customerID, customerType, customerBalance)
    {

    }

    public override double CalculateCost()
    {
        return base.CalculateCost() + this.voucher;
    }
}
```

We’ve created virtual classes to accommodate that, now we can override the method to make changes as it’s needed.

- Liskov rules, sometimes we may have some customers but they actually don’t buy anything from us so adding a sub class would force it to inherit all its methods, we can use interface to resolve this problem, this way this current does not have to inherit any methods of it’s parent.
- Interface segregation, if our application hits customers greater than we are capable of then we have to facilitate it, one way we can achieve this, if one type of customers just want to add and another type of customer wants to read, we can create 2 interfaces and inherit interface 2 from a, so customer A can only add but customer B can add and read.
- Dependency, earlier we’ve created an Interface “IEmail” with method “Send”, but what if we wanted to use the Send function to send different kind of emails, maybe we want to send offers to customers and send invoices to customers are overdue, now We can create a class SendEmail and we can create subclasses “SendOffer : SendEmail”, “SendInvoice: SendOffer”.

This method will allow us expand our application later without breaking working application.