Question 1.
Consider a simple shopping cart application that uses a Cart class to keep track of the items in the shopping cart and an Order class for processing a purchase as shown in the figure below. When the Order needs to determine the total value of the items in the shopping cart, the ***getOrderTotal*** method will iterates through the list maintain by Cart class. Modify the codes to produce a less coupled design for the classes.

```
public class CartItem{
    public String itemSKU;
    public double unitPrice;
    public int quantity;
}

public class Cart{
    public CartItem[] items;
}

public class Order{
    private Cart cart;
    private double tax;

    public Order(Cart cart, double tax){
        this.cart = cart;
        this.tax = tax;
    }

    public double getOrderTotal(){
        double cartTotal = 0;

        for(int i = 0; i < cart.items.Length; i++){
            cartTotal += cart.items[i].unitPrice * cart.items[i].quantity;
        }
        cartTotal += cartTotal * tax;
        return cartTotal;
    }
}
```

Answer:

```
public class CartItem{
    public String itemSKU;
    public double unitPrice;
    public int quantity;

    public double getItemTotal(){
        return unitPrice * quantity;
    }
}

public class Cart{
    public CartItem[] items;

    public double getCartItemsTotal()
    {
        double cartTotal = 0;
        foreach (CartItem item in items)
        {
            cartTotal += item.getItemTotal();
        }
        return cartTotal;
    }
}

public class Order{
    private Cart cart;
    private double tax;

    public Order(Cart cart, double tax){
        this.cart = cart;
        this.tax = tax;
    }

    public double getOrderTotal(){
        return cart.getCartItemsTotal() * (1 + tax);
    }
}
```

Question 2.

The Java classes below are tightly coupled, rewrite the code using Java interface to produce a loosely coupled code. (5 marks)

```
class CustomerService{
    public static void main(String args[]){
        Customer c = new Customer();

        c.SendMessage();
    }
}

class Customer{
    private Email e;

    public Customer(){
        e = new Email()
    }

    public void SendMessage(){
        e.Send("Hi Customer,");
    }
}

class Email{
    public void Send(String text){
        System.out.println(text + "\nSending product message using EMAIL service!!!");
    }
}
```

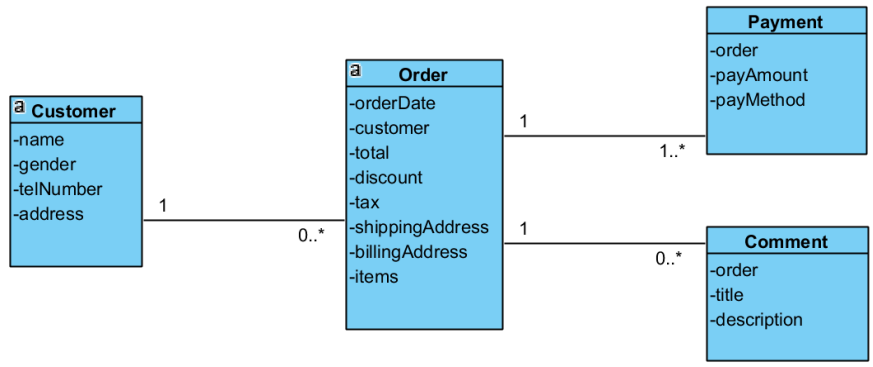
Answer：

```
interface IMessageService{
    public void send(String text);
}

class Email implements IMessageService{
    @Override
    public void send(String text){
        System.out.println(text);
        System.out.println("sending a product message using Email service!");
    }
}

class Customer{
    private IMessageService e;

    public void setM(IMessageService e){
        this.e = e;
    }
    public void sendMessage(String text){
        e.send(text);
    }
}

class CustomerService{
    public static void main(String[] args) {
        Customer c = new Customer();
        IMessageService e = new Email();
        c.setM(e);

        c.sendMessage("hi...");
        }
}
```

Question 3：

Refer to the below class diagram for a simplified order system, redraw the class diagram to include the listed operations. Your new class diagram must show highly cohesive classes.

Operation	Description
addItem()	Add the item to the order
getItems()	Return all items added to the order
pay()	Make payment to the order
getOrderTotal()	Return the total of all the items added to the order
cancelOrder()	Cancel the order and delete all the items added to that order
addComment()	Add a new comment to an order
getAllPayments()	Return all payment made to an order
cancelPayment()	Cancel a particular payment to an order
printOrder()	Print the content of the order
deleteComment()	Delete a comment



Answer