

|  |  |
| --- | --- |
| Name | ID |
| Abdelrahman Hagrass | 195948 |
| Seifeldin Khalil | 195423 |
| Maiada | 203398 |
| Mahmoud Assem | 196735 |
| Mostafa Gado | 192848 |

Group 2

## Software Engineering II (21CSSE01l)

ONLINE FOOD ORDERING SYSTEM

# Table of Contents

[Updated Class Diagram <<Please check the “Phase 3 – diagrams” folder if images are not clear here>> 2](#_Toc102849328)

[Contribution 3](#_Toc102849329)

[Design Patterns Used 5](#_Toc102849330)

[Abdelrahman Hagrass (195946): **Chain of Responsibility** 5](#_Toc102849331)

[Seifeldin Khalil (195423)**: Strategy Pattern** 6](#_Toc102849332)

[Maiada (203398): **Singleton Pattern** 7](#_Toc102849333)

[Maiada (203398): **Observer Subscriber Pattern** 7](#_Toc102849334)

[Mahmoud Assem (196735): **Observer-Subscriber Pattern** 8](#_Toc102849335)

[Mostafa Gado (192848): **Read-Only Pattern** 10](#_Toc102849336)

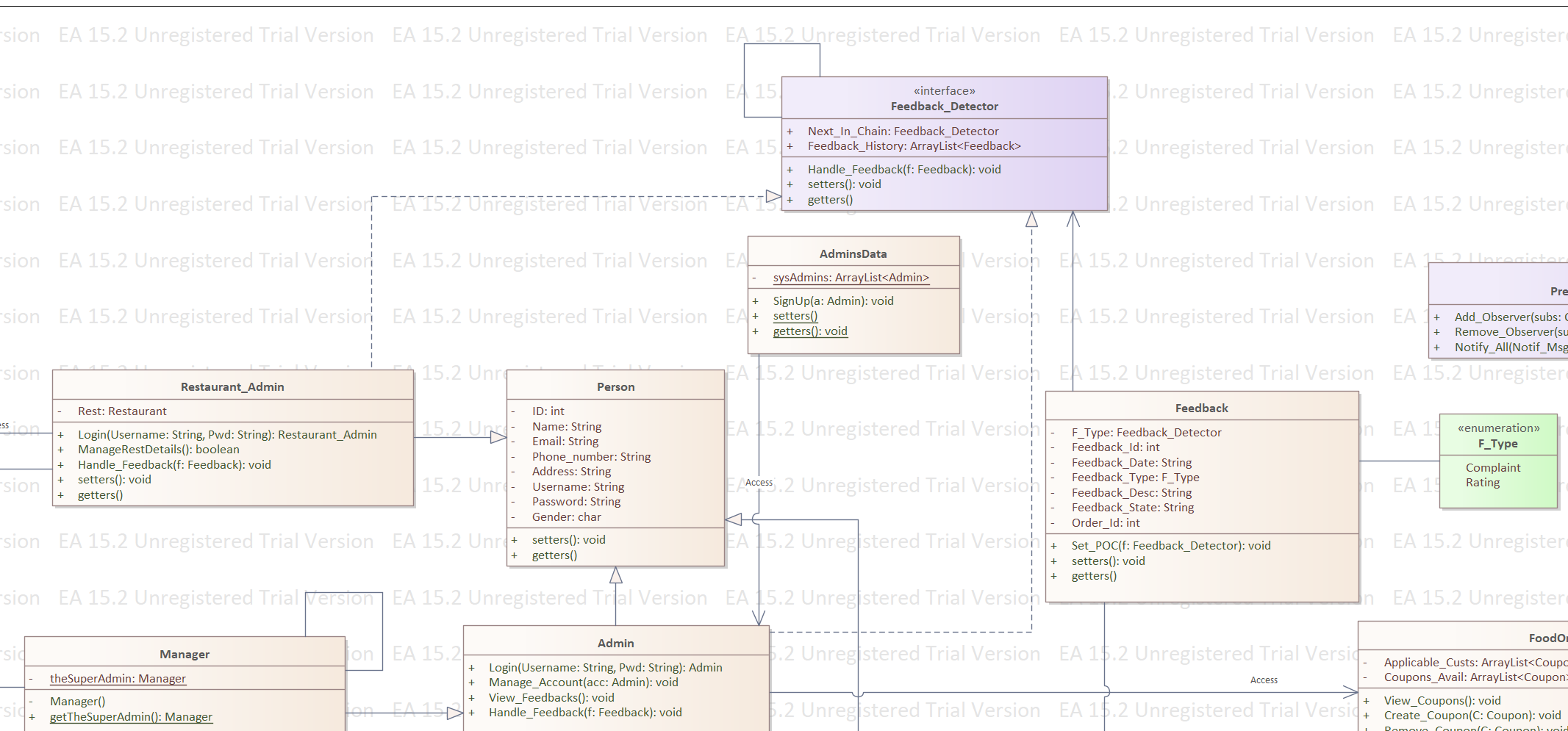
# Updated Class Diagram **<<Please check the “Phase 2 – diagrams” folder if images are not clear here>>**

# Contribution

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Patterns** | **Assigned Classes** | **GUIs** |
| Abdelrahman Hagrass | Chain of Responsibility Pattern | * DB\_Connection\_Hagrass * Person * Admin * Restaurant Admin   >> handle\_feedback()   * <Feedback\_Detector> * Feedback * **Customer** >> Create\_Order()   >> ~~View\_Order\_Details()~~  >> Create\_Feedback() | * AdminManageAccount * Rest\_AdminManageAccount * CreateFeedback * RatingFeedback * Feedback\_History |
| Seifeldin Khalil | Strategy Pattern | * Order * <Payment\_Method> * Cash * Credit * Order\_Item * **Customer** >> Cancel\_Order()   >> ~~Reorder()~~ |  |
| Maiada | Observer Subscriber Pattern | * Manager * FoodOrderingSysRests * RestsAdminData * Restaurant\_Admin * <New\_Rests> * <New\_Rests\_Observer> * **Customer** >> View\_Notifications() | * DB\_Connection\_Maiada * Manager\_AddRest\_GUI * Manager\_RemoveRest\_GUI * Manager\_SalesReport\_GUI * RestAdmin\_ManageRest\_GUI * Cust\_Notifications\_GUI |
| Mahmoud Assem | Observer Subscriber Pattern | * <Premium\_Coupon> * <Coupon\_Observer> * FoodOrderingSysCoupons * Coupon * Premium\_Customer * **Customer, Admin, Restaurant\_Admin, Manager** >> Login() |  |
| Mostafa Gado | Read-only Pattern | * <SysRests\_Interface> * Restaurant * Menu * Menu\_Item * **Customer** >> Sign\_Up()   >> Manage\_Account() | • viewMenuItem  • removeMenuItem  • customer\_SignUp  • allRestaurants  • addMenuItem  • Customer\_ManageAccount\_GUI |

# Design Patterns Used

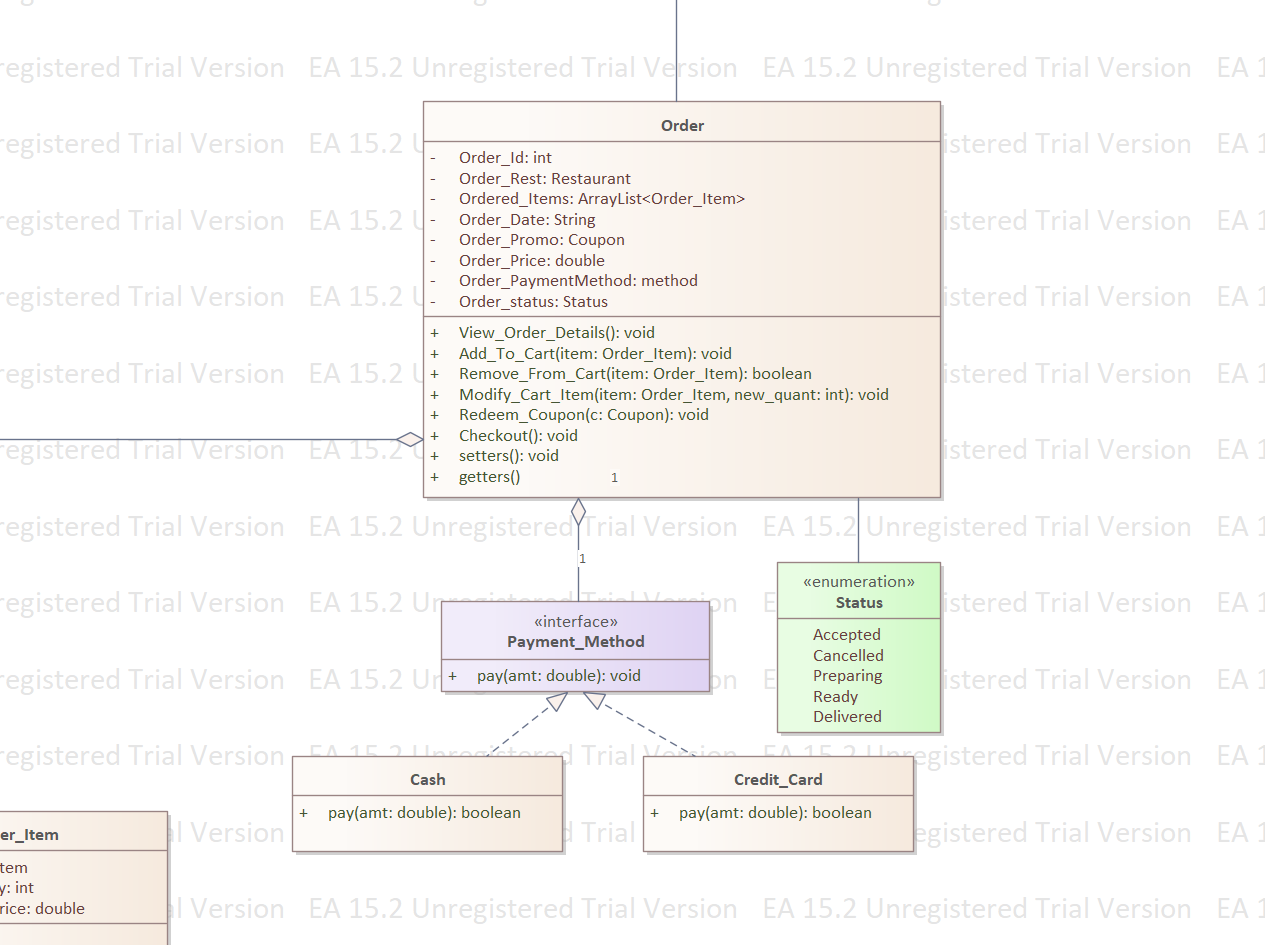
## Abdelrahman Hagrass (195946): **Chain of Responsibility**



Classes are: feedback - feedback\_detector - admin - restaurant admin

The chain of responsibility pattern is included in our system / class diagram in the feedback concept. The customer is capable of sending feedback to the system about their experience with a certain encounter or giving their opinion in a general way. When a customer is willing to send feedback, an object of **feedback** class is created after the function is executed, **Create\_FeedBack(int Order\_ID).** While the function is being executed, an object aggregated of **Feedback\_Detector**in the Feedback class will be assigned in the **Feedback ()** Constructor that include the calling of **Set\_Point\_Of\_Contact()** Function, this step is considered one of the main aspects of ***Chain Of Responsibility*** pattern. The purpose of calling such function is to assign the feedback object to the first element in the chain of responsible actors for handling feedbacks. Later on, the feedback object is passed to another function. **Handle\_Feedback(Feedback feedback)**, which is responsible for detecting if the feedback is a complaint or an opinion about the system in a general way. If it is, the first point of contact, Admin, will handle such types of feedbacks. If it is not, the **Admin** will pass it to the next responsible which is the **Restaurant Admin**. Restaurant Admins are responsible for types of feedbacks that are concerned with the orders or food in a general way. After the feedback is handled one way or another, it is stored in an arraylist in the interface **Feedback Detector.**

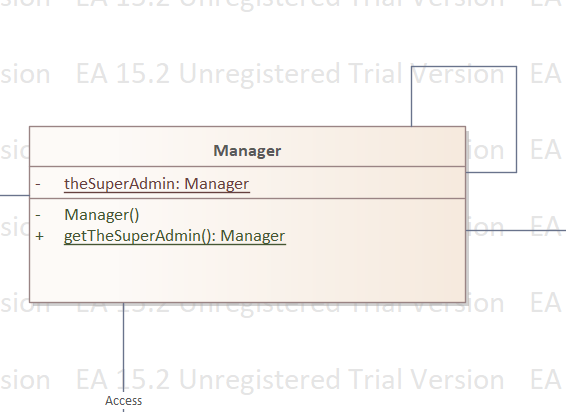
## Seifeldin Khalil (195423)**: Strategy Pattern**



The strategy pattern in our system is used to differentiate between the payment methods that the customer can use to pay for his order. The customer can either select to pay by credit card, or he could choose to pay by cash when receiving his order at the specified address.

When the Checkout() function is called from the order class, the customer is presented with a GUI which displays the available payment methods (cash or credit card). Based on the user’s choice in the GUI, the Payment\_Method interface then calls the class with the methods corresponding to the user’s choice.

## Maiada (203398): **Singleton Pattern**



The Singleton Pattern is applied in our system to maintain having only one Manager. This manager is the only one responsible for adding restaurants to the system upon having a contract with them and they are the only one having access to the sales report, for business needs.

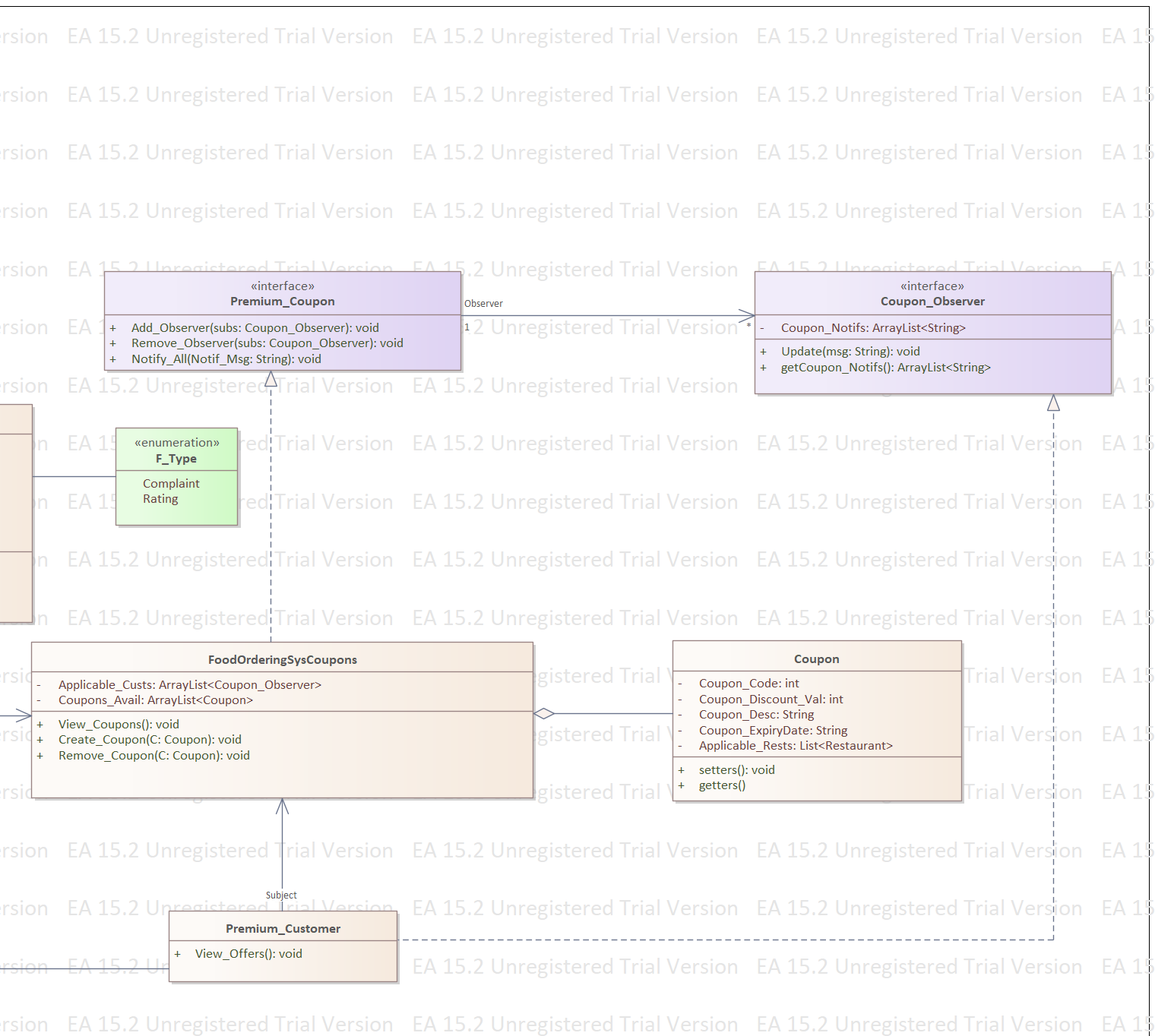
The pattern is applied by having a private constructor that sets the private Singleton attribute’s details and is returned/accessed via the public getSingelton method.

## Maiada (203398): **Observer Subscriber Pattern**

All registered customers will receive notifications upon adding a new restaurant to our application.

**FoodOrderingSysRests** >> Subject, **Customer** >> Subscriber

## Mahmoud Assem (196735): **Observer-Subscriber Pattern**



The observer pattern is known as publisher-subscriber pattern, which states that users can subscribe to a certain service to be notified of.

The observer pattern is described in our project somehow as a reward gifted to the users by the system. When a customer exceeds a certain number of orders done through the system, they are automatically changed into **Premium Customer***.* The steps of the following action are as follows:

1. When the customer is making an order through the system, the Create\_Order function contains a chunk of code that checks if the ordering customer exceeded the certain number of orders.
2. If they did exceed the certain number of orders, a new instance is created from class **Premium\_Customer**.
3. The object of **Premium\_Customer** contains the same exact data that was included in the **Customer** object.

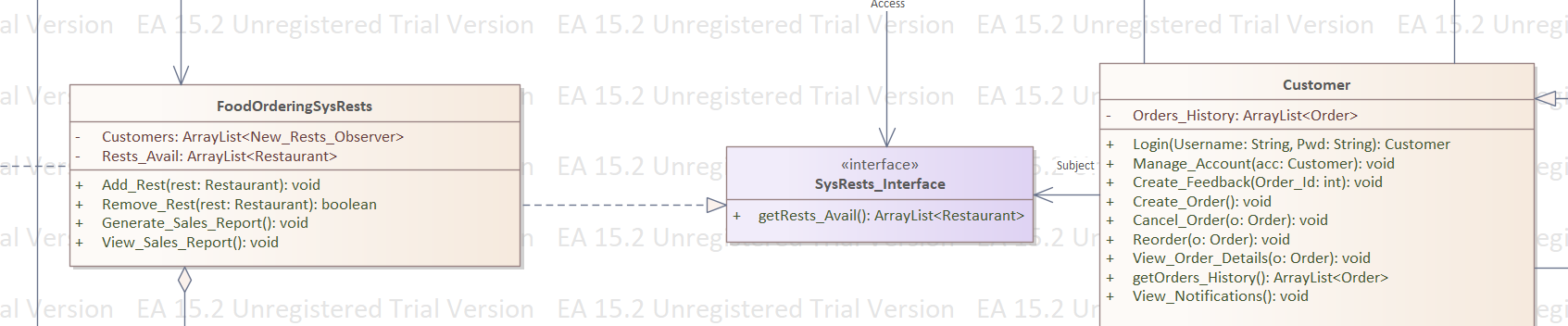
The observer pattern works only on the Premium Customers, since they are valued to the system. Premium Customers act as the ***Subscribers*** in the observer pattern. While the Coupons act as the ***Subject*** that the Customers are to be notified with. The coupons in this case are only available to the Premium Customers only.

**Admins** create coupons. Whenever an Admin creates a coupon, the Premium Customers are notified with it. The steps of the following step are as follows:

1. Admin calls the Create\_Coupon function.
2. Starts to set the details using setters
3. The Create\_Coupon function contains an arraylist of the “applicable restaurants” which is filled automatically with all restaurants in system, because all of them can accept any coupon.
4. The admin then calls the notify all function that loops over the “Applicable Customers” arraylist<observer> that delegates to the Observer Interface by calling the “update(string msg)” on each observer and sending the notification message as a parameter.
5. The coupon notification message is then added to the array list<String> of “Coupons\_available”

What was stated previously is the subscription technique. When it comes to the unsubscribing technique, Premium Customers are automatically assigned back as a normal Customer and removed from the Premium Customer category when they don’t use or make orders through the system for a certain period of time.

## Mostafa Gado (192848): **Read-Only Pattern**



The read-only interface in our system is between the customer and the manager.

Both can access the restaurant but in different ways.

The customer can only read the different data while the manager can edit the restaurant data.

The implementation of the patterns is through main classes the Unprivileged class, which is the customer class, the Read-Only-Interface, which is the restaurant interface, and the mutable class is the Manager class. The manager class can see the functions that are implemented in the restaurant class and can access the getters to view the data and can access the setters to edit the data. The relationship between the manager and the restaurant is association. The read-only interface is the restaurant interface. Which inherits only the getters to read the data in the restaurant class and not to edit them. The customer is the unprivileged class that can only read the restaurant data through the restaurant interface.

The relation between the admin and the restaurant interface is association.

# 