1 ILogger interface

I implemented an ILogger interface that contains a Log(string) function. Than, I created the ConsoleLogger class that inherits the ILogger interface. This way, we apply Liskov's Substitution Principle, because we can always use another class that inherits ILogger, to print all the messages in another environment. The Interface Segregation Principle is also applied, because we have a small interface with a function that we use.

I created the _logger variable inside the HotelReception class. The _logger variable is of type ILogger, this way we apply the Dependency Inversion Principle. This variable is private for a better encapsulation. Than I added the HotelReception constructor and I'm using it to assign a reference to _logger.

I put the ILogger interface and the Logger class into a separate folder for a better structure of the project.

2 File reading

I created an interface IFileReader with a function Read(). Than I made a FileReader class that inherits the interface. The Read() function from the FileReader class returns all the content from the json file. This class has, also, a constructor for assigning a reference to a _logger variable.

I decided to do that, because the Single Responsibility Principle was violated, since the reading of the text file should be in its own class. Than I created the _fileReader variable of type IFileReader, inside the HotelReception class. This way, we apply the Dependency Inversion Principle, because the dependency is of interface type, and, also we apply Liskov's Substitution Principle, because we can always use another class that inherits IFileReader. This variable is private for a better encapsulation.

In the constructor of the HotelReception class I assigned a reference to this variable.

In the ProcessOrder() function I used _fileReader.Read() to get the content of the json file.

I put the IFileReader interface and the FileReader class into a separate folder for a better structure of the project.

3 Order class to OrderData

I changed the Order class to an OrderData class and I'll read the data from the json file into an object of type OrderData.

4 OrderSerializer

I created an interface IOrderSerializer with a function Deserialize(). Than I made an OrderSerializer class that inherits the interface. The Deserialize() function from the OrderSerializer class returns the OrderData from the json string. This class has, also, a

constructor for assigning a reference to a _logger variable. I decided to do that, because the Single Responsibility Principle was violated, since the deserialization of the orderJson should be in its own class.

Than I created the _orderSerializer variable of type IOrderSerializer, inside the HotelReception class. This way, we apply the Dependency Inversion Principle, because the dependency is of interface type, and, also we apply Liskov's Substitution Principle, because we can always use another class that inherits IOrderSerializer. That variable is private for a better encapsulation. In the constructor of the HotelReception class I assigned a reference to that variable.

In the ProcessOrder() function I used _orderSerializer.Deserialize() to deserialize the OrderData.

I put the IOrderSerializer interface and the OrderSerializer class into a separate folder for a better structure of the project.

5 Orders

I made an Order abstract class as a base class for all the orders. This way, the project will follow the abstraction principle. I created an ILogger variable inside that class. That variable is protected, because it will be used in inherited classes. I also defined an abstract Process(OrderData) function. In the inherited classes, that function will calculate the Final Price and will return it. If the order was not processed, the function will return 0.

In Order class I defined an abstract function GetStringType() witch should return for every inherited class its type, as a string. This function is helpfull, because we observe that, in all the 3 order types, we have some validation for the data of the order. So, I created a function for each validation, because the logs were almost the same for all order types. This way, the code in the Process(OrderData) function is smaller, more visible and it's not repeated in all the inherited classes of the Order class.

Another good advantage of this method is that we avoid mistakes, like writing "Room" instead of "Product" and log the wrong message to the screen. Also, if in the future we want to add new orders, we can use those functions for them too and if those orders have some new properties, we can add new functions to validate them, as well. So, this way we apply the Open/Closed Principle.

I created the RoomOrder class for the room type of an order. In the Process(OrderData) function I stored the orderData.Quantity * orderData.Price into a variable, because its calculation appears multiple times and the code is repetitive.

For the ProductOrder and the BreakfastOrder classes, I made the same chances as in the RoomOrder.

This way, we apply the Single Responsibility Principle, because the HotelReception class shouldn't deal with the processing of each individual order.

6 Unknown Order

We need an Unknown order type for the default case of the switch. So, I added Unknown to the enum, and I created the UnknownOrder class, that inherits the Order class. The Process function of this class only returns 0.

I changed the Deserialize() function like this:

```
public OrderData DeserializeOrder(string orderJson)
{
    _logger.Log("Deserializing Order object from json data...");
    OrderData orderData = JsonConvert.DeserializeObject<OrderData>(
        orderJson, new StringEnumConverter());

    if (orderData == null)
    {
        orderData = new OrderData();
        orderData.Type = OrderType.Unknown;
    }

    return orderData;
}
```

The switch from the ProcessOrder() function was moved to a new class, OrderFactory, which inherits an IOrderFactory interface. This interface contains a Create(OrderData orderData) function that return an object of type Order.