OOP principles used to evaluate the code:

1. Encapsulate what varies
2. Favour composition over Inheritance
3. Program to an interface not implementation
4. Strive for loose coupling between objects that interact
5. Classes should be open for extension and closed for modification

1)Encapsulate what varies: Encapsulating what varies is an approach for dealing with details that change regularly. When code is constantly updated to accommodate new features or requirements, it tends to become tangled. We reduce the surface area that will be affected by a change in requirements by separating the sections that are prone to change.

In our project, we have implemented this principle by separately creating classes for Menu, Order, Customer, Admin, Person, OrderDetail and setting their attributes as private. So any changes made in any of these classes will not impact the working of the rest of the project. We have created the getter() and setter() methods to access or modify data of these classes if required.

2) Favour composition over Inheritance: In object-oriented programming, composition over inheritance refers to the idea that classes should achieve polymorphic behaviour and code reuse by holding instances of other classes that implement the needed functionality rather than inheriting from a base or parent class.

In our project, we have implemented this feature quite well as instead of inheriting the classes mentioned above we have created instances of these classes in the Restaurant class and controlled the flow of the code in the Restaurant class. We have just extended Admin and Customer class from Person class as basic features like user id, first name, last name, password are the same in both for customers and admin. So to implement run-time polymorphism for the person entering the restaurant, these two classes are only extended.

3) Program to an interface, not implementation: Always programme for the interface rather than the application; this will result in adaptable code that will function with any new interface implementation. In Java, employ interface types on variables, method return types, and method argument types. Instead of concrete implementations, applications should always use interfaces from other parts of the programme. This technique has various advantages, including maintainability, extensibility, and testability. This can be used to alter a program's behaviour while it is running. It also aids you in writing considerably superior programmes in terms of maintenance.

In our project, we have created different classes for each feature in the Restaurant such as Creating Order, Menu list, Table, customer and Admin class. So if there is a problem with any one of the features we can test that class separately and even add a new feature to the existing class without changing any class.

But there is one limitation, we are implementing all these classes using the Restaurant class. So if we change the number or type of parameters required in any of the classes mentioned above then changes in the Restaurant class also be made.

4) Strive for loose coupling between objects that interact: Loose coupling is an important design guideline in software programmes that should be followed. Loose coupling's major goal is to achieve loosely connected architectures amongst items that interact with one another. The degree of knowledge that one thing has about the other item with which it interacts is referred to as coupling. Because they lessen the dependency between numerous objects, loosely-coupled architectures enable us to create flexible object-oriented systems that can handle changes.

In our project, we have not implemented this principle with precision. There is a moderate coupling between Admin, Person, Customer, Menu, Order and Restaurant, between Restaurant and Data and between Restaurant and GUI class. But there is loose coupling between order and order detail class although one class input depends upon others as data is directly passed from Restaurant class to respective classes.

5) Classes should be open for extension and closed for modification: The general idea of this principle is that it tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes.

This principle is also not followed quite well as in order to add new features to the Restaurant management system we need to modify the Restaurant class. Although Data class need not be modified as data is being stored in files. So new file can be created for adding a new feature and opened in the Data class without modifying it.

Design Pattern used to analyse the code

Strategy Pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

In our program, we have encapsulated different attributes within a class thus the clients namely administrators and customers can use different features of the program independent from each other. For implementing this pattern first we have separated things that are common (like menu items, reserving tables, ordering facilities) from things that are different like management of restaurant state, editing menu, visibility of revenue available only for administrator.