Detailed analysis of code documentation based on oop principles and design patterns

1.)Encapsulate What Varies : Its is a technique that helps us to handle frequently changing details.

Done mainly by making fields private/protected and and using getter and setter methods to access and modify the values.

This principle is a bit followed in user class where the fields were made protected(so that visible only to user class and its subclasses). The Account Balance of each user is changing frequently with each booking so it ensured that balance gets updated safely and also safe access using getter methods.

However this principle was not followed in property class(specifically for the calender field in property class). As the calender dates were changing frequently this could have been done to ensure that bookings do not get manipulated. (Honestly, It was becoming very difficult to access "available_dates" field since it has a complex return type).However for Updating dates specific methods(book and cancel) are created and direct updation(using field name) is avoided.

2.)Favor Composition over Inheritance : Composition gives us a great flexibility in terms of modification of code over inheritance. The problem with parent-child relationships is that just a minute change in parent class can lead to problem if this change is not to be done in child class.

Composition is done by including the object(as a field) of the earlier parent class in the child class. This is neatly done in my program. By including calendar field in Property class I can access all methods of calendar in Property class. Similarly object of User is included in Control class. Inheritance relationship of User and (Customer ,Manager) is however maintained as Always Customer and Manager "is a" User.

3.)Program to an interface not implementation : Having SuperClass SubClass relationship and combining it with dynamic method dyspatch helps in easy modification of code. For example having initialised object of User with manager or customer, in future If I wish to have different implementation of a method of user class in Customer class I can override it in Customer. And having reference type of Customer will directly allow overriden methods to be called.

4.)Depend for abstraction, do not depend on concrete classes : Abstract classes allow having different functionings of same methods of child classes, as per required at moment. It is very useful when we are having several sub classes having different implementations of some methods and rest methods are same. In my application however there aren't many classes so this principle was not used.

5.)Strive for loose coupling between objects that interact : loose coupling is when two objects interact with each other but still making for one object wont affect other much. It refers to the degree one class knows about other class..

When one class is calling logic of another class then there exists a tight coupling between the two classes.

Design Patterns Which Could be used/are used(a bit 😐)

- Strategy Pattern :

Wikipedia defines strategy pattern as:

“In computer programming, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern

- defines a family of algorithms,
- encapsulates each algorithm, and
- makes the algorithms interchangeable within that family.”

In my program at most stages the flow of the program gets decided by the input of the user, However proper encapsulation is not done since most methods are public and accessible easily.

We need to follow the principle of “Encapsulate what varies” while using this design pattern. For example the balance field is encapsulated by making it protected and using getter and setter methods to retrieve its values.

For eg: During SignUp on prompt of customer/manager If user enters 0, the current user is reinitialized by child class reference types – algorithm’s behavior is selected at run time only.

For eg A User on entering 0 will lead to Search for properties on entering 1 will lead to cancel booking and so on…

So the flow of the program is governed largely by user input. Hence there is a large scope of using Strategic design pattern in this program. For eg: All fields of property class should be made private and getter and setter methods should be used to change fields, in this way encapsulation could be ensured and then methods based on runtime inputs should be called.

- Factory Design Pattern:

In factory design pattern or factory method pattern, we just define an interface/abstract class for creating an object but let the subclasses decide which class to instantiate.

This design pattern is used in the User and Customer/Manager inheritance relationship.

The constructor of Customer/Manager can be called based on the user input (at run time). It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.