Well-designed program is reusable, extensible and maintainable. That’s why we have designed our program to achieve both low coupling and high cohesion. To make our classes reusable they are designed to be highly cohesive and hence execute only one purpose. For example, class user is only responsible of storing the information relevant to a user and returning it when called.

By coupling one means the dependencies between classes. High coupling results in a tight web amongst different modules whereas in low coupling only necessary links are made between classes. One example of low coupling in MOBLIMA can be found from the seat booking system. There’s no connection between seat and Cinema. Instead seat is connected to a class called movie display which is linked to the booking class with cinema. For every display of a movie the booking of seats is unique. The constructor of the movie display takes as a parameter the maximum number of seats and then creates a layout of the cinema by creating an array of seats.

MOBLIMA also follows the important object-oriented concept of encapsulation. Relevant information stored in user or seat classes can only be accessed through the public methods of those classes. Thus, the main menu calls the different classes without knowing the underlying implementation.

SOLID is a set of five different design principles. They are single responsibility, open-closed, Liskov substitution, interface segregation and dependency injection principle. Single responsibility principle tries to achieve the same as cohesion as every class should only have one responsibility. Thus, it’s one of the corner principles in the MOBLIMA design.

In open-closed principle one should be able to extend class but not to modify it. That’s how our user class is designed to work. There can be different users, for example end customers and admins which inherit the user class and add their own methods and variables.

Liskov Substitution principle states that a subclass can expect anything more from a client than what is required by the base class and it has to offer at least the same. Is there anything following this principle?

According to interface segregation principle it is better to have specific interfaces than one general that all the classes are using because the it includes method that not all client classes are using. Not sure about this either?

Dependency injection principle is the last one of SOLID principle. It states that high level modules should not rely on the low level modules. Instead they should be connected by an abstraction layer. Same thing.