Layered architecture

# SOLID principles

## Single responsibility

In case of a layered architecture we might be almost sure that the Single responsibility rule applies. Since in every layer we have different type of functionalities, we can avoid the creation huge classes with a lot of functionalities encapsulated (e.g. we can’t have basic functionalities, complex processing and UI parts in a class because are parts of different layers).

## Open-closed principle

In case of this principle it helps a lot if our design is open from extension; by using interfaces, it makes our design open for extensions and it also helps for the above layers in order not to be changed a lot when the layer below is extended.

## Liskov substitution

Not much relation

## Interface segregation

By having layers, we avoid the case of huge interfaces, because he functionalities are divided between the layers, which decreases the size of them (vertically). In case we are having other components inside a layer, we divide the interfaces vertically too.

## Dependency inversion

We cannot have circularities in case of layered architecture, dependency inversion allows us that layers above do not depend on the layers below.

# GRASP principles

## Creator

In case of the Creator principle we decide who creates and which object. In case of Layered architecture always those classes create and instance of a class that are one layer above or in the same layer (ex: Validators are used by Business Logic classes, but they are in the same layer).

## Expert

Based on the expert principle we need to decide which responsibilities are handled by one particular class. A responsibility should be handled if all the information necessary for it is available. In case of layered architecture, we can come up with the case of executing processing operations – this should be done by the business layer, because it has also the input data from the layer above, but the database data as well, since it has access to the DAO classes. For a complex operation it might be needed to use multiple data tables. Because of this it is necessary to perform it in the business logic level (same in case of view – controller -business logic).

## Low Coupling

Low Coupling helps maintainability. In case of layered architecture it would be optimal if one DAO class would be dependent on only one Entity class, one BL class would be dependent on one or as less DAO classes as possible.

Also, Layers help that only classes of one layer above are dependent on a lower layer. We never access one class from two different layers above.

## Controller

The controller decides which one is the first object that receives the data from UI and further processes it. In case of layered architecture, it is always the business logic level and its classes. The business logic level further delegates the data to the data access layer. In this case we can talk about façade (all the business logic classes are attributes of a class that connects to the controller) or session controller (one object represents a use case). To avoid overloading the controller, add as many as logically needed.

## High Cohesion

High cohesion states that related operations should be in one manageable unit. It defines what the purpose of one class it – layered architecture helps with this one as well, because a layer itself defines what a class contained should perform – ex: we never do input handling, logical operations and data insertion in the database in one class, these 3 are clearly separated between layers. The individual components of a layer also do coherent operations – ex: operations on a particular tale are done in its DAO.

## Polymorphism

Operations with the same purpose may be defined differently in different classes, in this case we introduce subclasses – ex DAO objects have the same purpose, but they are executing differently, also may add up some behavior to the existing one. We later call that DAO that is needed.

## Pure fabrication

Fabricated class means to create a class for those operations that doesn’t belong to any domain class – in layered architecture this is what DAOs are for.

## Indirection

If we introduce interfaces and polymorphism in case of layered architecture, or even façade, it helps to ease up the connections between layers, thus leads to low coupling.

## Protected Variation

To avoid huge changes in an application every time something new is performed, we introduce interfaces. This way we can add to the behavior of one layer but we are also avoiding influencing the higher classes to much.