# Introduction To Clean Architecture

# Hello, In this post I’m going to talk about Clean Architecture.

# There are two main reasons that I create a post about Clean Architecture : I have started to learn and practice Microservices & .NET Core . When I mentioned my senior about my goal, he recommended me that there is a better way to start to help me understand why and how we use microservies : working on clean architecture, CQRS and async programming first. So I took the advice and act upon it. Also, one week ago I gave presentation about Clean Architecture for DevNot and it was my first technical presentation, quite exciting as you may guess ! I am leaving the link here for the ones want to check it up. ☺ After all the work, I decided to create a documentation that I hope you find useful.

Clean Architecture is an architecture that Uncle Bob, whom we all know and love, published on his own blog, in other words, it is actually a software design philosophy. It is a young, dynamic and modern architecture that has become more popular especially with the introduction of microservices into our lives. Clean architecture aims to divide a big project into pieces (classes, entities, modules etc.), make them independent from each other : our big and complex project become more maintainable, flexible and easy to work on together.

We divide a big&complex project into smaller pieces like classes or modules etc. Then, we divide these smaller pieces into more smaller components. Therefore, our pieces and even the whole project become more easy to maintain, more flexible, easily extendable and more readable. We can be able to make changes even without effecting any other module / class in our project and modify different technologies, tools. We create a loosely coupled structure which we aim in almost every design principle.

Uncle Bob emphasizes 4 main points about Clean Architecture :

* Independent of Frameworks. The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
* Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.
* Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
* Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
* Independent of any external agency. In fact your business rules simply don’t know anything at all about the outside world.

Clean Architecture says : It’s all about Dependency Rule !The inner circle is not allowed to know anything about the outer circle. Even, Uncle Bob says more :

In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes, variables, or any other named software entity.  
As you can see there are 4 main layers : Entities, Use Cases, Adapters and Infrastructure. Now, let’s see each level with details.

Entities are a set of business rules that are critical to the application. This layer knows nothing about the other layers, which means entities don’t depend on anywhere or anything. They may be exist even there is no application.

Use cases are the key point to automatize the system. Different from entities, use cases contain specific business rules. This layer knows about nothing about the other layers but the entity. Use cases layer has interfaces and or abstract classes that the other (further) layers can use.

Adapter layer is literally an adapter between domain and infrastructure. Remember Adaptor Pattern. The logic is literally the same here.

Infrastructure layer is the most social layer, actually ! ☺ This layer is so likely to change so it is as far as possible from the other stable domain layers. So, as you can realize, we keep the one that is more likely to change and dynamic separate from the other stable ones to prevent coupling and effecting the stable domain layers.

Until now, we talked about what clean architecture is, when to use it and take a general look into it. Now, I want to give more detail about the communication between layers.

Domain layer is the centermost. The domain layer knows nothing about the outer layers. Generally, this layer has no likely to change. To give an example, in the project where I experienced this architecture, in the domain layer. entities were involved. We determined the structures that are least likely to change as entities and placed them in the innermost layer. This is how it is in most practices.

Domain can be included in the Application Layer in some uses, but for example, if you are using an entity framework (ie if you are using ORM technologies), the Infrastructure layer should query the necessary information from the Application layer. Separation is a better solution and fit better with the philosophy of the architecture.

The Application layer is the layer which the models are mapped. Validations, CQRS operations are the main important operations for this layer. Again, as you can see in the figure, Domain and Application communicate with Domain Models, we can directly think about DTOs here. Data Transfer Objects are data containers between layers and they do not contain business logic, allow only the required data to be passed. The point we need to pay attention here is the direction of communication. Whatever the databases and web services in outer layers, we are not aware of any of them as long as we are in the Application layer. We don’t go out to ask for anything.

Persistence is the layer that contains mirror images of db tables and queries. Object mappers can be considered directly when the persistence layer is mentioned. Of course there may be different hybrid solutions according to your needs.

Infrastructure layer is basically for encapsulating the logic (For example external APIs.)

Presentation layer collects all that in the Application level things with APIs. This layer helps the communication between UI side and Application layer by APIs.

Before ending this part, as additional information, I would like to point out that :

Persistence, Infrastructure and Presentation layers are called Periphery layers.

Domain and Application layers are called Core layers.

If you just search for Clean Architecture, you can see these definitions so I also wanted to mention.

# Comparison

The purpose of Layered Architectures is to divide the application into different levels that have their own modules, classes, share similar responsibilities, and can work together with performance. They differ in how they accomplish this goal. Now I am gonna talk about a few different architectures in detail and make a comparison.

Let’s start with Onion Architecture first. The reason why it is called Onion is that its rings actually looks like an onion, you can see the intertwined rings and it is the closest architecture to Clean Architecture (that’s why I chose it first for comparison). The point the architecture focuses on most is loose coupling. UI, Business logic and Data are the basic layers. There is a hierarchical structure. Each layer is linked to a lower level layer. The UI layer can only communicate with the BL layer, it is forbidden to talk to other layers. As you can see in the figure, the Data layer acts as a kind of center. Here, one-way communication shows us that the changes made will affect the outer layers, not the inner ones. If we consider the layers one by one, Domain already has the same features that we mentioned in other architectures: entities, value objects, whichever we need, we group them in this layer.

The interface layer is where it is abstracted between the Domain and the business layers and one the parts that loose coupling is provided.

Persistence layer is the layer where operations such as DbContext and migration are performed. Interfaces provided by the Application layer are also implemented in this layer. The Infrastructure layer is actually used intertwined with the Persistence layer.

The Presentation layer is the layer where the user communicates with the application. Again, similar to other architectures.

So what are the advantages and disadvantages of onion architecture? It separates the layers according to their responsibilities, which leads us to the principle of Separation of Concerns. It increases testability and maintainability. As I mentioned, it is very similar to clean architecture, and in some sources clean architecture is called an extension of onion architecture. In fact, when we look at the communication of the layers, the loosely coupled design and the results obtained, this approach can be considered correct, but in these 3 architectures that I am going to mention, the results and structures are actually similar, so the Clean Architecture approach is not a revolution, but as you notice, the components that the layers contain and their communication with each other are different and that has so many pros for developers.

The disadvantages are that it is difficult to navigate the project, I am not talking about the communication that interacts with the core, especially, there are too many interfaces (at this point, it is a problem that they can not solve together with the hexagonal architecture). Another disadvantage is that there is too much logic loaded in the core. That’s why I just emphasized the word interaction. Any change made in the core can affect the behavior outwardly, of course, it is the same in clean architecture, but the extra interfaces here increase this effect even more, which may make choosing this architecture disadvantageous according to the case.

I want to reiterate, they have the same basic point of view, only Onion is more model-centric, It is a design that has been created with the repositories on top of the model layer and the services on top of it, and the presentation at the end. Clean Architecture is a design in which Entities are put in the core and in this way, it is progressed outward again as the presentation layer at the outermost. So Clean Architecture is more use case oriented.

Moving on to N-layered Architecture, it basically consists of 3 layers: Presentation Layer, Business layer and Data Access layers, it is an architecture used to separate responsibilities and manage dependencies.The general principle is database operations are done by the Data Access Layer, the manipulation of that data is done by the BL and the results are sent to the Presentation layer. In the presentation layer, it is desirable to write code only for data to be displayed. That is, there should be no database-related operations in the presentation layer.

This architecture is actually an evolution of traditional application modeling. It is a very common architecture. Reduces clutter, provides scalability, centralizes business rules, which ensures reusability. It is easy to implement, the deployment cost is low.

What are the disadvantages, first of all, of course, it is a monolithic structure. It does not allow the use of new features independently. Increases complexity and effort, can cause performance issues. Specially when the number of layers increases, (I will talk about hexagonal) its intelligibility decreases, becomes difficult to maintain depending on the number of layers. While coding, it takes longer to start because multiple components have to be touched, and of course, as a developer, touching many modules for a small change would not make any of us grateful :)

The difference with Clean Architecture is what the code knows, what it references and at what point it connects to other code. In N-Layered architecture, business Logic needs to know the API. In Clean Architecture and onion architecture, Infrastructure layer knows the APIs of the domain layer. Apart from that, you can basically see the difference even in their schematic representations. There are no nested and one-way communication between layers in N-layered architecture.

In hexagonal architecture, the most basic point of distinction is the inner and outer layers. This architecture is more commonly called Ports and Adapters. In fact, ports and adapters can be thought as plug-ins. The reason is that the user or system inputs reach the applications on the ports through the adapters. We can say it tries to preserve the core of the application, because it contains adapters that abstract from tools and technologies as an intermediary. The reason it’s called Hexagonal is each side has its own specific rules. So the only important point is not the direction of communication or the placement of layers. Externally, there are many systems, not just one. In this architecture, the most crowded layer is Infrastructure, and the layer with the most problems is also Infrastructure. Therefore, Dependency Injection must be implemented in this layer. In fact, in some scenarios, the Infrastructure layer becomes so large and complex that it has to be divided into other layers within itself, which means a different design philosophy is needed.

As for the negative aspects, you first noticed that decoupling is used in hexagonal architecture, so intermediate classes are needed, which means that performance is affected. A single extra call means that the service goes end to end which means extra code, extra class, extra performance, but more importantly, the application becomes extra difficult to understand and debugging neither. Also, if the application is already large and complex, these performance problems and the possibility of further complexity of the system will cause serious problems.

At the end of this section, I would like to emphasize that I use the word disadvantage in this post because I have discussed these architectures within the scope of Clean Architecture and I want to reach conclusions by way of comparison. So we’re not grouping everything as black and white here. I only mention the points where they differ, such that the Onion vs Hexagonal comparison has left its place to the hexagonal and onion architecture, which is the combination of the points that they are the best. I suggest you to take a look. The point is, these architectures differ in their implementation, not their goals.

# Advantageous Scenarios

First of all, in my own experience, what I’ve read and researched, the most important thing to do before choosing clean architecture is a serious review of the size and complexity of the application. That directly reminded me of the CQRS pattern. Command Query Responsibility Segregation. It’s very common in Clean Architecture and, of course, in microservice architecture. As the name suggests, it’s based on the principle of separating Commands and Query’s. In that case, if the application only performs small and basic crud operations, the use of this pattern is mostly not recommended, because it actually makes things more complicated and prolongs the execution of a simple operation. In fact, we can say the same situation is partially valid for Clean Architecture. If your application is large enough and complex enough, yes, choosing this architecture provides many advantages. But otherwise, it is not recommended because there is a risk of increasing the workload and you may make the structure more complex than it is.

Another criteria is if different databases and different UI technologies are must or necessary in your application, I will give an example of the concept of necessity here, I am sure we have all encountered or heard of it from our friends: Most of the time, it is more costly to switch to new technologies the more current technology is older . Because of this reason, old technology / architecture continues to be used for years in products, we all witness them unfortunately. So, preventing such situations is also included in the condition of necessity. Clean Architecture is the perfect solution to keep up with the technology, to stay up to date, and not to face any serious cost problems.

If it is desired to keep the adaptation times of new developers as short as possible: which is a very common need. Especially in busy periods the need for developers to adapt to the application as soon as possible is a situation mostly preferred, but can cause stress in some cases. At this point, while the independence of the layers and technologies is a big factor, the increase in readability and the fact that the use cases are collected in a certain layer, of course, accelerate and facilitate the familiarization process.

Another important criteria is if the point of arrival is uncertain. While talking about the advantages of Clean Architecture, I said it is easily extendable, I said it’s flexible. The uncertainty of the destination means these features are needed. At the same time, at any point, for example, when using sql, if there can be a possibility of a scenario where it may be necessary to switch to mongo db or oracle, or in case the project has a constantly changing and developing structure designing according to Clean Architecture improves maintainability and it also provides a great advantage that it increases readability.

To sum up, the new technologies and increasing demands, business logics of today’s world, makes us looking for more efficient, adaptable software designs and we have many approaches to make our software more efficient, well-performed, flexible, maintainable and clean. Today I talked about one of them — Clean Architecture : definition, key points, layers and communication between layers, comparison to some other architectures and scenarios to apply for best results.

If you are interested, recommend you this post which I found quite inspiring and useful :