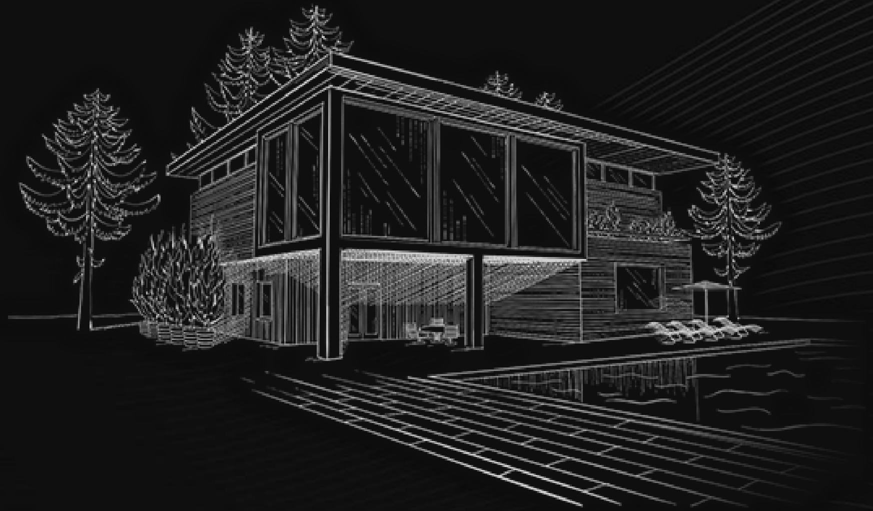


Clean Architecture

Alireza Kamarzardi

The goal of software architecture is to minimize the human resources required to build and maintain the required system.

— Robert C. Martin

Clean architecture gives us all these benefits:

Testable.

The business rules can be tested without the UI, database, web server, or any other external element.

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.

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.

Other clean architecture benefits:

always ready to deploy

It is always ready to deploy by leaving the wiring up of the object for last. Or by using feature flags, so we get all the benefits of continuous integration

Independent of any external agency

In fact, your business rules don’t know anything at all about the interfaces to the outside world.

Modularity.

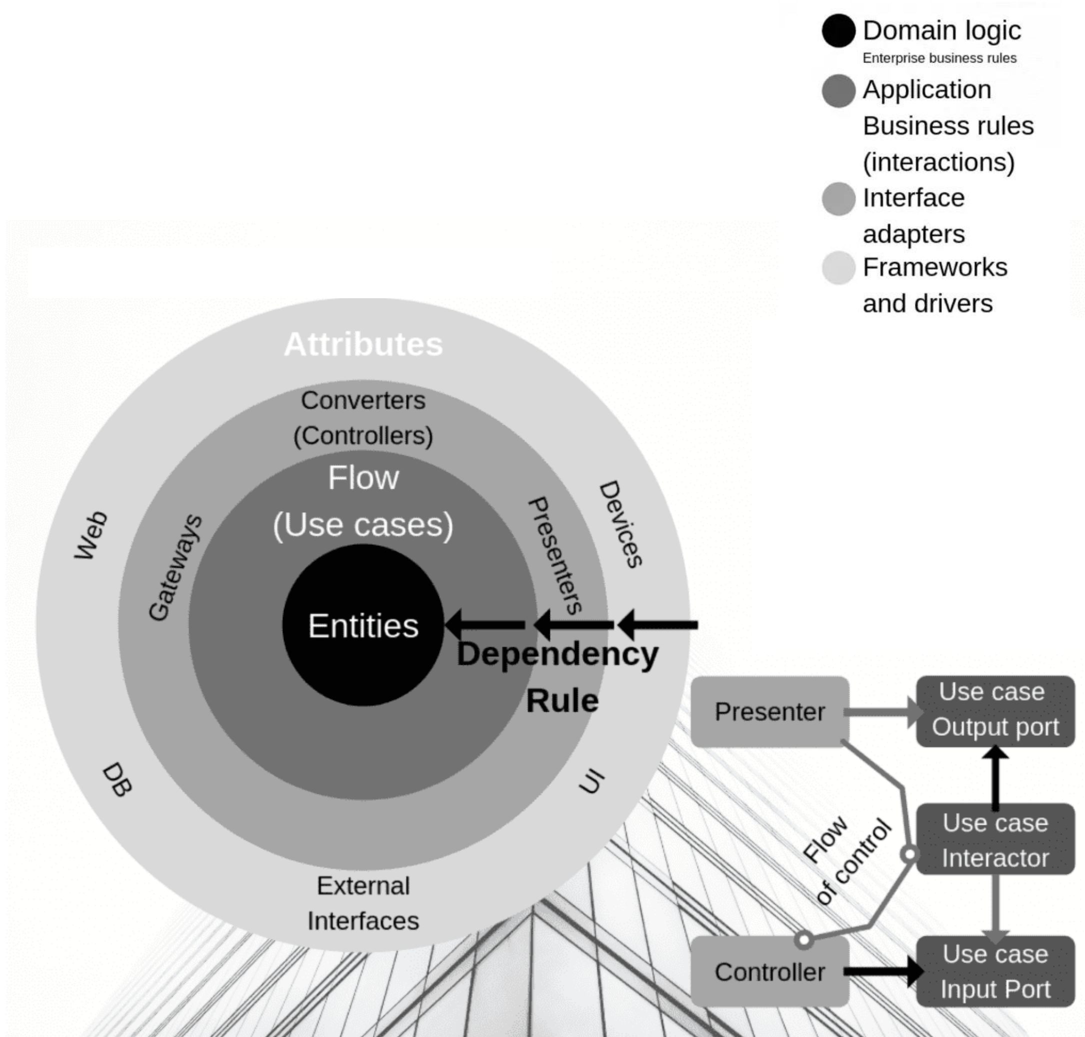
Clean Architecture helps to create a clear separation of concerns within the codebase. Each layer has a specific purpose and is decoupled from the others, making it easier to understand and modify and reuse components in other projects.

Maintainability

By separating the concerns of the various components and enforcing the dependency rule, it becomes much easier to understand and modify the code. Depending on abstractions allows you to design your business logic in a flexible way, without having to know the implementation details.

Clean architecture

The main rule for Clean Architecture is the Dependency Rule, which says that the dependencies of a source code can only point inwards, that is, in the direction of the highest level policies. That said, we can say that elements of an innermost layer cannot have any information about elements of an outermost layer. Classes, functions, variables, data format, or any entity declared in an outer layer must not be mentioned by the code of an inner layer.



The center of your application is not the database. Nor is it one or more of the frameworks you may be using. *The center of your application are the* ***use cases*** *of your application.*

—Uncle Bob

What is use-case?

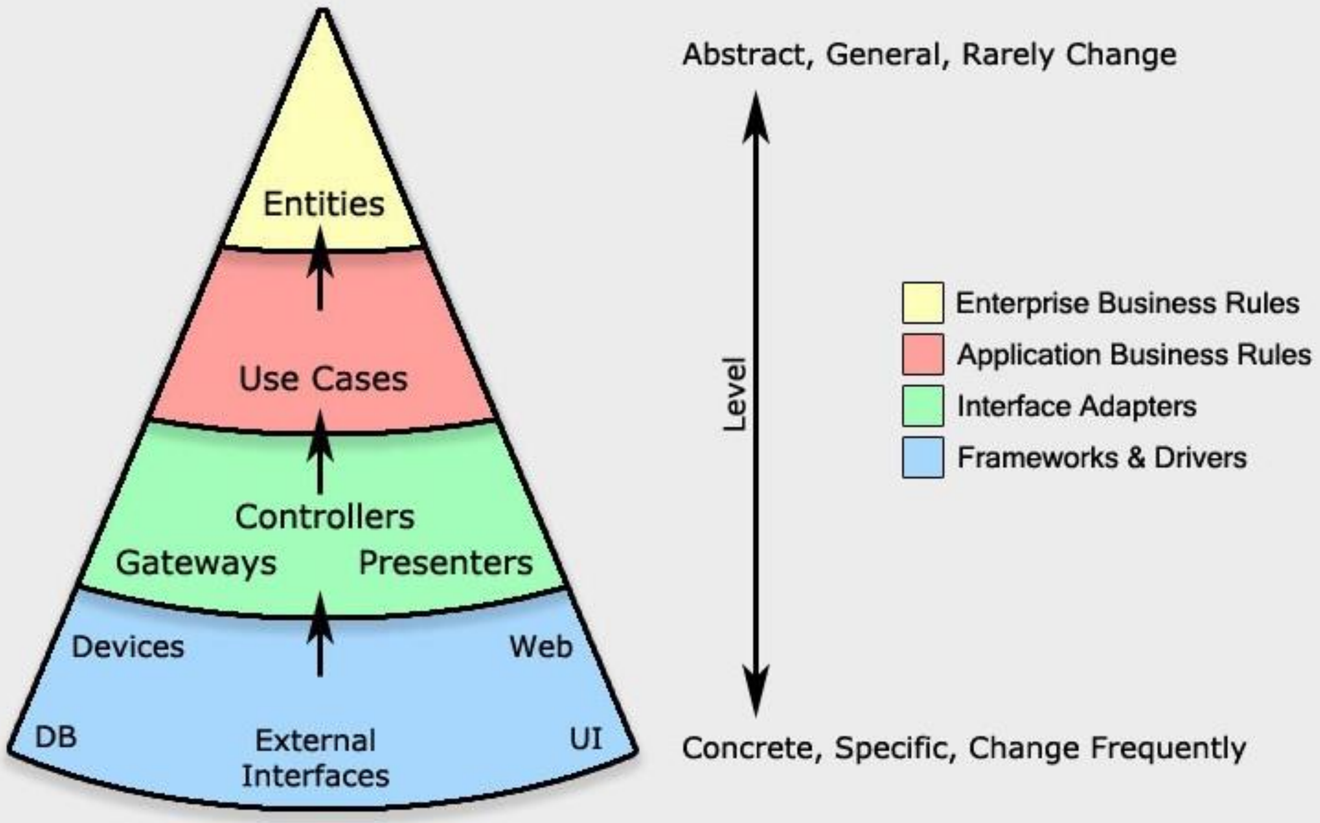
Wikipedia

A *use case* is a list of actions or event steps typically defining the interactions between a role (known in the Unified Modeling Language (UML) as an *actor*) and a system to achieve a goal.

Clean architecture book

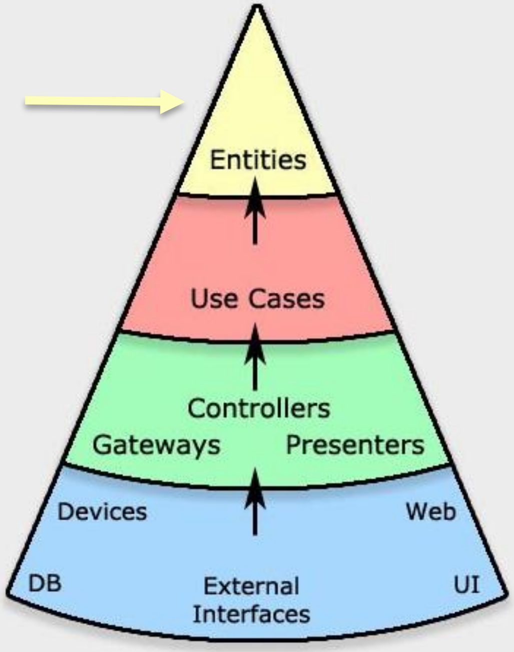
These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their *enterprise wide* business rules to achieve the goals of the use case.

The Clean Architecture Cone



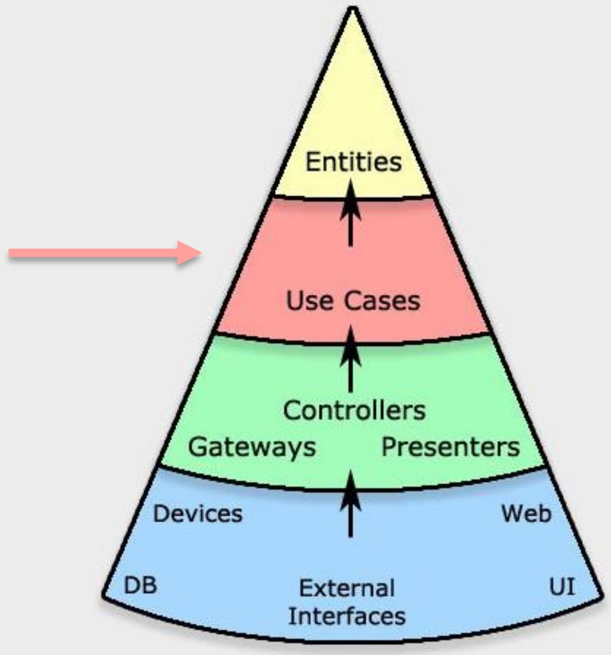
Entities

The innermost layer is the Entities layer, which contains the application's business objectives, containing the most general and highest level rules. An entity can be a set of data structures and functions or an object with methods, as long as that entity can be used by multiple applications. This layer must not be altered by changes in the outermost layers, that is, no operational changes in any application should influence this layer.



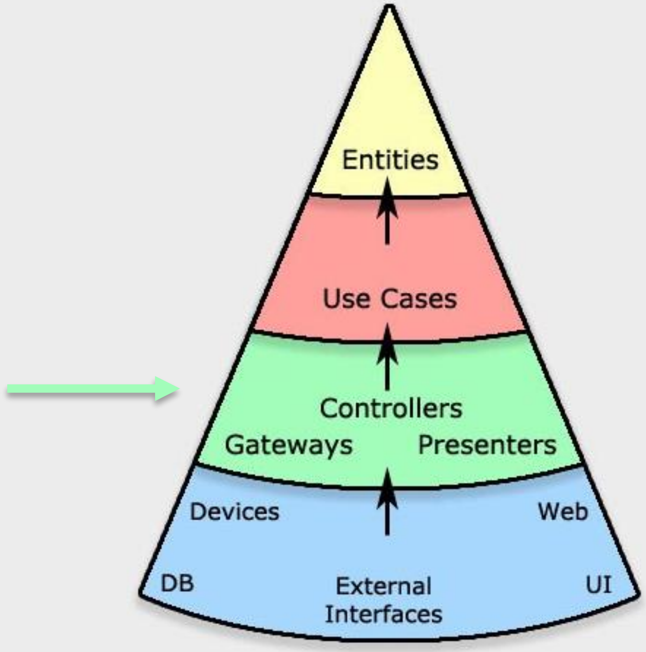
Use Cases

The use case layer contains the application-specific business rules, grouping and implementing all the use cases of the system. Use cases organize the flow of data to and from entities and guide entities in applying crucial business rules to achieve use case goals. This layer should also not be affected by the outermost layers, and your changes should not affect the entities layer. However, if the details of a use case change, some of the code in that layer will be affected.



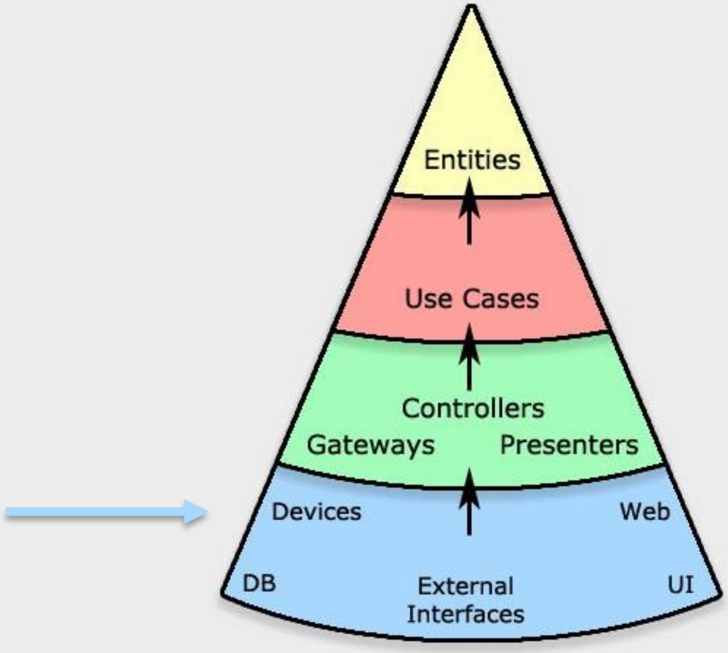
Interfaces / Adapters

The interface adapters layer has a set of adapters that convert the data to the most convenient format for the layers around it. In other words, it takes data from a database, for example, and converts it to the most convenient format for the entity layers and use cases. The reverse path of conversion can also be done, from the data from the innermost layers to the outermost layers. Presenters, views, and controllers belong to this layer.



Frameworks and Drivers

he outermost layer of the diagram is usually made up of frameworks and databases. This layer contains the code that establishes communication with the interface adapters layer. All the details are in this layer, the web is a detail, the database is a detail. All these elements are located in the outermost layer to avoid the risk of interfering with the others (MARTIN, 2017).



But if the layers are so independent, how do they communicate?

This contradiction is resolved with the **Dependency Inversion Principle**. Robert C. MARTIN (2017) explains that you can organize the interfaces and inheritance relationships so that the source code dependencies are opposite the control flow at the right points.

If a use case needs to communicate with the presenter, this call cannot be straightforward as it would violate the Dependency Rule, so the use case calls an interface from the inner layer and the presenter in the outer circle does the implementation. This technique can be used between all layers of the architecture. The data that travels between the layers can be basic structures or simple data transfer objects, these data being just arguments for function calls. Entities or records from the databases must not be transmitted in order not to violate the Dependency Rule.

Let's see an **example** from the clean architecture book

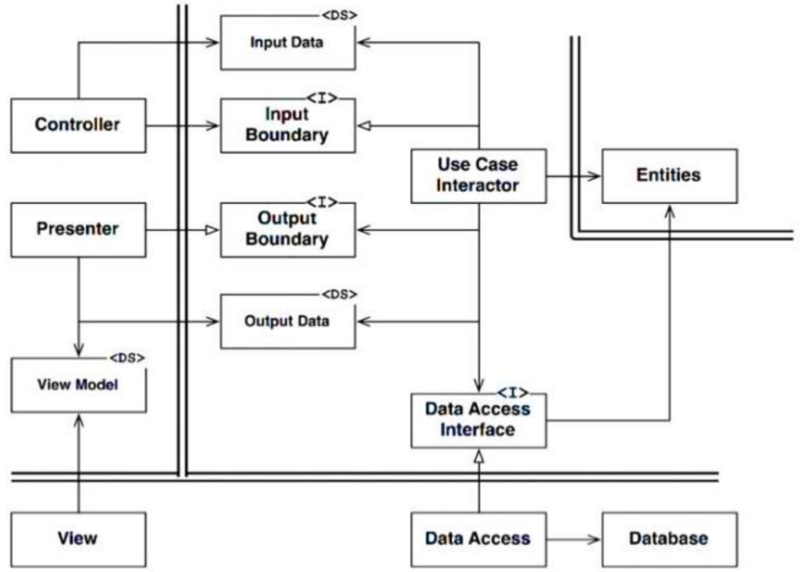


Figure 22.2 A typical scenario for a web-based Java system utilizing a database

The diagram in previous slide shows...

The diagram in previous slide shows a typical scenario for a web-based Java system using a database. The web server gathers input data from the user and hands it to the *`Controller`* on the upper left. The `Controller` packages that data into a plain old Java object and passes this object through the `InputBoundary` to the `UseCaseInteractor`. The `UseCaseInteractor` interprets that data and uses it to control the dance of the Entities. It also uses the `DataAccessInterface` to bring the data used by those `Entities` into memory from the `Database`. Upon completion, the `UseCaseInteractor` gathers data from the `Entities` and constructs the `OutputData` as another plain old Java object. The `OutputData` is then passed through the `OutputBoundary` interface to the Presenter.

Continue...

The job of the `Presenter` is to repackage the `OutputData` into viewable form as the `ViewModel`, which is yet another plain old Java object. The `ViewModel` contains mostly `Strings` and flags that the `View` uses to display the data. Whereas the `OutputData` may contain `Date` objects, the `Presenter` will load the `ViewModel` with corresponding `Strings` already formatted properly for the user. The same is true of `Currency` objects or any other business-related data. `Button` and `MenuItem` names are placed in the `ViewModel`, as are flags that tell the View whether those `Buttons` and `MenuItems` should be gray.

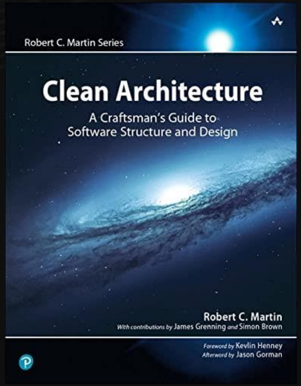
This leaves the View with almost nothing to do other than to move the data from the `ViewModel` into the `HTML` page.

See code example on GitHub

THANKS!

References :

- MARTIN, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. 1st. ed. USA: Prentice Hall Press, 2017. ISBN 0134494164.
- Dev.to/rubemfsv/clean-architecture-the-concept-behind-the-code-52do
- en.wikipedia.org/wiki/Use_case


Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman