Clean architecture y Programación Reactiva.

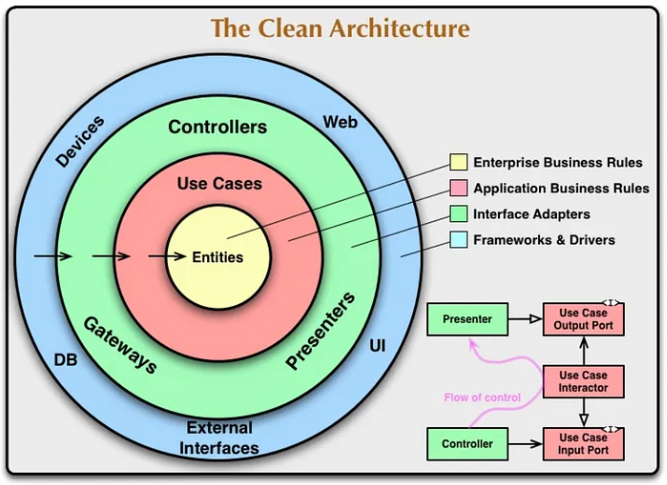
**CLEAN ARCHITECTURE**

**Why do we need to architect?**

“The goal of software architecture is to minimize the human resources required to build and maintain the required system.” *― Robert C. Martin, Clean Architecture.*

Advantages of Proper Architecture:

* Testable
* Maintainable
* Changeable
* Easy to Develop
* Easy to Deploy
* Independent



Each circle represents different areas of the software. The outermost layer is the lowest level of the software and as we move in deeper, the level will be higher. In general, as we move in deeper, the layer is less prone to change.

**The Dependency Rule**

The Dependency Rule states that the source code dependencies can only point inwards.

This means nothing in an inner circle can know anything at all about something in an outer circle. i.e. the inner circle shouldn’t depend on anything in the outer circle. The Black arrows represented in the diagram show the dependency rule.

Diagrama

Descripción generada automáticamente

Remember, the arrow should be read as “depend on”. Frameworks and Drivers should depend on Interface Adapters, which depend on Application Business Rules which depend on Enterprise Business Rules.

Nothing in the bottom layer should depend on the top layer.

**Frameworks and Drivers**

Software areas that reside inside this layer are

* User Interface
* Database
* External Interfaces (eg: Native platform API)
* Web (eg: Network Request)
* Devices (eg: Printers and Scanners)

**Interface Adapters**

This layer holds

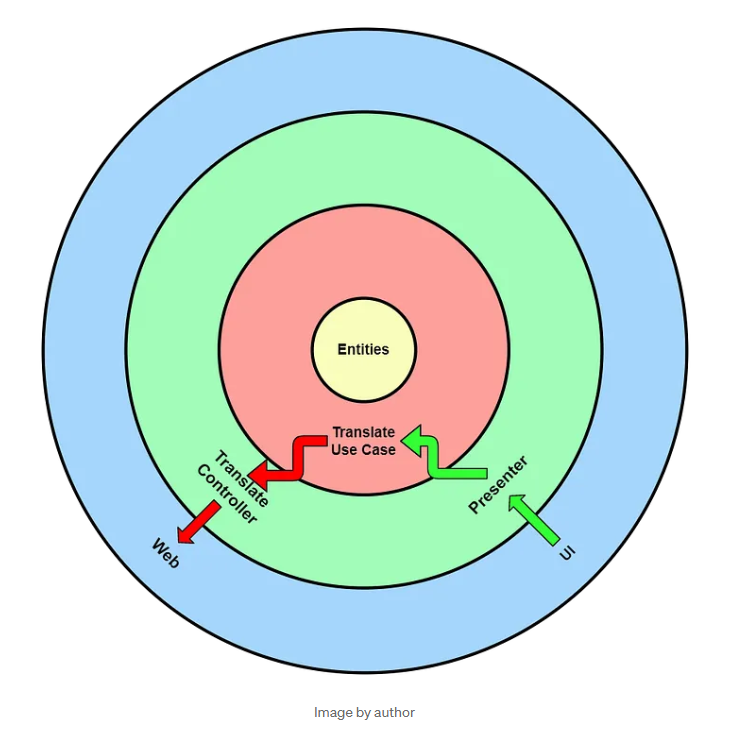
* Presenters (UI Logic, States)
* Controllers (UI Logic, States)(Interface that holds methods needed by the application which is implemented by Web, Devices or External Interfaces)
* Gateways (Interface that holds every CRUD operation performed by the application, implemented by DB)

**Application Business Rules**

Rules which are not Core-business-rules but essential for this particular application come under this. This layer holds Use Cases**.**As the name suggests, it should provide every use case of the application. i.e. it holds each and every functionality provided by the application.

Also, this is the layer that determines which Controller / Gateway to be called for the particular use case. Sometimes we need controllers from different modules.

Also, this is the layer that determines which Controller / Gateway to be called for the particular use case. Sometimes we need controllers from different modules.

Example wrong:  


**Enterprise Business Rules**

This is the layer that holds core-business rules or domain-specific business rules. Also, this layer is the least prone to change.

Change in any outer layer doesn’t affect this layer. Since Business Rules won’t change often, the change in this layer is very rare. This layer holds Entities.

Diagrama

Descripción generada automáticamente

*After all, how can we expect the*web*to throw some data to**the*Controller*without the*Controller*being dependent on it? Also, how can we expect the*Use Case*to get the proper data from the*Controller*without depending on it?*

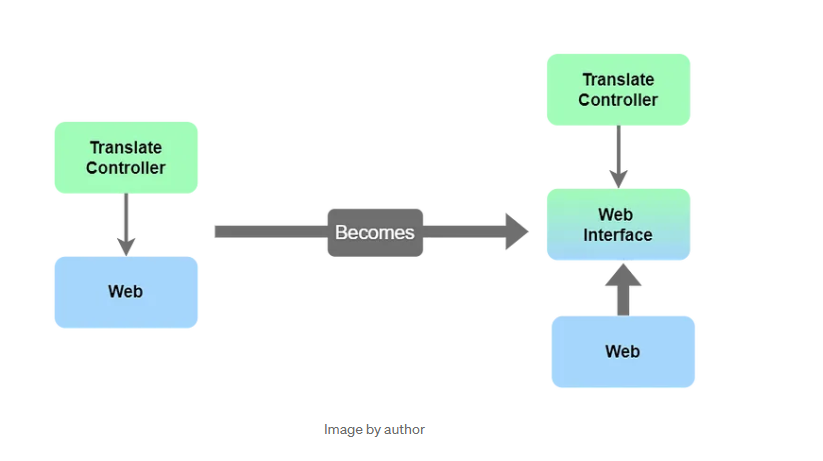
*But the Dependency Rule strictly says dependencies can only point inwards. It adds up by saying this is the rule that makes the architecture work.*

In order to pass this rule, we need to invert the arrow to the opposite direction. Is that possible? Here comes Polymorphism**.**When we include some Polymorphism here, something magic happens.

Simply by having an Interfacebetween these 2 layers, we could invert the dependency. This is known as The Dependency Inversion Principle.

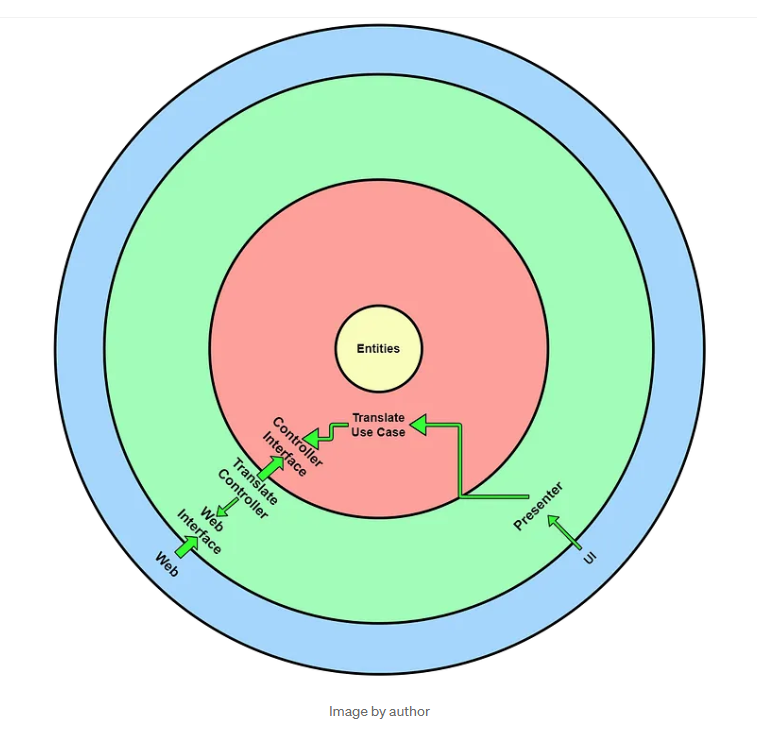
Let’s implement the Dependency Inversion Principle in the cases where the Dependency Rule is violated.

RIGTH WAY:



Diagrama

Descripción generada automáticamente



Imagine you’re in a hotel. We want the hotel to serve us what we want, but not what they offer right?. The same thing is happening here, we want the DB to give the data the application needs but not the data it has.

Also, the single-way dependency rule saves the application from the deadlock state. i.e. imagine in a 2 layer architecture, the first layer depends on the second layer, and the second layer depends on the first layer. In such a case, If we need to change anything in the first layer, it breaks the second layer. If we need to change anything in the second layer, it breaks the first layer. This can be rejected by following the deadlock state.

Application orders what data it wants and it doesn’t care how DB or API prepares the data. This way, the application doesn’t depend on DB or API. If we need/want to change the DB or API Schema in the future, we can simply change it. As far as it gives what the application asks for, the application doesn’t even know the change in DB or API.

Also, the single-way dependency rule saves the application from the deadlock state. i.e. imagine in a 2 layer architecture, the first layer depends on the second layer, and the second layer depends on the first layer. In such a case, If we need to change anything in the first layer, it breaks the second layer. If we need to change anything in the second layer, it breaks the first layer. This can be rejected by following the deadlock state.

Source: [The Clean Architecture — Beginner’s Guide | by Bharath | Better Programming](https://betterprogramming.pub/the-clean-architecture-beginners-guide-e4b7058c1165)

**Reactive Programming:**

Patrón **de diseño Observer**: necesidad de notificar cambios.

Analogía del canal de Youtube: un publicador (canal) y los observadores (suscriptores). El canal necesita que todos los suscriptores noten cuando él sube un nuevo vídeo a su canal de Youtube. Se necesita un método para registrar sus observadores.

Diagrama

Descripción generada automáticamente

**What’s reactive programming?**