TRABAJO PRÁCTICO N° 2
ARQUITECTURA CLEAN

Integrantes:

Cánepa Enzo

Jones Martin

Monzón Job

Tomasella Tobias

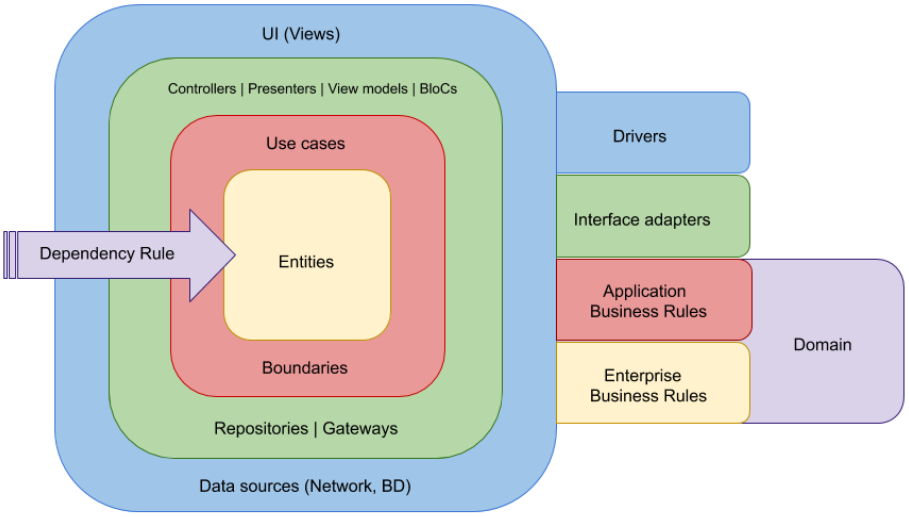
Profesor/a: Jose A. Fernandez

Cátedra: Paradigmas y Lenguajes de Programación III Cuatrimestre 2do

Carrera: Ingeniería en sistemas de información

¿Qué es?

Clean Architecture es un nombre popularizado por Robert Cecil Martin, conocido como “Uncle Bob” que se basa en la premisa de estructurar el código en capas contiguas, es decir, que solo tienen comunicación con las capas que están inmediatamente a sus lados. Basados en esta idea podemos encontrar artículos que hablan sobre Clean Architecture, Onion Architecture, Hexagonal Architecture, todas ellas tienen diferentes enfoques, pero comparten la idea de que cada nivel debe realizar sus propias tareas y se comunica únicamente con sus niveles inmediatamente contiguos.



La regla de dependencia:

La principal característica de Clean Architecture frente a otras arquitecturas es la regla de dependencia.

En Clean Architecture, una aplicación se divide en responsabilidades y cada una de estas responsabilidades se representa en forma de capa.

De esta forma tenemos capas exteriores y capas interiores:

- Las capa más exterior representa los detalles de implementación
- Las capas más interiores representan el dominio incluyendo lógica de aplicación y lógica negocio empresaria

La regla de dependencia nos dice que un círculo interior nunca debe conocer nada sobre un círculo exterior. Sin embargo, los círculos exteriores sí pueden conocer círculos interiores.
La lógica de dominio es lo que menos va a cambiar con el tiempo y por lo tanto se debe evitar que tenga dependencias de detalles de implementación que van a cambiar con más frecuencia.

En qué capas se divide Clean Architecture?

Clean architecture se divide en las siguientes capas:

- Entidades
- Casos de uso
- Adaptadores
- Frameworks y drivers

Tambíen podemos ver estas capas bajo la siguiente agrupación:

- Dominio -> entidades y casos de uso
- Adaptadores
- Detalles de implementación -> frameworks y drivers

Dominio

El dominio es el corazón de una aplicación y tiene que estar totalmente aislado de cualquier dependencia ajena a la lógica o los datos de negocio.

Entidades

La lógica de negocio empresarial es aquella lógica que existiría aunque no tengamos una aplicación para automatizar los procesos de una compañía.
Por ejemplo, para realizar un pedido es necesario que deba existir un cliente con nombre, apellidos, NIF y una dirección.

La lógica y datos de negocio empresarial se representa utilizando las entidades.
Las entidades contienen los datos de negocio así como las reglas de negocio empresarial.
Las entidades de una aplicación podrían ser compartidas entre diferentes aplicaciones dentro de una compañía.

Por ejemplo en el caso anterior la entidad cliente podría compartirse entre una aplicación móvil, si quisiéramos realizar las validaciones sin necesidad de una llamada de red, y su backend o API porque las reglas para dar de alta un cliente serían las mismas.

Casos de uso

Los casos de uso representan la lógica de aplicación, que existe principalmente debido a la automatización de procesos mediante la aplicación y es inherente a cada aplicación.
Por ejemplo para crear un pedido, el caso de uso entre la aplicación móvil y la API tiene sentido que sean diferentes.
En el caso de uso crear pedido en la aplicación móvil, puede tener unas validaciones mínimas necesarias si queremos evitar llamadas a red previas y la posterior llamada a la API para crear el pedido.
Sin embargo en la API, tiene sentido que sea diferente. A parte de crear el pedido, es posible que el caso de uso se encargue de coordinar el envío de email al usuario, gestión de stock, generación de factura etc.

Adaptadores

Los adaptadores se van a encargar de transformar la información como se entiende y es representada en los detalles de implementación o frameworks, drivers a como la entiende el dominio.
Es habitual utilizar en este punto junto con clean architecture diferentes patrones de presentación como MVC, MVP, MVVM o BloC donde los presenters, controllers, viewmodels o blocs serían los adaptadores encargados de transformar la información de las vistas a información que necesitan los casos de uso.

Características:

1. Es independiente de cualquier framework

La arquitectura limpia debe ser capaz de aplicarse a cualquier sistema sin importar el lenguaje de programación o las librerías que utilice. Las capas deben quedar tan bien separadas que puedan sobrevivir individualmente, sin necesidad de externos.

2. Testeable

Entre más pura sea una función, clase o módulo más fácil será predecir el resultado a obtener. Cuando hablamos de que algo sea puro nos referimos a que no tenga efectos colaterales. Lee este artículo para que entiendas mucho más sobre los efectos colaterales y funciones puras. (Importante leerlo para completar la lección). Cada módulo, tanto de UI, base de datos, conexión a API Rest, etc., se debe poder testear de forma individual.

3. Independiente de la interfaz de usuario (UI)

Uno de los componentes que sufren cambios constantemente es la interfaz de usuario. La UI debe ser capaz de cambiar sin alterar todo el sistema y, si vamos más allá, esta capa debería vivir tan independiente que podría ser desensamblada y sustituida por otra. Por ejemplo, cambiar una UI Móvil por una en modo consola.

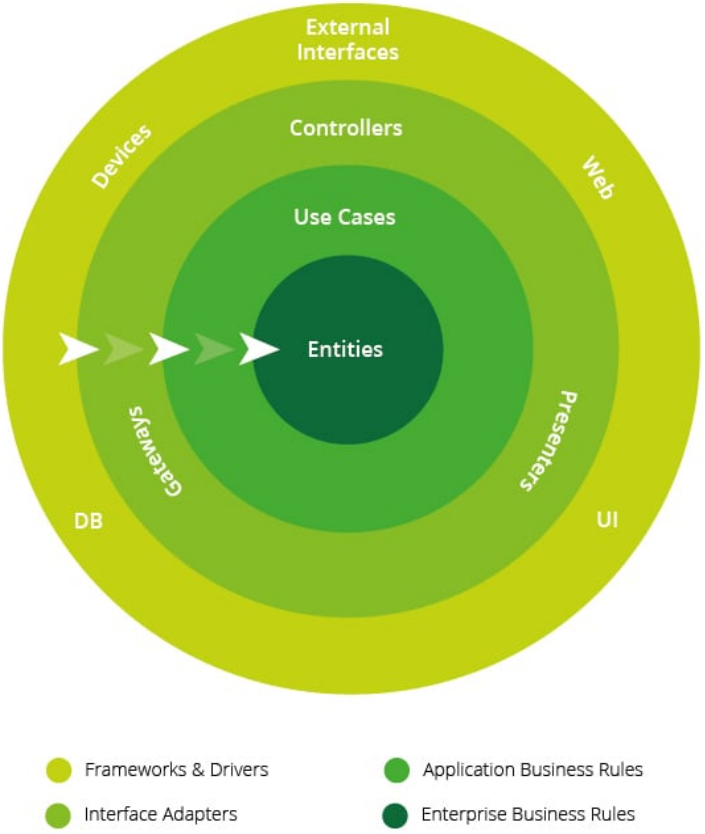
4. Independiente de la base de datos

Así como en el punto anterior, esta capa debe ser tan modular que sea posible agregarle múltiples fuentes de datos, e incluso múltiples fuentes del mismo tipo de datos. Por ejemplo, manejar varias bases de datos como MySQL, PostgreSQL, Redis, etc.

5. Independiente de cualquier elemento externo

Si en algún punto nuestro sistema necesita de una librería, otro sistema o cualquier elemento a conectar, debería ser fácilmente ensamblado y también debería ser modularizado. De hecho, para el sistema esta capa externa debería ser transparente.
Estos principios fueron graficados por el Tío Bob en el siguiente diagrama:

Este es el gráfico original que propone Uncle Bob y se puede apreciar cómo los diferentes niveles anteriormente descritos sólo se comunican con los inmediatamente contiguos.



A continuación, tenemos otra interpretación del gráfico tradicional, en la que se han añadido descripción de las tareas que se suelen realizar en cada uno de los niveles.

Cada uno de los niveles aquí representados podría considerarse dentro de la estructura de nuestra aplicación como carpetas independientes, diferentes módulos o librerías que se incluyen como dependencias del proyecto principal o incluso podríamos aplicar este tipo de estructuración sin la necesidad de reflejar de forma explícita en la estructura del proyecto.

¿Cuándo usar Clean Architecture?

Clean architecture tiene sentido cuando estamos desarrollando una aplicación que va a tener una duración de vida media o larga.
Si tienes que crear una aplicación para un evento como una feria y después no se va a mantener más tal vez no tenga sentido.

Si por el contrario va a tener una duración media o larga, donde la aplicación va a ser evolucionada con nuevas features, vamos a actualizar librerías de las que depende a nuevas versiones en un futuro o utilizar nuevas, tiene sentido usar Clean Architecture.

Ventajas

***Independencia**:* cada capa tiene su propio paradigma o modelo arquitectónico como si se tratara de una aplicación en sí misma sin afectar al resto de los niveles.
***Estructuración**:* mejor organización del código, facilitando la búsqueda de funcionalidades y navegación por el mismo.
Desacoplamiento: cada capa es independiente de las demás por lo que podríamos reemplazarla o incluso desarrollar en diferentes tecnologías. Además de reutilizar alguna de ellas en diferentes proyectos.
Facilidad de testeo: podremos realizar test unitarios de cada una de las capas y test de integración de las diferentes capas entre sí, pudiendo reemplazarlas por objetos temporales que simulan su comportamiento de forma sencilla.

Desventajas

Metodología: todo el equipo de desarrollo debe conocer la metodología que se está aplicando y cada desarrollador debe ser responsable de entender y aplicar las reglas establecidas a medida que se está desarrollando.

Complejidad: la velocidad de desarrollo al comienzo del proyecto es menor debido a que hay que establecer esta estructura y todo el equipo debe adaptarse a la nueva forma de trabajar, pero poco a poco y a medida que la aplicación va creciendo el mantenimiento y ampliación será más sencillo.

Conclusiones

Hemos visto porque utilizo clean architecture, una introducción a las principales partes de arquitectura y sus ventajas.
Al final se trata de utilizar las herramientas en tu trabajo con las que te sientas más cómodo y hacer tu trabajo mejor, de una forma más productiva.

Bibliografía:

https://dev.to/japhernandez/clean-architecture-with-nodejs-typescript-and-mongo-3ene

https://es.linkedin.com/pulse/clean-architecture-cristofer-padilla

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

https://platzi.com/clases/1619-arquitectura-android/21324-que-es-clean-architecture/

https://codersopinion.com/blog/clean-architecture-building-software-that-lasts/