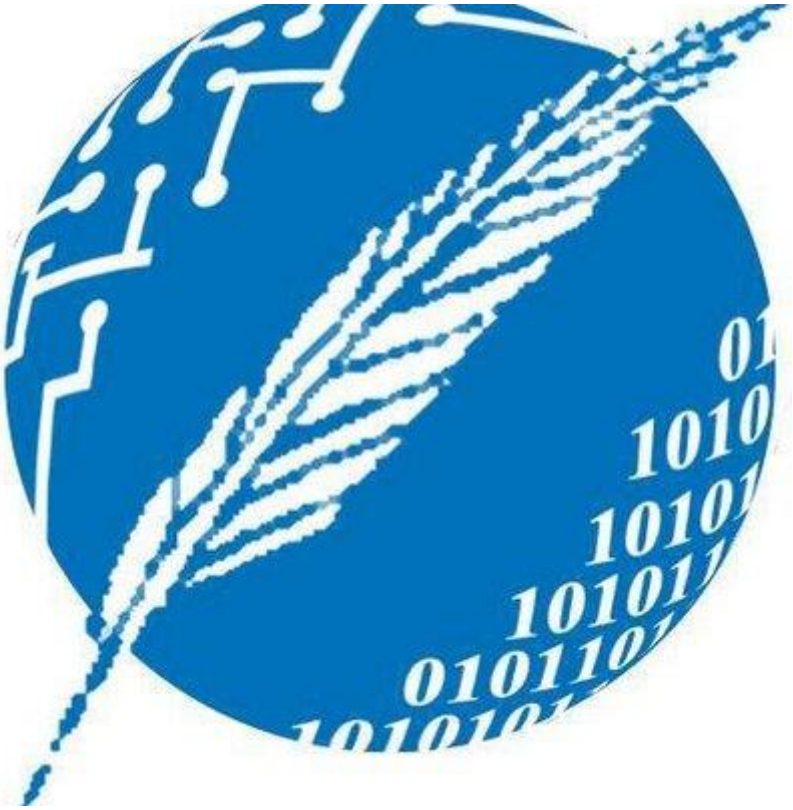
Arquitectura y Diseño de Sistemas

Informe de proyecto: Clean Architecture



Desarrollado por Labrisca Joaquín

LU 94445

Agosto del 2018

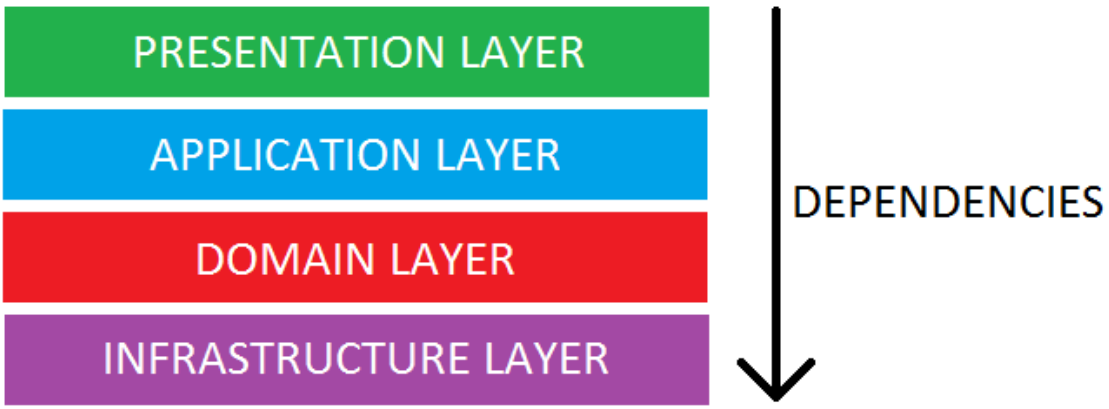
Indice de contenidos:

Sección 1: Conceptos previos

Antes de definir las características propias de la denominada Arquitectura Clean (*Clean Architecture)* resulta conveniente definir otros conceptos que están detrás de la idea y el funcionamiento de la misma. Estos conceptos son los siguientes: la arquitectura por capas *(Layered architecture*), la arquitectura hexagonal (*Hexagonal Architecture*), arquitectura cebolla (*Onion Architecture*) y paquete por característica (*Package by Feature*). Los mismos se detallan a continuación:

Layered architecture:

Se refiere a separar el código en capas, donde cada una de las capas tiene una responsabilidad diferente, como el principio de responsabilidad única en SOLID pero aplicado a la arquitectura del sistema en vez de a un conjunto de clases. Para eso se separa al sistema en cuestión en cuatro capas como sugiere la siguiente imagen:



Capa de presentación: Contiene todas las clases responsables de presentar la interfaz de usuario al usuario final o enviar la respuesta al cliente en caso de tratarse de una aplicación *back-end*.

Capa de aplicación: Contiene toda la lógica necesaria para que la aplicación cumpla con sus requerimientos funcionales.

Capa de dominio: Contiene el dominio subyacente, compuesta principalmente por las entidades del dominio y en algunos casos servicios. Además de las reglas de negocio.

Capa de infraestructura: También conocida como capa de persistencia. Contiene todas las clases responsables del trabajo técnico,

como guardar o recuperar datos desde o hacia la base de datos, comunicación con servicios externos, etc.

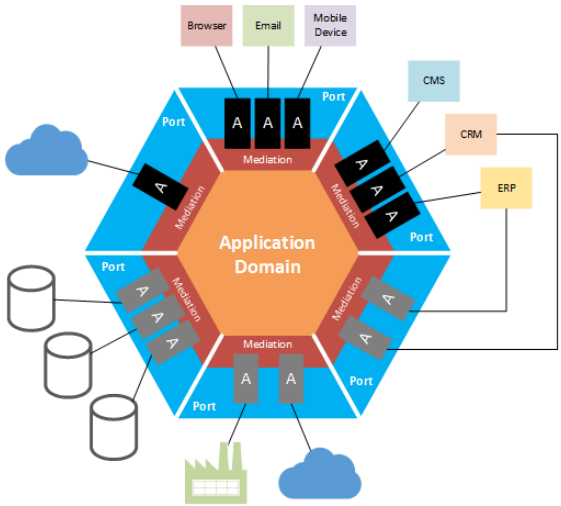
La idea de organizar el proyecto en capas sugiere que esta separación se logre mediante la organización del código fuente, pero no sugiere nada más acerca de la implementación del sistema en sí. Por este motivo debe ser complementada con alguna otra técnica o arquitectura.

Hay dos principios o reglas básicas que deben cumplirse para que la arquitectura por capas esté implementada de manera correcta:

- Como indica la flecha en la imagen, todas las dependencias van desde presentación hacia infraestructura. Esto se conoce como la regla de dependencia (*Dependency Rule*).
- Nada de la lógica correspondiente a una capa debe ser colocada en otra.

Hexagonal Architecture:

En este estilo de arquitectura se en vez de considerar capas conceptuales se hacen dos grandes distinciones. Por un lado, se considera la parte *interna* del sistema y por otro la parte *externa* del mismo. La parte interna del sistema es todo aquello que formaría las capas de aplicación y dominio en la arquitectura por capas detallada anteriormente. Y la parte externa es todo el resto (GUI, base de datos, etc.). La conexión entre la parte interna y externa se logra mediante abstracciones llamadas *puertos* y sus implementaciones llamados *adaptadores*. Por este motivo la arquitectura hexagonal también es conocida como la técnica de patrones y adaptadores. La siguiente imagen da una mejor idea sobre estos conceptos que a primera vista pueden resultar muy abstractos:

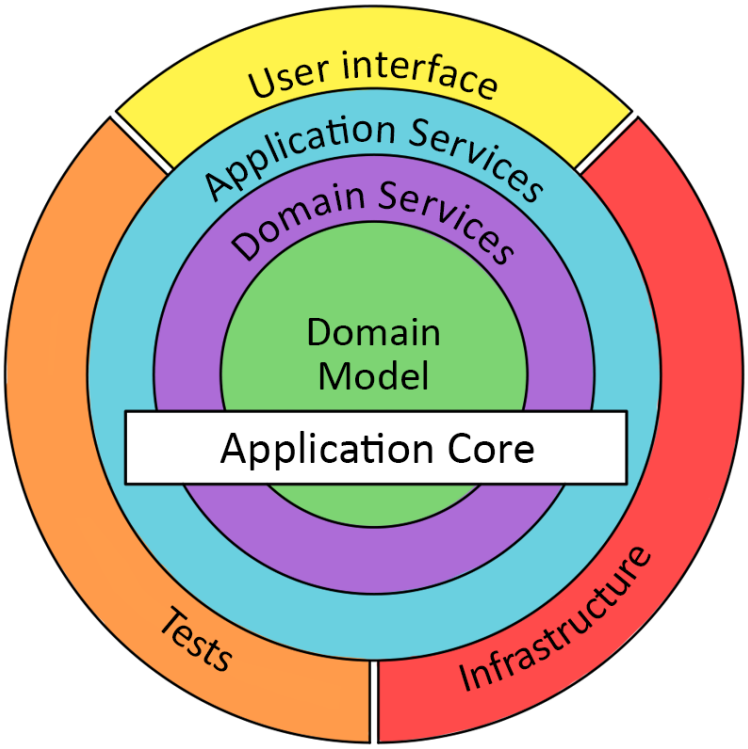


El aspecto mas importante a destacar de esta arquitectura es que separa el núcleo de la aplicación del resto del sistema y permite acoplar los componentes mediante los puertos y adaptadores. Lo que hace que la parte interna del sistema no dependa de la externa, es completamente independiente de la GUI, de la base de datos, de los *frameworks* que puedan estarse utilizando, etc. Por lo que los componentes que forman esta parte externa pueden ser reemplazados sin tener que modificar o recompilar ningún componente que forme el núcleo de la aplicación.

Al igual que en la arquitectura por capas, existe un conjunto de principios a cumplir:

1) La parte interna no conoce nada sobre la externa.
2) Debe poder utilizarse cualquier adaptador que encaje en un puerto.
3) Ningún caso de uso ni lógica de dominio va hacia la parte externa.

Onion Architecture:



Esta arquitectura está fuertemente relacionada a las dos introducidas anteriormente. Está definida por capas (al igual que la Layered Architecture) pero estas capas son ligeramente diferentes y se detallan a continuación:

Capa de modelo de dominio: Aquí residen las entidades y clases directamente relacionadas al mismo.

Capa de servicios de dominio: Aquí residen los procesos específicos del dominio.

Capa de servicios de la aplicación: Aquí reside la lógica que es específica de la aplicación. Esto es, los casos de uso.

Capa exterior: Contiene todo lo periférico del sistema. Aquí residen la GUI, la base de datos, cosas relacionadas con testing, etc.

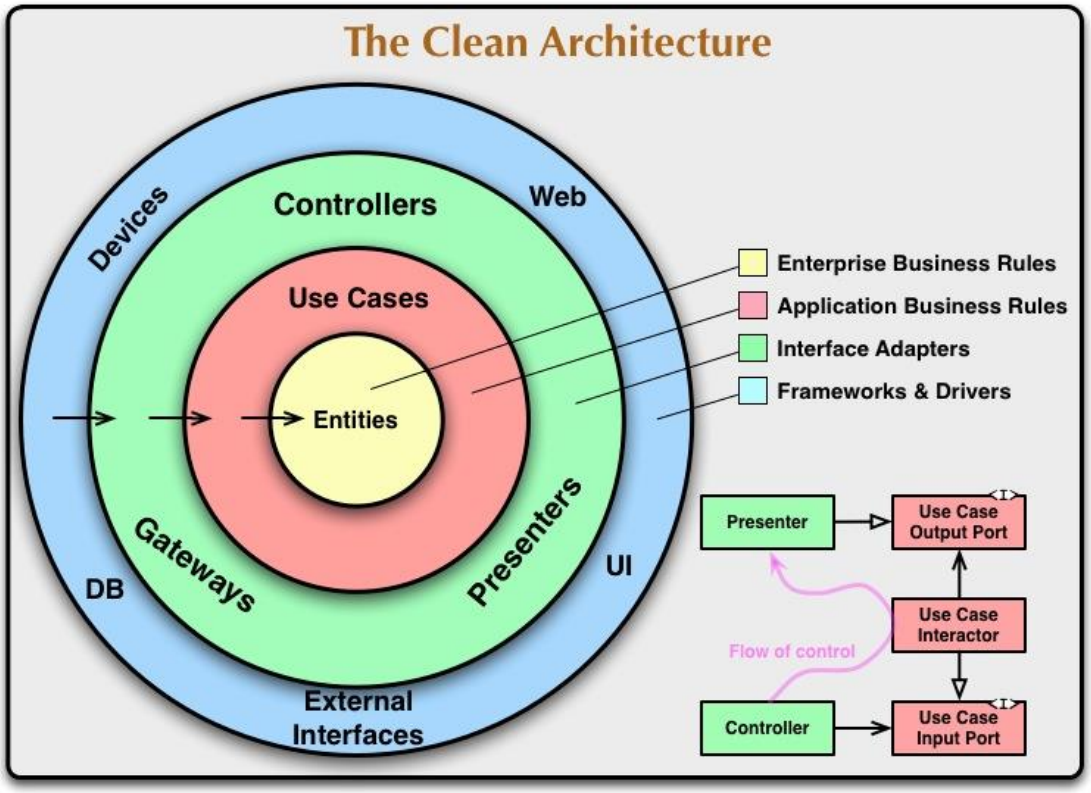
Como indica la imagen, cada una de estas capas envuelve a la interior. Formando así una cebolla (Onion). Entre estas capas rige fuertemente la regla de dependencia (Dependency Rule). Esto es, las capas exteriores pueden depender de las interiores, pero **nunca** inversamente. Esto es en esencia el famoso Principio de Inversión de Dependencias, pero aplicado a nivel de arquitectura en vez de a un conjunto de clases.

Package by Feature:

En una típica arquitectura por capas, la organización del código sigue una descomposición horizontal determinada por la división del sistema en capas. Esto comúnmente se conoce como *Empaquetamiento por capas*. El problema con esa aproximación es que la cohesión dentro de esos paquetes es muy baja y existe un alto acoplamiento entre los mismos. Esto parece ser exactamente lo contrario a lo que se pretende logar.

Para solucionar lo mencionado anteriormente surge el concepto de *empaquetamiento por característica*. Lo que sugiere es mover las capas un nivel más abajo (al nivel de clases), concentrarse en el acoplamiento y cohesión a un nivel más alto. Manteniendo todas las clases relacionadas a la misma característica en el mismo paquete/módulo. Permitiendo que sea posible darse cuenta a simple vista (mirando la organización de los mismos) qué es lo que hace el sistema. Esto ultimo es lo que Robert C. Martin llamó *Screaming Achitecture*.

Sección 2: Clean Architecture



Clean Architecture se construye sobre los conceptos introducidos anteriormente y alinea al proyecto con mejores prácticas como el *Principio de Inversión de Dependencias* o los *Casos de Uso*. También apunta a una máxima independencia de cualquier framework o herramientas que puedan estar interfiriendo con la testeabilidad del sistema o el reemplazo de dicho componente. Esta arquitectura divide al sistema en cuatro capas:

Entidades: Son los objetos de negocio de la aplicación. Encapsulan las reglas de negocio mas generales y de alto nivel. Son las que menos probabilidades tienen de cambiar cuando algo externo cambie.

Casos de uso: El software en esta capa contiene las reglas de negocio específicas de la aplicación. Encapsulan e implementan todos los casos de uso del sistema. Estos casos de uso orquestan el flujo de información desde y hacia las entidades. No es de esperar que cambios en esta capa afecten a las entidades, así como tampoco se espera que esta capa se vea afectada por cambios a las capas exteriores. Lo que sí ocurrirá

es que si hay cambios en la operación de la aplicación afecten los casos de uso y por lo tanto el software en esta capa.

Adaptadores de interfaces: El software en esta capa es un set de adaptadores que convierten los datos desde la forma mas conveniente a los casos de uso y entidades hacia los más convenientes para algún agente externo como puede ser la base de datos o la web. Además, dentro de esta capa se encuentran los adaptadores necesarios para convertir los datos obtenidos desde una fuente o servicio externo.

Frameworks y drivers: La capa exterior del sistema está compuesta generalmente por frameworks y herramientas como la base de datos, GUI, etc. Generalmente no hay mucho código dentro de esta capa, salvo el que comunica a la misma con la capa interior. En esta capa se agrupan todos los detalles, que son mantenidos afuera donde menos daño pueden hacer. Un cambio en los mimos no tiene porqué producir cambios en las capas internas.

Como se menciona anteriormente, las dos características principales que separan Clean Architecture de otros estilos son:

1) **Fuerte adherencia a la *Dependency Rule*:** Al igual que la *Onion Architecture*, establece de manera explícita las prioridades entre los diferentes objetos dentro del sistema. De una forma, la Clean Architecure lo hace de una mejor manera ya que dedica una capa entera solamente para los detalles por fuera de las demás capas que componen al sistema.
2) **Orientación a los *Casos de Uso*:** Similarmente a cómo se expuso en *Empaquetamiento por Característica*, Clean Architecture promueve la separación vertical del código dejando las capas casi a nivel de clases. La mayor diferencia entre las dos es que en lugar de concentrarse en el difuso concepto de *feature* (característica), reorienta el empaquetamiento hacia los Casos de Uso. Esto es fundamental ya que permite mapear funcionalidad definida por los casos de uso especificados directamente a los paquetes que contienen la lógica correspondiente a los mismo.

Sección 3: Ventajas

Clean Arquitecture consolida todas las ventajas que tienen individualmente las arquitecturas mencionadas en la sección 2. Se pueden resumir de la siguiente manera:

Screaming: Los Casos de Uso son claramente visibles en la estructura del proyecto, revelando la funcionalidad de la aplicación rápidamente.

Flexible: Si se implementa correctamente, se pueden cambiar frameworks, bases de datos o servidores muy fácilmente. Ya que nada dentro del proyecto depende de ellos.

Testeable: Las interfaces entre las capas permiten setear el alcance y las interacciones de cualquier forma posible.

Sección 4: Desventajas

Algo que no se detalló en ningún momento fueron las posibles desventajas o *trade-offs* que puede traer Clean Architecture. Algunas pueden ser:

No existe un framework: La regla de dependencia es implacable en este sentido y rige la arquitectura.

Curva de aprendizaje: Es un poco mas difícil de captar que otros estilos, especialmente considerando el punto anterior.

Pesada: En el sentido de que se terminan definiendo muchas mas clases que con algún otro estilo.

Sección 5: Conclusiones

La Clean Architecture es una arquitectura muy bien elaborada y podría considerarse como la evolución de otras anteriores, fusionando sus ventajas. Pero también requiere de mucho tiempo para ubicar cada elemento en la capa correspondiente y definir las interfaces y adapters entre las mismas teniendo mucho cuidado de no violar la regla de

dependencia. Por lo que se reduce a preguntarse si el sistema vivirá lo suficiente como para que las ventajas en el largo plazo paguen el esfuerzo extra inicial. ¿Al final del día, no es éste el motivo por el cual se dedica tanto esfuerzo a la tarea de definir lo antes posible la arquitectura de un sistema?

Sección 6: Cambios realizados al Proyecto de cursado.

Con el objetivo de llevar el proyecto de cursado de una arquitectura MVC (Model View Controller) a una Arquitectura Clean, se realizó un conjunto importante de cambios que se detalla a continuación:

-Se reemplazaron los paquetes correspondientes a Model View y Controller por tres módulos: Data, Model y J2CharlieDic que implementan el acceso a datos, el modelo del dominio de la aplicación y la vista respectivamente.

-Se eliminó el paquete Controller, ya que la vista interactúa directamente con el dominio para invocar a un caso de uso.

-Primero se identificaron los objetos de negocio de la aplicación y se colocaron dentro de un paquete llamado Entities dentro del módulo Model. Estas entidades son SearchResult y SearchError.

El primero tiene el término buscado, el resultado retornado por la capa de acceso a datos y la fuente de dicho resultado. Mientras que el segundo mantiene una descripción del error, así como la fuente en la cual se produjo dicho error.

-Luego se identificaron los casos de uso. En este caso solo hay uno, el caso de uso SearchTermUseCase que fue colocado dentro del paquete UseCases dentro del módulo Model. Dentro de este paquete también se declararon dos interfaces: Repository y ErrorHandler, que son necesarias por el caso de uso para realizar su trabajo. Se declararon aquí para no violar la Dependency Rule, de manera que si se desea cambiar de repositorio o el manejador de errores simplemente se implementa una clase que implemente dichas interfaces (en la capa correspondiente) y no es necesario modificar el core de la aplicación como consecuencia de la alteración de un detalle.

-Una vez finalizado el módulo Model (compuesto por las entidades y casos de uso) se prosiguió a implementar la capa de acceso a los datos, el

módulo Data. Aquí dentro se separaron las responsabilidades dentro de varios paquetes:

DataBase: Aquí se agrupan todas las clases relacionadas con el acceso a los datos persistentes de la aplicación.

DataEntities: Aquí se definieron entidades para el uso exclusivo dentro del módulo Data. Parece una duplicación innecesaria de las entidades definidas en la capa mas interna del sistema, pero esto se hace para que dentro de este módulo se trabajen con entidades independientes al core de la aplicación y así poder mantener mejor la independencia entre los módulos.

ErrorHandler: Dentro de este paquete hay una clase que implementa la interfaz definida en el caso de uso. Aunque este manejador de errores funciona de manera diferente al implementado en MVC. Ya que guarda una lista con todos los errores ocurridos durante una instancia de ejecución en particular de un SearchTermUseCase en lugar de notificar por cada uno que ocurre.

ExternalSearchServies: Aquí se agrupan las interfaces y adaptadores para definir, adaptar, agrupar y finalmente acceder a los servicios de las entidades externas. A la interfaz ExternalSearchService se le agregó el método getSource() que retorna un String correspondiente con el nombre del servicio externo de búsqueda que se está invocando. Se eliminó el enumerado definido anteriormente ya que si se lo ubicaba dentro del módulo Data debería poder ser visible por el Model para definir el Source de las entidades, mientras que si se definía dentro de Model se estarían incluyendo nombres de detalles en el core de la aplicación, violando principios de Clean Architecture en ambos casos. Cada adaptador que implementa dicha interfaz retorna el nombre del servicio correspondiente. Finalmente se modificó la forma de agrupar a dicho conjunto de servicios externos, en lugar de un mapeo se utilizó una lista iterable, de esta forma se puede iterar sobre la lista invocando al método SearchTerm() de la interfaz independientemente de los nombres de cada servicio.

ModelAdapters: Aquí dentro se implementaron los mappers entre la capa de acceso a datos al modelo, como se explicó

anteriormente. Para esto se implementó un adaptador correspondiente a cada entidad de la capa interior del sistema.

Repository: Dentro de este paquete se implementa la interfaz definida dentro de los Casos de Uso. El funcionamiento es casi igual al definido para MVC. La única diferencia importante es que, como se comentó, itera sobre los servicios externos independientemente de su nombre y cuando ocurre un error con alguno de ellos simplemente lo notifica al ErrorHandler.

-Por último, se definió el módulo JCharlieDic, que corresponde a la vista. Aquí dentro se implementa la GUI, un conjunto de entidades propias y sus adaptadores correspondientes para mapearlas a entidades del modelo. Además del MainModule que contiene la clase con el método main y configura la inyección del único caso de uso existente a la vista para que pueda invocarlo y además le inyecta a este caso de uso el Repository y el ErrorHandler que el mismo necesita para su funcionamiento.