ARQUITECTURA LIMPIA

La arquitectura limpia tiene como objetivo la separación de dependencias. Está separación se logra dividiendo el software en capas. Todas tienen al menos una capa para la lógica del negocio y otra para la interfaz gráfica.

El uso correcto de este tipo de arquitectura nos proporcionará:

- Independencia al framework.
- Testable.
- Independencia de la interfaz de usuario, BD o cualquier agente externo.

La Regla de Dependencia

Esta regla dice que las dependencias del código fuente solo pueden apuntar hacia adentro. Nada en un círculo interno puede saber nada sobre algo en un círculo externo. En particular, el nombre de algo declarado en un círculo exterior no debe ser mencionado por el código en un círculo interior.

Entidades

Las entidades encapsulan la lógica de negocio de todo el proyecto. Una entidad puede ser un objeto con métodos o puede ser un conjunto de estructuras y funciones de datos.

Son los que tienen menos probabilidades de cambiar cuando algo externo cambia. Por ejemplo, no esperaría que estos objetos se vean afectados por un cambio en la navegación de la página o la seguridad.

Casos de uso

El software de esta capa contiene reglas comerciales específicas de la aplicación. Encapsula e implementa todos los casos de uso del sistema.

Adaptadores de Interfaz

El software en esta capa es un conjunto de adaptadores que convierten los datos obtenidos en las capas exteriores al formato más conveniente para los casos de uso y entidades. Además se adaptarán al marco de persistencia que se esté utilizando. Ningún código interno de este círculo debería saber nada sobre la base de datos.

Frameworks y drivers

La capa más externa generalmente se compone de marcos y herramientas como la base de datos, el marco web, etc. Generalmente, no escribe mucho código en esta capa aparte del código adhesivo que se comunica con el siguiente círculo hacia adentro.

¿Solo cuatro círculos?

No, los círculos son esquemáticos. Puede encontrar que necesita algo más que estos cuatro. No hay ninguna regla que diga que siempre debes tener solo estos cuatro. Sin embargo, siempre se aplica la regla de dependencia.

Qué datos cruzan los límites

Normalmente, los datos que cruzan los límites son estructuras de datos simples. Puede utilizar estructuras básicas u objetos simples de transferencia de datos si se desea. O los datos pueden ser simplemente argumentos en llamadas a funciones. Lo importante es que las estructuras de datos simples y aisladas se traspasan los límites. No queremos engañar y pasar *Entidades* o filas de bases de datos. No queremos que las estructuras de datos tengan ningún tipo de dependencia que viole la regla de dependencia.

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

Primera Capa:

Están las aplicaciones. Cada aplicación tiene el código de presentación, este suele residir en capas desacopladas.

Todas las aplicaciones construyen el grafico de dependencias. Esto implica leer la configuración y ya que la aplicación contiene todos los enlaces del framework, se encuentran allí todos los controllers.

Al estar las aplicaciones en la capa superior y que las dependencias solo pueden apuntar a las de abajo, ningún código de las aplicaciones puede depender del código de otras aplicaciones. Eso significa que hay 0 enlace a mecanismos externos.

Segunda capa:

Aquí están los Bounded Contexts, uno por subdominio. En cada núcleo del BC se encuentra el Domain Model, después el Domain Services, conteniendo toda la lógica de negocio e incluyendo las interfaces de persistencia. Sus dependencias solo pueden apuntar hacia dentro así que el modelo no puede depender de nada externo.

Prosiguen los casos de uso, estos pueden usar ambos dominios internos y a su vez forman una barrera para que nada exterior pueda interactuar con los interiores.

Los BC tienen su propia capa de persistencia. Esta puede usar su propio sistema de información. También utilizan implementaciones de *dominios de servicio,* usados por los casos de uso.

Estas implementaciones son la única cosa que tiene permiso para comunicarse con los niveles inferiores de la capa de persistencia. Por otro lado, las únicas cosas que pueden usar estas implementaciones de servicios son otros *dominios de servicios y los casos de uso.*

Los casos de uso, tienen 0 enlaces a los mecanismos de persistencia de fuera del BC, también su código responsable de la *lógica del dominio* no puede ser a accedido directamente desde cualquier lado, como en la presentación de la capa de una aplicación.

Tercera Capa:

Las aplicaciones y BC contienen todo el código del dominio, este código puede usar las librerías y por supuesto el runtime(PHP) de la tercera capa.

Fuente: https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/

Implementación de *Clean Architecture* en *React*.

Fuente: "React as an Implementation Detail – Chris Kiehl (Jun 05, 2019)" [Inglés]

A la hora de desarrollar aplicaciones con *React*, la gente tiende a dejar que el *framework* de *React* se adueñe de la arquitectura de dicha aplicación. *React* es un *framework* que está derivándose cada vez

más y más a la programación funcional (véase *React Hooks*. No obstante, esta práctica suele concluir con que el desarrollador que lo programó sea el único capaz de comprender el código totalmente.

```
const Headline: React.StatelessComponent = ({stuff}) => (
    <Title> {stuff.map(x => x.thing.name).join('-")} </Title>
)
```

El acoplamiento estructural va a dar problemas de inversión de dependencias: cuando desarrollas un componente directamente dentro de su contexto, estás tomando *React* como arquitectura única, mientras que tendría que serlo de una pieza de la aplicación. Este componente no es reutilizable. También es probable que los componentes se apilen entre ellos.

```
const MyThingList: React.Stateless = ({things}) => (
    {_.orderBy(things, ['group', 'createdOn']).map(stuff =>
            <Headline stuff={stuff} />
    )}
)
```

La solución a este problema es **separar los asuntos** relativos a la aplicación y desarrollar una **arquitectura acorde a estos**. Debemos desplazar a una capa central la lógica relativa al dominio o a los asuntos de negocio de la aplicación, dejando que *React* se encargue de la capa de presentación al usuario, para lo cual está pensado este *framework*.

```
const Headline: React.StatelessComponent = ({title}) => (
    <Title> {title} </Title>
)
```

Fuente: "Arquitetura limpa para bases de código *React* – Eduardo Morôni (Jun 27, 2018)" [Portugués]

Para ejemplificar las capas de la arquitectura limpia, tomaremos un ejemplo simple en el que implementaremos un contador.

Entidad.

Se trata de las reglas universales de negocio, que son independientes de la aplicación que va a simularla.

```
export class Counter {
     count: number;
     constructor(startNumber: number) { this.count = startNumber; }
     increment() { this.count += qty ? qty : 1; }
     decrement() { this.count -= qty ? qty : 1; }
}
```

Interactor.

Son los responsables de las reglas de negocio específicas de la aplicación. Coordinan qué debe ser hecho, dejando a elección por parte de quien lo instancie el cómo.

```
import { Counter } from "../entities";
export class CounterInteractor {
     higherBound: number = 10;
     count: Counter;
     constructor( startNumber: number, higherBound: number = 10 ) {
          this.count = new Counter(startNumber);
          this.higherBound = higherBound;
     }
```

```
    increment(qty?: number): Counter {
        this.counter.increment(qty);
        if (this.counter.count >= this.higherBound)
            this.counter = new Counter(this.higherBound);
        return this.counter;
    }
    decrement(qty?: number): Counter {
        // Omitted for simplicity
    }
}
```

Tríada Adaptadores – Presentadores – Componentes.
Ejemplificada con Reducers / Redux – React-Redux – React.

Adaptadores. Gestionan todo aquello que esté entre la capa de dominio y la capa de presentación. Mapean en formato de presentación más conveniente para el dominio de negocio. Nos permite aprovechar la lógica de negocio para diferentes tipos de aplicación.

Componentes (en concreto, los **presentacionales** de *React*). Todo aquello que es mostrado al usuario.

```
// Rest of the file omitted

const incrementReducer = ( counter: StateSliceType, action: ActionType ):
StateSliceType => {
    const interactor = new CounterInteractor(counter);
    interactor.increment(action.qty);
    return new Counter(interactor.counter.count);
};
export const counterReducer =
    ( state: StateSliceType = INITIAL_STATE, action: ActionType ): StateSliceType
=> {
    switch (action.type) {
        case INCREMENT:
            return incrementReducer(state, action);
        case DECREMENT:
            return decrementReducer(state, action);
        default:
            return state;
    }
};
```

En esta última parte, estaríamos siguiendo el patrón State – Reducer.

Fuente: “The state reducer pattern – Kent C Dodds (Feb 19, 2018)” [Inglés]