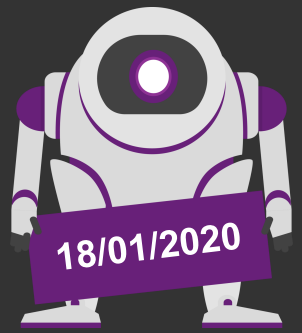
2020

Netcoreconf

Ensúciate con “Clean Architecture”



Jose María Flores Zazo

Development & Cloud Consultan in Tokiota

@jmfloreszazo

Sponsors

Microsoft

avanade

plain concepts

devs dna

everis
an NTT DATA Company

encamina
PIENSA EN COLORES

KABEL

SolidQ

intelequia

東京 TOKIOTA

birchman
VISEO GROUP

GIN MG
CASA FUNDADA
EN 1835

monday

Acerca de…

@jmfloreszazo

Jose María Flores Zazo

Tengo más de 24 años de experiencia en el campo de desarrollo, análisis, diseño y entrega de aplicaciones con diversas tecnologías.

Actualmente desempeño mi actividad laboral en TOKIOTA.

Cosas que me motivan

A partir de mi profesión, hacer la vida más fácil a las personas.

Disfrutar de la familia, leer y hacer deporte.

Esto no sería posible sin…

Robert C. **Martin**

Clean Architecture

Alistair **Cockburn**

Hexagonal Architecture

Ivar **Jacobson**

Object Oriented Software Engineering:
A Use Case Driven Approach

Manifiesto

SENTIDO COMÚN – Desarrolladores y Arquitectos

Tanto Robert C. Martin, Alistair Cockburn, Jeffrey Palermo, Martin Fowler, etc. no han inventado nada nuevo.

Concretamente Robert C. Martin, solamente a plasmado formalmente en un libro, artículo o conferencia una serie de aseveraciones del sentido común en el desarrollo del software y se han ganado la medalla correspondiente.

Muchas de las cosas que os mostraré son consecuencia directa del sentido común y del día a día de nuestra profesión.

Supongo a que, a alguien todo lo que cuente le resulta nuevo; me alegro por aportar algo en su vida. Y a otros tantos solo les servirá de recordatorio; que nunca está de más volver la vista atrás.

Sin más, comencemos.

Agenda

01 – ¿Qué tienen de malo los layers?
02 – Inversión de dependencias, acoplamiento y cohesión.
03 – ¿Por qué Clean Architecture?.
04 – Organizando el código.
05 – Demo: Implementación de un caso de uso, adaptador web y adaptador de persistencia.
06 – Testing.
07 – Mapeo entre los límites de la arquitectura.
08 – Cumplir con los límites de la arquitectura.
09 – Coger atajos conscientemente.
10 – Decidir un estilo de arquitectura.

¿Qué tienen de malo los layers?

LAYERS – Capas

Los más probable es que en el pasado hayas desarrollado aplicaciones por capas o incluso en la actualidad lo estés haciendo, es más, en la actualidad yo lo estoy haciendo.

Las **capas** están con nosotros desde hace tiempo. Seguramente esta filosofía de una aplicación web con un planteamiento convencional, nuestro hilo conductor a lo largo del curso te suene:

¿Qué tienen de malo los layers?

Usar LAYERS es un patrón sólido de arquitectura – "Clean Architecture" Robert C. Martin.

Si lo hacemos bien, podemos construir una lógica de dominio que sea independiente de la web y la persistencia. Podremos cambiar las tecnologías web y de persistencia y no afectarían a la lógica de dominio si ese es nuestro deseo. Incluso podremos incluir nuevas características sin afectar a las ya existentes.

Entonces, ¿qué tienen de malo las capas?. En mi experiencia, una arquitectura por capas tiene demasiados frentes abiertos que permiten la introducción de malos hábitos y hacer que el software sea cada vez más difícil de cambiar con el tiempo.

Ahora os diré el por qué.

¿Qué tienen de malo los layers?

Promueve un diseño basado en DATOS (1/3)

La base de una arquitectura basada en capas, por convención es la base de datos.

Principalmente intentamos modelar el comportamiento y no el estado. Sí, el estado es una parte importante de cualquier aplicación, pero el comportamiento es lo que cambia el estado y, por lo tanto, impulsa el negocio.

Entonces, ¿por qué estamos haciendo de la base de datos el cimiento de nuestra arquitectura y no la lógica del dominio?.

Piensa en los últimos casos de uso que ha implementación en cualquier aplicación. ¿Empezasteis por implementar la lógica de dominio o por la capa de persistencia?.

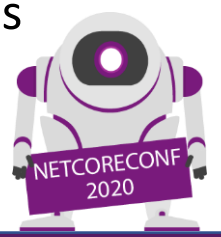
¿Qué tienen de malo los layers?

Promueve un diseño basado en DATOS (2/3)

Lo más probable es que hayas pensado en cómo se vería la estructura de la base de datos y sólo entonces te paraste a pensar la lógica de dominio que está encima de ella. Esto tiene sentido en una arquitectura convencional de capas ya que seguimos el flujo natural de las dependencias. Pero desde el punto de vista empresarial, no tiene ningún sentido.

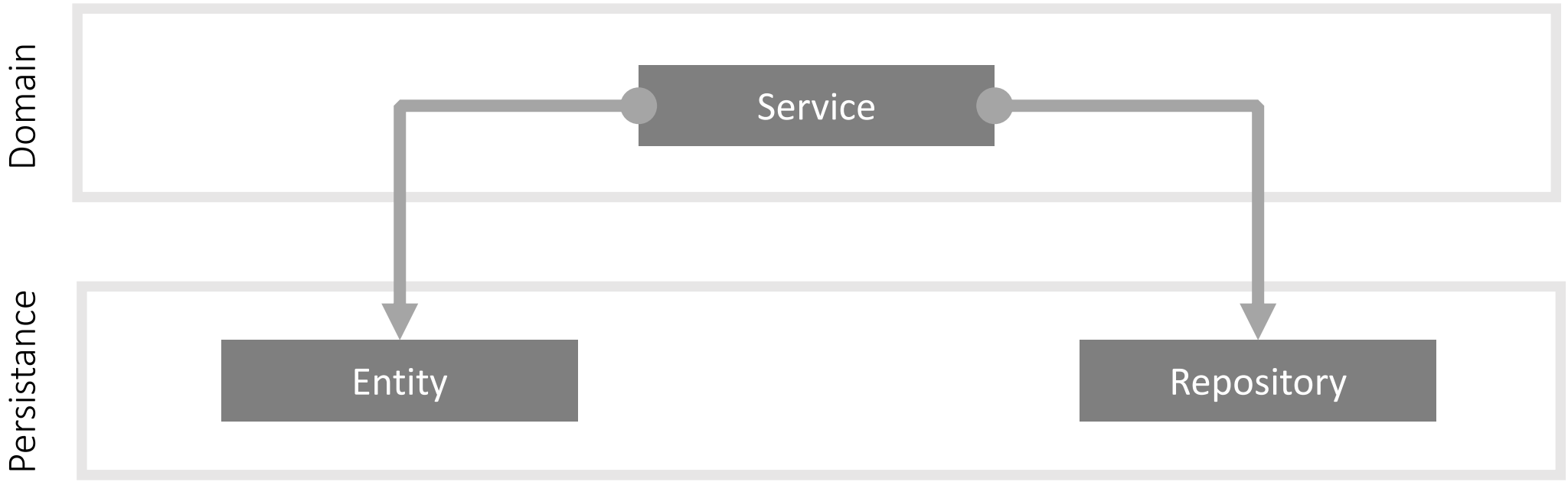
Debemos construir la lógica de dominio antes de hacer cualquier otra cosa. Sólo entonces podremos averiguar si lo hemos entendido correctamente. Y sólo una vez que sepamos que estamos construyendo la lógica de dominio correcta, es cuando debemos pasar a construir una capa de persistencia y web alrededor de ella.

La fuerza motriz de una arquitectura centrada en la base de datos es el uso de ORM. A diario los uso, concretamente EF, pero si combinamos un ORM con una arquitectura de capas, nos sentiremos tentados a mezclar reglas de negocio con aspectos de persistencia.



¿Qué tienen de malo los layers?

Promueve un diseño basado en DATOS (3/3)



¿Qué tienen de malo los layers?

Es propenso a los ATAJOS (1/3)

En una arquitectura convencional de capas, la única regla global es que, a partir de una determinada capa, sólo podemos acceder a componentes de la misma capa o de una capa inferior.

Nuestro equipo de desarrollo puede acordar muchas otras reglas, e incluso algunas vendrán impuestas por la herramienta que usemos, pero la arquitectura de capas por si misma no nos impone ninguna regla más que la indicada.

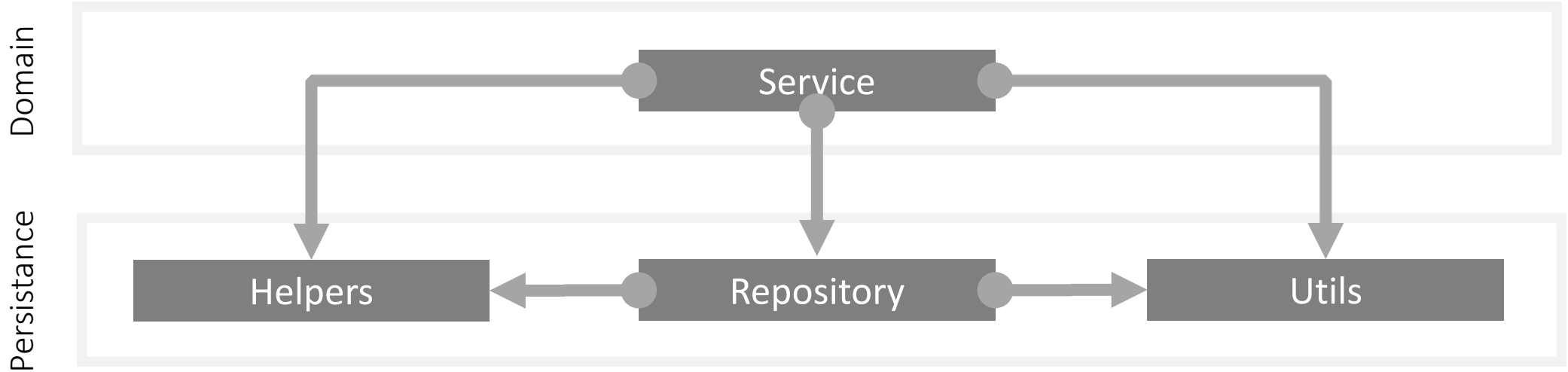
Por tanto, si necesitamos acceder a un determinado componente de una capa superior, podemos empujar el componente hacia a bajo y podremos acceder a el. Problema resuelto. Hacer esto una vez puede estar bien. Pero al hacerlo una vez se deja la puerta abierta para hacerlo una segunda vez. Y si a alguien se le permitió una vez, yo también podré hacerlo, ¿verdad?. No nos tomemos estos atajos a la ligera. Ya que, si existe esta opción para hacerlo, alguien lo seguirá haciendo, máxime cuando estamos en una fecha límite de entrega de un producto.

¿Qué tienen de malo los layers?

Es propenso a los ATAJOS (2/3)

Suele pasar que si algo ya se a hecho antes, el umbral para que alguien lo haga de nuevo baja drásticamente. Este es un efecto psicológico denominado “Teoría de las ventanas rotas” definido por George L. Kelling y Catherine Coles.

A lo largo de la vida de desarrollo y mantenimiento de un producto, la capa de persistencia puede terminar de la siguiente forma:



¿Qué tienen de malo los layers?

Es propenso a los ATAJOS (3/3)

La capa de persistencia, en términos más genéricos, la más baja, engordará a medida que empujemos los componentes hacia abajo.

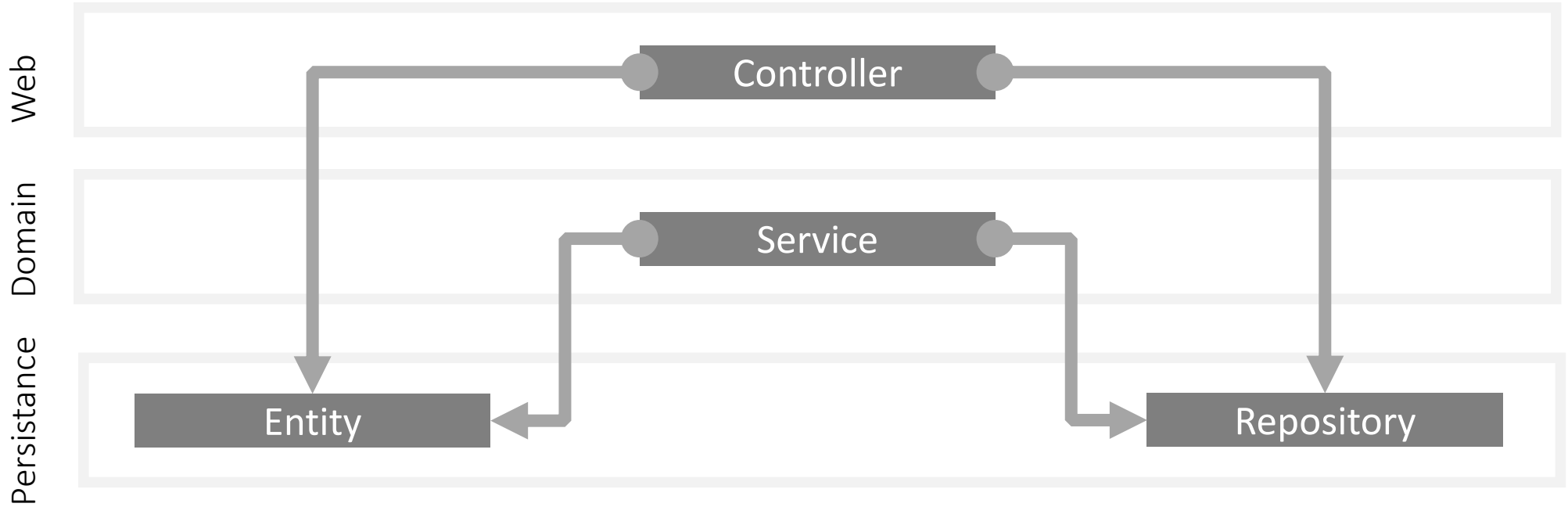
Los candidatos perfectos para esto son los componentes destinados a utilidades, ya que siempre parece que no pertenecer a ninguna capa específica.

Por tanto, si queremos deshabilitar el “modo abreviado” para nuestra arquitectura, las capas no son la mejor opción, al menos no sin forzar algún tipo de reglas adicionales en la arquitectura. Y por “forzar” no me refiero a que un desarrollador senior se dedique a revisar el código, si no a reglas que impidan crear una build cuando se estas reglas se están rompiendo.

¿Qué tienen de malo los layers?

Se hace difícil PROBAR (1/3)

A veces se omiten capas. Accedemos a la capa de persistencia directamente desde la web, ya que cómo sólo estamos manipulando un campo de una entidad, no es necesario molestarnos en pasar por la capa de Dominio, ¿verdad?.



¿Qué tienen de malo los layers?

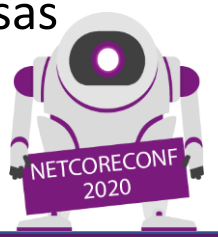
Se hace difícil PROBAR (2/3)

Saltarse la capa de dominio tiende a dispersar la lógica de dominio a través de todo el código.

Una vez más, esto si se consiente una vez por una circunstancia especial y no es refactorizado inmediatamente, no tiene la mayor importancia. El problema es que se realice por desidia, el primer paso ya está dado y, por tanto, el resto de los desarrolladores tendrá la escusa perfecta para “seguir” la norma.

¿Qué pasa si el caso de uso de “ese único campo” se expande en un futuro? Lo más probable es que terminemos añadiendo lógica de dominio en la web y acabemos mezclando responsabilidades extendiendo la lógica de dominio esencial por toda la aplicación.

¿Qué pasa con los test?, qué tendremos que hacer mock de la capa de dominio además de saltarnos cosas de la capa de persistencia. Esto añade complejidad a los test unitarios. Una configuración de test complejo es el primer paso para la ausencia de test, por que no tenemos tiempo para implementarlos.



¿Qué tienen de malo los layers?

Se hace difícil PROBAR (3/3)

A medida que el componente web crece con el tiempo, acumulará muchas dependencias con diferentes componentes de persistencia y por tanto añadirá complejidad a las pruebas.

Finalmente, en algún momento, nos llevará más tiempo entender el funcionamiento del caso de uso y por tanto las pruebas no tendrán ninguna validez, ya que es fácil hacer mock ciertas dependencias obligatorias para dar por buena una prueba.

Y lo que es más grave, es que, ante esta situación muchos desarrolladores, manipularan los test simplemente para que pasen y no de error.

¿Qué tienen de malo los layers?

Se ocultan CASOS DE USO (1/3)

Como desarrolladores, nos gusta crear código que implemente nuevos y brillantes casos de uso. Pero normalmente pasamos mucho más tiempo cambiado código existente que creando código nuevo. Esto ocurre tanto en los temidos proyectos legacy como en los proyectos greenfield tras haber implementado algunos casos de uso.

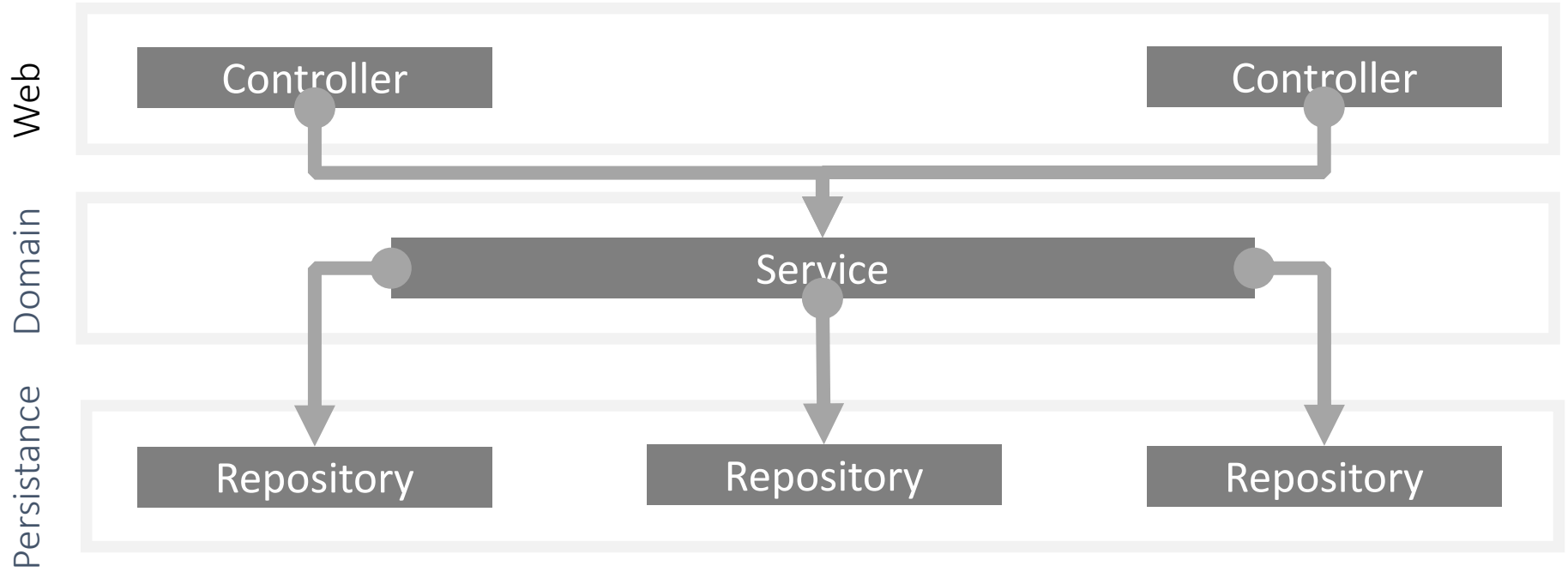
Dado que estamos buscando muchas veces el lugar correcto para añadir o cambiar una funcionalidad, nuestra arquitectura debería ayudarnos a navegar rápidamente por el código base. Pero ¿cómo se mantiene una arquitectura en capas en este sentido?

La lógica del dominio puede dispersarse a través de otras capas. Puede existir en la capa web si nos saltamos la lógica de dominio para un caso de uso fácil. Puede que exista en la capa de persistencia si hemos empujado un componente para que pueda acceder a el tanto desde el dominio como desde la persistencia. Hacer esto complica encontrar el lugar adecuado para añadir nuevas funcionalidades.

¿Qué tienen de malo los layers?

Se ocultan CASOS DE USO (2/3)

Pero, es más, una arquitectura de capas que no impone reglas sobre lo “grande” que pueden ser los servicios de dominio. Con el tiempo, esto a menudo conduce a servicios muy amplios que sirven para muchos casos de uso, tal y como se puede observar en el siguiente dibujo:



¿Qué tienen de malo los layers?

Se ocultan CASOS DE USO (3/3)

Un servicio amplio tiene muchas dependencias en la capa de persistencia y muchos componentes en la capa web dependen de ella. Esto no sólo hace que el servicio sea difícil de probar, si no, que también dificulta encontrar el servicio responsable del caso de uso en el que queremos trabajar.

Todo sería más fácil si tuviéramos servicios de dominio más pequeños y altamente especializados que sirvan para cada caso de uso, en lugar de buscar el caso de uso en un macro servicio poco especializado. Por poner un ejemplo, es mucho más sencillo añadir funcionalidad nueva al registro de un usuario, en un servicio pequeño y altamente especializado llamado `RegisterUserService` que en uno llamado `UserService`, que a saber cuantas cosas puede llegar a realizar este macro servicio.

¿Qué tienen de malo los layers?

Es difícil TRABAJAR EN PARALELO (1/2)

Supongo que conocen la famosa conclusión:

"Adding manpower to a late software project makes it later" – The Mythical Man-Month: Essays on Software Engineering by Frederick P. Brooks, Jr., Addison-Wesley, 1995.

La dirección a parte de pedirnos que terminemos nuestro software en un tiempo y coste estimado, cuando nos desviamos, añadir más gente debería funcionar. Es un pensamiento lógico, pero si esto se apoya en una arquitectura que realmente lo permita. Y ciertamente, una arquitectura de capas no nos ayuda a que esto se cumpla.

Imagine que estamos añadiendo un nuevo caso de uso en la aplicación. Tenemos tres desarrolladores disponibles. Uno para la web, otro para el dominio y el tercero para la persistencia. Debería funcionar ¿verdad?.

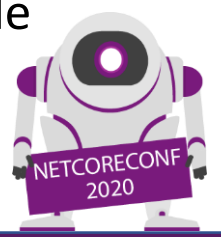
¿Qué tienen de malo los layers?

Es difícil TRABAJAR EN PARALELO (2/2)

Normalmente esto no funciona así con la arquitectura de capas, dado que todo se construye en base a la persistencia, la cual se debe desarrollar primero, luego la capa de dominio y finalmente la web. Por tanto, un desarrollador solo podrá trabajar en una característica al mismo tiempo.

Estaréis pensado que las interfaces serían nuestra salvación, ya que no hace falta esperar a la implementación real. Claro, esto es posible, si estamos trabajando con bases de datos, tal y como comentamos anteriormente y donde nuestra lógica de persistencia está mezclada con la lógica de dominio.

Y si tenemos servicios amplios, puede que trabajar en paralelo en diferentes características sea tarea imposible, ya que ese servicio podrá ser modificado por varios equipos y la consolidación del código se convierta en un infierno, además de provocar errores colaterales, aunque realicemos muchas pruebas de regresión para minimizar el riesgo.



Inversión de dependencias, acoplamiento y cohesión

Principio de responsabilidad única S (1/2)

La mayoría de los desarrolladores entienden este principio como “un componente debe hacer sólo una cosa, y hacerla bien”.

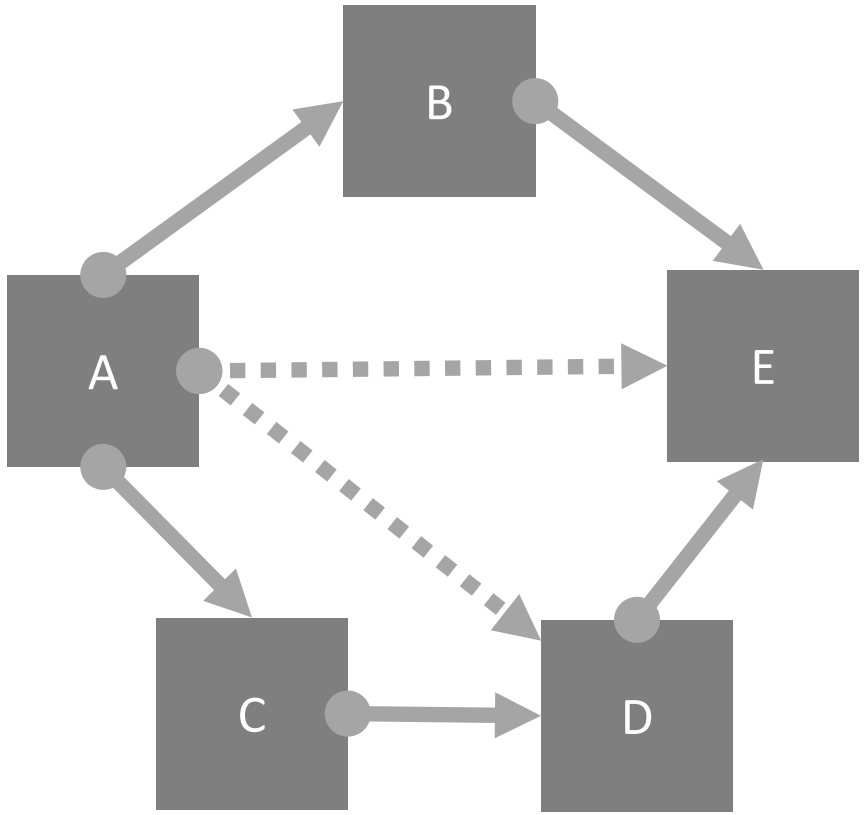
Pero en el ámbito de una arquitecta debe usarse la siguiente acepción: “un componente debe tener sólo una razón para cambiar”.

Si un componente tiene sólo una razón para cambiar, no tenemos que preocuparnos en absoluto por ese componente si cambiamos el software por cualquier otra razón, porque como bien sabemos, seguirá funcionando como se esperaba.

Por desgracia es muy fácil que por una razón para cambiar acabemos propagándolo por diversos componentes.

Inversión de dependencias, acoplamiento y cohesión

Principio de responsabilidad única S (2/2)



El componente A depende de muchos otros componentes (ya sea directa o transitoriamente) mientras que el componente E no tiene ninguna relación. El único motivo para modificar el componente E es cuando la funcionalidad de E debe cambiar por una nueva necesidad. Sin embargo, el componente A tendrá que cambiar ya que depende de ellos.

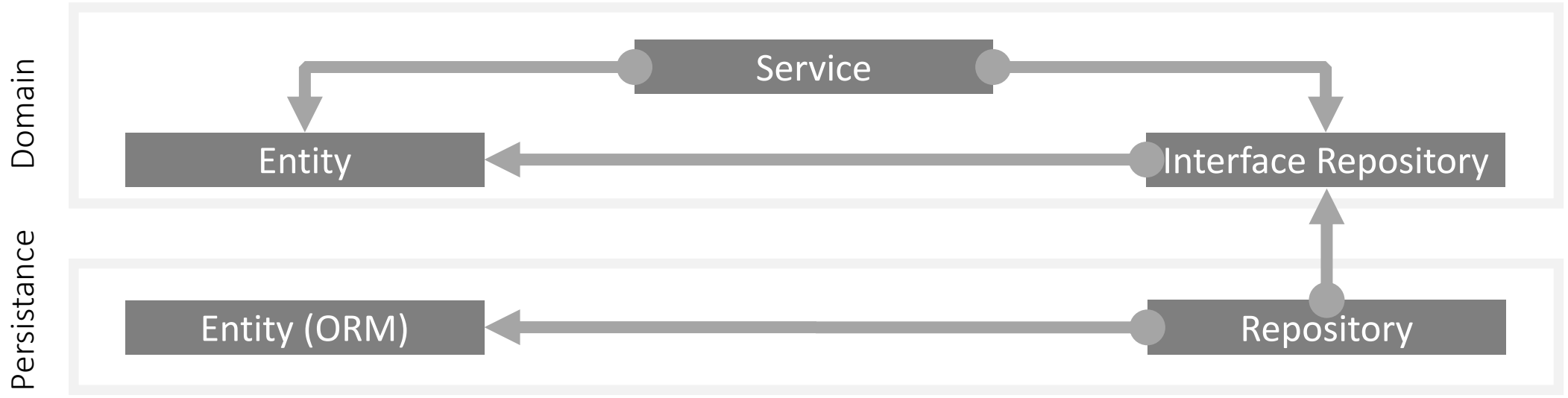
Muchas arquitecturas se vuelven más difíciles y caras de mantener por violar el SRP. Con el tiempo, un componente reúne más y más razones para cambiar. Tras arreglar ese componente por una necesidad, ese cambio causa que otros componentes fallen. Se producen los conocidos efectos secundarios y daños colaterales.

Inversión de dependencias, acoplamiento y cohesión

Principio de inversión de dependencia D

En una dependencia de capas si cambiamos la capa superior por el principio SRP deberíamos tener razones suficientes para cambiar las capas inferiores.

Afortunadamente tenemos el principio de inversión de dependencia, que deshace dicha dependencia con las capas.



Inversión de dependencias, acoplamiento y cohesión

¿Qué es ACOPLAMIENTO?

Es la propiedad que un módulo se ve obligado cambiar porqué otro lo hace.

Existen varias formas de acoplamiento entre módulos: por dependencia o por conocimiento.

Dos módulos están a desacoplados si los cambios en la implementación de uno no afectan a la implementación de otro.

Por tanto, lo deseable es que el acoplamiento sea lo más relajado posible.

Una de las consecuencias del Code Smell es la duplicidad de código por un fuerte acoplamiento.

Inversión de dependencias, acoplamiento y cohesión

¿Qué es COHESIÓN?

Es la propiedad por la que las piezas de funcionalidad de un módulo están relacionadas entre sí.

Un módulo que tiene una baja cohesión tiene muchas razones para cambiar. Es un módulo que sufre mucho cambio, el código que se comparte carece de la más mínima cohesión. Para que se entienda mejor, pensar el módulo UTILS, un lugar donde suelen acabar funcionalidades sin ninguna relación entre sí.

Por tanto, la baja cohesión es otra de las razones por las que existe Code Smell.

¿Por qué Clean Architecture?

¿Cómo me ayuda a construir SOFTWARE MANTENIBLE? (1/2)

Una responsabilidad clave de lo arquitectura es particionar el sistema.

Un objetivo principal de la arquitectura es reducir el acoplamiento en el sistema y mejorar la cohesión para favorecer su modificación.

Una ARQUITECTURA LIMPIA puede considerarse como aquella en la que la partición conduce a un bajo acoplamiento y una alta cohesión.

¿Por qué Clean Architecture?

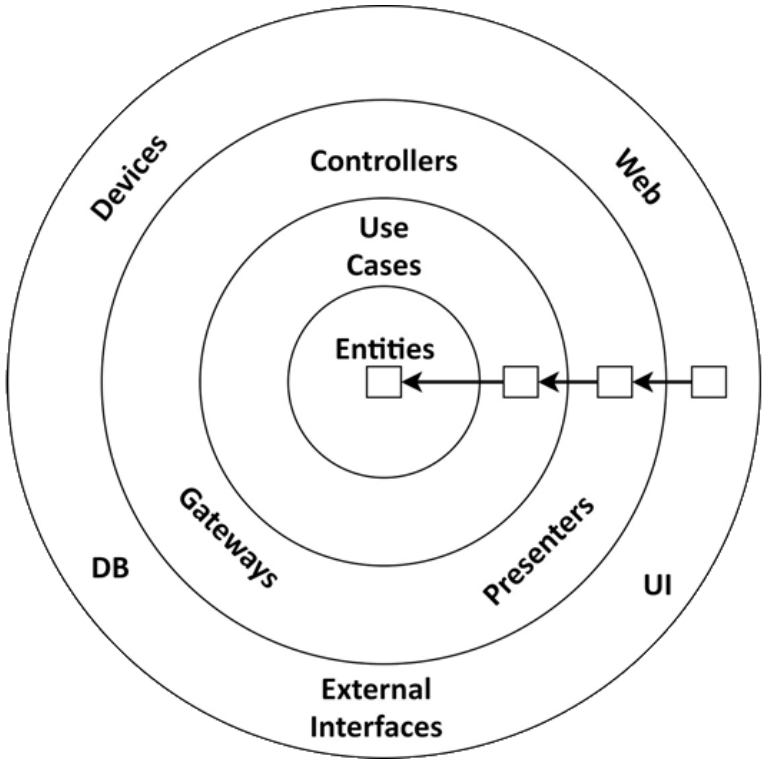
¿Cómo me ayuda a construir SOFTWARE MANTENIBLE? (1/2)

En resumen, lo llames como lo llames: clean, hexagonal u onion architecture, invirtiendo nuestras dependencias para que el código de dominio no tenga dependencias con el exterior, podremos desacoplar la lógica de dominio de todos los problemas que acarrea la persistencia o lo de la interface de usuario. Así reduciremos de razones para hacer cambios a lo largo de la base de código. Y menos razones para cambiar significa una mejor mantenibilidad.

El código de dominio es libre de ser modelado como mejor se ajuste a los problemas de negocio mientras que el código de persistencia y UI es libre de ser modelado como mejor se ajuste a los problemas de persistencia y UI.

¿Por qué Clean Architecture?

La propuesta: CLEAN ARCHITECTURE (1/2)



¿Por qué Clean Architecture?

La propuesta: CLEAN ARCHITECTURE (2/2)

Las capas de la arquitectura se envuelven entre sí con la intención que las dependencias apunten de fuera hacia dentro.

El núcleo de la arquitectura contiene las entidades de dominio, a las que acceden los casos de uso circundantes. Los casos de uso, anteriormente los hemos llamado servicios, pero son más pequeños para tener una sola responsabilidad (es decir, una sola razón para cambiar), evitando así el problema de los servicios grandes que hemos discutido anteriormente.

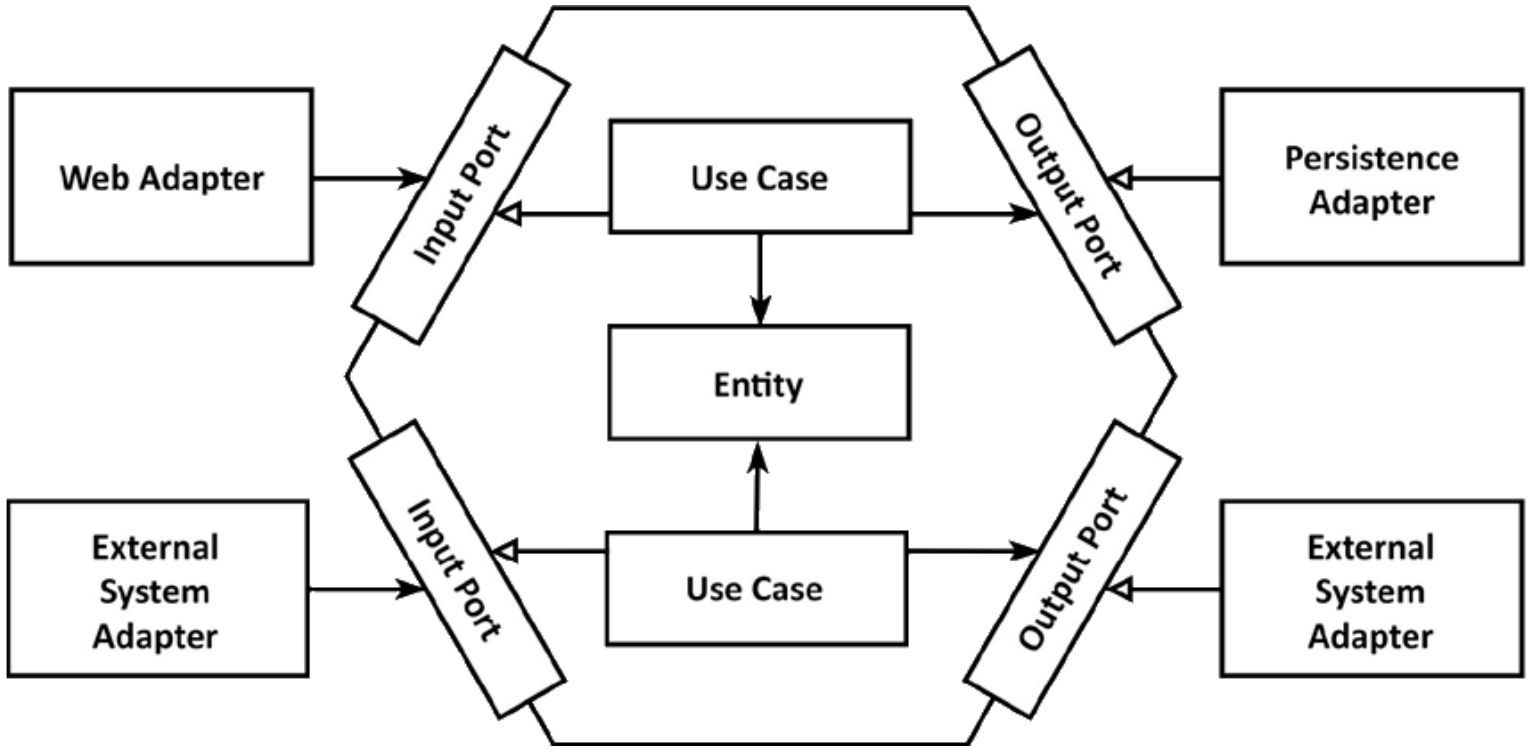
Alrededor de este núcleo, podemos encontrar todos los demás componentes de nuestra aplicación que soportan las reglas de negocio. Pueden ser persistencia, una interfaz de usuario, adaptadores a cualquier otro componente de terceros.

Debido a que el código del dominio no sabe nada acerca de qué persistencia o marco de UI se utiliza, no podrá contener ningún código específico de esos marcos y se centrará en las reglas de negocio. De este modo tendremos la libertad para modelar el código de dominio. Podríamos, por ejemplo, aplicar Domain Driven Design (DDD) puro. No tener que pensar en la persistencia o en problemas específicos de la interfaz de usuario hace que esto sea mucho más fácil.

Pero esto tiene un coste: la capa de dominio está completamente desacoplada de las capas externas, como la persistencia y la interfaz de usuario, por tanto, tenemos que mantener un modelo de las entidades de nuestra aplicación en cada una de las capas.

¿Por qué Clean Architecture?

El origen: HEXAGONAL ARCHITECTURE (1/2)



¿Por qué Clean Architecture?

El origen: HEXAGONAL ARCHITECTURE (2/2)

El término "arquitectura hexagonal“ existe desde hace bastante tiempo. Aplica los mismos principios que Robert C. Martin describió más tarde en términos más generales en su “arquitectura limpia”. Y dado que la propuesta de Robert C. Martin es algo abstracta, os emplazo a estudiar la "arquitectura hexagonal", que trata los principios de la arquitectura de una forma más concreta.

A esta arquitectura también se le conoce como “port-and-adapter” ya que el núcleo de la aplicación proporciona puertos específicos para que interactúe con cada adaptador.

El nombre hexagonal, no es por nada específico, podían haber usado otra figura, un octógono y llamarla “arquitectura octogonal”, según cuenta, se usó un hexágono para poder mostrar más casos “puerto-adaptador” que los que podría mostrarse con un cuadrado, nada más.

Dentro del hexágono, encontramos las entidades de dominio y los casos de uso, con las dependías apuntando hacia adentro, por tanto, se mantiene la propuesta e Robert C. Martin.

Y fuera de hexágono, encontramos varios adaptadores que interactúan con la aplicación. Puede ser un adaptador web que interactúa con un navegador web, otros que interactúan con sistemas externos e incluso un adaptador que interactúe con una base de datos.

Podrá observar que existen adaptadores en el lado izquierdo, los que manejan nuestra aplicación (llaman al núcleo) y los adaptadores del lado derecho, los que maneja el núcleo de la aplicación (son llamados por el núcleo): Input / Output. En esta arquitectura las interfaces son un factor clave.

Organizando el código

¿No sería bueno reconocer una ARQUITECTURA con solo mirar el código?

Rotundamente sí: a mi me gustaría llegar a medias de un proyecto y ser razonablemente autónomo, para molestar mínimamente a mis compañeros. Y si a esto le sumas una “aceptable” documentación, el impacto para el cliente es despreciable y nuestra salud mental como desarrolladores no se verá afectada.

En un proyecto greenfield, lo primero que debemos hacer es una estructura que nos permita construir unos sólidos cimientos, pero para ellos debemos mentalizarnos en mantener la estructura.

Y a lo largo del desarrollo proyecto, cuando las aguas se agiten, que nuestra organización del código no sólo sea una bonita fachada para un desorden de código no estructurado.

A continuación, mostraré diferentes formas de organizar el código.

Organizando el código

```
PayPal
  |- domain
    |- Account
    |- Activity
    |- AccountRepository
    |- AccountService
  |- persistance
    |- AccountRepositoryImpl
  |- web
    |- AccountController
```

Por CAPAS

¿Qué hace esto aquí si ya hemos visto lo malo que tienen las capas?

Por qué muchas veces no debemos hacer sobre-arquitectura y, a pesar de sus problemas, es más que suficiente este planteamiento.

Organizando el código

```
PayPal
  |- account
    |- Account
    |- AccountController
    |- AccountRepository
    |- AccountRepositoryImpl
    |- AccountService
```

Por CARACTERÍSTICAS

Cuando organizamos el código por características, la arquitectura subyacente tiene a no ser evidente. Se hace aun menos visible que el enfoque por Capas.

Cada nuevo grupo de características obtendrá un nuevo paquete de alto nivel y podemos reforzar los límites del paquete entre las características utilizando la visibilidad privada del paquete para las clases a las que no se debería acceder desde el exterior. Los límites de los paquetes, combinados con la visibilidad privada de los paquetes, nos permiten evitar dependencias no deseadas entre las características.

Hacer visible la funcionalidad de la aplicación en el código es lo que Robert C. Martin llama una "screaming architecture", porque nos grita su intención.

Organizando el código

```
PayPal
  |- account
    |- adapter
      |- in
        |- web
          |- AccountController
      |- out
        |- persistance
          |- AccountPersistanceAdapter
          |- AccountDataRepository
    |- domain
      |- Account
    |- application
        |- AccountService
        |- port
          |- in
            |- AccountUseCase
          |- out
            |- LoadAccountPort
            |- UpdateAccountStatePort
```

Por EXPRESIÓN (1/2)

Cada elemento de la arquitectura puede ser mapeado directamente a uno de los paquetes. El nivel más alto en este caso es `account`, indicando que este módulo implementa casos de uso que cubren `account`.

En el siguiente nivel tendremos:

- `domain`, que contiene nuestro modelo de dominio.
- `application`, que contiene la capa de servicio que envuelve al anterior modelo de dominio. Donde `AccountService` implementa la interface del puerto entrante, `AccountUseCase` y utiliza las interfaces del puerto saliente, `LoadAccount` y `UpdateAccount`, que se implementan mediante adaptarores web y de persistencia.

En resumen: el `adapter` contiene los adaptadores entandes que llaman a puertos entrantes de la capa de aplicación y los adaptadores salientes que proporcionan implementaciones para los puertos salientes de las capas de aplicación. En el ejemplo que presento, estamos construyendo una aplicación web muy sencilla con adaptadores web y de persistencia, cada uno de ellos con su propio sub-paquete.

¿Os resulta confuso?

Organizando el código

Por EXPRESIÓN (2/2)

Imaginar que tenemos una vista de alto nivel de nuestra arquitectura hexagonal dibujada en una pizarra de nuestra oficina y que estamos hablando con un compañero sobre la modificación de un cliente de una API de un tercero que estamos consumiendo. Mientras hablamos sobre esto, podemos apuntar en el dibujo cual será el adaptador saliente para que nos entendamos mejor. Y luego, cuando terminamos de hablar, nos sentamos en nuestro IDE y comenzamos a codificar inmediatamente, porqué el código de la API del que hemos hablado se encuentra en la zona adapter/out/<nombre-del-adaptador>.

Menos confuso, ¿no?.

Esta expresiva estructura de paquete promueve el pensamiento activo sobre la arquitectura. Tenemos muchos paquetes y tenemos que pensar en qué paquete poner el código en el que estamos trabajando actualmente.

Cuidado, no significa el tener tantos paquetes que todo tiene que ser público para permitir el acceso a través de los paquetes, unos serán privados (evita dependencias accidentales) y otros públicos (estos deberían hacer uso extensivo de las interfaces).

Otra ventaja de esta estructura de paquetes es que se mapea directamente a los conceptos de DDD. El paquete de alto nivel, `account`, en nuestro caso, es un contexto acotado que tiene puntos de entrada y salida dedicados (los puertos) para comunicarse con otros contextos acotados. Dentro del paquete de dominios, podemos construir cualquier modelo de dominio que queramos, utilizando todas las herramientas que nos proporciona DDD.

Se necesita disciplina para mantener esta estructura a lo largo de la vida de un proyecto de software y habrá casos en los que la estructura de paquete no encaje, y no vemos otra forma de ampliar la brecha entre la arquitectura y el código y crear un paquete que no refleje la arquitectura. No existe la perfección. Pero con una estructura de paquete expresiva, podemos al menos reducir la brecha entre el código y la arquitectura.

Organizando el código

El Rol de la INYECCIÓN DE DEPENDENCIAS

Disculpármele, pero la estructura de paquetes anteriormente descrita va más allá de la arquitectura limpia, pero es un requisito esencial: la capa de aplicación no debe tener dependencias entre los adaptadores de entrada y salida.

Para la parte web, es sencillo ya que el flujo de control apunta en la misma dependencia que el adaptador. Pero para los adaptadores salientes tendremos que usar el Principio de inversión de Dependencias.

La idea es introducir un componente neutral que tiene una dependencia de todas las capas. Este componente se encargará de instanciar la mayoría de las clases que componen nuestra arquitectura.

Organizando el código

¿Cómo me ayuda a construir SOFTWARE MANTENIBLE?

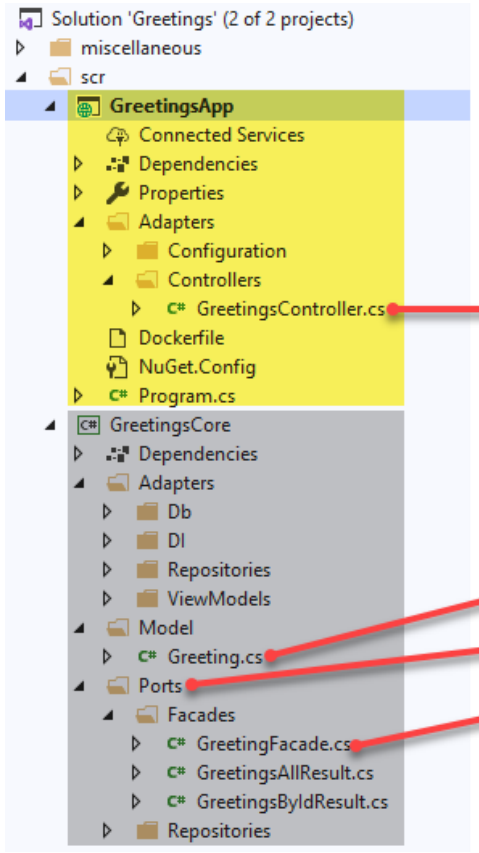
La estructura de paquetes para la arquitectura hexagonal intenta llevar la estructura del código real tan cerca de la arquitectura de destino como sea posible.

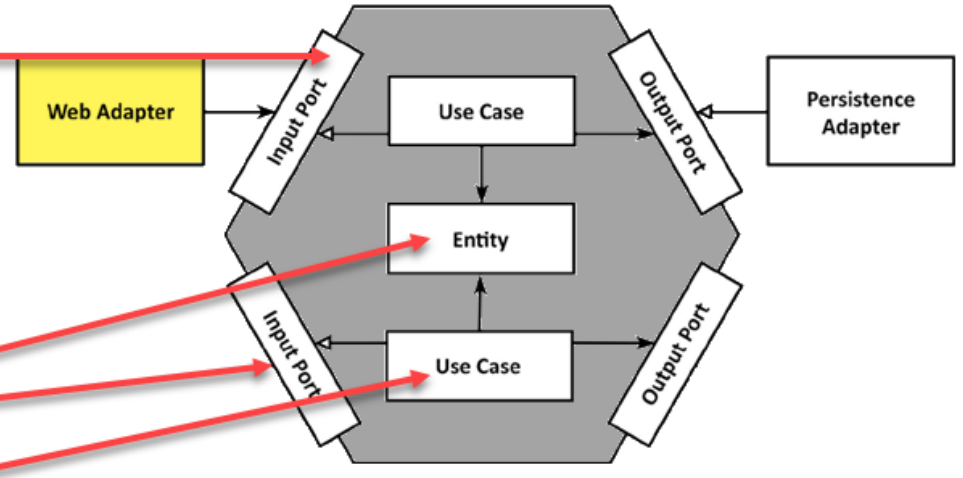
Encontrar un elemento de la arquitectura en el código es ahora una cuestión de navegar por la estructura de paquetes junto con los nombres de ciertos elementos del diagrama de la arquitectura, nos ayudará en la comunicación, desarrollo y mantenimiento del software.

Demo

Demo

¿Quién es quien en la DEMO?





Demo

Para ESTUDIAR

- https://docs.microsoft.com/es-es/dotnet/architecture/modern-web-apps-azure/
- https://paulovich.net/
- https://github.com/JasonGT/CleanArchitecture

Testing

Y no puede ser CLEAN ARCHITECTURE sin TESTING

Es algo que ya conocen, aplican y es un parte más del ADN del desarrollo diario, ¿verdad?.

Aquí solamente quiero mencionar que, gracias a este tipo de arquitectura, podremos hacer:

- Testing en la entidad de dominio con pruebas unitarias.
- Testing de casos de uso con pruebas unitarias.
- Testing del adaptador web con pruebas de integración.
- Testing del adaptador de persistencia con pruebas de integración.
- Testing de los principales flujos con pruebas de sistema.

Testing

¿Cuántas prueba son SUFICIENTES?

Permitirme dar un pequeño consejo, la métrica de cobertura es una muy mala métrica para medir el éxito de las pruebas. Cualquier objetivo que no sea el 100% no nos asegura que todos los objetivos de las pruebas se cumplan.

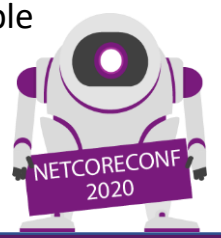
Soy más subjetivo, sugiero medir el éxito de las pruebas, en la medida que nos sintamos con la confianza suficiente para desplegar nuestro software.

Sí, os propongo un salto de fe las primeras veces que liberamos software; si alguien nos culpa por no haber detectado ese caso que falla en producción, nada más sencillo como añadirlo y en la próxima iteración, habremos cubierto más código, así sucesivamente. Y así tendremos cada vez más confianza en que nuestro software es cada vez más fiable.

Una buena estrategia es: mientras implementas una entidad, caso de uso, adaptador, flujos, crea un test unitario/integración/sistema para cada caso correspondiente.

Y como consejos finales:

- Por favor, si cambia un caso de uso (por una mala arquitectura) y no se entiende que es lo que hacía el test, no os dediquéis a arreglarlo para que pasen los test y listo esta mala praxis es peor que haber quitado el test, otros compañeros o mi yo del futuro en tendrá que esto esta cubierto. Un simple error será el detonante para n errores.
- Y si a la hora de arreglar un test por un nuevo campo, nos tiramos varias horas, es probable que estemos haciendo cosas mal en nuestra arquitectura y estos sean muy vulnerables a cambios estructurales de código. Los test pierden valor si se modifican en cada refactorización.



Testing

¿Cómo me ayuda a construir SOFTWARE MANTENIBLE?

La arquitectura limpia, nos ayuda a definir una estrategia clara que cubre la lógica de dominio central con pruebas unitarias y a los adaptadores con pruebas de integración.

Los puertos de entrada y salida proporcionan un punto de mocking muy visible para las pruebas. Para cada puerto, podemos decir hacer mocking o realizar implementaciones reales: si los puertos son pequeños hacer mocking puede ser suficiente, si no, lo que recomiendo son implementaciones reales.

Si hacer mocking se convierte en una carga muy pesada o no sabemos que es lo mínimamente necesario para hacer la prueba, es una señal de advertencia. En este sentido, las pruebas funcionarán como un Canary Test, nos advertirá de los defectos de la arquitectura y nos hará reaccionar para tomar un camino donde el software sea más mantenible.

Mapeo entre los límites de la arquitectura

El omnipresente MAPEO

Supongo que os reconoceréis en esta discusión ficticia.

Desarrollador Pro-Mapeo

Si no mapeamos entre capas, tenemos que usar el mismo modelo en ambas capas, lo que significa que las capas estarán intrínsecamente acopladas.

Desarrollador Contra-Mapeo

Si hacemos mapeo entre capas, produciremos un montó de código basura, lo cual es una exageración; solo hacemos CRUD y es el mismo a través de todas las capas.

Desde mi punto de vista ambos tienen razón, existen estrategias con pro-contra.

Mapeo entre los límites de la arquitectura

Estrategias de MAPEO

- Estrategia de “no” mapeo:
  Pros: si todas las capas necesitan lo mismo, no es necesario otra estrategia.
  Contras: los más probable es que violes el PSR, los más probable es que estemos haciendo viajar mucha información innecesaria.
- Estrategia “único sentido” del mapeo.
  Pros: es sencillo, envías datos de una capa a otra de forma unidireccional.
  Contras: en la mayoría de software conceptualizar esta esta estrategia es la más compleja de todas.
- Estrategia “doble sentido” del mapeo.
  Pros: cada capa tiene su modelo y puede modificar su modelo, es fácil de mantener. No afecta a otras capas. Es muy simple.
  Contras: genera mucho código basura, depurar es mucho más molesto (cuidado con los nombres), vulnerable a cambios entre capas.
- Estrategia “todo” mapeado.
  Pros: es muy fácil de implementar y mantener ya que cada mapeo será prácticamente para cada caso de uso.
  Contras: el que más código basura genera y no es buena idea usarlo entre capas de aplicación, más bien para uso interno de capas.

Desafortunadamente no disponemos de ninguna bala de plata. En muchos proyectos, elegir un mal modelo de mapeo ralentiza innecesariamente el desarrollo. Y evidentemente en los proyectos ~~pueden~~ deben mezclarse las estrategias.

Mapeo entre los límites de la arquitectura

¿Cómo me ayuda a construir SOFTWARE MANTENIBLE?

Los puertos de entrada y salida de nuestra aplicación actúan como guardianes entre las capas, definen cómo las capas se comunican entre sí y por lo tanto cómo mapeamos entre las capas.

Podemos elegir diferentes estrategias de mapeo para diferentes casos de uso, e incluso evolucionarlos a lo largo del tiempo sin afectar a otros casos de uso, seleccionando así la mejor estrategia para una determinada situación en un momento determinado.

La elección de estrategias de mapeo es ciertamente difícil, usar la misma estrategia de mapeo para todas las situaciones no tiene por que ser un error, siempre y cuando todo el equipo tenga conocimiento de las pautas para hacer los mapeos, de este modo se establece una base y el software se hace más fácil de mantener.

Cumplir con los límites de la arquitectura

Tendencia a la EROSIÓN (1/2)

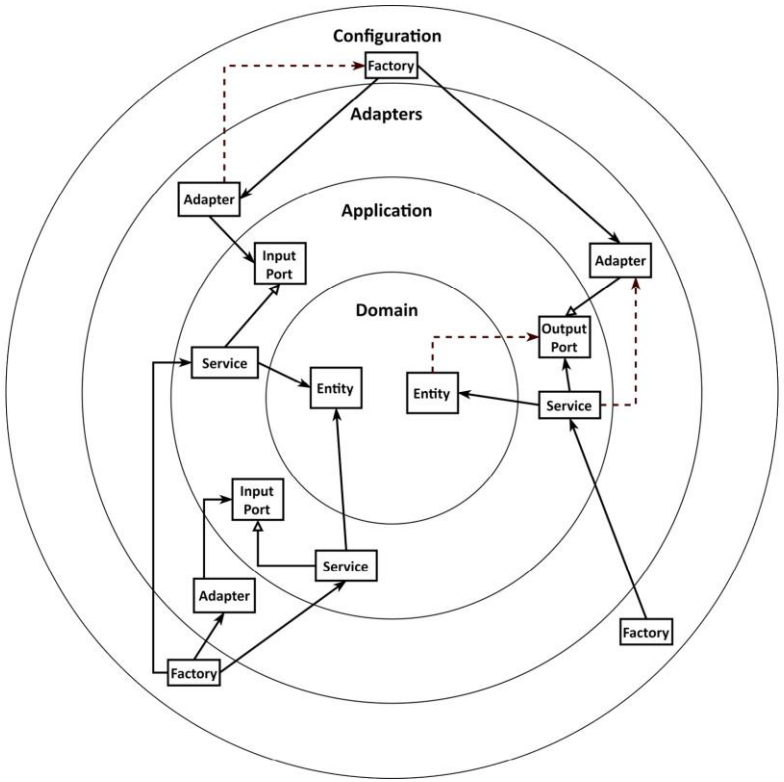
Tras hablar largo y tendido de arquitectura, dónde y como poner el código, en todo proyecto la arquitectura tiende a degradarse con el tiempo. Los límites entre capas se debilitan, el código se vuelve más difícil de probar y generalmente necesitamos más y más tiempo para implementar nuevas características.

Os propongo algunas medidas para reforzar esos límites y luchar contra la erosión.

Pero antes de que hables de límites con el equipo de desarrollo debes saber que significado vas a dar a “hacer cumplir un límite”.

Sobre la siguiente figura:

Cumplir con los límites de la arquitectura



Tendencia a la EROSIÓN (2/2)

Hacer cumplir los límites de la arquitectura significa hacer cumplir que las dependencias apunten en la dirección correcta. Las flechas punteadas marcan las dependencias que no están permitidas según nuestra arquitectura

En este diagrama, los límites de nuestra arquitectura son bastante claros.

De acuerdo con la regla de dependencia, las dependencias que cruzan tales límites de capa siempre deben apuntar hacia adentro.

Cumplir con los límites de la arquitectura

Algunas técnicas para luchar contra la EROSIÓN

Visibilidad y Modificadores:
Por especificaciones del lenguaje C# dispone de `Public`, `Private`, `Protected`, `Internal`, … Estas directivas van más allá de la erosión, podríamos relacionarlas hasta con la seguridad de la aplicación.

Chequeo Post-Compilado:
Disponemos de herramientas que chequean en tiempo de ejecución, por supuesto que debe ponerse en los Test durante el CI, que las dependencias apuntan en la dirección correcta. Como: ArchUnitNET.

Construir artefactos:
Desde que podemos usar NuGet no tenemos escusa en encapsular y consolidar. Desde este planteamiento lo que estamos haciendo es reforzar las dependencias y por tanto los límites de las capas de nuestra arquitectura.

Cumplir con los límites de la arquitectura

¿Cómo me ayuda a construir SOFTWARE MANTENIBLE?

La arquitectura de software consiste básicamente en gestionar las dependencias entre los elementos de la arquitectura. Si las dependencias se convierten en una bola de barro, la arquitectura se convertirá en una gran bola de barro.

Por lo tanto, para poder preservar la arquitectura a lo largo del tiempo, necesitamos asegurarnos continuamente de que las dependencias apunten correctamente.

Cuando creamos código nuevo o cuando refactorizamos, debemos tener en cuenta la estructura y visibilidad para evitar dependencias indeseadas.

Cuando consideremos que una arquitectura es lo suficientemente estable, es cuando podremos extraer elementos e ir un paso más allá con nuestra arquitectura.

Coger atajos conscientemente

Para evitar atajos debemos ser capaces de IDENTIFICARLOS

El desarrollo de software tiene una característica de doble filo que yo he usado y sugerido a veces.

Cuando un dead-line no depende de ti, ni es un capricho del cliente, si no de una decesión administrativa: léase gubernamental como la GDPR o la directiva PSD2. Podemos ser laxos y tomar atajos, ya que el software por muy bien que quieras hacerlo, no es nuestro, es del cliente y es valor tangible en su negocio.

En algunas ocasiones es más económico (conscientemente) tomar un atajo primero y arreglarlo después (o nunca).

Por eso vamos a comentar unos aspectos a tener en cuenta:

Coger atajos conscientemente

ASPECTOS a tener en cuenta (1/3)

¿Por qué los accesos directos son como ventanas rotas?

Tan pronto como algo se ve desgastado, dañado, [insertar el adjetivo negativo aquí], o en general no atendido, el cerebro humano cree que es correcto desgastarlo, dañarlo, o [insertar el adjetivo negativo aquí] más.

Recomiendo la lectura del siguiente artículo: https://www.theatlantic.com/ideastour/archive/windows.html

La responsabilidad de comenzar a limpiar

Estamos sujetos inconscientemente a la psicología de las “ventanas rotas”. Por eso es importante comenzar con limpieza y poco atajo, así estaremos sujetos a la mejor deuda técnica posible.

En los proyectos largos y costosos, mantener a raya las “ventanas rotas” es una responsabilidad hacia nosotros y nuestros compañeros. Es indudable que cuando heredamos algo, el umbral para crear ventanas rotas baja.

Si en ocasiones somos pragmáticos y por ciertas razones debemos atajar, deberíamos documentar dichos atajos: https://github.com/joelparkerhenderson/architecture_decision_record

Coger atajos conscientemente

ASPECTOS a tener en cuenta (2/3)

Compartir modelos entre casos de uso

Si dos casos de uso comparten mismo modelo quiere decir que “tienen una razón para cambiar” y cumplimos con PSR. Es decir, ambos casos de uso están vinculados funcionalmente. En caso contrario no deberían compartir el mismo modelo.

Uso de entidades de dominio como modelos de entrada y salida

El peligro de este caso no esta en el inicio de un desarrollo, su caso de uso principal será crear o modificar. Si no, durante su evolución terminan por tener lógica compleja. En un entorno Agil es donde suele suceder mucho este caso.

Saltarse los puertos de entrada

Los puertos de salida siempre son necesarios para invertir la dependencia entre capa de aplicación y adaptadores de salida, pero los de entrada no. Podremos dejar que los apeadores entrantes accedan directamente a los servicios de la aplicación. Esto es un problema muy grande, debemos conocer con exactitud la aplicación para implementar un nuevo caso de uso. Sin embargo, si los mantenemos de un solo vistazo podremos identificar los puntos de entrada y facilitar la vida a los desarrolladores a la hora de orientarse en el código.

Otra razón para mantenerlos es que refuerzan la arquitectura y no exponemos por accidente algo que no debería ser accesible.

Coger atajos conscientemente

ASPECTOS a tener en cuenta (3/3)

Si una aplicación es pequeña o sólo tiene un único adaptador entrante para que podamos captar todo el flujo de control sin la ayuda de los puertos entrantes, podríamos querer prescindir de los puertos entrantes.

Sin embargo, ¿cuántas veces podemos decir que una aplicación se quedará pequeña o que sólo tendrá un único adaptador entrante a lo largo de toda su vida?

Saltarse los servicios de la aplicación

A parte de querer saltarte los puertos de entrada, también para ciertos casos de uso querrás saltarte la capa de ampliación en su totalidad.

Sin un servicio de aplicación, no tendremos una ubicación para la lógica de dominio.

Es tentador hacer esto para casos simples de CRUD, ya que solamente se trata de un reenvío de información, podemos dejar que sea el adaptador de persistencia quien lo use directamente. ¿Verdad?

Sin embargo, esto hace que “desaparezcan” los casos de uso, descentralice la lógica de dominio y sea complejo de mantener.

Tan pronto como el software crezca debería hacer las cosas bien y no saltarse la capa de servicio.

Coger atajos conscientemente

¿Cómo me ayuda a construir SOFTWARE MANTENIBLE?

Como todo en la vida, a veces los atajos tienen sentido desde un punto de vista. En el software o bien es el económico y o bien el tiempo, si no ambos.

Es tentador introducir atajo para casos simples de uso, tipo CRUD, ya que más bien parece sobre-arquitectura, y los atajos no se entienden como tal o no se ven como tal. Debemos ser pragmáticos.

Sin embargo, os emplazo a ser más estrictos y hacer un poco más de trabajo, si no estáis seguros de que la aplicación siempre será CRUD.

En cualquier caso, la arquitectura se debe documentar junto a las decisiones por las cuales elegimos un atajo para que nosotros o nuestros herederos podamos o puedan reevaluar las decisiones en el futuro.

Decidir un estilo de arquitectura

CONCLUSIONES (1/3)

Os he presentado un enfoque para construir una aplicación bajo un estilo de Clean Architecture usando una definición Hexagonal, desde donde habéis visto la organización del código hasta el razonamiento para tomar atajos. Sin embargo, muchas cosas de esta presentación aun podéis aplicarlas en arquitectura de capas convencional y otras solo pueden ser implementadas en DDD. Y espero que con vuestro sentido crítico ni siquiera estéis de acuerdo con cosas que he planteado por vuestra propia experiencia.

Debería se obvio que la característica principal de esta arquitectura libre de desviaciones, sin preocuparnos de de la persistencia o dependencias externas es el dominio. El dominio se mantendrá libre de influencias externas.

El dominio es el Rey.

Por eso este tipo de arquitectura es una simbiosis ideal para el DDD, obviamente en DDD el dominio dirige el desarrollo. Y me atrevería a razonar que, sin preocuparnos de las otras capas, podemos diseñar un mejor dominio e incluso que esta arquitectura es un facilitador del DDD.

Por tanto, si el dominio no es lo más importante de tu aplicación, probablemente no necesites este tipo de arquitectura.

Decidir un estilo de arquitectura

CONCLUSIONES (2/3)

Todos tenemos hábitos y estos automatizan ciertas decisiones por nosotros, para no dedicarles más tiempo del necesario, lo más probable es que use un estilo de capas.

No estoy diciéndoos que los hábitos generen necesariamente malas decisiones. Los hábitos igual de buenos para ayudarnos a tomar una decisión correcta como incorrecta. Todos nos sentimos cómodos cuando no salimos de nuestra zona de confort, si en el pasado lo hemos hecho así, ¿para qué o por qué deberíamos cambiar algo?.

La experiencia es la Reina.

Por tanto, la única manera de tomar una decisión sobre un estilo es teniendo experiencia en diferentes estilos de arquitectura. Si no está seguro de aplicar Clean Architecture en alguna de sus formas hexagonal, onion, etc. Crea un pequeño módulo de una aplicación que estés construyendo actualmente. Acostúmbrate a os conceptos y acomódate.

Al final la experiencia guiará la próxima toma de decisión de una arquitectura.

Decidir un estilo de arquitectura

CONCLUSIONES (3/3)

Lo siento no os puedo proporcionar un check-list que como resultado nos diga: usa esta u otra arquitectura.

Como consultor que lleva mucho tiempo en este mundillo, mi respuesta siempre es depende:

- Depende del tipo de software a construir.
- Dependen del papel que jugará el dominio en la aplicación.
- Depende de la experiencia del equipo profesional.
- Y finalmente, depende de lo cómodo que se sientan con esa decisión.

No pretendo hacer proselitismo, solamente aportar ideas y herramientas para usarla adecuadamente.

si sólo tienes un martillo, todo parece un clavo

The Psychology of Science, Abraham Maslow, 1966.

Muchas Gracias

Puedes encontrarme buscando por jmfloreszazo en

Sponsors

Microsoft

avanade

plain concepts

devs dna

everis
an NTT DATA Company

encamina
PIENSA EN COLORES

KABEL

SolidQ

intelequia

東京 TOKIOTA

birchman
VISEO GROUP

GIN MG
CASA FUNDADA
EN 1835

monday

Más información:

info@netcoreconf.com
@Netcoreconf

Visítanos en:
netcoreconf.com