13 Octubre 2017Actualizado 31 Enero 2018, 18:30 **RUBENFA**

Estoy seguro de que si te dedicas a programar, conoces a Robert "Uncle" Martin. Su libro *Clean Code* es uno de los más recomendados en la lista de libros que todo desarrollador debería leer. Martin, con sus cosas buenas y malas, es uno de los desarrolladores más influyentes del panorama ingenieril. Fuerte defensor de TDD, de la cobertura de tests y otras buenas prácticas, y además cuenta con muchas personas que siguen sus enseñanzas a rajatabla.

Recientemente, Bob Martin, ha publicado un nuevo libro llamado Clean Architecture. ¿Pero qué se entiende por arquitectura limpia?

Clean Code

Como comentaba antes, *Clean Code* es un libro muy recomendable, que desgrana algunas ideas importantes para poder escribir código limpio.

El código limpio es aquel código que **está estructurado de forma compresible, que es claro en sus intenciones, fácil de leer, que es fácilmente mantenible y que está testeado**. En el libro se van dando algunas ideas para conseguir escribir código limpio, hablando de principios SOLID, de la importancia de dar nombres a variables y clases etc. En GenbetaDev ya hemos hablado del libro y de sus ideas en alguna ocasión

Principios de una arquitectura limpia

Aunque seamos capaces de escribir código limpio, podemos encontrarnos que al crecer nuestro sistema, la arquitectura del mismo sea un lastre. Y es que **no es lo mismo escribir código limpio para un proyecto sencillo, que para un proyecto complejo compuesto de varios componentes obligados a cooperar**. A veces las arquitecturas son demasiado complejas, nos obligan a repetir código, o nos hacen tener demasiadas dependencias entre componentes, causándonos muchos problemas.

> Los conceptos de cohesión y acoplamiento, también pueden aplicarse a nivel de arquitectura.

Si utilizáis programación orientada a objetos, seguro que conocéis los conceptos de cohesión y acoplamiento. Esos conceptos también pueden aplicarse de forma parecida a los componentes de un sistema, ya sean *dlls* o archivos *jar*, estos tienen que cooperar unos con otros. Y la manera en la que cooperen, pueden hacer un sistema fracasar. Pero si seguimos una serie de principios para controlar estas dos variables, nuestra arquitectura será más limpia y manejable.

Cohesión

- **The Reuse/Release Equivalence Principle**: que nos dice que **los componentes deben poder ser desplegados de forma independiente sin afectar a los demás**. Las clases, o código que van en ese componente, deben tener una relación, y por tanto deben poderse desplegar de forma conjunta.
- **The common closure principle**: se podría decir que hablamos del principio de responsabilidad única (SRP) aplicado a componentes. **La idea es agrupar clases que puedan cambiar por la misma razón en un solo componente**. Si tenemos que hacer un cambio, y hay que tocar varios componentes, esto supondrá tener que desplegarlos todos, en lugar de sólo uno.

intentar que sea porque necesita todas las clases que lo componen. Lo contrario nos obligará a trabajar más cuando nos toque hacer el despliegue. De esta manera será más fácil reutilizar componentes.

Conseguir cumplir estos tres principios a la vez es algo bastante difícil, por lo que a veces hay que aceptar compromisos. Por ejemplo es común sacrificar un poco la reusabilidad, para conseguir que los componentes sean fáciles de desplegar.

Acoplamiento

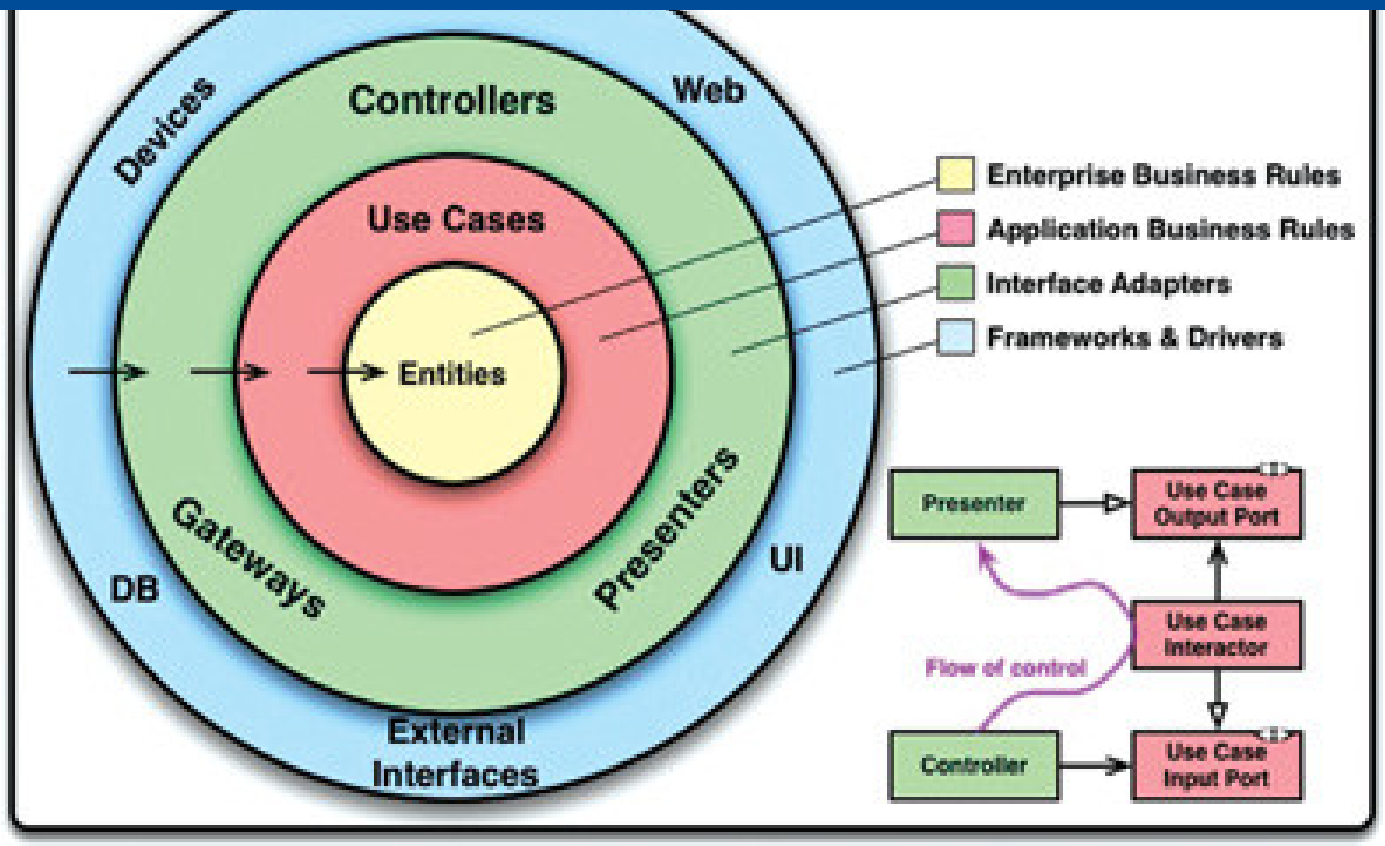
- **The Acyclic Dependencies Principle**: si trazamos líneas entre los componentes para representar las dependencias entre ellos, tenemos que intentar que no existan ciclos. Es decir, **que el cambio en un componente, no acabe desencadenando en la necesidad de hacer cambios en cadena en los demás componentes**, que obliguen a volver a modificar el componente inicial. Cuando eso sucede, es difícil conseguir una versión estable del sistema, ya que hay que hacer multitud de cambios en los distintos componentes hasta que todo vuelve a funcionar.
- **The stable dependencies Principle**: todo sistema tiende a cambiar y evolucionar, pero no todos los componentes cambian con la misma frecuencia, ni es igual de fácil modificarlos. Este principio nos dice que **un componente que cambia a menudo no debería depender de otro que es difícil modificar**, ya que entonces será también difícil de modificar.
- **The stable Abstractions Principle**: este principio nos dice que **si un componente de nuestro sistema va a cambiar poco ya que es difícil modificarlo, debe estar compuesto mayoritariamente por interfaces y clases abstractas**. De esta manera el componente será fácilmente extensible, y no afectará tanto al resto de la arquitectura.

Características de una arquitectura limpia

Además de cumplir los principios anteriormente descritos, una arquitectura limpia se caracteriza por:

- **Independiente de los frameworks**. Los frameworks deberían ser herramientas, y no obligarnos a actuar de una determinada manera debido a sus restricciones.
- **Testable**. Debemos poder probar nuestras reglas de negocio sin pensar en base de datos, interface gráfica u otros componentes no esenciales de nuestro sistema.
- **Independiente de la UI**. Si la UI cambia a menudo esto no puede afectar al resto de nuestro sistema, que tiene que ser independiente.
- **Independiente de la base de datos**. Deberíamos poder cambiar de Oracle, a SQL Server, a MongoDB, a Casandra o a cualquier otra base de datos sin que afectara demasiado a nuestro sistema.
- **Independiente de cualquier entidad externa**. No deberíamos saber nada de entidades externas, por lo que no deberemos depender de ellas.

Todas estas características, según Bob Martin, se agrupan en el siguiente gráfico:



Partes de una arquitectura limpia

Entidades

Las entidades son las que **incluyen las reglas de negocio críticas para el sistema**. Estas entidades pueden ser utilizadas por distintos componentes de la arquitectura, por lo que son independientes, y no deben cambiar a consecuencia de otros elementos externos.

Una entidad deberá englobar un concepto crítico para el negocio, y nosotros tendremos que separarlo lo más posible del resto de conceptos. Esa entidad recibirá los datos necesarios, y realizará operaciones sobre ellos para conseguir el objetivo deseado.

Casos de uso

En este caso nos encontramos con las **reglas de negocio aplicables a una aplicación concreta**. Estos casos de uso siguen un flujo para conseguir que las reglas definidas por las entidades se cumplan. Los casos de uso, solo definen como se comporta nuestro sistema, definiendo los datos de entrada necesarios, y cual será su salida. Los cambios en esta capa no deberían afectar a las entidades, al igual que los cambios en otras capas externas no deberían afectar a los casos de uso.

Es importante que no pensemos en como los datos que genera un caso de uso serán presentados al usuario. No deberemos pensar en HTML, o en SQL. Un caso de uso recibe datos estructurados y devuelve más datos estructurados

Los datos generados por los casos de uso y las entidades, tienen que transformarse en algo entendible por la siguiente capa que los va a utilizar y de eso se encarga esta capa. Pensando en MVC por ejemplo, los controladores y las vistas, pertenecerían a esta capa, y el modelo, serían los datos que se pasan entre los casos de uso y los controladores para luego poder presentar las vistas.

Lo mismo aplicaría para por ejemplo, presentar información a un servicio externo, ya que en esta capa definiríamos la manera en la que los datos de las capas internas se presenta al exterior.

Frameworks y drivers

En la capa más externa es, como dice Bob Martin, donde van los detalles. Y la base de datos es un detalle, nuestro framework web, es un detalle etc.

Fronteras o límites

Una frontera (o como dicen los aglosajones, *boundaries*) es una **separación que definimos en nuestra arquitectura para dividir componentes y definir dependencias**. Estas fronteras tenemos que decidir dónde ponerlas, y cuándo ponerlas. Esta decisión es importante ya que puede condicionar el buen desempeño del proyecto. Una mala decisión sobre los límites puede complicar el desarrollo de nuestra aplicación o su mantenimiento futuro.

> Una mala decisión sobre los límites entre componentes puede complicar el desarrollo de nuestra aplicación o su mantenimiento futuro

Por ejemplo, podemos sentirnos tentados de pensar que las reglas de negocio deben poder guardar información directamente en la base de datos. Como ya hemos visto antes, la base de datos es un detalle, así que esto deberíamos evitarlo. En ese punto deberíamos trazar una frontera. Nuestras reglas de negocio, se comunicarían siempre con una interface, sin saber nada sobre la base de datos. La base de datos en cambio, si sabrá cosas sobre las reglas de negocio, ya que tiene que transformar los datos en sentencias SQL que puedan almacenar la información.

Otra ventaja adicional de este enfoque, es que podemos retrasar ciertas decisiones. Podemos empezar a desarrollar todas nuestras reglas de negocio, sin tener en cuenta su persistencia, ya que esa parte se realiza a través de una interface. Primero podemos utilizar objetos en memoria, y según avancemos, ir añadiendo sistemas más sofisticados. Al final podremos elegir entre usar una base de datos relacional, NoSQL, o incluso guardar la información en archivos.

En definitiva, debemos pensar en nuestro sistema, como un sistema de plugins, de forma que los componentes estén aislados y podamos sustituir unos por otros sin demasiados problemas.

Las fronteras de una arquitectura limpia

En el esquema de arquitectura limpia que hemos visto anteriormente, podemos ver dónde se han trazado las fronteras o límites. Entre entidades y casos de uso, hay una frontera. Lo mismo con los adaptadores de [illegible] son importantes, porque añadirlas cuando no las

La separación en fronteras es importante, pero mucho más importante es la gestión que hagamos de las dependencias entre estas capas. Para ello siempre hay que seguir la regla de las dependencias.

La regla de las dependencias

Esta regla es muy importante, ya que sin ella, nuestra arquitectura no sería más que un bonito diagrama. **Las capas interiores de una arquitectura limpia, no deben saber nada de las capas exteriores**. Por ejemplo la capa de entidades, no puede saber de la existencia de los casos de uso, y los casos de uso no deben saber nada de la existencia de los adaptadores de interface. Así las dependencias están controladas y van siempre en un solo sentido.

Estructuras de datos simples

A la hora de traspasar una frontera, deberemos utilizar estructuras de datos simples, evitando utilizar conceptos como *DatabaseRows* o similares. Pensando en los casos de uso, estos deben recibir estructuras de datos como datos de entradas, y deben devolver estructuras de datos como salida. Como decía antes, no nos interesa que un caso de uso tenga conocimientos sobre HTML o SQL. Lo contrario nos lleva a una falta de independencia, con todo lo que eso conlleva (despliegue, actualización, tests etc.)

> Las capas interiores de una arquitectura limpia, no deben saber nada de las capas exteriores

En ocasiones al pasar datos a los casos de uso, podemos pensar que es buena idea utilizar las entidades como datos de entrada o salida. Al fin y al cabo comparten mucha información. Pero esto no deja de ser un error, ya que aunque al principio la información parezca similar, en el futuro los casos de uso y las entidades cambiarán de muy diferentes maneras, obligándonos a tratar con la dependencia que hemos creado.

Fronteras parciales

A veces, por motivos de organización y mantenimiento, nos interesa crear **fronteras parciales**. Este tipo de fronteras las tenemos que planificar de forma similar a una frontera real, pero en lugar de empaquetarla en un componente aislado, la dejamos que forme parte de otro componente. Así nos ahorramos parte del esfuerzo de crear un componente nuevo, que no estamos seguros de que vaya a necesitarse. Obtenemos algunas de sus ventajas, dejando todo preparado por si es necesario dar ese último paso.

Conclusión

Aunque puede que no sea la parte más importante de un proyecto de software, la arquitectura juega siempre un papel importante. Ignorar esta fase pude traernos muchos problemas en el futuro, por lo que nunca está de más prestarle un poco de atención. Cuando diseñamos una arquitectura hay que tener muchas cosas en cuenta, como la separación de componentes, las dependencias entre ellos y la manera en la que cruzaremos las fronteras entre los mismos.