**La arquitectura limpia**



En los últimos años hemos visto toda una serie de ideas con respecto a la arquitectura de los sistemas. Éstas incluyen:

* [Arquitectura hexagonal](http://alistair.cockburn.us/Hexagonal+architecture) (también conocido como puertos y adaptadores) por Alistair Cockburn y adoptada por Steve Freeman, y Nat Pryce en su libro maravilloso [de software orientado a objetos creciente](http://www.amazon.com/Growing-Object-Oriented-Software-Guided-Tests/dp/0321503627)
* [Arquitectura de cebolla](http://jeffreypalermo.com/blog/the-onion-architecture-part-1/) por Jeffrey Palermo
* [Gritando Arquitectura](https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html) de un blog de la mina el año pasado
* [ICD](http://www.amazon.com/Lean-Architecture-Agile-Software-Development/dp/0470684208/) de James Coplien, y Trygve Reenskaug.
* [BCE](http://www.amazon.com/Object-Oriented-Software-Engineering-Approach/dp/0201544350) por Ivar Jacobson de su libro *de objetos Ingeniería de Software Orientada: un enfoque de casos de uso Driven*

A pesar de todas estas arquitecturas varían algo en sus detalles, que son muy similares. Todos ellos tienen el mismo objetivo, que es la separación de las capas.Todos ellos lograr esta separación dividiendo el software en capas. Cada uno tiene al menos una capa de reglas de negocio, y otro para las interfaces.

Cada una de estas arquitecturas de producir sistemas que son:

1. Independiente de los marcos. La arquitectura no depende de la existencia de alguna biblioteca de software cargado función. Esto le permite utilizar herramientas tales como marcos, en lugar de tener que meter el sistema en sus limitaciones limitados.
2. Comprobable. Las reglas de negocio pueden ser sometidos a ensayo sin la interfaz de usuario, base de datos, servidor web, o cualquier otro elemento externo.
3. Independiente de la interfaz de usuario. La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario Web podría sustituirse por una interfaz de usuario de la consola, por ejemplo, sin cambiar las reglas de negocio.
4. Independiente de la base de datos. Puede intercambiar Oracle o SQL Server, por Mongo, BigTable, CouchDB, o alguna otra cosa. Sus reglas de negocio no están obligados a la base de datos.
5. Independiente de cualquier agente externo. De hecho sus reglas de negocio simplemente no sabe nada en absoluto sobre el mundo exterior.

El diagrama en la parte superior de este artículo es un intento de integrar todas estas arquitecturas en una sola idea procesable.

**La Regla de Dependencia**

Los círculos concéntricos representan diferentes áreas de software. En general, a medida que te vas, el nivel más alto es el software se convierte. Los círculos exteriores son mecanismos. Los círculos internos son las condiciones.

La regla primordial que hace que esta obra de arquitectura es *la Regla de Dependencia*. Esta norma establece que las *dependencias de código fuente* sólo pueden apuntar*hacia el interior* . Nada en un círculo interno puede saber nada en absoluto sobre algo en un círculo exterior. En particular, el nombre de algo declarada en un círculo exterior no deberá ser notificada por el código en el interior de un círculo. Eso incluye, funciones, clases. variables, o cualquier otra entidad de software llamado.

Por la misma razón, los formatos de datos utilizados en un círculo exterior no debe ser utilizado por un círculo interno, especialmente si esos formatos se generan por un marco en un círculo exterior. No queremos que nada en un círculo externo para impactar los círculos internos.

***entidades***

Entidades encapsulan *toda la empresa* reglas de negocio. Una entidad puede ser un objeto con métodos, o puede ser un conjunto de estructuras de datos y funciones. No importa, siempre que las entidades podrían ser utilizados por muchas aplicaciones diferentes en la empresa.

Si usted no tiene una empresa, y se acaba de escribir una sola aplicación, a continuación, estas entidades son los objetos de negocio de la aplicación. Ellos encapsulan las reglas más generales y de alto nivel. Ellos son los menos propensos a cambiar cuando algo cambia externos. Por ejemplo, no es de esperar que estos objetos podrían verse afectados por un cambio a la página de navegación, o la seguridad. Ningún cambio operacional para cualquier aplicación particular debería afectar a la capa de entidad.

**Casos de uso**

Los programas de esta capa contienen *específicos de la aplicación* de reglas de negocio. Se encapsula e implementa todos los casos de uso del sistema. Estos casos de uso orquestar el flujo de datos hacia y desde las entidades, y dirigen esas entidades para utilizar sus *anchas empresariales* reglas de negocio para lograr los objetivos del caso de uso.

No esperamos que los cambios en esta capa para afectar a las entidades. Tampoco esperamos que esta capa se vea afectada por los cambios en factores externos tales como la base de datos, la interfaz de usuario, o cualquiera de los marcos comunes.Esta capa está aislado de estas preocupaciones.

Nosotros *sí* , sin embargo, esperamos que los cambios en el funcionamiento de la aplicación *se* afectará a los casos de uso y por lo tanto el software en esta capa. Si los detalles de un cambio de casos de uso, a continuación, algo de código en esta capa será, sin duda afectada.

**Adaptadores de interfaz**

Los programas de esta capa es un conjunto de adaptadores que convierten los datos en el formato más conveniente para los casos de uso y entidades, para el formato más conveniente para algún agente externo, como la base de datos o en la Web. Es esta capa, por ejemplo, que en su totalidad contendrá la MVC arquitectura de unainterfaz gráfica de usuario. Los presentadores, vistas y controladores de todos pertenecen aquí. Los modelos son probables estructuras de datos sólo que se transmiten de los controladores de los casos de uso, y luego de vuelta de los casos de uso para los presentadores y puntos de vista.

Del mismo modo, los datos se convierten, en esta capa, de la forma más conveniente para las entidades y los casos de uso, en la forma más conveniente para cualquier framework de persistencia se están utilizando. Es decir, la base de datos. Ningún código hacia el interior de este círculo debe saber nada en absoluto sobre la base de datos. Si la base de datos es un SQL base de datos, entonces todo el SQL debe limitarse a esta capa, y en particular a las partes de esta capa que tienen que ver con la base de datos.

También en esta capa es de cualquier otro adaptador necesario para convertir los datos de una cierta forma externa, tal como un servicio externo, a la forma interna utilizada por los casos de uso y entidades.

**Marcos y conductores.**

La capa más externa se compone generalmente de marcos y herramientas tales como la base de datos, el Marco de Web, etc. Generalmente no se escribe mucho código en esta capa que no sea un código de adhesión que comunica a la siguiente círculo hacia el interior.

Esta capa es donde van todos los detalles. La Web es un detalle. La base de datos es un detalle. Mantenemos estas cosas en el exterior donde pueden hacer mucho daño.

**Sólo cuatro círculos?**

No, los círculos son esquemáticas. Usted puede encontrar que usted necesita algo más que estos cuatro. No hay una regla que dice que siempre hay que tener sólo estos cuatro. Sin embargo, *la regla de Dependencia* se aplica siempre. Dependencias de código fuente siempre apuntan hacia el interior. A medida que avanza hacia el interior aumenta el nivel de abstracción. El círculo más externo es bajo nivel de detalles concretos. A medida que avanza hacia el interior del software crece más abstracto, y encapsula las políticas de nivel superior. La más interior del círculo es el más general.

**cruzar fronteras.**

En la parte inferior derecha del diagrama es un ejemplo de cómo cruzar los límites del círculo. Muestra los controladores y los presentadores que se comunican con los casos de uso en la siguiente capa. Tenga en cuenta el flujo de control. Se inicia en el controlador, se mueve a través del caso de uso, y luego termina la ejecución en el presentador. Tenga en cuenta también las dependencias de código fuente. Cada uno de ellos apunta hacia dentro, hacia los casos de uso.

Por lo general, resolver esta aparente contradicción con la [dependencia Inversión Principio](http://en.wikipedia.org/wiki/Dependency_inversion_principle) . En un lenguaje como Java, por ejemplo, arreglaríamos interfaces y relaciones de herencia tales que las dependencias de código fuente se oponen al flujo de control en los puntos de derecho sólo a través de la frontera.

Por ejemplo, considere que el caso de uso tiene que llamar al presentador. Sin embargo, esta llamada no debe ser directa, porque eso violaría *la regla Dependencia* : Sin nombre en un círculo externo puede ser mencionado por un círculo interno. Así tenemos el caso de uso llamar a una interfaz (Mostrado aquí como puerto de salida de casos de uso) en el círculo interior, y tienen el presentador en el círculo exterior implementarlo.

La misma técnica se utiliza para atravesar todas las fronteras en las arquitecturas.Nos aprovechamos de polimorfismo dinámico para crear dependencias de código fuente que se oponen al flujo de control para que podamos cumplir con *la regla de la dependencia* sin importar la dirección del flujo de control va en.

**¿Qué datos cruza las fronteras?.**

Por lo general los datos que cruza las fronteras es estructuras de datos simples. Puede utilizar estructuras básicas u objetos simple de transferencia de datos si lo desea. O simplemente los datos pueden ser argumentos en llamadas a funciones. O bien, puede empacar en un mapa hash, o construir en un objeto. Lo importante es que, aislados, estructuras de datos simples se pasan a través de las fronteras. No queremos hacer trampa y pasar *entidades* o filas de base de datos. No queremos que las estructuras de datos que tienen cualquier tipo de dependencia que viola *la regla de Dependencia* .

Por ejemplo, muchos marcos de base de datos devuelven un formato de datos conveniente en respuesta a una consulta. Podríamos llamar a esto un RowStructure. No queremos pasar esa estructura fila hacia el interior a través del límite. Eso violaría *la regla de Dependencia* porque obligaría a un círculo interno saber algo sobre un círculo exterior.

Así que cuando se pasa datos a través de una frontera, siempre es en la forma que sea más conveniente para el círculo interior.

**Conclusión**

Conforme a estas reglas simples no es difícil, y le ahorrará muchos dolores de cabeza en el futuro. Al separar el software en capas, y conforme a *la Regla de Dependencia* , va a crear un sistema que es intrínsecamente comprobable, con todos los beneficios que ello implica. Cuando cualquiera de las partes externas del sistema se vuelven obsoletos, como la base de datos, o el framework de desarrollo web, se puede sustituir aquellos elementos obsoletos con un mínimo de esfuerzo.