# Lección 1: Introducción.

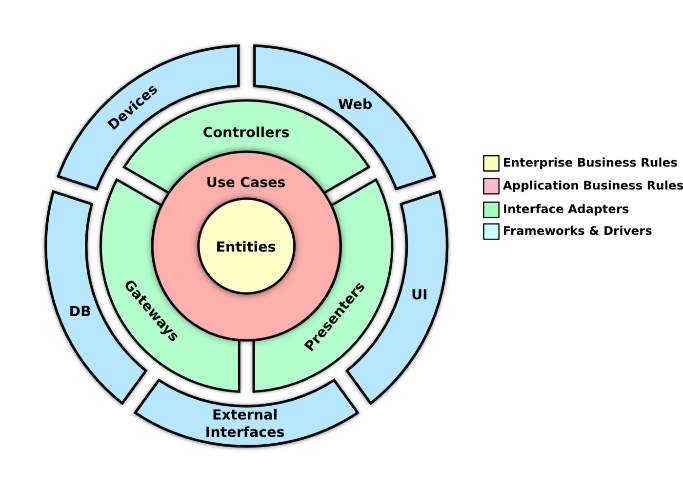
## Principios de arquitectura.

*“Si los constructores construyeran los edificios de la misma forma que los programadores escriben los programas, el primer pájaro carpintero que apareciera destruiría la civilización”*

*Gerald Weinberng.*

*\*\*\* Repaso \*\*\**

### Un poco sobre el diagrama.

* Las necesidades que se requiere solucionar se implementan en “Entidades”, en esta parte va el “que se necesita” no el “como lo debo hacer “, es la parte que requiere más abstracción.
* En “Casos de Uso” se implementa la lógica de negocios, en otras palabras, el “cómo se debe hacer”, lo que no se definió en Entidades.
  + - * Para obtener los datos que requiere el “Caso de Uso” se requiere de un Repositorio de Datos, todo el código relacionado al acceso a datos de ubica en el 3er nivel del diagrama en “Gateways”, pero también los proyecto requieren de algo para mostrar información hacia los usuarios y para eso utilizaremos el “Presentador” o “Controlador”, y el controlador dice a los elementos de adentro que es lo que se quiere hacer.
* En la capa exterior tenemos elementos como las bases de datos e Interfaces de usuario, mismas que serán comunicada con sus elementos correspondientes, por ejemplo, en BD con Gateways O Presentador con UI.

*\*\*\* 9 sesion \*\*\**

### **Inversión de dependencias.**

***La dirección de dependencia dentro de la aplicación debe estar en la dirección de la abstracción, no de los detalles de implementación.***

Ejemplo:

Cuando un hablamos con un cliente, él nos dice ¿**Que Necesita?** ynosotros damos solución, comenzando a crear las abstracciones, mismas que van en la capa de “Entidades”, aunque puede que esa capa no exista entonces las abstracciones irían en la capa de “Casos de Uso”.

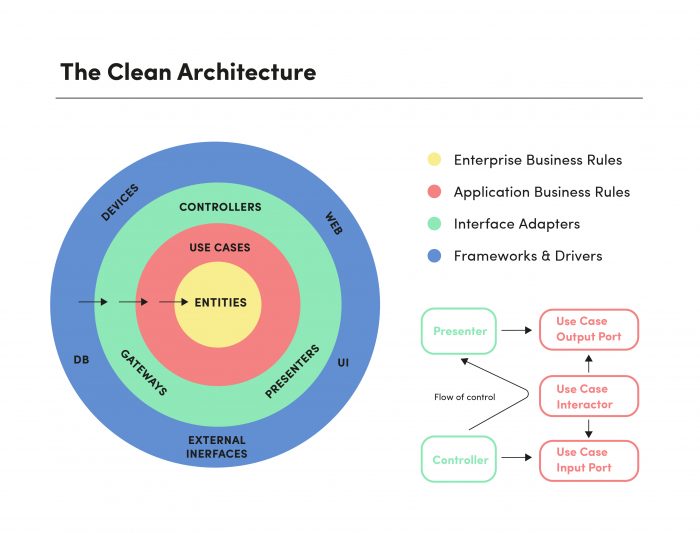
Desde las capas internas “Entidades” o “Casos de Uso” se crean las abstracciones y en capas exteriores se hace la implementación de esas abstracciones.

En el caso de “Arquitectura Limpia” las dependencias de las capas externas van en dirección de la atracción de capas internas, “por eso muchas interfaces”.

Diferencias entre Dependencia Directa y Dependencia Invertida.

|  |  |
| --- | --- |
|  | Dependencia Directa.  La “Clase A” invoca un método de la “Clase B” misma que invoca un método de la “Clase C”.  *Las clases están acopladas y tienen una dependencia directa, también el consumo de las clases es directo.* |
|  | Dependencia Invertida.  Permite que la “Clase A” invoque métodos de una abstracción de la “Clase B” esto por medio de una interfaz y la clase “Clase B” implementa la abstracción.  *La “clase a” necesita funcionalidad que esta en “interfaz b” y “clase b” da la funcionalidad y lo mismo pasa con b y c.*  *Así la “Clase A” define la abstracción en “Interfaz b” y “Clase B” implementa esa abstracción, obteniendo como resultado que clase A y B no se conozcan, ni tengan dependencia entre si y estén* ***desacoplados.*** |

*Otro ejemplo: En dependencia la “Clase A” implementa “Clase B” con un new Clase B, PERO en dependencia invertida “Clase A” dice “quiero alguien que implementa* ***Interfaz B*** *para consumir* ***clase B****” y no con new Clase B, además que con Interfaces estaremos seguros que se implementara todo lo que se requiere.*

***\*\*\* Regresando al Diagrama \*\*\****

*Cuando la* ***UI*** *solicita algo, el* ***controlador*** *atiende, pero el controlador no lo puede dar, se lo tiene que pedir a* ***Casos de Uso “Interactor”,*** *dado que tiene la lógica de negocios, PERO el “controlador” y los “Casos de Uso” no se comunican directamente, así que hacen uso del “Input Port” que es un Interfaz.*

### **Dependencias Explicitas.**

***Los métodos y las clases deben requerir explícitamente de todos los objetos de colaboración que necesiten para funcionar correctamente.***

***El código se vuelve más auto explicativo y los contratados de codificación más fáciles de usar.***

En otras palabras, debemos de crear clases y métodos que indiquen explícitamente u obligatoriamente que elementos necesitan para funcionar, en el caso de las clases tenemos los constructores para que inicialice todos los campos necesarios para funcionar, en el caso de los métodos podemos requerirlos desde los parámetros.

### **Responsabilidad Única.**

***Los objetos solo deben tener una responsabilidad y solo una razón para cambiar.***

***El objeto debe cambiar solo si hay que actualizar la mera en la que lleca a cabo su única responsabilidad.***

***Este principio nos lleva a los Microservicios.***

Ejemplo:

Necesitamos un método para obtener un registro en específico y uno para para eliminar, *¿Estos métodos deben estar en la misma clase, según este principio?...* NOO son operaciones distintas “Consulta” y “Agregar” que deberán estar separadas dado a que este principio dice que solo una responsabilidad por objeto.

Resultado:

Ayuda a generar sistemas mas modulares y de acoplamiento flexible, cada clase con una función, así cuando le mueva no le vas a pegar otra cosa, *“agregar clases nuevas es mas seguro, que cambiar las existentes”*

*Microservicios: Es un servicio con una sola responsabilidad, usualmente se se aplica en las Web Api y expone un solo endPoint.*

### **Una vez y solo una “Don’t repeat toueself”.**

***La aplicación debe evitar especificar el comportamiento relacionado con un determinado concepto en varios lugares.***

***En lugar de duplicar la lógica, se puede encapsular en una construcción del programa.***

***“SI TIENE EL MISMO OBJETIVO, SE ENCAPSULA DE LO CONTRARIO SE DEBE DUPLICAR"***

Ejemplo:

Queremos detectar si un cliente tiene un adeudo de 100 pesos, tenemos un método que lo revisa en un módulo, pero ahora lo queremos en otro por ejemplo en reportes pues ahora metemos la función en ese lugar, para que al final el cliente dice que sean 200, hay que cambiar en 2 o mas lugares y puede que olvidemos una.

Nota:

*“las funcionalidades se parecen mucho solo por un dato, tenemos que crear dos clases distintas,* ***POR QUE SON DOS OBJETIVOS DISTINTOS****”*

### **Omisión de persistencia**

***El código de los tipos que se deben conservar (persistir) no debe verse afectado por la elección de la tecnología de persistencia.***

En .NET, estos tipos a veces se denominan objetos (POCO, clases que usamos en entidades), ya que no necesitan heredar de una clase base concreta o implementar una interfaz determinada.

*Algunos ejemplos de las infracciones de este principio son estos:*

* *Una clase base requerida.*
* *Una implementación de interfaz requerida.*
* *Clases responsables de guardarse a sí mismas (por ejemplo, el patrón de registro activo).*
* *Constructor sin parámetros requerido.*
* *Propiedades que requieren la palabra clave virtual.*
* *Atributos requeridos específicos de la persistencia.*

*La omisión de persistencia es útil porque permite conservar el mismo modelo de negocio de varias formas, lo que ofrece flexibilidad adicional a la aplicación.*

### **Contextos delimitados.**

***Abordar la complejidad en organizaciones o aplicaciones de gran tamaño dividiéndola en módulos conceptuales independientes.***

***Después, cada módulo conceptual representa un contexto que está separado de otros contextos (por tanto, delimitado) y que puede evolucionar independientemente.***

***Los contextos delimitados se asignan estrechamente a los microservicios que, idealmente, también se implementan como sus propios contextos delimitados individuales.***

En otras palabras, se trata de entender la complejidad de una aplicación y dividirla en módulos independientes, con contextos independientes, e incluso podrían tener BBDD separadas.