UNIVERSIDAD DON BOSCO

INGENIERÍA DE SOFTWARE

TRABAJO DE INVESTIGACIÓN

DOCENTE:

ING. ALEXANDER SIGÜENZA

INTEGRANTES:

ACOSTA BELTRÁN, WILFREDO JOSÉ AB181924

GUTIÉRREZ SOLÓRZANO, HENRY BRYAN GS181939

CHÁVEZ GARCÍA, JOSUÉ ALEXANDER CG172415

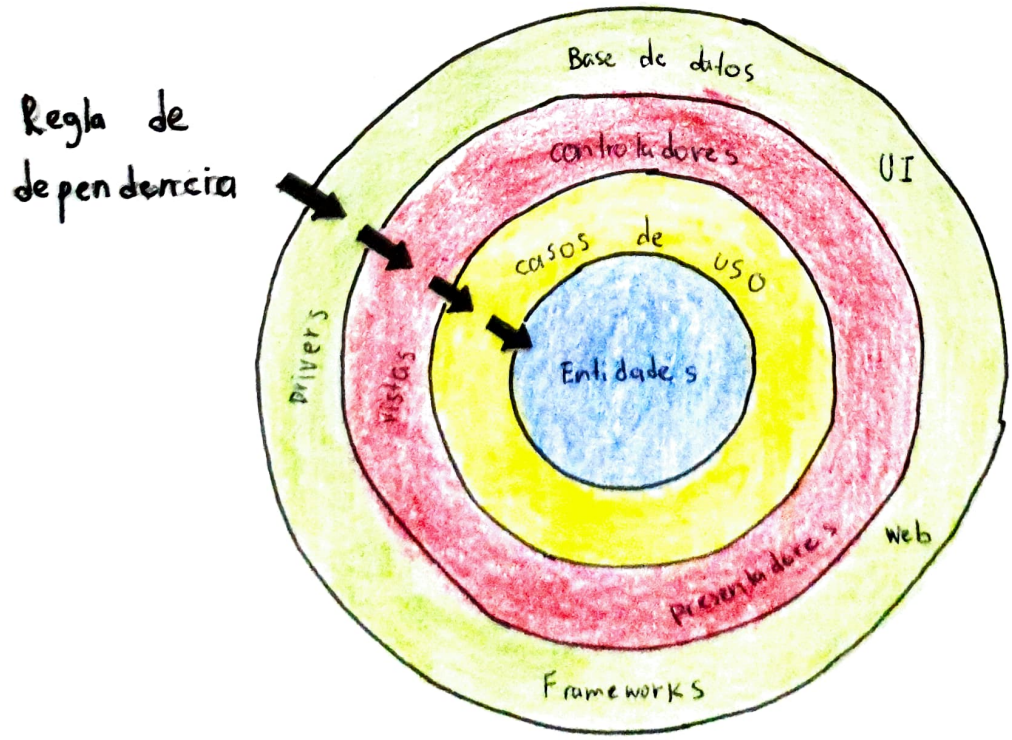
RIVAS GONZÁLEZ, CÉSAR JOSUÉ RG180141

SOYAPANGO 18 DE JUNIO DE 2021

ARQUITECTURA CLEAN

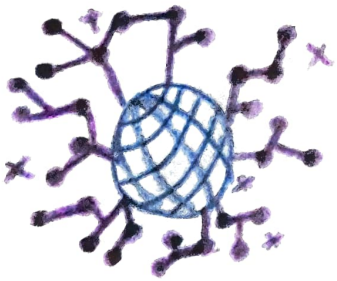
Se basa en estructurar el código en niveles. Nos podemos imaginar las distintas capas de software como círculos concéntricos de distinto tamaño, en la que cada capa solo puede comunicarse y heredar elementos de capas más interiores, nunca de una capa más externa a ella. A esto se le llama "regla de dependencia". Entre más adentro del círculo esté la capa, mayor será la complejidad del código. Mediante estas técnicas, la lógica del programa se mantiene aislada y el código se vuelve más mantenible y escalable

La arquitectura clean se compone de la siguiente manera:



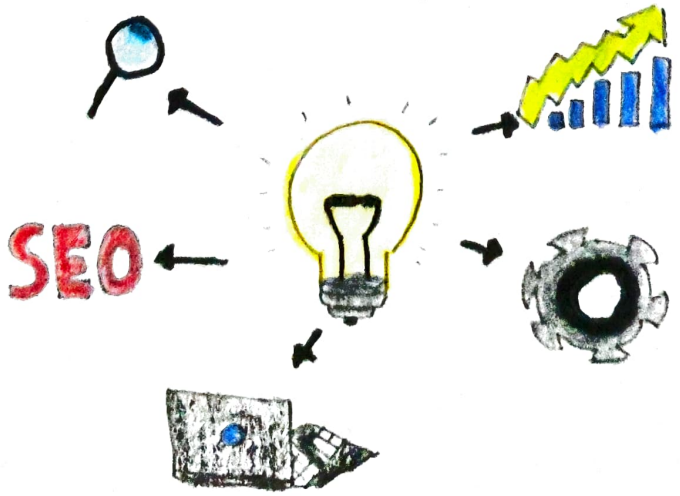
1. ENTIDADES

Contienen las reglas más fundamentales con las que funciona la lógica de una empresa, reglas que difícilmente si acaso llegan a cambiar. Estas pueden ser objetos con métodos, estructuras de datos o funciones que sean usadas en muchas partes de nuestra empresa.



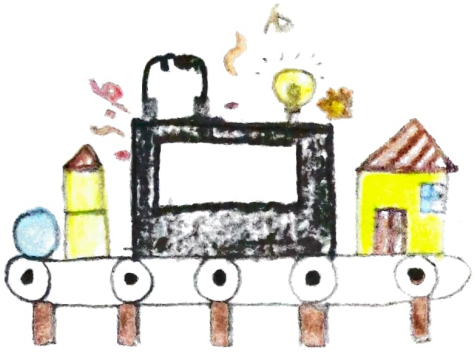
2. CASOS DE USO

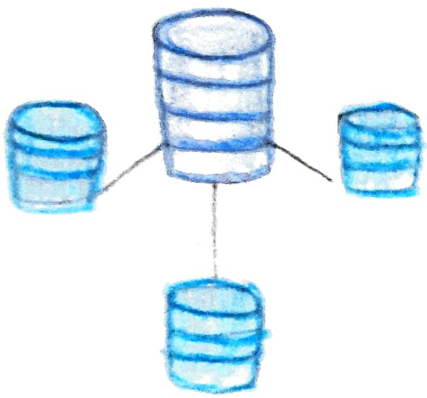
Hacen referencia a la lógica de la aplicación, encapsula e implementa todos los casos de uso con los que va a trabajar la aplicación, estos hacen llegar los datos a las entidades y logran que las entidades trabajen para cumplir con las tareas que necesitan. Esta capa no realiza cambios a las entidades ni tampoco debe sufrir cambios. La única manera en que se ve afectada es si alguno de los casos de uso es modificado.



3. ADAPTADORES

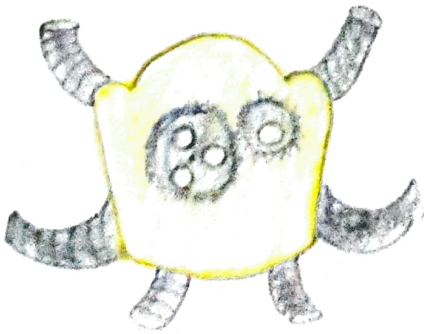
En esta capa de software, se transforman los datos para que puedan ser utilizados y presentados en la capa exterior, en el caso de la arquitectura MVC, los controladores y las vistas se encuentran encapsulados en esta capa, mientras que el modelo son los datos que viajan entre los controladores y los casos de uso.

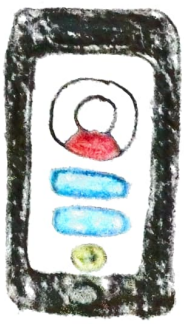




4. UI - DRIVERS - BD - FRAMEWORK...

Esta capa se compone generalmente de herramientas como bases de datos, Frameworks y todos aquellos componentes que no influyen en la lógica de la aplicación y por lo tanto, no dependen ni hacen dependiente a la aplicación, por lo que se vuelve sencillo cambiar la UI o framework si fuera necesario.





PRINCIPIOS S.O.L.I.D.

Conjunto de principios que cuando son aplicados a la vez, generan un código fácil de mantener y escalar con el tiempo, su objetivo principal es hacer un código más entendible y reducir su complejidad. Estos principios son:

- SINGLE RESPONSIBILITY -

Nos dice que en cada clase o método en nuestro código solo debe tener un propósito. Esto causa que haya menos Bugs en nuestro código y si los hay, arreglarlos no causaría más problemas en otra parte.

- OPEN / CLOSED -

Nos indica que cuando se añadan más funcionalidades a una clase, se debe hacer extendiendo el código y no modificándolo, por ejemplo, heredar desde una clase a una subclase y añadir nuevas funcionalidades.

También se puede aplicar a UI's existentes, agregando miembros en vez de modificar los existentes.

- LISKOV SUBSTITUTION -

Cuando una clase hija no puede hacer las mismas acciones que su clase padre, esto puede causar bugs en el código. La clase hija debe poder realizar lo mismo que la clase padre. Este proceso se conoce como herencia de clases. La clase hija debe ser capaz de arrojar el mismo resultado que la clase padre o como mínimo un resultado del mismo tipo.

- INTERFACE SEGREGATION -

Las clases solo deben realizar acciones que sean útiles y necesarias, de lo contrario puede causar bugs. Las acciones que una clase no deba realizar para cumplir con su rol deben quitarse o moverse a otro lado si estas van a ser usadas por otras clases en el futuro.

- DEPENDENCY INVERSION -

Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones (interacción entre dos clases). Esto nos dice que una clase no debe estar amarrada a una herramienta para ejecutar una acción, sino a la interfaz que permite a la herramienta trabajar con la clase.

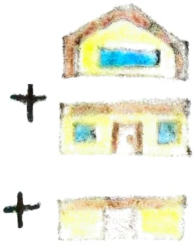
PATRONES DE DISEÑO

Son una colección de técnicas utilizadas en el desarrollo de software para resolver problemas comunes. Estos pretenden estandarizar la forma en que se realiza el diseño, facilitar el aprendizaje, proporcionar elementos reutilizables, entre otros.

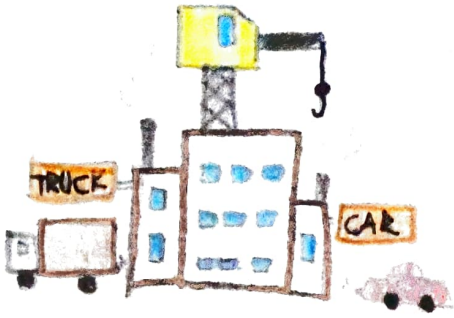
Existen 23 patrones de diseño individuales que a su vez se clasifican en tres grandes categorías:

PATRONES CREACIONALES

Proporcionan mecanismos de creación de objetos que aumentan la reutilización de código, se basan sobre todo en creaciones de clases y objetos.

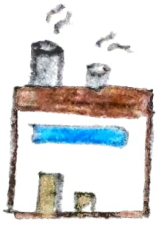


FACTORY METHOD



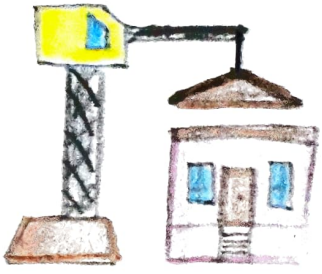
Se crea una superclase constructora de objetos, que permite a las subclases modificar el tipo de objeto creado para su conveniencia.

ABSTRACT FACTORY



Permite crear fábricas de objetos específicos sin tener que especificar su clase concreta.

BUILDER



Permite crear objetos complejos paso por paso y diferentes tipos de un objeto usando el mismo código. Se tienen todos los métodos en un objeto constructor, pero solo se invocan los necesarios para construir los distintos tipos de objetos según se requiera.

PROTOTYPE

Permite clonar un objeto a partir de una clase con todos los valores del objeto viejo al nuevo. Sirve como alternativa a la creación de subclases.

SINGLETON

Permite asegurar que una clase tenga una única instancia y también proporciona un único punto de acceso global evitando que se sobreescriba. Cuando se llama a un objeto de nuevo, se accede al original en vez de crear una nueva instancia.

PATRONES ESTRUCTURALES

En summa objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.

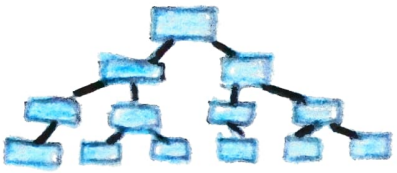
ADAPTER

permite la colaboración entre objetos incompatibles. En esencia, convierte la interfaz de un objeto para que otro pueda comprenderla y trabajar con ella.

BRIDGE

Permite dividir una clase grande o grupo de clases relacionadas en dos jerarquías separadas que pueden desarrollarse por separado independientes de la otra.

COMPOSITE



Permite componer objetos en estructuras de árbol y trabajarlas como objetos individuales. Solo puede usarse cuando el modelo central se puede representar en forma de árbol.

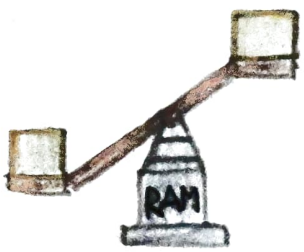
DECORATOR

Permite añadir funcionalidades a objetos existentes colocándolos dentro de objetos encapsuladores especiales que tienen esta funcionalidad.

FACADE

Proporciona una interfaz simplificada a una biblioteca, framework o grupo de clases con alta complejidad.

FLYWEIGHT



Permite mantener más objetos dentro de la cantidad disponible de RAM. Comparte estados comunes entre varios objetos en vez de guardar la información de cada objeto individual.

PROXY

permite tener un substituto de posición para un objeto, para controlar el acceso al objeto original, permitiendo realizar tareas antes de que la solicitud llegue al original.

PATRONES DE COMPORTAMIENTO

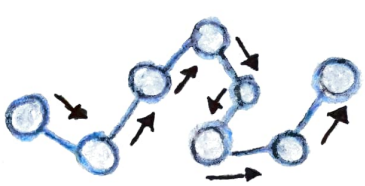
CHAIN OF RESPONSIBILITY

permite pasar solicitudes a lo largo de una cadena de handlers, al pasarla, cada handler decide si procesarla o pasarla al siguiente.

COMMAND

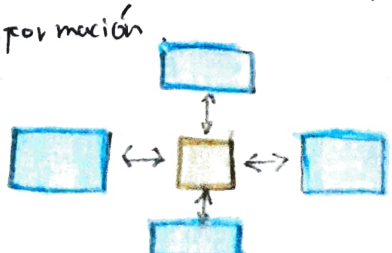
convierte la solicitud en un objeto independiente que tiene toda la información sobre la solicitud y permite a la clase realizar las tareas con la información proporcionada por cualquier solicitud.

ITERATOR



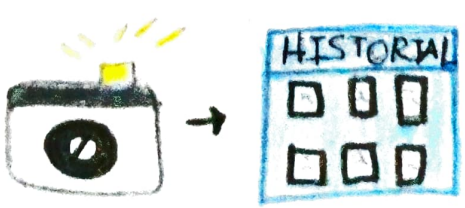
se utiliza para recorrer estructuras de datos como listas, árboles, grafos, etc. Sin realizar ningún intercambio de información.

MEDIATOR



permite reducir las dependencias entre objetos restringiendo las comunicaciones directas entre objetos.

MEMENTO

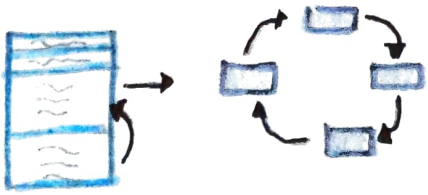


permite guardar y restaurar el estado previo de un objeto sin revelar detalles importantes sobre su implementación

OBSERVER

permite definir un mecanismo de "suscripción" que permite notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

STATE

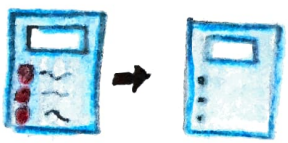


permite a un objeto cambiar su comportamiento cuando su estado interno cambia.

STRATEGY

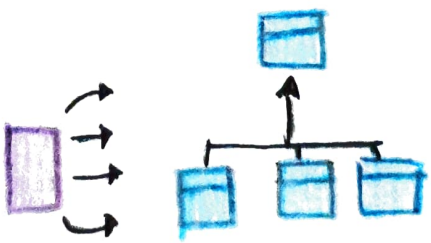
permite definir una familia de algoritmos, colocarlos en clases separadas y hacer sus objetos "intercambiables" (cumplen la misma función pero de distinta manera, con una distinta estrategia).

TEMPLATE METHOD



Define el esqueleto del algoritmo en una superclase pero permite a las subclases sobreescribir pasos del algoritmo sin alterar la estructura.

VISITOR



permite ejecutar una operación sobre un grupo de objetos con diferentes clases, el objeto visitante implementa diferentes variantes de la misma operación que corresponden a las clases objetivo.