**Arquitectura de Software 1**

Alumno: Santiago Vietto

Docente: Martin Marcucci

DNI: 42654882

Institución: UCC

Año: 2021

**Introducción**

**Contenido**

\_ Los patrones de diseño son básicamente bloques para construir cosas. En el mundo del software también existen patrones, en los que cuando alguien se sienta a programar y se le presenta un problema, es muy probable que alguien ya lo haya resuelto mediante ese patrón y por ende nosotros lo aplicamos. Cada patrón soluciona un problema que surge en una situación particular. Los video juegos son un lugar donde sucede el uso de los patrones debido a la complejidad del proyecto.

* Página para ver los patrones de diseño: <https://refactoring.guru/es/design-patterns>

\_ También tenemos MVC (modelo vista-controlador) que es un patrón de arquitectura, en este tenemos varios frameworks ya construidos donde uno solamente los usa, algunos son:

* Si queremos programar Javascript con NodeJS podemos usar Sails.js, Meteor, Express.
* Si queremos programar con Python tenemos Django.
* Si queremos programar con PHP tenemos Symphony.

\_ Tanto los patrones de diseño como MVC serian dos grandes partes del diseño de software.

**Herramientas**

Librería SFML (Simple and Fast Multimedia Library): es una librería grafica multiplataforma y multilenguaje.

<https://www.sfml-dev.org/>

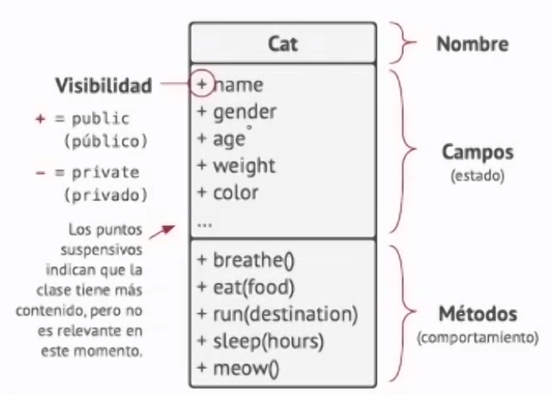
Tiled: herramienta para crear o generar mapas.

Assets: son los recursos de juegos (personajes, texturas, armas, objetos).

Tileset: conjunto de patrones que se usan para armar un mapa.

**Repaso conceptos**

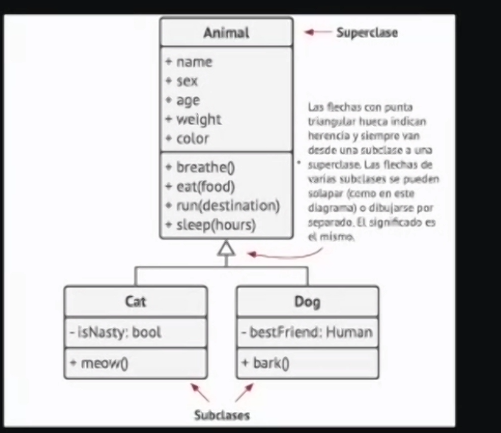
Diagramas UML: son dibujos que nos van a permitir representar lo que nosotros hacemos en un software. Es una forma de plasmar de manera grafica cuestiones de diseño de software. El diagrama de clases en particular nos ayuda a representar clases justamente. Si queremos hacer una clase, primero hacemos un rectángulo y luego se divide en 3 partes, la primera sección es el nombre de la clase, la segunda son todos los campos o atributos de esa clase, donde le colocamos un + si el atributo es público o un - si es privado, y la tercera parte es donde se describen todos los métodos y comportamientos tratando de poner la firma lo más completa posible y usamos también la analogía + o - si es un método público o privado.



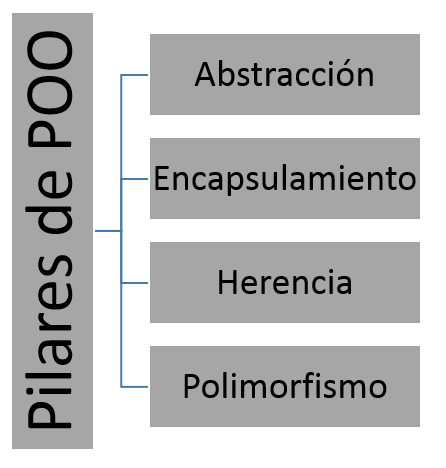
\_ Luego pasamos a definir lo que son los objetos, estos son instancias de esas clases y es directamente algo físico o que existe que tiene los atributos particulares y que pasa a hacer un objeto diferente de otro.



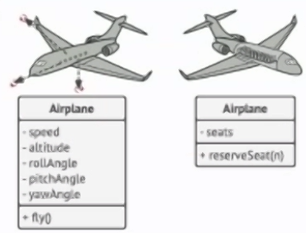
\_ Después pasamos a tener una jerarquía de clases. Las clases generalmente según que se utilice (herencia, encapsulamiento), tienen una jerarquía, una clase va a ser que otra exista, la contenga, herede de esta, etc, entonces cuando tengo una clase y tengo otras clases se unen con flechas y según el tipo de flecha va a indicar lo que yo quiero. Entonces como vemos en el ejemplo, la unión nos da jerarquía y nos dice que la clase animal es algo más genérico que el gato o el perro, y en particular la flecha triangular hueca indica herencia. Cuando dos clases heredan de una misma se pueden unir.



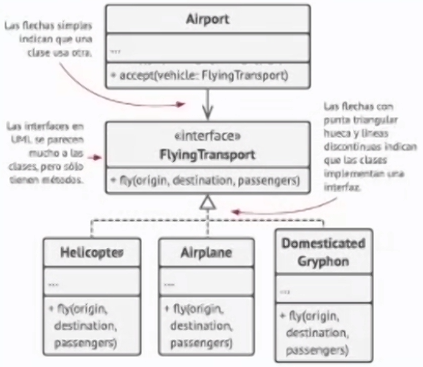
POO: la idea de usar programación orientada a objetos es que podamos tener abstracciones, polimorfismo, encapsulamiento, y herencia.



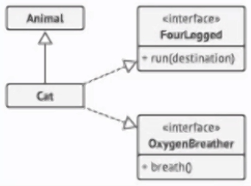
* Abstracción: es la capacidad de poder describir solamente lo que me hace falta en el contexto en el que estoy utilizando. Por ejemplo si estamos haciendo un simulador de aviones, el objeto o clase avión va a tener atributos como la velocidad, ángulo de giro, altitud, etc y los métodos que va a tener son volar, acelerar, frenar, etc. Pero si trabajamos en una aerolínea y estamos haciendo un gestor de pasajes aéreos, lo que nos va a hacer falta saber es cuantos asientos tiene el avión para saber si podemos reservar alguno o no. Entonces esto es lo que nos permite la abstracción, no tenemos que crear un avión y poner todas sus características sino que abstraemos solamente las que nos hacen falta para simplificar el uso de cada entidad.



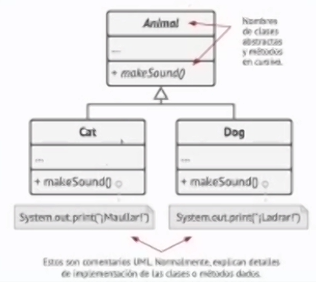
* Encapsulamiento: nos permite hacer que algo quede oculto al resto del software, entonces es cuando encapsulamos atributos y los hacemos privados adentro de una clase. En la clase helicóptero por ejemplo, las características están encapsuladas y son propias del helicóptero y si las hacemos privadas nadie más las va a poder ver. La otra posibilidad que tenemos para usar encapsulamiento es utilizar lo que se llama interfaz. En C++ no existe la interfaz como tal pero lo podemos tomar como si fuese una clase abstracta, donde esta hace que las demás puedan heredar y cuando damos a conocer la clase abstracta, el que la utiliza sabe que puede hacer lo que la misma permite pero no sabe que atributos o que cosas va a tener esa clase abstracta o la que hereden de esta. Las líneas discontinuas indican que es una interfaz. La flecha simple indica que la clase utiliza en este caso a la clase abstracta, es decir, una clase que utiliza otra clase.



* Herencia: lo que permite es que generamos comportamientos o atributos genéricos en una clase y hago que otras clases más particulares hereden esos atributos y funcionalidades, a la vez podemos tener herencia múltiple, donde por ejemplo la clase gato hereda de la clase animal y a la vez hereda de las otras dos interfaces.



* Polimorfismo: es la capacidad en POO de que un método en una clase sea sobre escrito por otro método en otra clase y tengan el mismo nombre, por lo tanto si estamos trabajando con un gato a través de la clase abstracta animal, si llamamos al método makeSound en realidad llamamos el del gato y no el de animal porque estamos usando polimorfismo, donde el mismo nombre del método es sobre escrito o utilizado por otro. En los diagramas UML el nombre del método se pone en cursiva para denotar de que no se implementa en la clase genérica sino en las clases que hereda.



**Principios de diseño**

\_ Los principios de diseño son reglas que nos van ayudar a tener un diseño más mantenible, o sea a que nuestro software pueda ser mantenido fácilmente. Esto es importante porque el software que nosotros desarrollamos para un cliente muy probablemente debamos agregarle funcionalidades o mejorarlo a medida que pase el tiempo, y si nosotros no tenemos en cuenta esto desde un principio veremos que se puede hacer más compleja esta tarea.

**SOLID**

\_ Es importante decir que el desarrollo de software no es un juego de “jenga”, en donde no sirve apilar bloques sin sentido y luego cuando queramos agregarle funcionalidades o tratar de hacer algo que este todo frágil e inestable ahí y que sea muy complejo poder realizar un cambio. La idea es que nuestro diseño sea un cimiento fuerte para poder seguir construyendo arriba de eso por más que cambie el tiempo.

**Problemas en el código**

\_ Los problemas que podemos tener en el código son:

* El código puede ser grande y tener muchas líneas, haciendo que nuestro código sea más difícil de mantener. Mientras más código reutilicemos y diseñemos mejor, más chico será y fácil de comprender ese software.
* Puede ser que haya mucho código repetido (copy paste), esto de tener una sola funcionalidad implementada en muchos lugares mediante el mismo código, y esto no es bueno porque si quiero modificar esa funcionalidad debo hacerlo en cada lugar en donde este implementada.
* Otro problema es que una funcionalidad este mal implementada y que este distribuida por muchos lugares, entonces si queremos modificar esa funcionalidad, tenemos que ir a cada uno de esos lugares donde este la función y solucionar los problemas. Para realizar un cambio hay que modificar muchas cosas.
* La complejidad ciclo matica elevada, que es la cantidad de estructuras que tengo anidadas, dicho de otra forma es cuantos if, for, while, switch, estas estructuras de control tengo unas dentro de otras. Si yo tengo un for con adentro un if luego un while y así, seguir que está haciendo ese código se va a hacer muy complejo.

\_ Para solucionar estos problemas hay muchas buenas prácticas, como por ejemplo comentar bien el código, poner buenos nombres a las variables, crear funciones, etc. En cuanto al diseño hay buenas prácticas y principios a seguir que nos van a ayudar a que ese código este bien hecho. Algunos de estos principios son por ejemplo los SOLID.

**Repaso de POO**

\_ La programación orientada a objetos es un estilo de programación. Es un paradigma de programación como el paradigma funcional, procedural, lógico. El paradigma orientado a objetos se basa en cuatro pilares:

Encapsulamiento: el poder guardar en un solo lugar el conjunto de funcionalidades encapsuladas.

Herencia: poder derivar en funcionalidades en común desde un solo lugar y no tener que re implementarlo en varios lados.

Polimorfismo: el poder cambiar una clase por otra y que sus nombres sean los mismos, y que sus métodos sean los mismos.

Abstracción: el poder hacer algo abstracto y poder después especializarlo en distintos objetos.

\_ Esto está basado en clases y objetos. Cada objeto tiene responsabilidades únicas y contienen campos, atributos, métodos que hacen que pueda cumplir sus objetivos.

\_ Siempre se aconseja estas dos medidas en la POO como buenas prácticas:

* Acoplamiento: el bajo acoplamiento es un término de software que se refiere a la relación que se guarda entre los módulos del sistema y la dependencia entre ellos. Si hay bajo acoplamiento quiere decir que los módulos no conocen a otros módulos del mismo sistema, hay muy poca interacción entre esos módulos, por eso no están acoplados, esto esta bueno porque quiere decir que ese modulo si esta acoplado muy poco lo podemos sacar con mucha facilidad y utilizarlo en otro lado, pero si está muy acoplado y su funcionalidad depende de los otros módulos o clases del sistema, no podemos sacar esa clase sola sino que debemos sacar todas las clases juntas.
* Cohesión: se busca tener una alta cohesión, en términos de software esto se refiere a la medida que tanto un módulo del sistema tiene una sola responsabilidad, solo un módulo tiene una responsabilidad, no está esa responsabilidad derivada en varios módulos. Entonces el módulo con alta cohesión será aquel que guarde una alta relación entre sus funcionalidades y manteniendo un enfoque de su único propósito. Esto quiere decir que esa cohesión es un módulo que sabe hacer algo particular y que solamente en todo el sistema él es el que lo sabe hacer, ese es su único propósito, no tiene otro, y no hay otro modulo que lo ayude.

\_ Para realizar estas cosas veremos algunas formas de empezar a ver que tenemos problemas, estos se llaman Desing Smells.

Desing Smells

\_ Se sabe decir que el código huele, cuando vemos algo raro en el código o funciona de forma correcta pero hay algo que nos dice que ese código no está bueno, que hay algo que lo podríamos hacer de una forma mejor, que esta feo básicamente. Ahora cuando hacemos diseño, también hay algunas cosas que nos dicen que el diseño huele cuando se presiente que nos traerá problemas más adelante y que podría estar mejor. Algunos de ellos son:

* Rigidez: quiere decir que para poder hacer un cambio en el software, aunque sea un cambio simple, se me hace muy complejo. Un cambio causa una cascada de cambios dentro de los módulos de mi proyecto porque están muy acopladas. La rigidez está muy relacionada con el acoplamiento. Por ejemplo si de repente estamos trabajando con enteros y queremos pasar a float en un cálculo en particular, no tengo que cambiar solo los int y los float de esa clase sino que debemos cambiar en un montón de lados, entonces ahí hay algo que está demasiado rígido.
* Fragilidad: esta es que en un cambio del código hace que este no funcione más. No debemos tener miedo al momento de cambiar algo en el código porque debemos estar seguros que en nuestro diseño, hacer un cambio para agregar una determinada funcionalidad o lo que sea, no va a hacer que todo el programa deje de funcionar.
* Inmovilidad: esto quiere decir que el código no se puede reutilizar o que es muy difícil de reutilizarlo, están particular, grande, o acoplado, que yo no lo puedo sacar y llevarlo a otro lado porque no lo podemos reutilizar, entonces es inmóvil.
* Viscosidad: esta nos dice que un código o diseño es viscoso cuando me induce a hacer lo incorrecto, es tan difícil seguir programando bien con buenas prácticas de diseño en ese código, que el código solo me lleva a hacer cosas mal. Entonces el código huele respecto a la viscosidad cuando por ejemplo es más fácil hacer un copy paste que implementar una clase nueva y heredar las cosas, porque como ya estuvo mal hecho el diseño debemos de tratar de ir por ese lado.
* Complejidad innecesaria: cuando existen elementos de diseño que soportan requerimientos que no se necesitan, hacemos por ejemplo una escalada de cinco herencias comenzando por la clase ser vivo y luego bajo hasta la hormiga, donde pase por mamíferos, reptiles, haciéndolo demasiado complejo, cuando lo único que hubiésemos necesitado es la clase insecto y de ahí heredar. Cuando sobre diseño el código y lo hago demasiado complejo estoy teniendo complejidad innecesaria.
* Repetición innecesaria: cuando hay código repetido por muchos lugares, donde los módulos se vuelven difícil de mantener porque para modificar algo tengo que modificarlo en muchos lugares.
* Opacidad: esto es que el código sea difícil de entender cuando se lo hace muy complejo en cuanto a lo que hace al código y poco intuitivo, haciendo que el código sea opaco. Como el código va a ir aumentando o creciendo con el tiempo, si este se lo hace de mala forma, va a ser difícil seguir como se realizó ese crecimiento y poder seguir utilizándolo.

Principios de diseño en POO

\_ Ahora venimos con unas buenas prácticas, algunos principios muy simples, que pueden arreglar siempre nuestro diseño o código:

Principio DRY (Don’t Repeat Yourself): es decir no hacer copy paste o formas muy similares. Si tenemos cosas similares, debemos buscar la forma de abstraer y reutilizar.

Principio KISS (Keep It Simple Stupid): tratar de hacer las cosas simples, no hacerlas complejas al vicio sino tratar de hacerlo lo más simple porque esa complejidad lo único que va a traer es que sea complejo.

Principio YAGNI (You Are Going To Need It): siempre pensar que lo que hacemos lo vamos a necesitar de vuelta o reutilizar.

**Introducción a principios de diseño SOLID**

\_ Los principios de diseño SOLID son un conjunto de principios, no son reglas, no es un framework, ni una tecnología o librería, ni metodología. No me van a decir cómo hacer las cosas, o no son ya cosas hechas, sino que son principios, nosotros vamos a tener que ver si después lo que hacemos los siguen o no a estos principios. Al problema lo podemos resolver de 20 maneras distintas y las 20 maneras pueden seguir los principios SOLID.

\_ El propósito de estos principios son generar código fácil de entender y de mantener. Al seguir estos principios uno hace un buen diseño que va a ser fácil poder agregarle funcionalidad.

\_ SOLID representa cinco de los principios básicos de la POO, estos son:

Single responsability: esto quiere decir que una clase tiene que tener una única responsabilidad, y que si tenemos que modificar algún cambio en el sistema debemos tocar una sola clase, ahora si tenemos que tocar dos clases quiere decir que ese cambio en la funcionalidad que quería modificar esta implementada en dos clases distintas, o que hay una clase que está teniendo dos funcionalidades.

Open closed: nos dice que una clase debe ser abierta para extender su funcionalidad, pero cerrada para ser modificada, uno debe ser capaz de extender el comportamiento de una clase sin modificar su contenido o su código.

Liskov sustitution: este principio de sustitución nos dice que una clase derivada, una clase hija de una abstracta, puede ser reemplazada por cualquier otra que use su clase base sin ser alterada. Por ejemplo si tenemos una clase vehículo y abajo tenemos un montón de implementaciones de distintos vehículos, nosotros deberíamos poder cambiar cualquiera de esas clases derivadas y no debería pasar nada, si llega a pasar algo lo más probable es que hicimos mal la herencia.

Interface segregation: esto quiere decir que cuando uno define una interfaz esta tiene que ser pequeña y que resuelva un problema específico, y no tener interfaces grandes que hagan muchas cosas. Es preferible tener muchas interfaces pequeñas que hagan cosas distintas a tener una interfaz que haga muchas cosas y desperdiciar funcionalidades al usar alguna en particular.

Dependency inversion: este principio nos dice que los módulos o clases de alto nivel no deben depender de los módulos o clases del bajo nivel, sino que ambos deben depender de clases abstractas (abstracciones) y no de clases concretas. Otra forma de verlo, es decir, que las abstracciones no dependen de los detalles sino que los detalles deben depender de las abstracciones.

**Principios de diseño SOLID**

Principio de responsabilidad simple (single responsability)

\_ Este principio nos dice que una clase debe tener uno y solo una razón para cambiar. Esto quiere decir que no tenemos que tener una clase que sepa hacer de todo, sino que se pueda hacer una sola cosa y que sea por la única que cambia, o sea debe tener una sola funcionalidad o responsabilidad. A esto lo podemos ver como que cada clase debe tener una única responsabilidad, o que cada clase debe tener una, y solo una, razón de cambio.

\_ Este principio es aplicable en:

* Métodos
* Clases
* Bibliotecas
* Módulos
* Sistemas

\_ Algunos lineamientos como para ver cuando tenemos que ver si tenemos algún problema con este principio es cuando:

* Cuando es una clase muy larga (300+ líneas), y probablemente esta clase no esté haciendo una sola cosa.
* Cuando los métodos son muy largos (40+ líneas), puede hacer que haga más de una cosa a la vez también.
* Cuando existen muchas dependencias a otros objetos (20+), si una clase de repente pasa a tener muchas clases probablemente este haciendo muchas cosas distintas porque cada una de las otras clases debería hacer una sola cosa.
* Cuando hay baja cohesión, es decir, cuando una clase realiza una sola cosa por lo tanto si es baja la cohesión significa que está haciendo más de una sola cosa.
* Cuando se usan nombres muy genéricos, por ejemplo cuando no sabemos cómo ponerle a la clase y usamos Util, Manager, Process.
* Y cuando esta implementado el anti patrón spagethi code.

\_ Hay algunos argumentos en contra de este principio que son:

* Se generan demasiadas clases.
* Es complicado entender el panorama general de diseño al haber tantas cosas.

\_ Es necesario tener en cuenta que no necesariamente si tenemos muchas clases el código va a ser más complicado, hay casos y casos. Si está bien diseñado y seguimos todos los otros principios, el tener demasiadas clases no será un problema por lo tanto tampoco será problema entender todo.

Ejemplo: hacemos una página personal, en este caso un blog:

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**class Blog{**

**public:**

**string titulo;**

**vector<string> entradas;**

**void add\_entry(string txt){**

**entradas.push\_back(txt)**

**}**

**void save(string file){**

**ofstream ofs(file);**

**for(auto e: entradas){**

**ofs << “Entrada: “ << e << ‘\n’;**

**}**

**};**

**int main(){**

**Blog miBlog;**

**miBlog.titulo = “Personal”;**

**miBlog.add\_entry(“Hoy esta soleado”);**

**miBlog.add\_entry(“Con netflix no me aburro en la cuarentena”);**

**for(auto d: miBlog.entradas){**

**cout<<”Blog: “ << d << ‘\n’;**

**}**

**MiBlog.save(“salida.txt”);**

**return 0;**

**}**

\_ Le podemos agregar persistencia que es que todo lo que guardamos en el vector pueda mantenerlo, para eso hacemos la funcion save.

\_ Ahora suponemos que nos dicen que queremos que aparte del blog tengamos una pagina con historial de proyectos.

\_ Aca nuestra clase blog no esta cumpliendo con el principio de responsabilidad simple porque esta haciendo muchas cosas, en donde aparte de mantner la entradas del blog y gestionandolo, hace lo que se llama persistencia. Para esto debemos modificar la calse blog. Entonces dado a una razon de cambio, si queremos cambiar el lugar donde se van a guardar las cosas, en vez de un archivo txt en una base de datos, tenemos que cambiar la clase Blog, y si tenemos que modificar como es una entrada del blog tenemos que modificar la clase tambien, entonces por dos razones distintas estamos teniendo dos cambios. Por lo tanto deribamos la responsabilidad a otra clase.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**class Blog{**

**public:**

**string titulo;**

**vector<string> entradas;**

**void add\_entry(string txt){**

**entradas.push\_back(txt)**

**}**

**};**

**Class PersisteceManager{**

**public:**

**void save(Blog b, string file){**

**ofstream ofs(file);**

**for(auto e: b.entradas){**

**ofs << “Entrada: “ << e << ‘\n’;**

**}**

**};**

**int main(){**

**Blog miBlog;**

**miBlog.titulo = “Personal”;**

**miBlog.add\_entry(“Hoy esta soleado”);**

**miBlog.add\_entry(“Con netflix no me aburro en la cuarentena”);**

**for(auto d: miBlog.entradas){**

**cout<<”Blog: “ << d << ‘\n’;**

**}**

**PersisteceManager pm;**

**pm.save(miBlog, “salida.txt”);**

**return 0;**

**}**

\_ Por lo tanto, ahora si queremos cambiar que se guarde en una base de datos solo tocamos la clase persisteceManager, y si queremos cambiar el blog tocamos la clase Blog.

Principio abierto/cerrado (open closed)

\_ Este principio según Mayer, Bertand dice que todas las entidades deben ser abiertas a ectension pero cerradas a ser modificadas, esto quiere decir que una vez que hagamos una clase y la terminemos y quede finalizada, si despues queremos hacer que nuestro sistema tenga mas funcionalidades o haga algo nuevo no deberiamos tocar las clases que ya programamos, sino que deberiamos programar clases, modulos o funciones nuevas pero no modificar el codigo que ya esta escrito. La analogia de esto es que no hace falta hacer una cirujia de torax para ponerse un abrigo, podemos pasar de tener frio a calor sin la necesidad de modificarme a mi para tener esa nueva funcionalidad.

\_ Entonces una clase debe ser abierta para extender, pero cerrada para modificar. Uno debe ser capaz de extender el comportamiento de una clase, sin modificar su contenido. Los artefactos a los que este principio aplica son a funciones, clases y modulos, dentro de las funciones estarian los metodos de las clases en donde si queremos extender la funcionalidad de un metodo probablemente este haciendo algo mal y debo modificarlo, ya que si quiero extender la funcionalidad de un metodo debiera estar modificando alugna otra cosa.

\_ Los beneficios que trae trabajar con este principio son:

* Reduce el riesgo de agregar nuevos bugs al codigo, ya que el codigo esta cerrado por lo tanto no modificarlo no introduce problemas nuevos agregando siempre cosas nuevas.
* Reduce el acoplamiento, porque si esa clase que yo no modifico ya hace una cosa y la hace bien, al no modificarla y extenderla mediante otra clase estoy haciendo que sea autonoma y que no este acoplada a otras.
* Obtenemos mas flexibilidad.
* El sistema es mas facil que evolucione y cambie, esto es con respecto al mantenimiento, porque permite ir extendiendolo o agregando funcionalidades sin tener que revisar codigo que se hizo.

\_ Nosotros debemos aplicar este principio cuando hay o se detecta una clase que sera sujeta a cambio, si vemos que hay una clase que tiene que cambiar vemos si podemos aplicar este principio y de alguna forma hacer que esa clase no cambie o cambiarla una sola vez para que nunca mas tenga que cambiar y pase a cumplir con el principio. Tambien debemos aplicar cuando no se puede cambiar una librería de terceros, es decir si tenemos una librería de terceros y queremos que esta tenga una funcionalidad nueva no deberiamos tocar ese modulo sino que deberiamos extender su funcionalidad de alguna manera.

\_ Los argumentos que tiene en contra este principio son que requiere mas esfuerzo diseñar las clases, ya que debemos pensarla desde antes, y de la misma forma uno tiene que tener mas experiencia porque el darse cuenta que se esta violando el principio y hacer para solucionarlo requiere de que uno haya trabajado un poco mas.

Ejemplo: tenemos productos que van a poder tener color y tamaño:

\_ enum es poder enumerar determinadas cosas para poder usarlas despues.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**enum class Color { rojo, verde, azul};**

**enum class Size { chico, mediano, grande};**

**struct Product {**

**string name;**

**Color color;**

**Size size;**

**};**

**int main() {**

**Product manzana{"Manzana", Color::rojo, Size::chico};**

**Product Sandia{"Sandia", Color::verde, Size::grande};**

**Product Pomelo{"Pomelo", Color::rojo, Size::mediano};**

**vector<Product \*> productos {&manzana, &sandia, &pomelo};**

**for(auto p: productos){**

**cout<<p->name<<endl;**

**}**

**return 0;**

**}**

\_ Ahora viene nuestro jefe y nos dice que queremos hacer un filtrado de los productos, y poder crear una lista de productos filtrados, por ejemplo para que venga el cliente y filtre todos los productos que son rojos. Para eso creamos una clase nueva, donde la funcion byColor le pasamos la lista de los productos y un color, y nos devuelve una nueva lista de productos.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**enum class Color { rojo, verde, azul};**

**enum class Size { chico, mediano, grande};**

**struct Product {**

**string name;**

**Color color;**

**Size ssize;**

**};**

**class ProductFilter {**

**vector<Product \*> byColor(vector<Product\*> items, Color Color) {**

**vector<Product \*> res;**

**for(auto i: items) {**

**if(i->color == color) {**

**res.push\_back(i);**

**}**

**}**

**return res;**

**}**

**};**

**int main() {**

**Product manzana{"Manzana", Color::rojo, Size::chico};**

**Product Sandia{"Sandia", Color::verde, Size::grande};**

**Product Pomelo{"Pomelo", Color::rojo, Size::mediano};**

**vector<Product \*> productos {&manzana, &sandia, &pomelo};**

**for(auto p: productos) {**

**cout<<p->name<<endl;**

**}**

**cout<<endl<<endl;**

**ProductFilter pf;**

**vector<Product \*> rojos;**

**pf.byColor(productos, Color::rojo);**

**for(auto p: rojos) {**

**cout<<"Rojo: "<<p->name<<endl;**

**}**

**return 0;**

**}**

\_ Ahora si nuestro jefe nos pide lo mismo pero para los tamaños. Podemos hacer en la misma funcion ProductFilter una funcion que se llame bySize pero estariamos violando el principio, ya que una vez que esta implementado el filtro por color si la modificamos estamos cometiendo lo contrario al principio. Para solucionar esto vamos a usar el patron de especificacion, en donde lo que nos va a permitir es poder hacer que sea muy facil extender nuestro sistema de filtros sin la necesidad de modificar el codigo. Para esto creamos una clase abstarcta que va tener templates, dentro de la clase vamos a tener un metodo virtual que devuelve un bool para decir si se satisface una determinada condicion. Lo otro que creamos ahora va a ser la clase para los filtros, que tambien va a ser abstracta con templates, y va a tener el metodo que va a filtrar en donde va a filtrar según la especificacion. Ahora hacemos la clase BetterFilter que hereda de la clase Filter, donde implementamos la funcion de esta ultima. Creamos la clase para filtrar los rojos, donde el constructor espera un color, e implementamos la funcion Satisfied en donde si esta bien tira un itemque es si el color es igual al que tenemos.

\_ De nuevo, en BetterFilter implementamos la funcion filter, donde le pasamos una lista de productos igual que antes solo que en vez de pasarle un color le pasamos una especificacion y lo que vemos es que esa especificacion se cumple.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**enum class Color { rojo, verde, azul};**

**enum class Size { chico, mediano, grande};**

**struct Product {**

**string name;**

**Color color;**

**Size ssize;**

**};**

**template<typename T> class Specification {**

**public:**

**virtual bool isSatisfied(T\* item) = 0;**

**};**

**template<typename T> class Filter {**

**public:**

**virtual vector<T\*> filter(vector<T\*> items, Specification<T> spec) = 0;**

**};**

**class ColorSpec: public Specification<Product> {**

**Color color;**

**public:**

**ColorSpec(Color color) : color(color) {}**

**bool isSatisfied(Product \*item) override {**

**return item->color = color;**

**}**

**};**

**class BetterFilter: Filter<Product> {**

**public:**

**vector<Product\*> filter(vector<Product\*> items, Specification<Product> spec) override {**

**vector<Product\*> res;**

**for(auto i: items) {**

**if(spec.isSatisfied(i))**

**res.push\_back(i);**

**}**

**return res;**

**}**

**};**

**int main() {**

**Product manzana{"Manzana", Color::rojo, Size::chico};**

**Product Sandia{"Sandia", Color::verde, Size::grande};**

**Product Pomelo{"Pomelo", Color::rojo, Size::mediano};**

**vector<Product \*> productos {&manzana, &sandia, &pomelo};**

**for(auto p: productos) {**

**cout<<p->name<<endl;**

**}**

**cout<<endl<<endl;**

**BetterFilter bf;**

**ColorSpec sp\_rojo(Color:: rojo);**

**vector<Product \*> rojos;**

**rojos = bf.filter(productos, sp\_rojo);**

**for(auto p: rojos) {**

**cout<<"Rojo: "<<p->name<<endl;**

**}**

**return 0;**

**}**

\_ Entonces el funcionamiento es exactamente el mismo nada mas que ahora tenemos un BetterFilter y una especificacion de color. En donde si ahora viene mi jefe y me dice que ahora se tiene que poder filtrar por tamaño, lo que hacemos es no modificar la clase Filter y tampoco la ColorSpec, sino que creamos una nueva clase SizeSpec para filtrar solo por size. Generamos codigo nuevo en una clase nueva sin tocar ninguna de las anteriores.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**enum class Color { rojo, verde, azul};**

**enum class Size { chico, mediano, grande};**

**struct Product {**

**string name;**

**Color color;**

**Size ssize;**

**};**

**template<typename T> class Specification {**

**public:**

**virtual bool isSatisfied(T\* item) = 0;**

**};**

**template<typename T> class Filter {**

**public:**

**virtual vector<T\*> filter(vector<T\*> items, Specification<T> spec) = 0;**

**};**

**class ColorSpec: public Specification<Product> {**

**Color color;**

**public:**

**ColorSpec(Color color) : color(color) {}**

**bool isSatisfied(Product \*item) override {**

**return item->color = color;**

**}**

**};**

**class SizeSpec: public Specification<Product> {**

**Size size;**

**public:**

**SizeSpec(Size size) : size(size) {}**

**private:**

**bool isSatisfied(Product \*item) override {**

**return item->size = size;**

**}**

**};**

**class BetterFilter: Filter<Product> {**

**public:**

**vector<Product\*> filter(vector<Product\*> items, Specification<Product> spec) override {**

**vector<Product\*> res;**

**for(auto i: items) {**

**if(spec.isSatisfied(i))**

**res.push\_back(i);**

**}**

**return res;**

**}**

**};**

**int main() {**

**Product manzana{"Manzana", Color::rojo, Size::chico};**

**Product Sandia{"Sandia", Color::verde, Size::grande};**

**Product Pomelo{"Pomelo", Color::rojo, Size::mediano};**

**vector<Product \*> productos {&manzana, &sandia, &pomelo};**

**for(auto p: productos) {**

**cout<<p->name<<endl;**

**}**

**cout<<endl<<endl;**

**BetterFilter bf;**

**ColorSpec sp\_rojo(Color:: rojo);**

**SizeSpec sp\_chicos(Size:: chico);**

**vector<Product \*> rojos;**

**vector<Product \*> chicos;**

**rojos = bf.filter(productos, sp\_rojo);**

**chicos = bf.filter(productos, sp\_chicos);**

**for(auto p: chicos) {**

**cout<<"Chico: "<<p->name<<endl;**

**}**

**return 0;**

**}**

\_ Y si ahora el jefe nos dice que debemos filtrar por color y por tamañio, entonces como tenemos ColorSpec y SizeSpec creamos una nueva clase AndSpec de tipo template con dos especificaciones. En cuanto el diseño es lo mas potente que se pudo hacer.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**enum class Color { rojo, verde, azul};**

**enum class Size { chico, mediano, grande};**

**struct Product {**

**string name;**

**Color color;**

**Size ssize;**

**};**

**template<typename T> class Specification {**

**public:**

**virtual bool isSatisfied(T\* item) = 0;**

**};**

**template<typename T> class Filter {**

**public:**

**virtual vector<T\*> filter(vector<T\*> items, Specification<T> spec) = 0;**

**};**

**class ColorSpec: public Specification<Product> {**

**Color color;**

**public:**

**ColorSpec(Color color) : color(color) {}**

**bool isSatisfied(Product \*item) override{**

**return item->color = color;**

**}**

**};**

**class SizeSpec: public Specification<Product> {**

**Size size;**

**public:**

**SizeSpec(Size size) : size(size) {}**

**private:**

**bool isSatisfied(Product \*item) override {**

**return item->size = size;**

**}**

**};**

**class BetterFilter: Filter<Product> {**

**public:**

**vector<Product\*> filter(vector<Product\*> items, Specification<Product> spec) override {**

**vector<Product\*> res;**

**for(auto i: items){**

**if(spec.isSatisfied(i))**

**res.push\_back(i);**

**}**

**return res;**

**}**

**};**

**template<typename T> class AndSpec: public Specification<T> {**

**Specification<T>& primera;**

**Specification<T>& segunda;**

**public:**

**AndSpec(Specification<T> &primera, Specification<T> &segunda) : primera(primera), segunda(segunda) {}**

**bool isSatisfied(T \*item) override {**

**return primera.isSatisfied(item) && segunda.isSatisfied(item);**

**}**

**}**

**int main() {**

**Product manzana{"Manzana", Color::rojo, Size::chico};**

**Product frutilla{"Frutilla", Color::rojo, Size::chico};**

**Product Sandia{"Sandia", Color::verde, Size::grande};**

**Product Pomelo{"Pomelo", Color::rojo, Size::mediano};**

**vector<Product \*> productos {&manzana, &sandia, &pomelo};**

**for(auto p: productos){**

**cout<<p->name<<endl;**

**}**

**cout<<endl<<endl;**

**BetterFilter bf;**

**ColorSpec sp\_rojo(Color:: rojo);**

**SizeSpec sp\_chicos(Size:: chico);**

**AndSpec<productos> sp\_rojoYchico(sp\_rojo, sp\_chicos);**

**vector<Product \*> rojos;**

**vector<Product \*> chicos;**

**rojos = bf.filter(productos, sp\_rojo);**

**chicos = bf.filter(productos, sp\_chicos);**

**ambos = bf.filter(productos, sp\_rojoYchico);**

**for(auto p: ambos){**

**cout<<"Rojo y chico: "<<p->name<<endl;**

**}**

**return 0;**

**}**

Principio de sustitucion de Liskov (Liskov sustitution)

\_ Este principio fue anunciado por Barbara Liskov en donde dice que los requisitos de un subtipo, si el subtipo tiene un metodo X y es una propiedad probable del objeto X entonces otra propiedad Y debe ser verdadera si el objeto depende de S que es parte de un subtipo. Haciendo una analogia, basicamente lo que este principio nos dice es que si utilizamos herencia por mas que actue y flote como un pato (bato de juguete), si necesita baterias es probable que hayamos hecho la herencia mal, donde el tipo de abstraccion que quisimos poner esta mal.

\_ Lo que viene a decir este principio es que una clase derivada puede ser reemplazada por cualquier otra que use su clase base sin alterar su correcto funcionamiento. Esto es el principio del polimorfismo. Si tenemos una clase abstracta pato, en donde de esa clase heredo y creo el pato de agua, pato frances y pato de goma, probablemente uno de los metodos sea hacerCuack, Volar, y en articular el pato de goma va a tener que cambiar baterias y resulta que hacerCuak van a poder hacer los tres, y el Volar solo los dos patos vivos. Entonces aca estamos violando el principio porque no podemos sustituir distintas clases por mas que tenagn una misma clase base, y esto trae soluciones feas por ejemplo que el pato de goma de repente ante una llamada de volar, porque tiene el metodo, no haga nada, otra seria que devuelva falso, etc, pero sigue siendo algo que no queda bien.

\_ Al artefacto al que solo afecta este principio es a las clases, ya que como el problema viene de la herencia, solo las clases heredan. Los beneficios que trae trabajar con este principio son:

* Flexibilidad, al estar implementando bien herencia vamos a poder usar polimorfismo y sustituir cualquiera de los objetos por el que corresponde, y verificamos de que funciona bien.
* Esto deriva en que el mantenimiento sea mas simple, y cambiar y elvlucionar el sistema sea mas facil.

\_ Nosotros debemos aplicar este principio cuando se quiere extender el funcionamiento usando clases derivadas (herencia) sin tocar el codigo base. Tambien cuando existan clases que comparten el mismo comportamiento ya que probablemente usamos herencia. Ademas cuando aplicamos el principio open closed.

Ejemplo: suponemos que tenemos un sistema que tiene una clase rectangulo con las propiedades del rectangulo que son protected para que la clase hijo pueda utilizarlos. Creamos una funcion test para verificar que nuestro rectangulo funciona. Hacemos cambios como calcular y mostrar el area, y luego de cambiar mediante los set los valores de ancho y altura.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**class Rectangulo {**

**protected:**

**int width, height;**

**public:**

**Rectangulo(int width, int height): width(width), height(height){}**

**int getWidth() const{**

**return width;**

**}**

**void setWidth(int width){**

**Rectangulo::width = width;**

**}**

**int getHeight() const{**

**return height;**

**}**

**void setHeight(int width){**

**Rectangulo::height = height;**

**}**

**int area(){**

**return width \* height;**

**}**

**};**

**void test(Rectangulo &r){**

**r.setWidth(10);**

**int w = r.getWidth();**

**r.setHeight(5);**

**int h = r.getHeight();**

**cout<<w<<" "<<h<<endl;**

**cout<<r.area()<<" "<<w\*h<<endl;**

**}**

**int main(){**

**Rectangulo r(3,5);**

**test(r);**

**return 0;**

**}**

\_ Ahora suponemos que nuestro jefe nos dice que tambien quiere un cuadrado. Creamos una clase cuadrado que va a heredar de la clase rectangulo, en el constructor seteamos solamente el ancho porque el alto no es necesario. Deberiamos reescribir los sets porque seguramente quicieramos cambiar el alto y ancho, por eso en la clase rectangulo los hacemos virtuales. Entonces volviendo a la clase cuadrado seteamos entonces las variables y con esto garnatizamos de que si queremos cambiar alto o ancho siempre se mantenga cuadrado.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**class Rectangulo {**

**protected:**

**int width, height;**

**public:**

**Rectangulo(int width, int height): width(width), height(height){}**

**int getWidth() const{**

**return width;**

**}**

**virtual void setWidth(int width){**

**Rectangulo::width = width;**

**}**

**int getHeight() const{**

**return height;**

**}**

**virtual void setHeight(int width){**

**Rectangulo::height = height;**

**}**

**int area(){**

**return width \* height;**

**}**

**};**

**class Cuadrado : public Rectangulo{**

**public:**

**Cuadrado(int width): Rectangulo(width,width){}**

**void setWidth(int width) override{**

**this->height = this->width = width;**

**}**

**void setHeight(int height) override{**

**this->height = this->width = height;**

**}**

**};**

**void test(Rectangulo &r){**

**r.setWidth(10);**

**int w = r.getWidth();**

**r.setHeight(5);**

**int h = r.getHeight();**

**cout<<w<<" "<<h<<endl;**

**cout<<r.area()<<" "<<w\*h<<endl;**

**}**

**int main(){**

**Rectangulo r(3,5);**

**Cuadrado c(5);**

**test(r);**

**test(c);**

**return 0;**

**}**

\_ Al probar vemos que no funciona como esperariamos, ya que la funcion test testea rectangulos, y nosotros tenemos ademas cuadrados, por lo tanto estariamos violando el principio de Liskov ya que la clase cuadrado que hereda de rectangulo no se comporta de la misma manera que la calse rectangulo. Una de las formas de solucionar esto es teniendo una clase padre que se llame figura geometrica y que de esta hereden rectangulo y cuadrado en donde pasa a ser otro el comportamiento, y la otra forma seria con el patron factory en donde creamos una clase construya cuadrados y construya rectangulos. Este no se desarrolla pero seria algo como lo siguiente:

**class Rect\_Factory {**

**Rectangulo new\_Rectangulo(int w, int h);**

**Rectangulo new\_Cuadrado(int w);**

**};**

Principio de segrecacion de la interfaz (interface segregation)

\_ Este principio nos dice que ningun cliente deberia ser forzado a depender de metodos que no vaya a usar, de otra forma una interfaz o clase abstarcta no deberia forzar a sus hijos la implementacion de un metodo que ese hijo no utiliza, o no pueda o sepa implementar. Entonces un cliente, que es la calse hijo, no debe ser forzado a implementar interfaces o metodos que no necesite. Esto va a hacer que definamos interfaces pequeñas que resuelvan un problema especifico (role iterface), en lugar de tener interfaces grandes que hagan muchas cosas (head interface) y que cada una de las clases que implementan esta interface pasen a ser parte de la interface, que no esta bueno.

\_ Los artefactos a los cuales aplica este principio son las clases y las interfaces, porque la interface es la que va a definir como habra que implementarce y las clases van a implementarlo. Entonces si una clase tiene que implementar algo a la fuerza estamos violando este principio. Los veneficios que trae trabajar con este principio son:

* Bajo acoplamiento, porque es una clase padre que implementa muchas cosas va a tener muchas clases hijas y todas ellas van a implementar partes de esa clase entonces esto va a conllevar de que haya mucho acomplamiento donde muchas clases dependen de una sola, en cambio si segregamos y hacemos interfaces pequeñas una va a depender de dos otra de tres o una y asi seria, donde esta solucion seria mediante herencia multiple, implemento dos o tres clases padres y las hijas implementan solo una o dos de las clases padres.
* Alta cohesion, ya que al hacer mucahs clases padre, cada clase padre se dedica a hacer una sola cosa.
* Ademas tendriamos codigo mas facil de cambiar y mantener.

\_ Nosotros debemos aplicar este principio cuando existan clases con metodos vacios o que devuelvan valores por defecto. Cuadno existan clases con metodos que devuelvan excepciones tipo NotImplemented. Y ademas cuando un cliente usa solo algunos metodos de una clase.

Ejemplo: vamos a tener un sistema de loggin. Queremos una clase que nos permita guardar logs o implementar un log. Creamos una clase Log con las funciones de conectarse y desconectarse de la base de datos y por ultimo la funcion mas importante de loguear. Una de las implementaciones seria FileLog que implementa todos los metodos de Log, en donde abre, cierra, y utiliza el archivo.

**#include <iostream>**

**#include <vector>**

**#include <fstream>**

**using namespace std;**

**class Log{**

**public:**

**virtual void connect() = 0;**

**virtual void disconnect() = 0;**

**virtual void log(string txt) = 0;**

**};**

**class FileLog: public Log{**

**ofstream salida;**

**public:**

**void connect() override{**

**salida.open("log.txt");**

**}**

**void disconnect() override{**

**salida.close();**

**}**

**void log(string txt) override{**

**salida << "LogArchivo: " << txt << endl;**

**}**

**};**

**int main(){**

**FileLog fl;**

**fl.connect();**

**fl.log("prueba1");**

**fl.log("prueba2");**

**fl.disconnect();**

**return 0;**

**}**

\_ Esta seria una opcion. Pero a la vez queremos que el sistema loguee tambien en pantalla, en donde creamos una clase ConsoleLog, en donde lo usamos uno de los tres metodos que hereda, en donde si no los implementamos no compila. Una forma de solucionarlo es tirando excepciones en los metodos no usados, podriamos no hacer nada, o con un cout mostrar un aviso, pero cualquiera de estas soluciones nos trae el problema de que el resultado no va a ser el esperado.

**class ConsoleLog: public Log{**

**public:**

**void connect() override{**

**cout << "No usado" <<endl;**

**}**

**void disconnect() override{**

**throw 404;**

**}**

**void log(string txt) override{**

**cout<<"Log: " << txt << endl;**

**} };**

\_ Una de las opciones para solucionar esto es segregar la clase Log. Entoneces esta clase para lo unico que se utilice sea para hacer un log y despues creamos otra clase abstracta que se llame Connection que implemente los metodos virtuales de connect y disconnect. Donde ahora la clase FileLog hereda de las dos clases padre y la clase ConsoleLog solo implementa la de Log. Entonces asi segregamos las interfaces.

**#include <iostream>**

**#include <vector>**

**#include <fstream>**

**using namespace std;**

**class Log{**

**public:**

**virtual void log(string txt) = 0;**

**};**

**class Connection{**

**public:**

**virtual void connect() = 0;**

**virtual void disconnect() = 0;**

**};**

**class FileLog: public Log, public Connection{**

**ofstream salida;**

**public:**

**void connect() override{**

**salida.open("log.txt");**

**}**

**void disconnect() override{**

**salida.close();**

**}**

**void log(string txt) override{**

**salida << "LogArchivo: " << txt << endl;**

**}**

**};**

**class ConsoleLog: public Log{**

**public:**

**void log(string txt) override{**

**cout<<"Log: " << txt << endl;**

**}**

**};**

**int main(){**

**FileLog fl;**

**ConsoleLog cl;**

**fl.connect();**

**fl.log("prueba1");**

**fl.log("prueba2");**

**fl.disconnect();**

**cl.log("prueba1");**

**cl.log("Prueba2");**

**return 0;**

**}**

Principio de inversion de dependencia (Dependency inversion)

\_ Este principio nos dice que los modulos de alto nivel no deberian depender de los modulos de bajo nivel, sino que ambos deben depender de abstracciones. La analogia seria que nosotros no soldariamos una lampara a un enchufe sino que tenemos un cable para eso. Las abstracciones no deben depender de los detalles sino que los detalles deben depender de las abstracciones. Cuando una abstraccion se pone en los detalles deja de ser una abstraccion.

\_ Los artefactos a los cuales se aplica este principio son los modulos y las clases. Los beneficios que trae trabajar con este principio son:

* Bajo acoplamiento, porque la calse de alto nivel no va a depender de las calses de bajo nivel.
* Ayuda a la testeabilidad, porque podemos testear cada una de las clases por separado sin que haya dependencia.
* Nos da mas flexibilidad.

\_ Debemos aplicar este principio cuando hace falta desacoplar piezas de software que pueden cambiar en el futuro, por lo tanto deberia ser casi siempre. Ademas cuando el nivel de acoplamiento en el codigo es alto.

\_ los argumentos en contra de este principio son:

* Requiere mayor experiencia, porque hay que darse cuenta cuando uno esta equivocado.
* Requiere mas esfuerzo el diseñar los artefactos del software.
* Se empiezan a crear muchas clases e interfaces.
* Es complicado entender el panorama general del diseño, porque por ahí queda mas rebuscada la solucion.

Ejemplo: hacemos un programa que va a guardar personas, en este caso desarrolladores, programadores, y testers. Para eso creamos una clase Desarrollador, Programador y Tester. Despues tenemos una clase Manaer que va a ser el encargado de gestionar este tipo de trabajo.

**#include <iostream>**

**#include <vector>**

**#include <fstream>**

**using namespace std;**

**class Desarrollador{**

**public:**

**Desarrollador(){**

**cout<<"Nuevo desarrollador"<<endl;**

**}**

**};**

**class Programador{**

**public:**

**Programador(){**

**cout<<"Nuevo Programador"<<endl;**

**}**

**};**

**class Tester{**

**public:**

**Tester(){**

**cout<<"Nuevo Tester"<<endl;**

**}**

**};**

**class Manager{**

**public:**

**vector<Desarrollador> desa;**

**vector<Programador> prog;**

**vector<Tester> test;**

**public:**

**void addDesarrollador(Desarrollador d){**

**desa.push\_back(d);**

**}**

**void addProgramador(Programador p){**

**prog.push\_back(p);**

**}**

**void addTester(Tester t){**

**test.push\_back(t);**

**}**

**};**

**int main(){**

**Manager m;**

**m.addDesarrollador(Desarrollador());**

**m.addProgramador(Programador());**

**m.addTester(Tester());**

**m.addDesarrollador(Desarrollador());**

**m.addDesarrollador(Desarrollador());**

**m.addProgramador(Programador());**

**return 0;**

**}**

\_ El programa funciona correctamente, pero tenemos un problema que es que la clase Manager que deberia ser de mas alto nivel o sea mas general, depende de las clases mas particulares, ya que si o si debe conocer las otras tipos de clases y no es lo que estariamos esperando. Entonces debemos romper ese ciclo de dependencias. Par esto usamos otra herramienta como por ejemplo la herencia. Creamos una clase abstracta Empleado, en donde todas las clases son dependientes de esta (Manager no). Y despues modificamos manager en donde en vez de tener tres listas vamos a tener una sola que es empleado. Y se supone que la funcionalidad es la misma.

**#include <iostream>**

**#include <vector>**

**#include <fstream>**

**using namespace std;**

**class Empleado{**

**};**

**class Desarrollador: public Empleado{**

**public:**

**Desarrollador(){**

**cout<<"Nuevo desarrollador"<<endl;**

**}**

**};**

**class Programador: public Empleado{**

**public:**

**Programador(){**

**cout<<"Nuevo Programador"<<endl;**

**}**

**};**

**class Tester: public Empleado{**

**public:**

**Tester(){**

**cout<<"Nuevo Tester"<<endl;**

**}**

**};**

**class Manager{**

**vector<Empleado> empl;**

**public:**

**void addEmpleado(Empleado e){**

**empl.push\_back(e);**

**}**

**};**

**int main(){**

**Manager m;**

**m.addEmpleado(Desarrollador());**

**m.addEmpleado(Programador());**

**m.addEmpleado(Tester());**

**m.addEmpleado(Desarrollador());**

**m.addEmpleado(Desarrollador());**

**m.addEmpleado(Programador());**

**return 0;**

**}**

\_ Creamos una clase abstracta, donde las clases de bajo nivel pasan a heredar de ella y las clases de alto nivel, en vez de conocer a las de bajo nivel, conoce a esa abstraccion nada mas. Entonces de esta forma rompemos el ciclo de que las clases de alto nivel dependan de las de bajo nivel, donde las clases de bajo nivel tambien dependen de esa abstraccion. Y permite que esa clase de alto nivel utilice las clases de alto nivel pero sin necesidad de conocerlas. Por lo tanto cumplimos con lo que dicta este principio.

Integracion continua (github)

\_ Es una herramienta de desarrollo, en donde cada vez que se hace un commit y un push, el servidor en el caso de github que es action va a correr ciertas acciones sobre su codigo.

Linter (Lint): es un programa que hace analisis estatico de codigo, se fija si el codigo esta bien por ejemplo el formato, etc, pero solo lo que esta escrito. En el archivo CPPLINT filtramos depuraciones, por defecto saca lo siguiente:

* Whitespaces: si hay espacios de mas entre una linea de codigo y otra, deja poner hasta dos espacios entre lineas de codigo, si hay mas lo saca.
* Legal/copyright: se saca que cada archivo .c o .h tiene que tener la leyenda de copyright arriba.
* Build/include\_order: se saca que los include tengan que estar ordenados por orden alfabetico, para que esten ordenados primero las librerias externas y despues los propios.
* Runtime/references
* Hacemos excluir los archivos que son de librerias y otras cosas.

**Patrones de diseño**

\_ Diseñar software OO es difícil y diseñarlo de manera reusable es incluso más difícil. Descubrir objetos, clasificarlos, hacerlo a la granularidad correcta, definir interfaces, herencia, relaciones entre ellos, etc. Todo esto puede ser un trabajo muy arduo, por lo que los Diseñadores experto, utilizan algunos trucos.  
Básicamente, aplican principios de diseño y re-usan soluciones probadas que han aplicado anteriormente a problemas similares. Estas soluciones que se usan y re-usan, son patrones recurrentes de estructuras y relaciones entre clases y objetos que resuelven el problema de diseño específicos.  
Si ya se conocen estos patrones de diseño, se pueden aplicar directamente sin necesidad de redescubrirlos.

Patrones creacionales

Builder: este patron permite crear objetos complejos, objetos muy grandes o que su construccion es muy compleja, paso a paso, sin necesidad de hacerlo nosotros en nuestro codigo. Permite crear diferentes representaciones del objeto usando el mismo codigo para construirlo.

* Motivacion: generalmente cuando un objeto es muy complejo requiere muchos datos para inicializarce en el constructor u otros objetos que le haga falta para poder ser construido, y empieza a crearce codigo bastante feo y dificil de mantener.
* Intencion: es separar la construccion de los objetos complejos de su representacion.
* Aplicabilidad: se separan distintos objetos cada uno dedicado a algo distinto y el algoritmo para crear el objeto complejo, debe ser independiente del objeto y de como se construye. Y todo el proceso de construccion va a permitir aparte que pueda haber distintas representaciones del objeto (permitimos que siga existiendo la herencia).
* Estructura: en este patron tenemos, nosotros somos el cliente que usamos todo el patron builder, el patron builder tiene un director que es el encargado de construir el objeto (que pasos seguir para construir un determinado objeto), es quien se encarga de ejecutar cada proceso de instanciamiento, despues hay una clase bastracta o interfaz que se llama builder que esta va a ser la que va a definir que cosas va a poder hacer el constructor, esta crea partes del producto. Despues vamos a tener los concreteBuilder que son las clases que van a implementar todos los metodos de la clase abstracta y van a hacer realmente la construccion del objeto, es decir, implementa la interfaz builder y sabe construir el producto concreto, y el producto de los concreteBuilder es el objeto en si.

\_ Un ejemplo simple seria tener una clase Casa donde definimos parametros como puertas, ventanas, garage, pileta, arboles, habitaciones, pisos, baños, etc. Luego creamos el constructor donde en si mismo definimos si tiene o no, o que cantidad hay de los parametros definidos antes, en donde en esto se hace bastante compleja la lectura o se acumulan false, true, 1s o 0s cuando en si no son los datos que deberian tener los objetos, por ende queda muy compleja su construccion. O tambien cuando hacemos un constructor simple y hacemos los adds, por ejemplo add\_Puerta, add\_Ventana y asi se van creando como objetos externos, y el problema de esto es que viola el principio de responsabilidades simples.

Singleton: este es uno de los mas simples en cuanto a estructura y uno de los mas controversiales en cuanto a que se lo puede considerar un antipatron. Este patron permite asegurarse que una clase va a tener solamente una instancia (podemos llamar todas las veces que queramos a esa clase y solamente voy a obtener el mismo objeto y no otro), y a la vez provee una forma de acceder a esta instancia desde cualquier lado.

* Motivacion: muchas veces queremos que haya una clase que tenga solo una instancia de un objeto porque este objeto utiliza algun recurso unico y no queremos que haya varias cosas usando este recurso, por ejemplo, una conexión a una base de datos debe ser unica, un archivo de logs debe ser abierto solo una vez, el acceso a un conversor A/D debe ser uno solo, etc.
* Aplicabilidad: este debe ser utilizado cuando solo debe existir una unica instancia de una clase, y esta debe ser accesible desde varios varios clientes desde cualquier lugar.
* Estructura: es solamente una clase que tiene una variable estatica que es su instancia, la variable al ser estatica va a ser instanciada una sola vez y tiene todos los datos que correspondan a su uso. Despues tiene un miembro estatico que es el que va a debolver esa instancia, y ademas va a tener todos los metodos del singleton para poder hacer su funcionalidad. Entonces tenemos un solo participante que es el singleton que tiene dos funciones, una es definir una instancia de la clase y dejar acceder a los clientes a esta unica instancia, y la otra es que puede ser responsable de crear su propia instancia.

Factory Method:

Abstract Factory:

Prototype:

**Protocolo IP (Internet Protocol)**

\_ Este protocolo identifica a cada dispositivo de internet mediante un único número. Hasta hace tiempo y en la actualidad se usa IPv4 (versión 4 del protocolo) y ahora salió el IPv6. Este numero identificador son 4 números que van entre 0 y 256. No puede haber dos dispositivos con la misma IP en internet, por lo tanto se identifican unívocamente a los dispositivos en internet. Pueden haber IPs publicas y privadas, es decir, que los dispositivos que están conectados a internet tienen una IP publica, pero no toda cosa que esta conectada a internet tiene una IP, porque si estamos en una IP privada, por ejemplo en nuestra casa al estar conectados a internet, quien tiene la IP de internet no es el teléfono o la PC sino el router o modem del proveedor, donde el router hace de enrutador, es decir, un dispositivo que genera de alguna manera una internet privada en nuestra casa y enruta los paquetes que los dispositivos internos hacen a través de esa IP publica, entonces podemos tener en nuestra casa 50 dispositivos conectados a internet y ocupar una sola IP. Podemos tener 2564 = 4294296 de IPs o dispositivos conectados consumiendo IPv4 privados y no IPv4 públicas (en realidad es un poco menos porque hay un rango de combinaciones IP que están restringidas, que son el rango privado).

**Protocolo TCP (Transfer Control Protocol)**

\_ Este protocolo permite comunicar una aplicación con otra, un cliente con un servidor, en donde como dispositivos se comunican a través de IP, pero cuando tal “cosa” del cliente se quiere comunicar con tal “cosa” del servidor se hace a través del TCP. Este protocolo se basa en un sistema de puertos, que lo que va a hacer es, que cuando nos queramos comunicar, un servidor pueda prestar varios servicios, un servidor puede ser un servidor web, servicio de streaming de música, servicio de FTP, servicio de VPN, es decir, para poder brindar todos esos servicios al mismo tiempo, TCP hace que cada uno de esos servicios escuche en un puerto distinto. Los puertos están identificados con un numero que va entre 0 a 65536 (216), entonces si por ejemplo queremos crear un software que sea un servidor podemos escuchar en cualquiera de todos esos puertos. Los puertos mas comunes son 80 para lo que son las paginas web, por defecto en el navegador si no ponemos el puerto, este se va a estar comunicando en el puerto 80, y si ponemos por defecto HTTPS va a ir al puerto 443, que es lo mismo que el puerto 80 pero con una capa de encriptación.

\_ Este protocolo esta previsto como un protocolo cliente - servidor, en donde tiene que haber un servidor que esta escuchando en un determinado puerto, y el cliente va a ir a esa IP en ese puerto y se va a poder comunicar. Entonces si ponemos un servidor web, tenemos que poner el puerto 80 abierto y darle a alguien la IP para que alguien se comunique a mi dispositivo a través de la IP al puerto 80 a la aplicación que tenemos de servidor.

**Protocolo HTTP (Hypertext Transfer Protocol)**

\_ Este es un protocolo de capa de aplicación del modelo OSI por la World Wide Web desde 1990. Se creo para transferir hipertexto, es decir, un texto con un enlace en donde al hacerle click cargaba otro texto. Este protocolo esta basado en un request y un response, es decir, en un pedido y en una respuesta, entonces si mando un pedido me van a responder con una cosa y así. Este se utiliza para el desarrollo web, es muy difícil hacer un sistema web sin conocer por lo menos las bondades que ofrece este protocolo.

URL (Uniform Resource Locator): el protocolo HTTP utiliza lo que se llama URL, es decir, una forma única de localizar recursos básicamente. Entonces si queremos identificar unívocamente una pagina lo podemos hacer a través de una URL. La URL no está identificada únicamente a paginas sino que también podemos identificar cualquier cosa como por ejemplo una base de datos MySQL. Las partes son las siguientes:

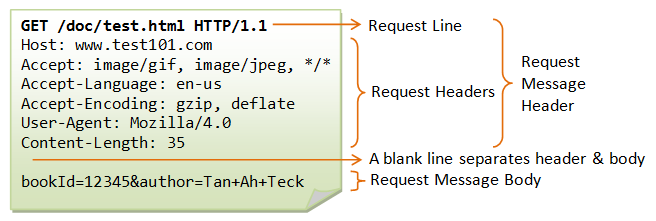
http:// www.example.com:80/blog/test/post?cant=4&pag=5

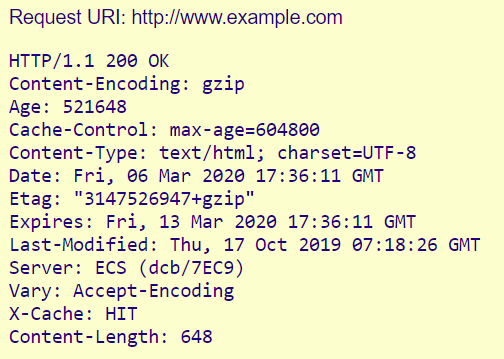
* Protocolo: acá hacemos la comunicación HTTP, FTP, HTTPS.
* Hostname: es el identificador del servidor al cual me quiero conectar, entonces acá podemos poner el nombre del dominio (DNS) o la IP del servidor.
* Puerto: por lo general esta parte no se ve o no se coloca ya que al poner HTTP que es 80 o HTTPS que es 443 que son los puertos estándares, los podemos obviar, por eso no hace falta poner el puerto.
* Ruto de directo: es como si fuera la ruta del directorio de nuestra PC.
* Recurso: haciendo referencia a que cosa quiero de ese servidor.
* Variable: en el ejemplo pide 4 post (query).
* Valor: en el ejemplo pide 4 post de la página 5 (query).

Verbos: a la URL le podemos hacer varios pedidos y esto va a depender de que verbos usemos o que métodos del protocolo HTTP usemos, y estos son:

* GET: sirve para pedir un recurso.
* POST: sirve para enviar datos al servidor, cuando quiero enviar datos como mail y contraseña para loguearme lo más probable es que utilice un POST.
* PUT: sirve para enviar datos y guardar en el servidor, como por ejemplo cuando estamos llenando un formulario de cuenta de un nuevo usuario.
* DELETE: sirve para borrar un dato del servidor.
* HEAD: sirve para pedir la cabecera de un recurso.
* TRACE
* OPTIONS
* CONNECT

Headers (cabecera): el navegador traduce una URL en una cabecera y realiza un request. El request es texto plano que se manda y está formado de esta forma:



\_ Ahora si el request está bien formado, el servidor va a responder con cierta información:

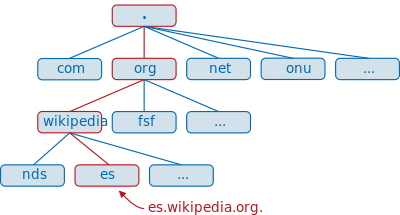
Códigos de error: el servidor informa un código de error en su respuesta, y con estos podemos como programadores saber e informar distintos errores al navegador:

* 1xx: errores de información.
* 2xx: significan que todo estuvo bien pero puede haber más detalle dentro de ese todo bien.
* 3xx: redirección de recursos, por ejemplo de una URL a otra.
* 4xx: códigos de error.
* 5xx: errores de servidor, por ejemplo cuando no puede responder.

**Protocolo DNS (Domain Name Service)**

\_ Este es un protocolo que sirve para traducir un nombre a una IP, entonces por ejemplo no nos tenemos que estar acordando la IP de Google que es 172.217.173.4 sino que usamos [www.google.com](http://www.google.com). Lo que se hizo fue un sistema de nombres de dominio donde podemos traducir una IP a un determinado nombre. También se pueden hacer cosas como traducir un nombre a varios IPs para no tener un solo servidor con una sola IP sino que tenemos uno en donde al entrar a por ejemplo [www.123.com](http://www.123.com) un dispositivo tiene una IP otro tiene otra IP y asi se balancea carga, de esta manera funciona Google.

\_ La jerarquía con la que se arman los nombres de dominio es la siguiente. La primera jerarquía es punto “.” que es una organización de internet que gestiona todos los nombres, luego esa organización genera la primera parte de la jerarquía que es lo último en el nombre dl dominio y da la gestión de esa primera jerarquía a alguien, por ejemplo los “com” los gestiona EEUU, “ar” lo gestiona Argentina (NIC) y así. Después viene la segunda jerarquía a la cual, el que gobierna la primera jerarquía puede generar distintos nombres de dominio de la segunda. Luego se le da la gestión del nombre del dominio a quien lo solicito en alguna instancia.



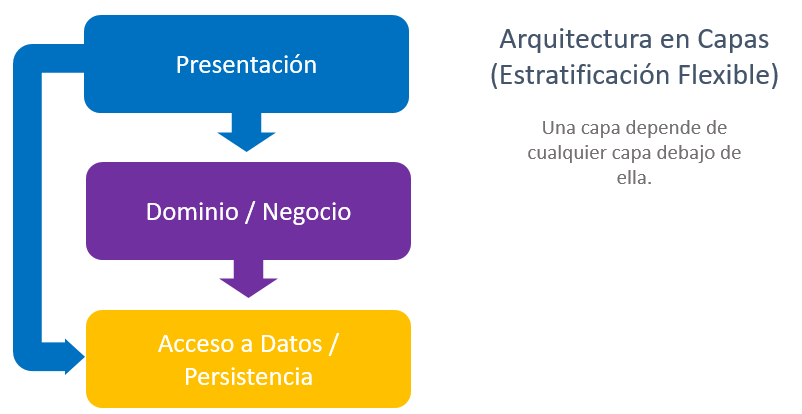
**Patrones arquitectónico**

\_ Un patrón arquitectónico es una solución general y reutilizable a un problema común en la arquitectura de software dentro de un contexto dado. Los patrones arquitectónicos son similares al patrón de diseño de software pero tienen un alcance más amplio. Bajo un contexto determinado vamos a tener que hacer una aplicación muy visual, donde vamos a tener una interfaz gráfica muy presente que es la página web y el usuario va a interactuar directamente con esa interfaz gráfica, entonces ante ese problema de como acomodar mi código para que todo funcione fácil, sea escalable y funcione bien, hay un patrón arquitectónico como el MVC.

**MVC**

\_ Un patrón arquitectónico es algo de más alto nivel, es decir, que por abajo vamos a usar patrones de diseño para lo que haga falta, pero vamos a usar el patrón Modelo Vista Controlador. Este patrón es muy utilizado para todo lo que es web por la forma en la que trabaja. Hay otros patrones como el hexagonal, el patrón de 3 capas, patrón cliente -servidor, entre otras formas para definir cómo se va a comportar el software para solucionar un problema.

\_ Este patrón es muy parecido al patrón de 3 capas (3 Layers Pattern), con capas hablamos de un conjunto de objetos, donde básicamente tenemos una capa de presentación donde se muestran los datos y esa misma capa valida los datos, esto es más utilizado en aplicaciones gráficas y es por ejemplo cuando llenamos un formulario esta capa nos va a decir si esos datos están bien o están mal. Luego vamos a tener otra capa de negocio que se va a encargar de manejar los datos como crear objetos nuevos o eliminarlos, y por último tenemos una capa de acceso a los datos que es la capa de persistencia, que es un lugar donde finalmente guardamos los datos, como lo puede ser una base de datos, un archivo de texto, o lo que elijamos como método de persistencia.



\_ En el caso del patrón MVC este también tiene 3 capas que son el Modelo la Vista y el Controlador. Este está definido para sistemas donde hay mucha interacción del usuario, donde el usuario se tiene que comunicar si o si con la maquina y que generalmente es muy utilizado en lo que son las páginas web, ya que estas tienen una alta interacción del usuario con un contenido visual muy grande (toda la página y el entorno gráfico), y nos permite abstraer la lógica del negocio de la interfaz de usuario de cómo se guardan los datos con criterios de seguridad.

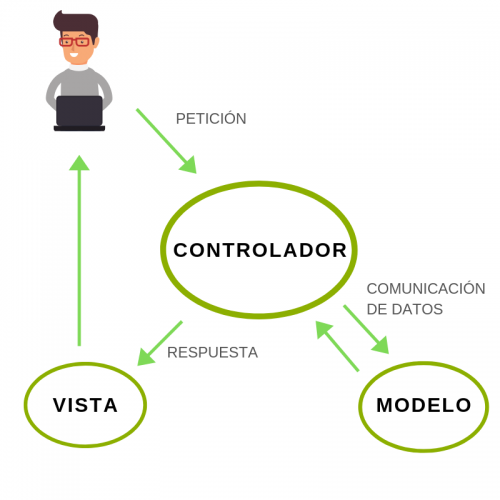
\_ Las capas son:

Capa de datos: es el modelo.

Capa de interfaz: es la parte que el usuario va a ver, esta es la vista.

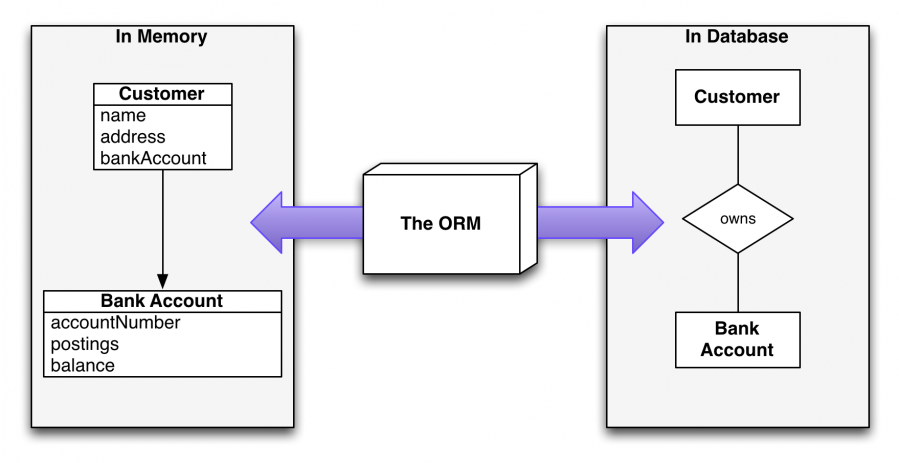
Capa lógica: es la parte lógica del uso del software, es el controlador.

\_ La interacción entre las capas seria la siguiente, al usuario se le muestra una vista, donde el servidor renderiza una página agarrando el HTML llenándolo con datos y se la muestra al usuario, el usuario interactúa con esa vista dándole click a un botón, llenando un formulario y la da subir, o algún tipo de interacción, entonces utiliza el controlador ya que el evento que realiza el usuario va a parar al controlador. Dependiendo del evento el controlador puede modificar o manipular datos del modelo, es decir, lo que este guardado en la base de datos, también puede requerirle datos al modelo para completar alguna cosa de la vista, lo que hace después el controlador es actualizar los datos de la vista, donde por ejemplo el usuario pide al controlador la lista de alumnos, entonces el controlador busca en la base de datos la lista de alumnos, los obtiene, agarra la vista donde hay una tabla con nombre, apellido y nota del alumno, y llena esa vista con la lista de alumnos, una vez que la vista está completa se manda al usuario para que la pueda ver.



Modelo: este representa la lógica del negocio y esto hace referencia por ejemplo a si una persona tiene 160 años está mal, si una persona tiene -1 años está mal, es decir, el modelo chequea la validez de los datos. Otro ejemplo seria si alguien paga un ticket y esa entidad ticket del modelo tiene asociada una cantidad de productos, el modelo debería ser el encargado de descontar el stock de esos productos porque ya se pagaron, también seria si alguien agrega un producto a una lista de compras y ese producto se fue de stock, la lógica de negocio nos diría que no debemos poder tener disponible ese producto. El código del modelo es el que tiene acceso directo a los datos actuando de intermediario entre la base de datos y el sistema. El modelo es el encargado de que los datos persistan, es decir, quien va a guardar los datos. Por lo general se implementa mediante:

* ORMs (Object Relational Model): es el encargado de establecer la comunicación entre la base de datos y la aplicación. Lo que hace es abstraer esa base de datos (estructura de entidades y relaciones) como objetos propios del lenguaje (programación orientada a objetos). Nosotros vamos a tener una base de datos que tenga un customer que es dueño de una cuenta, y en nuestro código nosotros lo vamos a ver como un objeto customer que tiene sus atributos donde uno de ellos va a ser una referencia a otro objeto. Entonces el ORM traduce lo que diseñemos en la base de datos a objetos en nuestro lenguaje.



Por otro lado, como el ORM se encarga de hablar con la base de datos, puede hablar varios protocolos de base de datos, como Postgres o MySQL, ambas bases de datos relacionales, y también puede hablar con otras bases de datos como MongoDB que son no relacionales y son de tipo documental, y también puede hablar con pseudo base de datos que son archivos de texto creados por la base de datos misma por lo que no hace falta instalar un servidor MySQL y configurar el framework para que se comunique con el servidor, sino que trabajamos y por detrás está escribiendo archivos de texto. No hace falta configurar el ORM si saltamos de un tipo de base de datos a otra distinta.

Vista: es el encargado de mostrar al usuario la información de forma gráfica y humanamente legible. Hace referencia a la interfaz gráfica. Generalmente la vista está formada o construida con datos que se obtuvieron del modelo. Por otro lado la vista va a tener las partes visuales para generar los eventos o acciones que realiza el usuario y que van a ser enviadas al controlador.

Controlador: es un intermediario que va a estar entre el modelo y la vista para que el usuario no tenga acceso crudo a los datos, sino que pase por este controlador. Es el que va a controlar toda la interacción o eventos que haga el usuario para solicitarle, según el tipo de evento, que el modelo realice cambios o solicitarle datos al modelo y poder formar una vista para mostrar. El controlador va a responder a los eventos que genere el usuario en la interfaz gráfica y va a decidir lo que se debe ver en esa vista que se tiene que formar.

\_ Las ventajas de usar el patrón MVC son:

* El código queda bien organizado, el flujo de trabajo consiste en que el usuario le pide a un controlador, el controlador utiliza el modelo para hacer una vista y se la muestra al usuario y de esa vista el usuario vuelve a llamar al controlador al hacer una acción y asi se repite.
* Reutilización de código ya que a los modelos se los programa una sola vez y después en todo el programa vamos a usar esos modelos, entonces es muy reutilizable.
* Desarrollo en paralelo, donde cada uno de estos flujos puede estar siendo desarrollado por un programador distinto, entonces puede haber un programador haciendo el alta de un usuario, otro haciendo la modificación del usuario, otro dando el alta de productos y asi, donde cada programador se dedica al controlador y otros a las vistas.
* Desarrollo rápido de la aplicación, ya que utilizando un framework no tenemos que hacer un montón de código sino que estamos utilizando código ya hecho y nos dedicamos solamente a las funcionalidades del sistema.
* Podemos representar la misma información de maneras diferentes, donde las vistas pueden ser hechas para distintas cosas. Por ejemplo podemos tener 3 vistas para un mismo modelo, en donde tenemos una que muestra una tabla, otra muestra una lista desplegable, otra muestra un gráfico y el modelo sigue siendo el mismo, donde lo único que cambia es la vista que vamos a utilizar.

**Node.Js**

\_ Es un intérprete de Javascript para la computadora. Javascript es un lenguaje de programación interpretado donde el principal interprete es el navegador. Cuando hacemos una página web con HTML, CSS y Javascript, el código en Javascript corre en el navegador web. Si queríamos hacer un sistema web teníamos que programar el frontend con Javascript en el cliente o en el navegador porque es lo único que interpreta y por otro lado teníamos que usar otro lenguaje de backend como Go, Java, Python, Ruby, etc, en el servidor. Entonces por eso surgió Node.JS, ya que es un intérprete para poder correr Javascript en el servidor, por lo que es muy utilizado para programar frontend y backend en Javascript. Los comandos a continuación son tanto para la consola en Windows como en Linux:

\_ Para ver la versión de Node.js hacemos:

* node -v

\_ Para ver la versión del npm hacemos:

* npm -v

\_ Para correr el intérprete hacemos:

* node.
* console.log(“hola mundo”).
* var a = 5 + 10;

console.log(a).

* De esta manera corremos líneas de código Javascript localmente.

\_ Para ejecutar un archivo .js hacemos:

* node miapp.js

Npm (node package manager): es un gestor de paquetes para node. Un gestor de paquetes seria como un gestor de bibliotecas de código, donde leguajes modernos como Node, Javascript, Python o Java tiene gestores de paquetes. En Javascript pero más en Node, el gestor oficial es npm, ya que los mismos que desarrollan el intérprete de Node, desarrollan el intérprete de npm en paralelo, como en Python que tiene su propio gestor de paquetes llamado py. Npm permite que haya un repositorio o un lugar central donde uno si hace una librería o biblioteca la puede publicar y si alguien la quiere consumir simplemente la usa. Para poder usar este gestor debemos estar en un proyecto de node

* npm init: genera un archivo que describe el proyecto node, este archivo es JSON (JavaScript Object Notation) y es la forma de describir objetos en Javascript.
* La carpeta node\_modules va a tener todas las dependencias que instalemos con npm.

**Sails**

\_ Sails es un framework, es decir, un conjunto de código que hizo gente en conjunto y que da funcionalidades, y es un framework que utiliza el patrón MVC, lo que nos permite a nosotros solamente realizar el sistema. Este tiene un ORM, está hecho en Javascript y es para Node.js.

\_ Instalación:

* npm install sails -g

\_ Ver versión:

* sails -v

\_ Crear un nuevo proyecto:

* sails new test-project

Elegimos “Empty” para arrancar con una app vacía solo con el esqueleto.

\_ Arrancar el proyecto:

* cd test-project
* sails lift
* En el navegador vamos a localhost:1337, es decir, levantamos el proyecto en el puerto 1337.
* Cerramos con ctrl + C.

\_ Dentro del proyecto vamos a tener las siguientes carpetas:

* API que va a tener las carpetas:
* Controllers que va a tener los controladores.
* Models: que va a tener los modelos.
* Helpers: que sirve para tener código y utilizarlo por si es necesario.
* Policies: que sirve para la parte de seguridad. Por ejemplo podemos crear una policie que diga que si el usuario no está logueado, que no se le permita usar determinados controladores, o si el usuario no es administrador, que no pueda ingresar a tal lado. Restringimos el acceso a ciertos controladores.
* Assets es todo lo que va a poder acceder el navegador web sin la necesidad de que el servidor haga algo:
* Imágenes
* Styles: podemos guardar los archivos CSS.
* Js: para archivos Javascript que corran en el navegador y van a poder ser pedidos justamente por el navegador.
* Favicon.ico: es el icono que se muestra en la pestaña de la página web.
* Config tiene archivos Javascript que tienen configuraciones que debemos cambiar según como queramos que se comporte el framework.
* Views es donde tenemos las distintas páginas:
* Layout: es un marco o contenedor genérico en el cual se van a insertar adentro las distintas pages, entonces podemos generar un menú por ejemplo y hacer que las páginas se inserten abajo, por lo que ese menú o cabecera va a ser como todo nuestro sistema. Lo que yo cambie acá se cambia en todas las páginas. En la página se hace click derecho y ver código fuente.
* Pages: las páginas que vamos a tener de la página.

3 modelos entidad relación con controladores.

Ejemplo: tenemos una aplicación donde como relaciones tenemos un usuario que va a tener un ID, un nombre, un mail, un password que se guarda como un hash y con eso logramos que si se filtra la base de datos no se vea el texto plano, sino que se vea un conjunto de números, el hash no es un encriptado, ya que el hashing es transformar cualquier cosa a un numero de determinado largo, en donde con ese número nunca podremos volver a saber el valor de la contraseña original, después tenemos otros atributos como un update y create que se generan con el ORM. Después, de este usuario vamos a tener tweets, que sería la otra entidad, en donde como atributos tendrá un ID, el contenido, el usuario que lo hizo, cuando fue creado y la actualización. Luego tenemos una entidad relationships que va a ser la relación entre usuarios y quien sigue a quien, esta sería como una tabla pasarela donde tenemos el ID de que usuario sigue a tal usuario.

\_ Vamos a tener un modelo que es la tabla usuarios, una vista que es la página de login y vamos a tener un controlador que va a ser el encargado de decir si se logueo o no.

\_ Nunca vamos a ejecutar una sentencia SQL porque no vamos a abstraer la base de datos.

\_ En la carpeta api del proyecto, en la parte de models creamos un nuevo archivo js, donde es importante el nombre porque va a ser el nombre del modelo. Como trabajamos con objetos, el objeto va a ser un solo usuario con sus atributos. A los atributos les podemos aclarar si son requeridos o no para que sea obligatorio ponerlo, les podemos poner un valor por defecto, entre otras cosas.

// api/models/user.js

module.exports = {

attributes: {

nameOnMenu: { type: 'string', required: true },

price: { type: 'string', required: true },

percentRealMeat: { type: 'number', defaultsTo: 20, columnType: 'FLOAT' },

numCalories: { type: 'number' },

},

};

\_ Podemos crear dentro de un modelo métodos en particular de lógica de negocios para ayudar simplificar las cosas.

\_ Los atributos se definen de esta forma:

// api/models/User.js

{

attributes: {

emailAddress: { type: 'string', required: true, },

karma: { type: 'number', },

isSubscribedToNewsletter: { type: 'boolean', defaultsTo: true, },

},

}

\_ Hay distintos parámetros en los atributos como un valor por defecto para que se inicialice con un dato, los tipos de datos pueden ser string, numero, boolean, json, etc, se le puede aclarar si son requeridos o no, podemos setear un valor null, también tenemos validaciones, para aclarar que tal atributo tiene que ser tal en la base de datos, podemos forzar el nombre de la columna que se va a ver desde la base de datos, y asi otras cosas más.

\_ Por otro lado tenemos las asociaciones que son las relaciones que hay entre una entidad de otra, es decir, la asociación entre un modelo y otro. Lo que ayuda la relación es que cuando vamos a realizar una búsqueda, por ejemplo, de los usuarios encontramos el 123, y se supone que el usuario tiene a dentro un campo que es mascotas, y ahí vemos la relación, y lo llenamos con el arreglo de mascotas. Lo vamos a ver desde el lado objetos como si hubieran hecho un join y vamos a ver el objeto usuario con un campo arreglo que es pet y ahí adentro vamos a tener un montón de objetos mascotas:

// Find a single user, including its pets

var userWithPets = await User.findOne(123).populate('pets');

\_ Las relaciones que tenemos son:

* Muchos a muchos.
* Uno a muchos.
* Uno a uno.
* Relación de una sola vía: de allá para allá me importa de acá para allá no.

\_ En nuestro caso muchos usuarios van a poder seguir a muchos usuarios y ese usuario va a poder seguir a muchos usuarios. Como ejemplo de relación muchos a muchos, siguiendo con el ejemplo de usuarios y mascotas, creamos el modelo desde el punto de vista del usuario, con los atributos del usuario y después vamos a tener un atributo mascotas que va a venir de una colección de mascotas. La tabla mascotas va a tener un owner que va a ser el ID del usuario.

// myApp/api/models/User.js

// A user may have many pets

module.exports = {

attributes: {

firstName: {

type: 'string'

},

lastName: {

type: 'string'

},

// Add a reference to Pet

pets: {

collection: 'pet',

via: 'owners'

}

}

};

\_ Luego desde el punto de vista de las mascotas, vamos a tener un owner que va a ser una colección de usuarios vía pets.

// myApp/api/models/Pet.js

// A pet may have many owners

module.exports = {

attributes: {

breed: {

type: 'string'

},

type: {

type: 'string'

},

name: {

type: 'string'

},

// Add a reference to User

owners: {

collection: 'user',

via: 'pets'

}

}

};

\_ Para crear estas relaciones tenemos dos opciones, una es crear en la carpeta modelos directamente un nuevo archivo con el nombre y generamos las sintaxis. La otra opción es un comando de sails que ya genera una base, entonces en el IDE abrimos la terminal y estando en la carpeta de la aplicación tiramos el comando:

* Sails generate api user: este comando genera un modelo y un controlador de user.

Api REST: hace que el servidor o página web exponga los datos del modelo entonces las podemos consumir desde Android y no solo desde vistas generadas por el servidor.

* <http://localhost:1337/user>

Carpeta Config:

* La carpeta boostrap es un buen lugar para setear datos falsos o de inicialización. Analizando el código tenemos que si la cantidad de usuarios que hay en la base de datos es mayor a cero quiere decir que la base de datos ya fue inicializada y ya hay datos ahí, por lo tanto, retorno, caso contrario creamos cada uno de los usuarios que pasamos, entonces pasamos objetos de Javascript entre llaves con los nombres de las variables y los valores que van a tener. Entonces creamos un arreglo de usuarios pasando los datos que pide el modelo user.
* La carpeta routes es donde establecemos las rutas.
* Cuando uno trabaja con modelos como códigos o agregamos atributos en un modelo, tenemos básicamente tres opciones a la hora de ejecutar sails lift, tenemos tres opciones porque esto repercute como un cambio en la base de datos:
* Alter: limpiar la base de datos y tratar de reinsertar todos los datos que teníamos (recomendado).
* Drop: limpio todo y empezamos de cero, acá perdemos la base de datos.
* Safe: no auto migra la base de datos, sino que espera que nosotros lo hagamos a través de MySQL.

Para establecer uno por defecto, lo establecemos en la carpeta models.

Vista LogIn: tenemos la acción que hace referencia a que ruta voy a usar del servidor que en nuestro caso es un /login, y tenemos el método que puede ser un get o un post. Los get se usan bastante, pero lo malo de enviar datos por get a través de un servidor, como es un cambio en la dirección de la URL, va a hacer que en el historial guarde los datos, por lo tanto, si mandáramos un usuario y contraseña de esta forma, quedaría en el historial guardado cada uno de las contraseñas que mandamos permitiendo que se filtre la contraseña. Por ende, hacemos que el método sea un post ya que los datos no quedan pegados en la URL.

Controlador: pide un request y un response, donde el request es lo que nos pide el navegador y el response es lo que le vamos a responder. Entonces por ejemplo en el request nos va a llegar el mail y la contraseña que enviamos como variables y después le vamos a responder al navegador enviando por ejemplo una vista.

Rutas:

* Si quiero que las rutas respondan ante un get, tenemos que poner un get y la ruta:

'GET /signup': { view: 'conversion/signup' },

* Si quiero que las rutas respondan ante un get, tenemos que poner un get y la ruta:

'POST /signup': { action: 'entrance/signup' },

Controller:

* Req.session: es una variable que queda permanente y tiene persistencia entre una llamada del navegador y otra. Si el navegador hace un pedido y el usuario ya estaba logueado, la variable req.session va a seguir existiendo y va a mantener el valor user.
* Argon2: algoritmo de hash que usamos para proteger las contraseñas, en donde transformamos los datos vulnerables en un numero de hash.
* FindOne(): buscamos un usuario por el mail y si existe lo trae. Pero no verifica en el pedido si la contraseña es la misma.
* Por ende, seguido de lo anterior, se verifica si el usuario se encontró y verificamos si la contraseña del usuario de la base de datos es la misma que la contraseña que se envió en el formulario. En el caso de que sea correcto reenvía al homepage para hacer logout y de lo contrario si no es asi reenvía al homepage pero para hacer login porque la cuenta no existe.