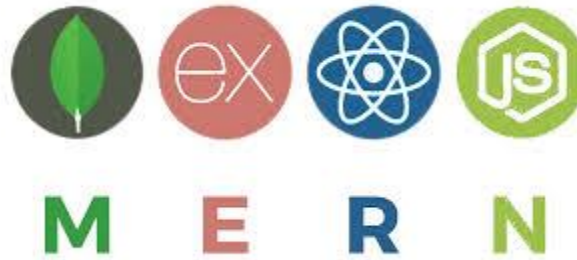


PROGRA 101



JavaScript



Paolo Sandoval Noel

Gmail: paolosandovaln@gmail.com

Email: paolo.sandoval@jalasoft.com

Skype: paolo.sandoval.jala

Github: jpsandovaln

AGENDA

Introducción

- UML - Diagrama de clases
- Building tools - research
- Git

Principios de código limpio

- Estilo de código / convenciones

Paquetes

Sintaxis Básica

- Variables, expresiones y sentencias
- Condicionales
- Funciones
- Iteraciones
- Cadenas
- Arreglos

OOP

- **Pilares de la programación Orientada a objetos**

- **Clases, Objetos e Instancias**

- Sobrecarga
- constructores
- Métodos estáticos

- **Herencia y Polimorfismo**

- Herencia
- Polimorfismo
- Clases Abstractas(emular)
- Interfaces (emular)

- **Manejo de excepciones**

- Try-catch-finally
- Throws

- **Web App**

- **Back end:**
 - REST Services
 - Base de datos
- **Front end**

AGENDA

- **Pruebas unitarias**

- Escribir unit test
- Buenas practicas

Diseño orientado a objetos

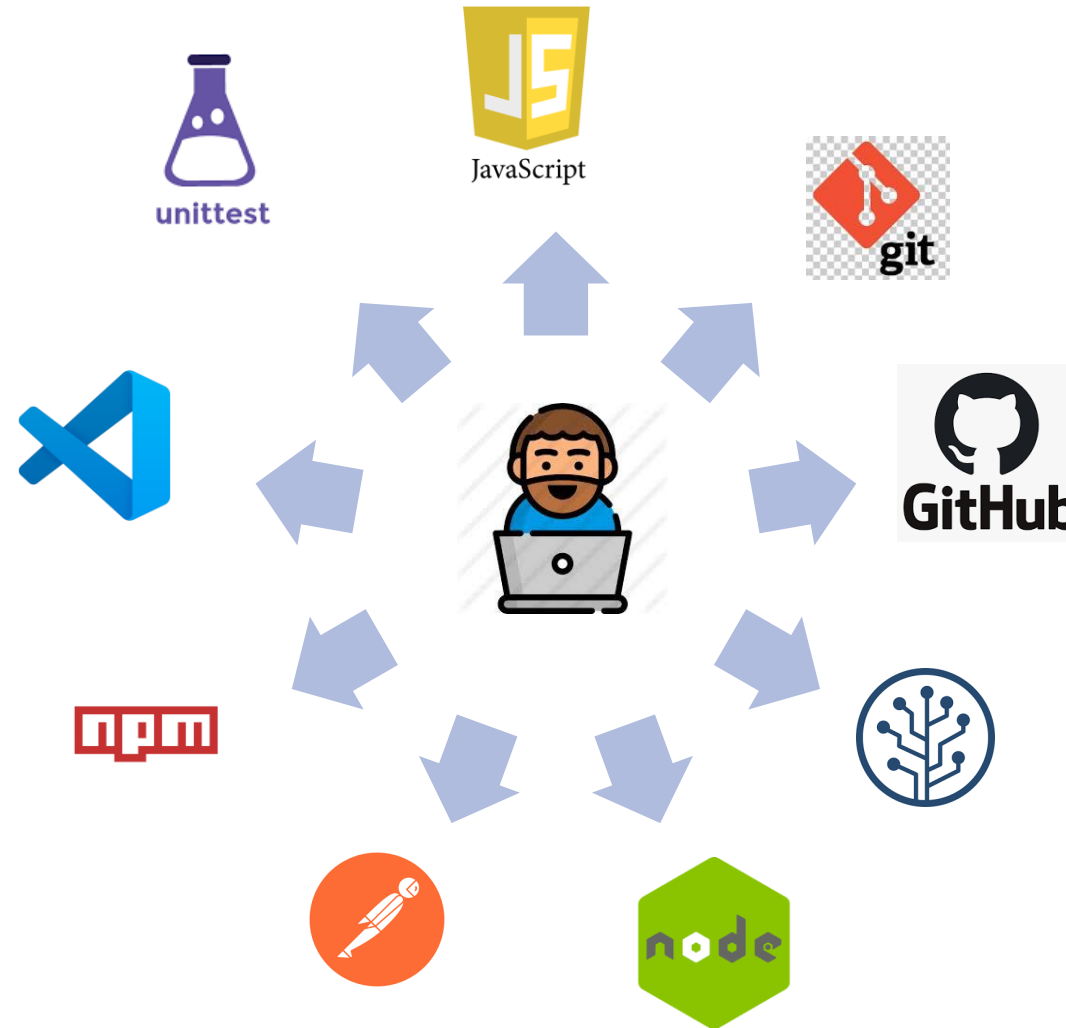
- **Principios S.O.L.I.D**
- **Diseño de patrones**
 - Como crear clases singleton

- **PROYECTO**

Implementar un proyecto donde los estudiantes puedan aplicar lo que aprendieron.

Tecnologías y Herramientas

- JavaScript
- Visual studio code
- Git
- GitHub
- SourceTree
- Postman
- node
- npm
- Unit test
- **Swagger**



SourceTree <https://www.sourcetreeapp.com/>
Postman <https://www.postman.com/downloads/>

- Instalar Git
- Instalar sourcetree
- Instalar Visual Studio Code
- Crear cuenta GitHub
- Crear cuenta en skype

- Investigación 20%
- Prácticas 30%
- Proyecto 50%
 - Convenciones de código
 - Code review
 - Progreso de codificación

Nota de aprobación 70%

Investigar

- Programación Funcional – Carlos Angel Tito Caceres
- Programación Orientada a Objetos – Cristian Torrico Vasquez
- SOLID – Edwin Jaime Patiño Abasto (SOL) Esther Huarayo Lalle (ID)
- Building tools – Milena Rodriguez Montecinos
- Micro servicios – Paulo Sergio Salinas Velez
- Interfaces y clases abstractas en java scripts – Javier Choque Matos

Practica 1

Realizar diagrama de clases para el siguiente caso de estudio

The image shows a web application interface for video analysis. On the left is a sidebar with input fields and controls. On the right is a large main area with a table header and a search button.

Left Sidebar:

- Video Path:** A text input field with a "Browse" button below it.
- Word:** A text input field.
- Neural network Model:** A dropdown menu currently showing "VGG16".
- Percentage:** A dropdown menu currently showing "0".
- Search:** A button.

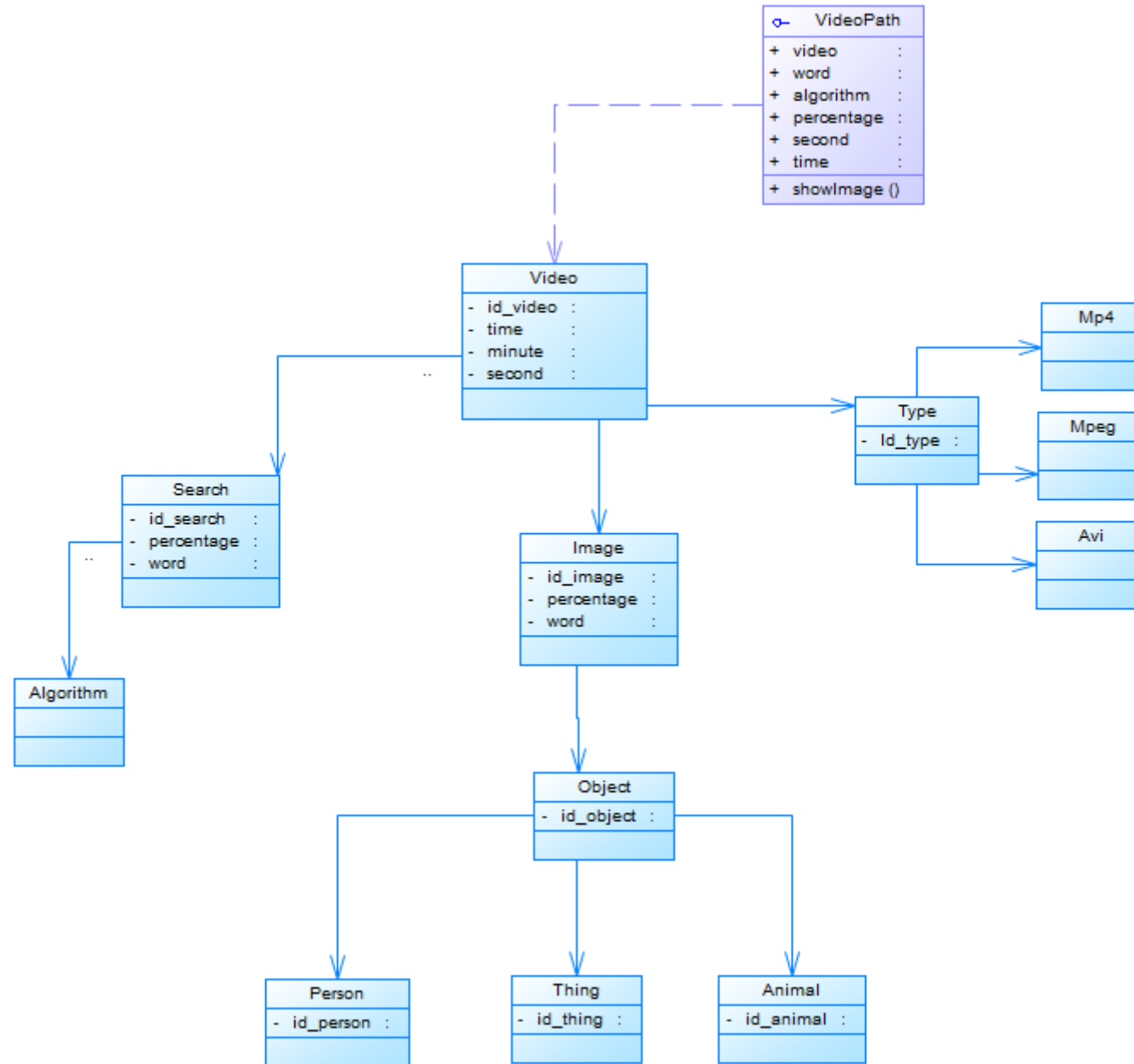
Main Area:

- A table with the following headers: "Algorithm", "Word", "Percentage", "Second", and "Time". The table body is empty.
- A "Show Image" button at the bottom center.
- Text at the bottom right: "Activar Windows" and "Ve a Configuración para activar Windows."

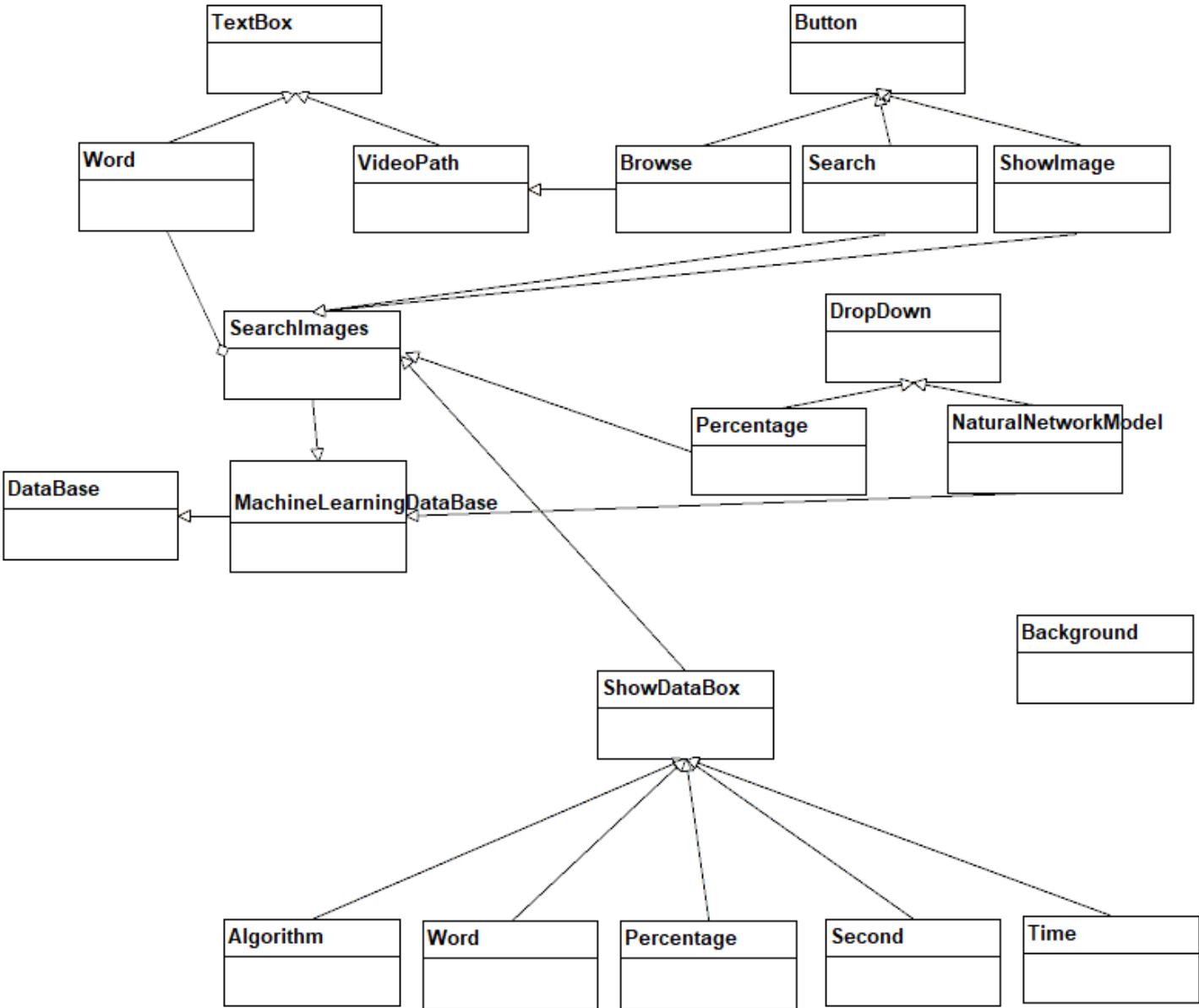
Objetivo:

- Medir el grado de Abstracción del problema

Practica 1

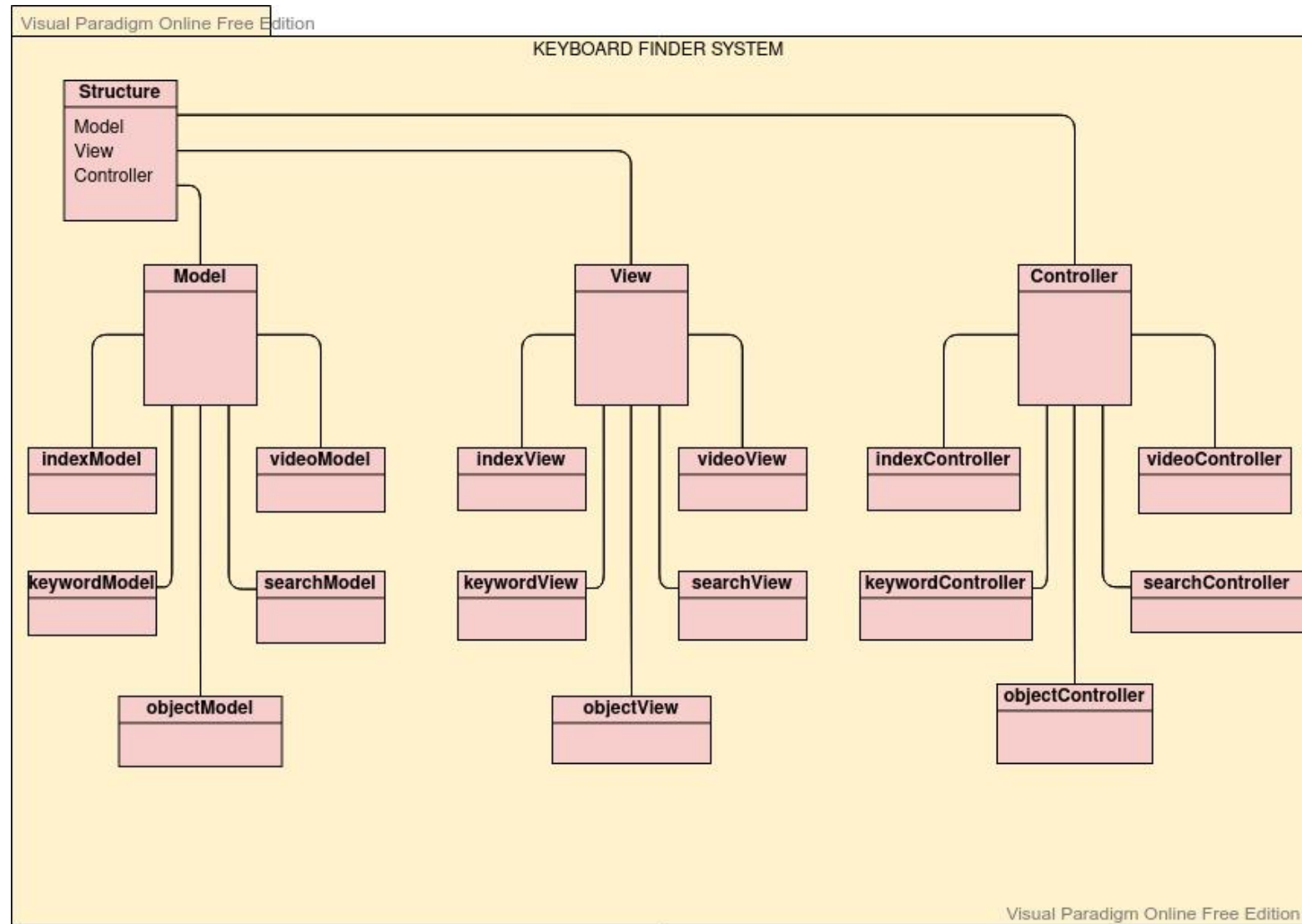


Practica 1

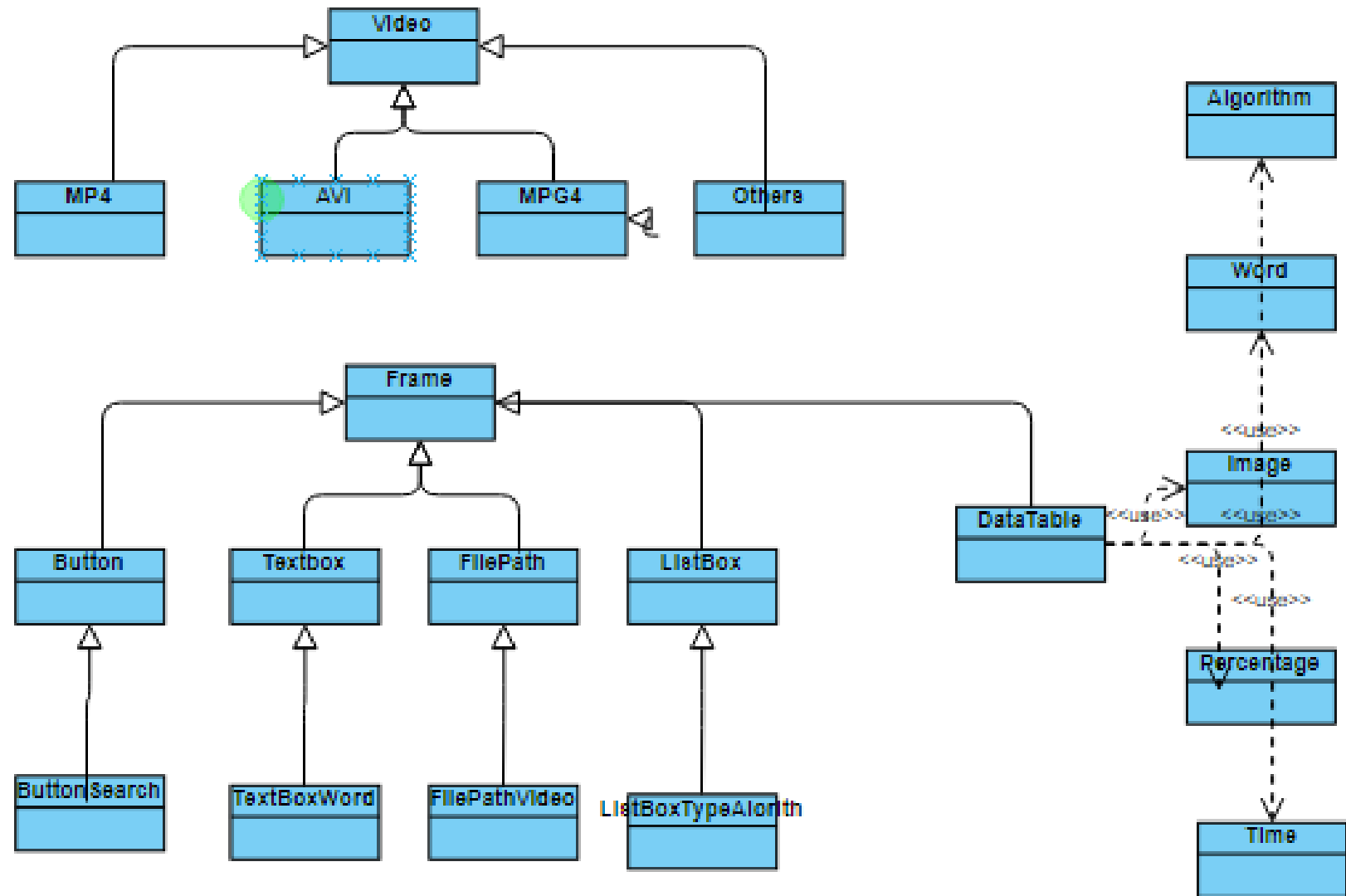


Paulo Sergio Salinas Velez

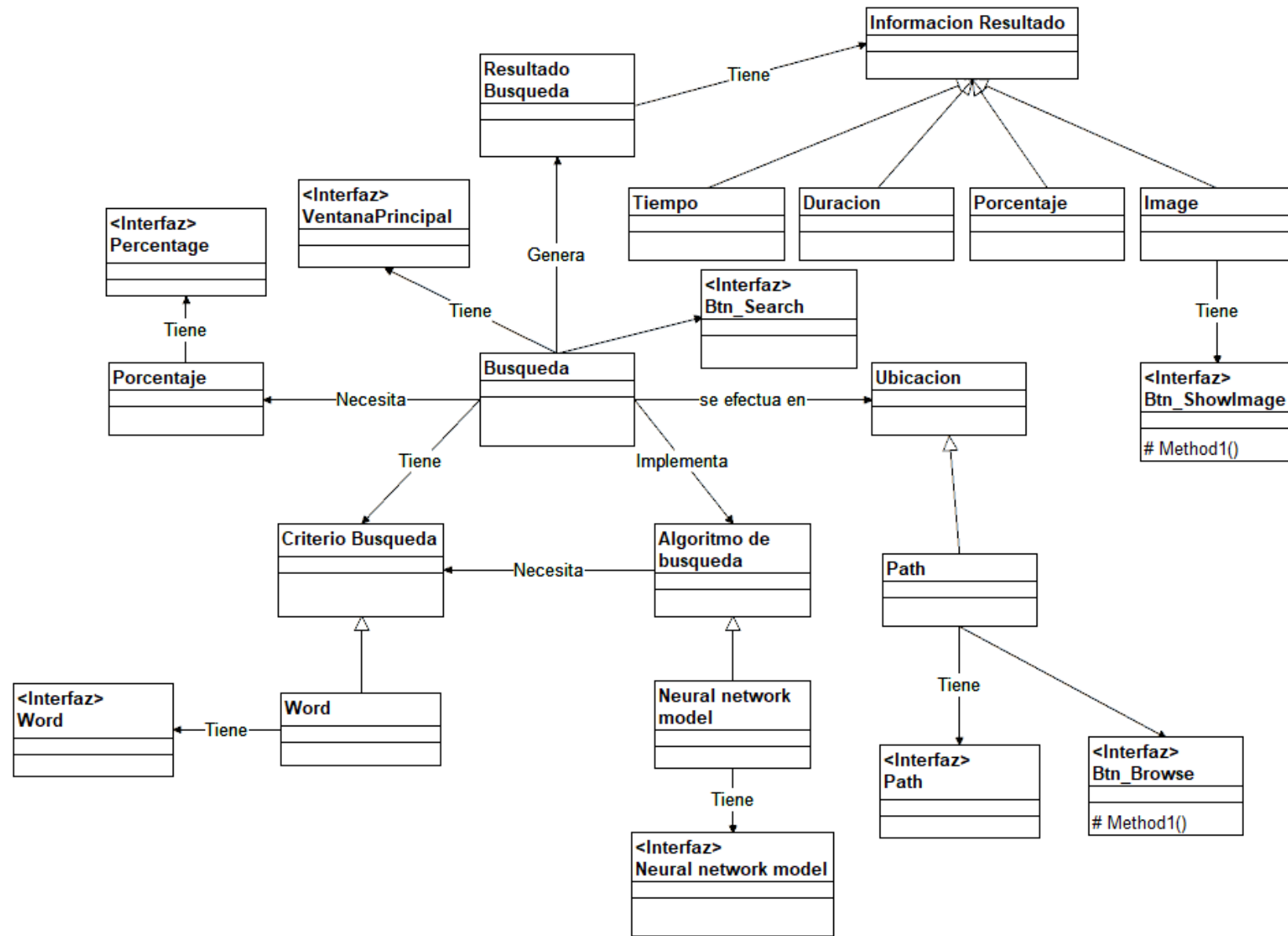
Practica 1



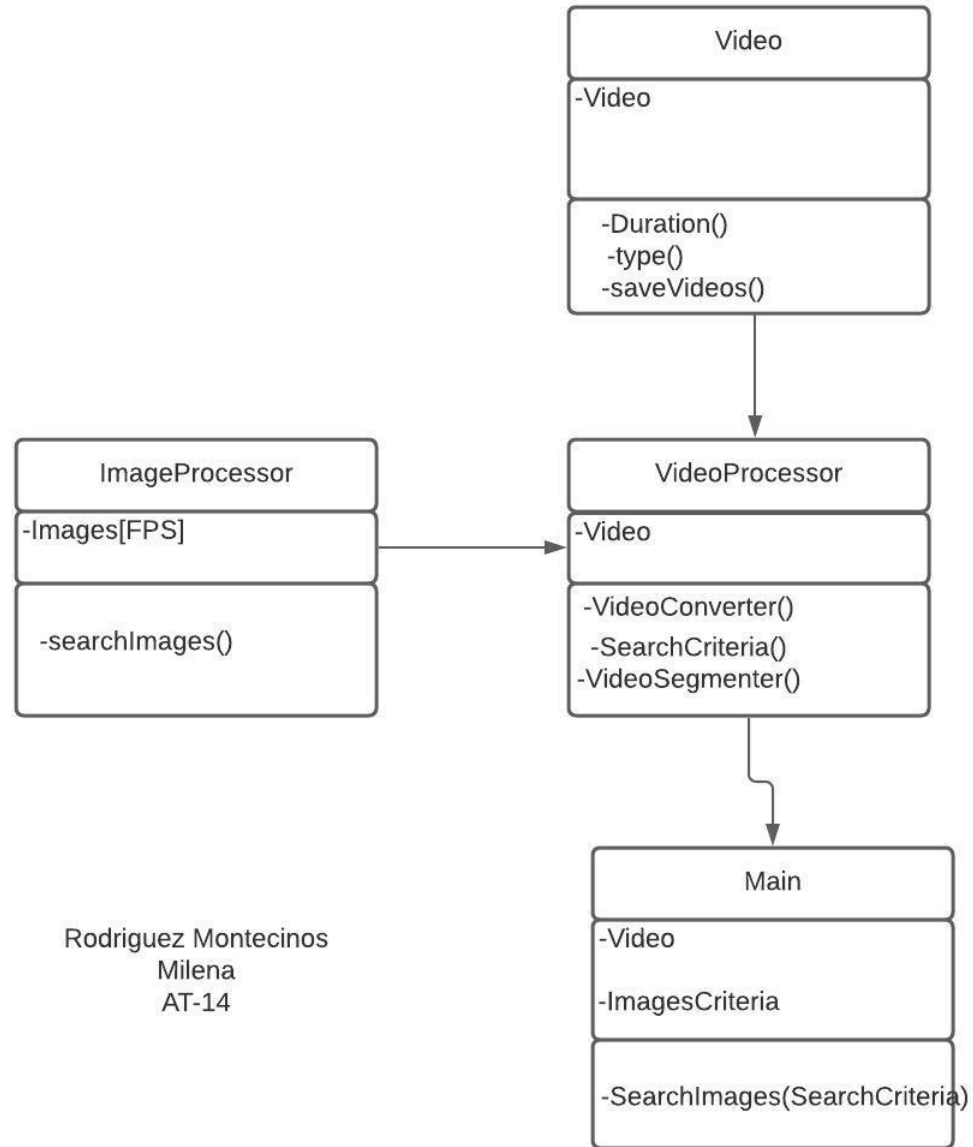
Practica 1



Practica 1

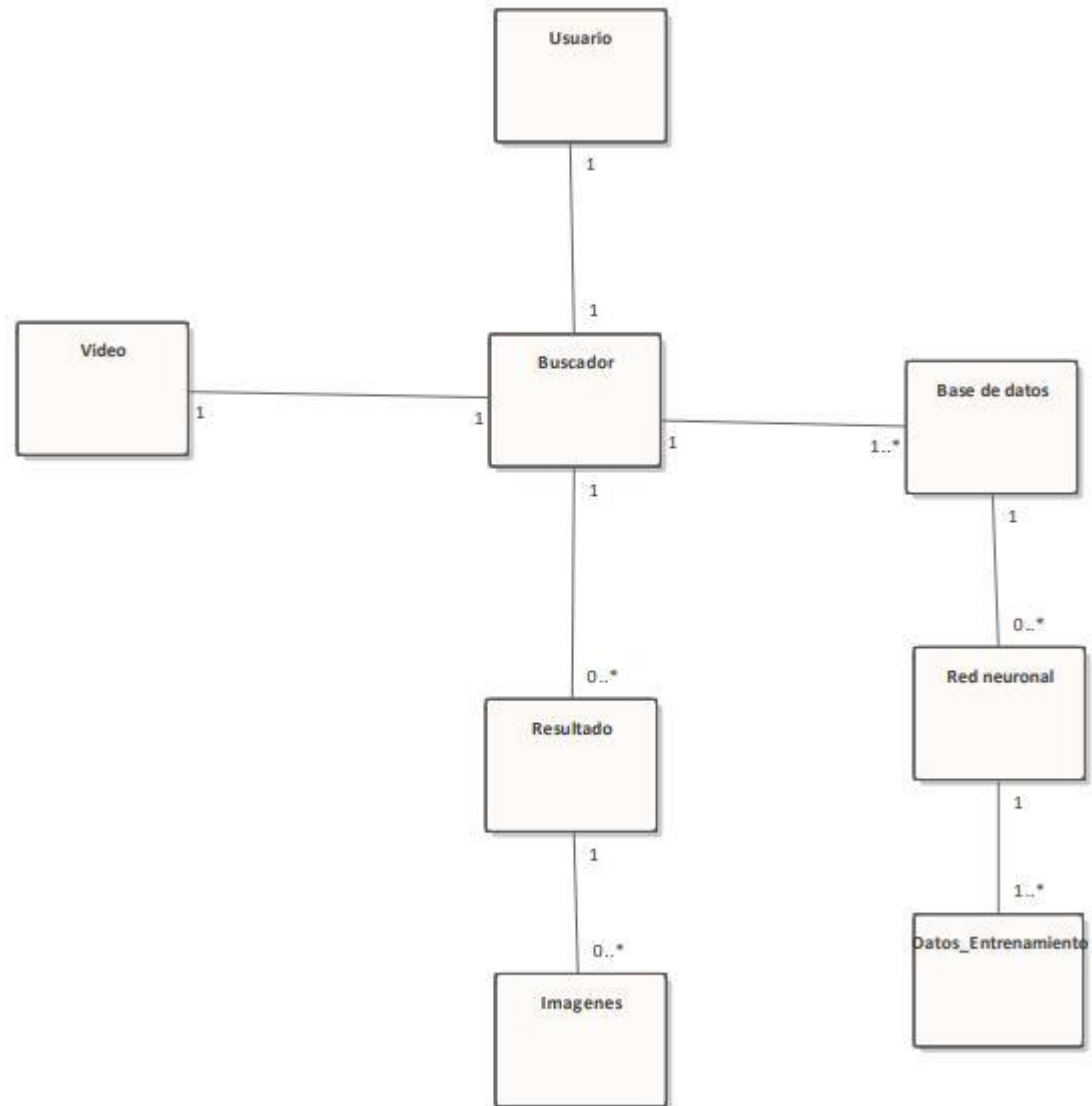


Practica 1



Rodriguez Montecinos
Milena
AT-14

Practica 1



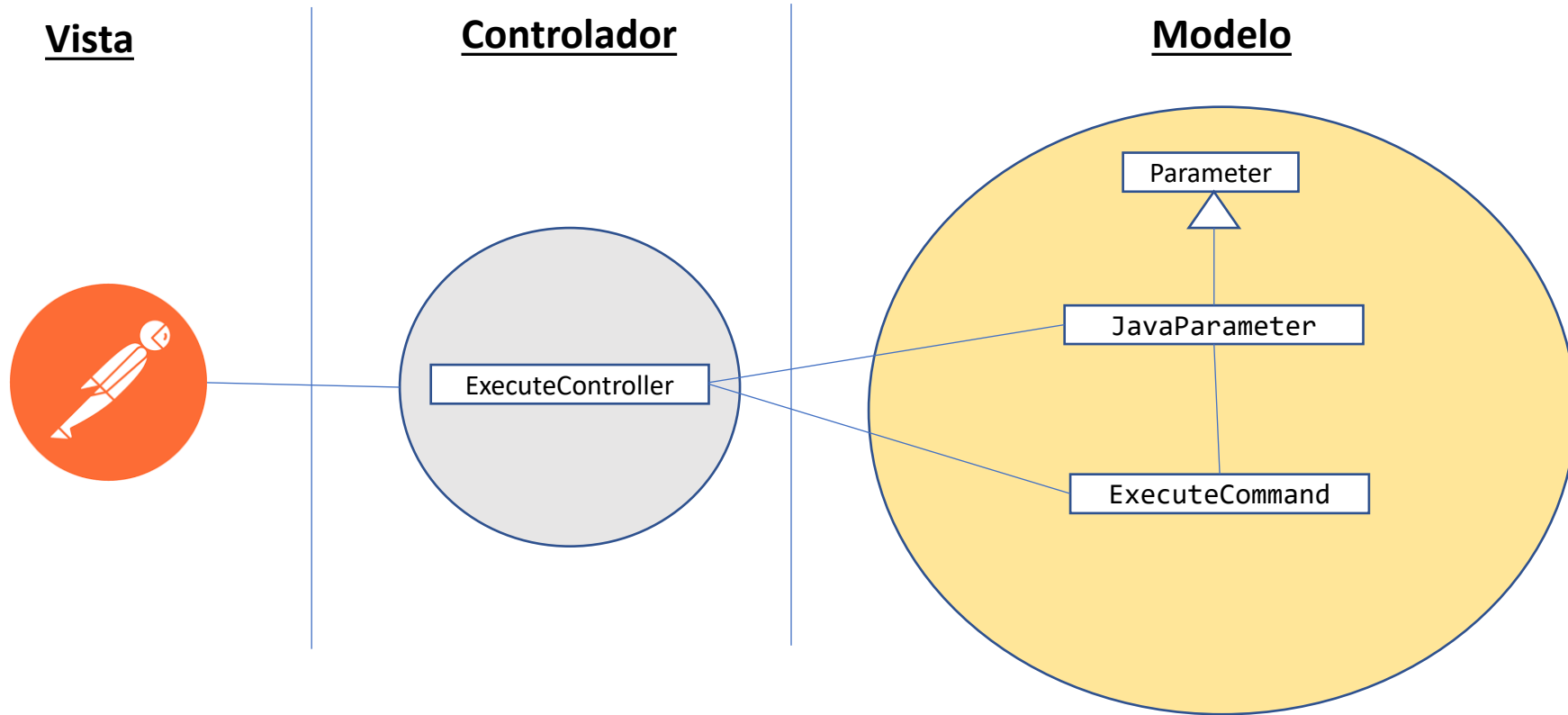
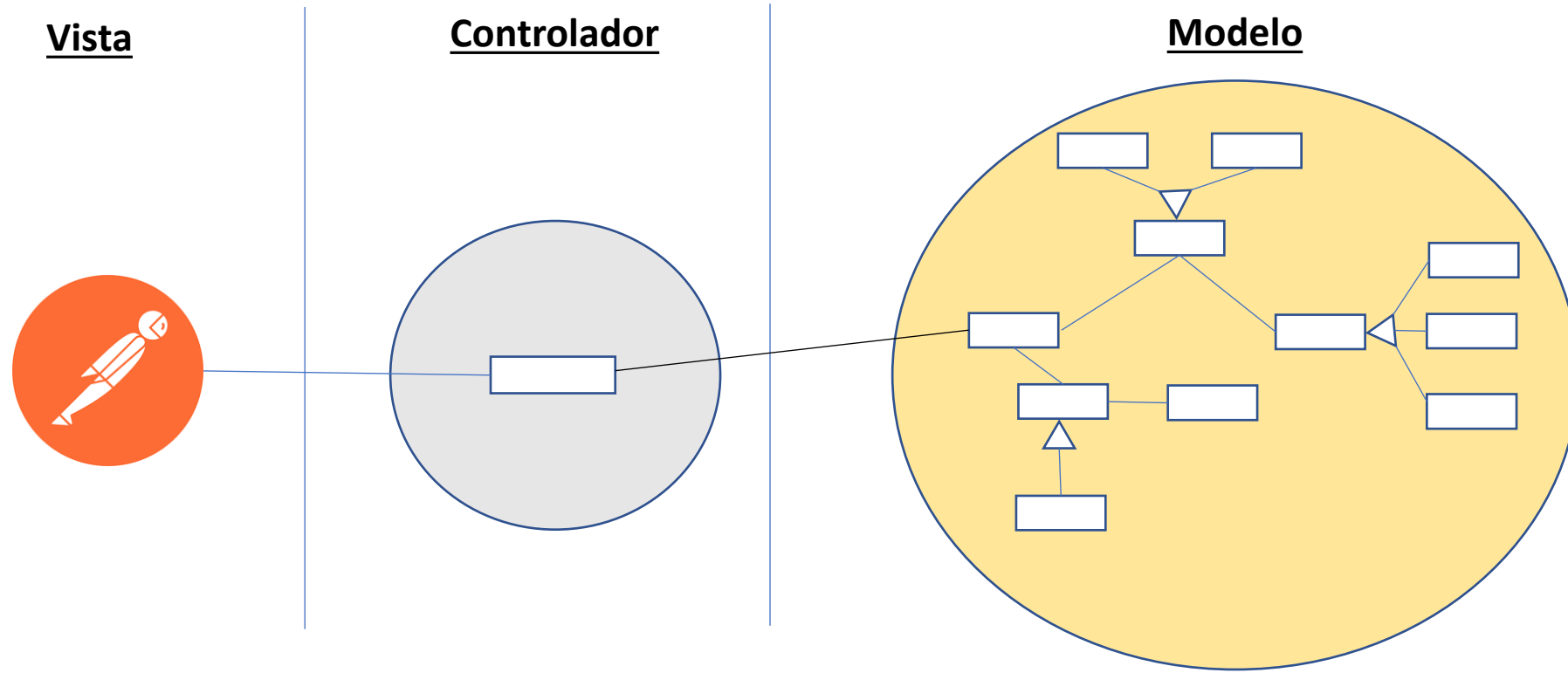


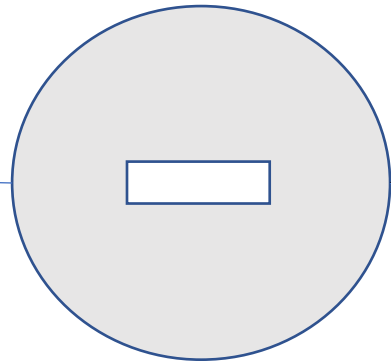
Diagrama de clases – Ejecutar Comando



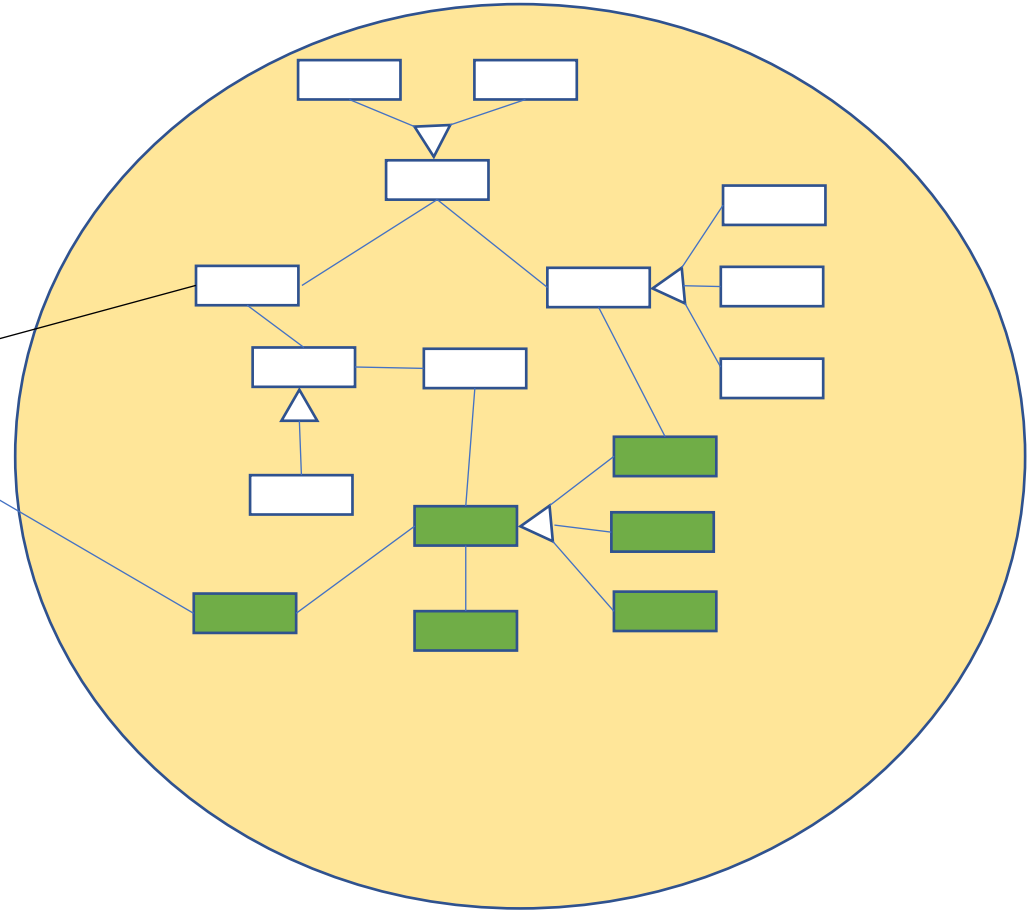
Vista



Controlador



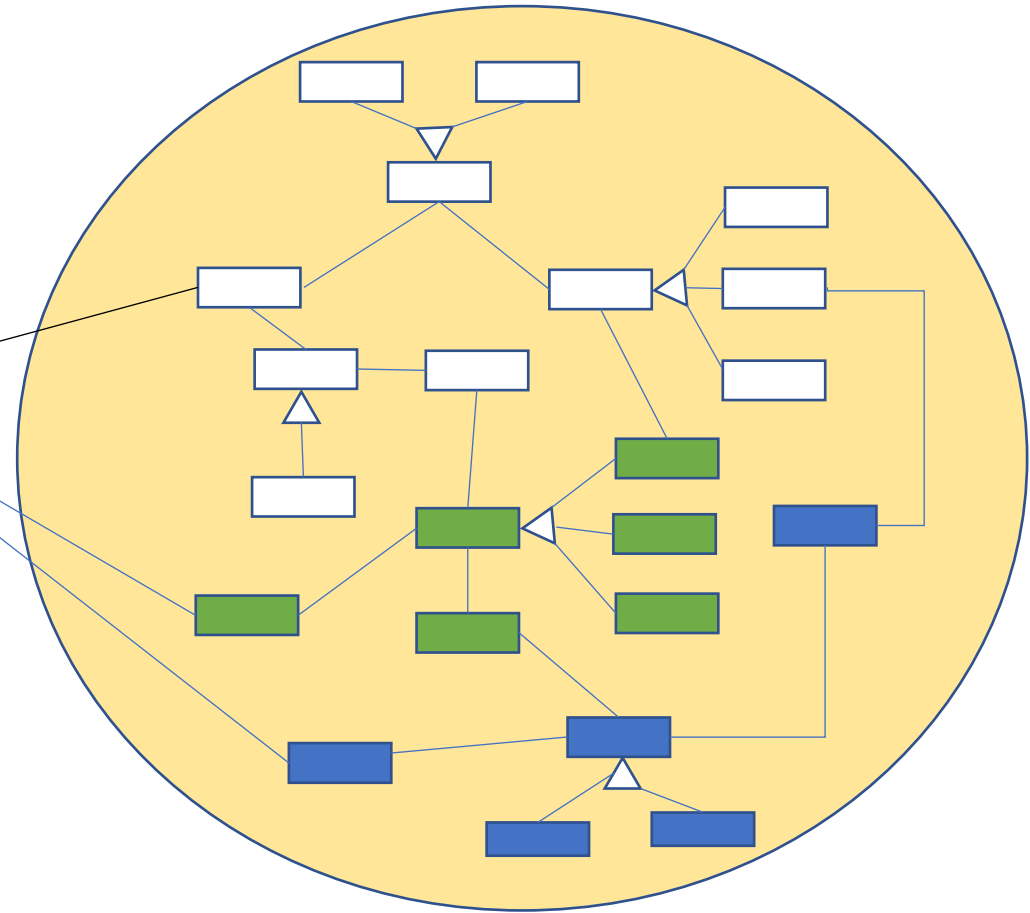
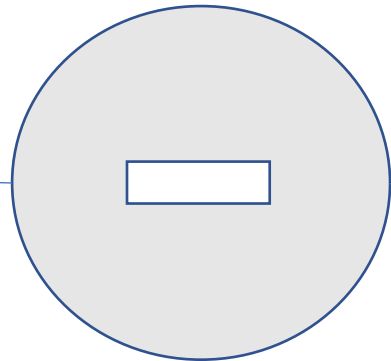
Modelo



Modelo

Vista

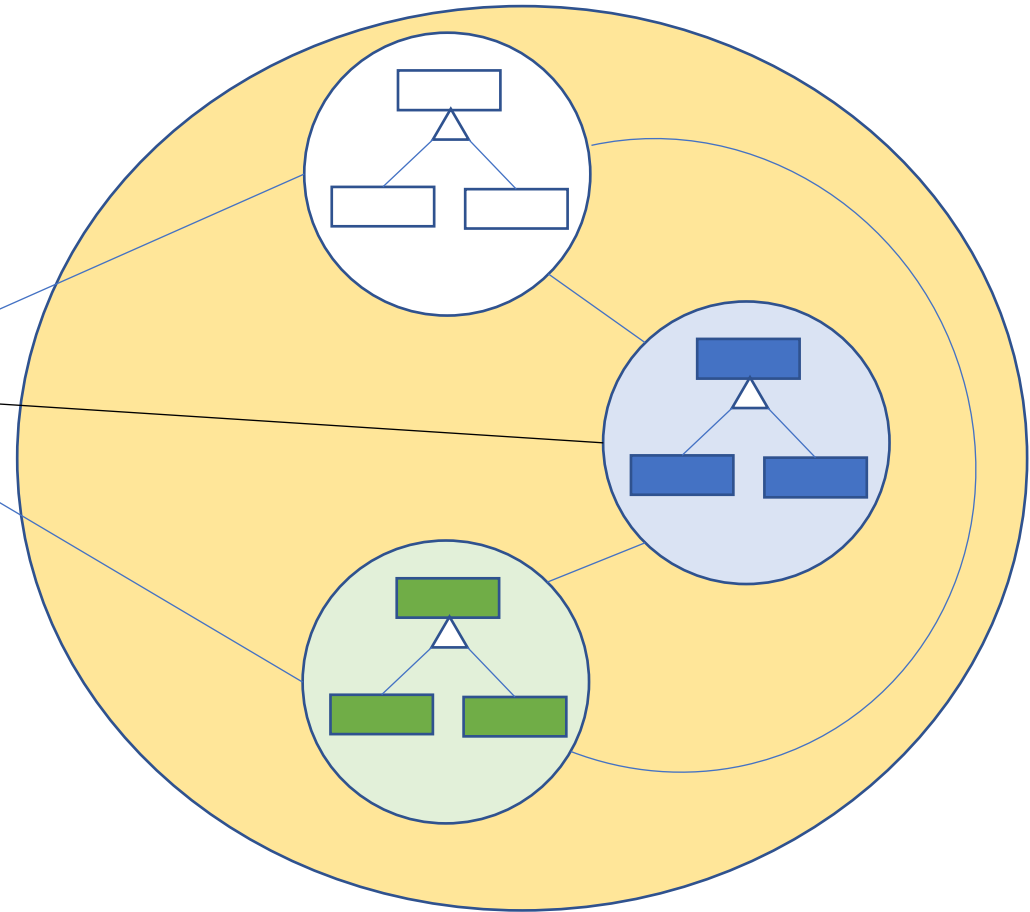
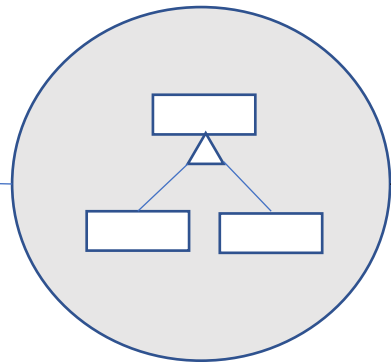
Controlador



Modelo

Vista

Controlador



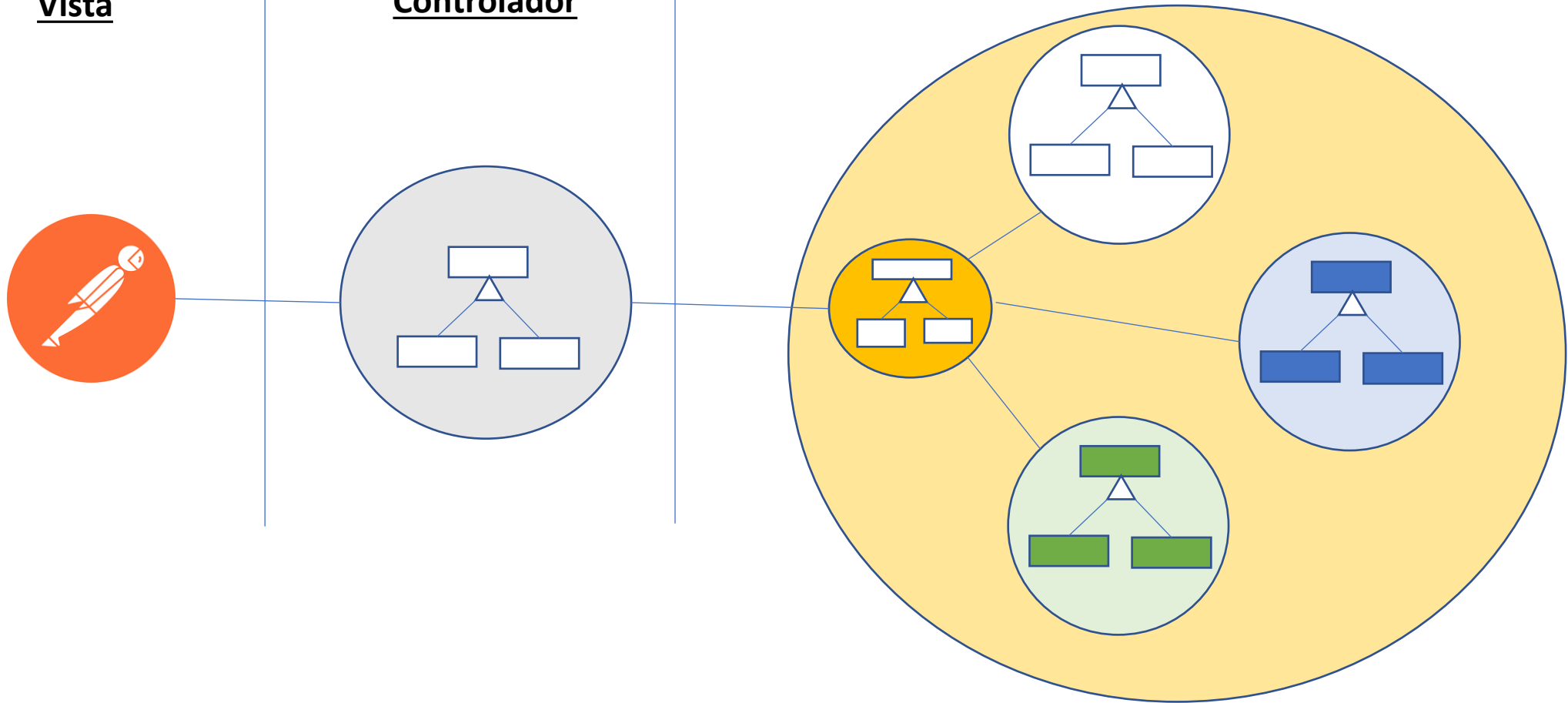
Abstracción

Diagrama de clases – Ejecutar Comando, Usuario y Conversor

Modelo

Vista

Controlador



Introducción

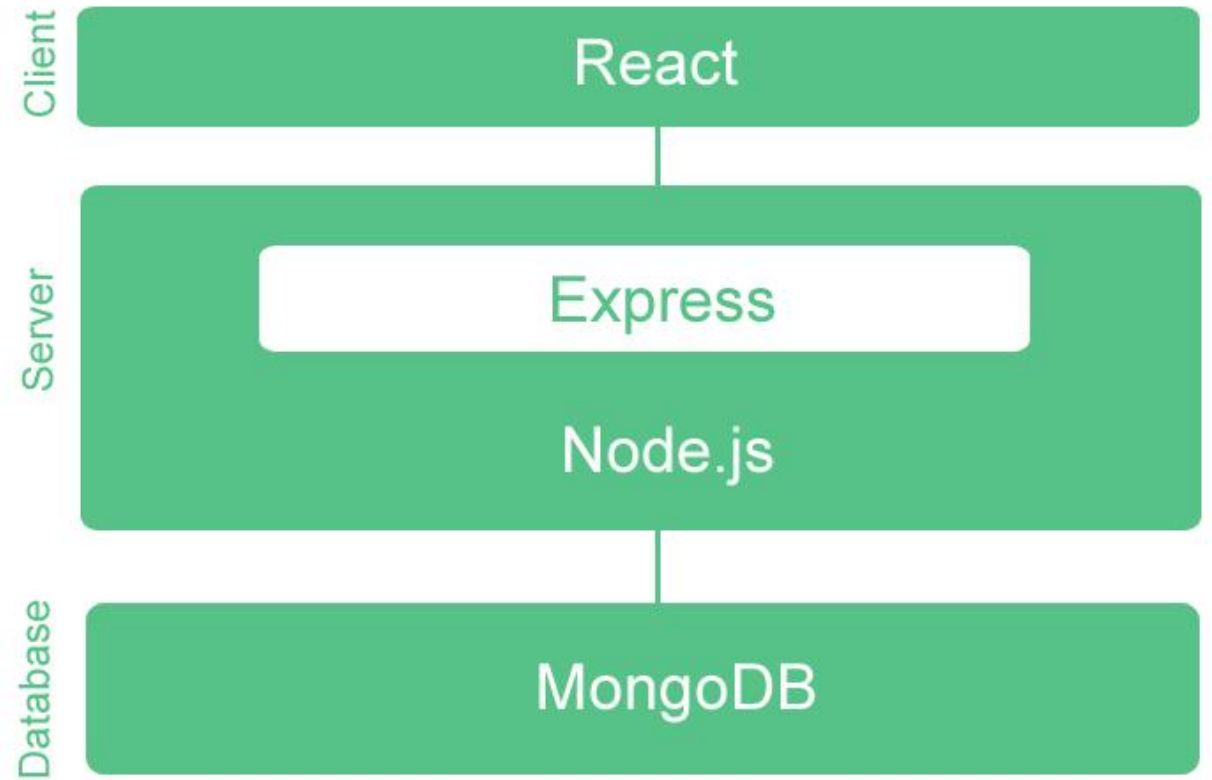
¿Qué hace un desarrollador Full Stack?

Un desarrollador Full Stack es el encargado de manejar cada uno de los aspectos relacionados con la creación y el mantenimiento de una aplicación web. Para ello es fundamental que el desarrollador Full Stack tenga conocimientos en desarrollo Front-End y Back-End. Además de manejar diferentes sistemas operativos y lenguajes de programación.



MERN

El stack MERN utiliza JavaScript como único lenguaje, por ello no tendremos dificultades al familiarizarnos con cualquiera de estas tecnologías, las cuales son mongoDB, Express, React y Node.js. La ventaja que encontramos al utilizar este stack en específico, es que nos permite profundizar en un solo lenguaje de programación, logrando así, enfocar y reforzar nuestros conocimientos para especializarnos en JavaScript y con ello ser más productivos.

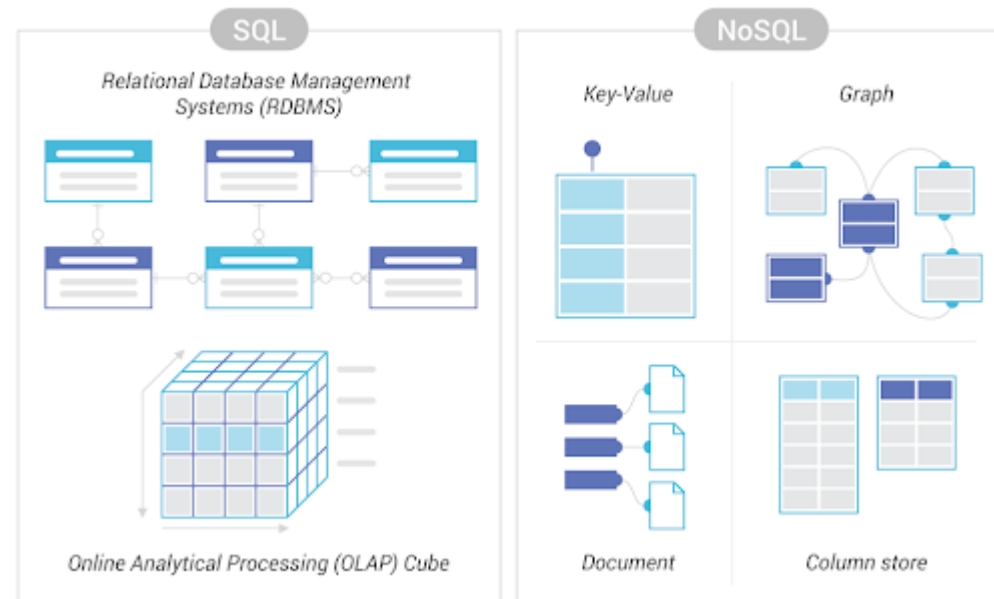
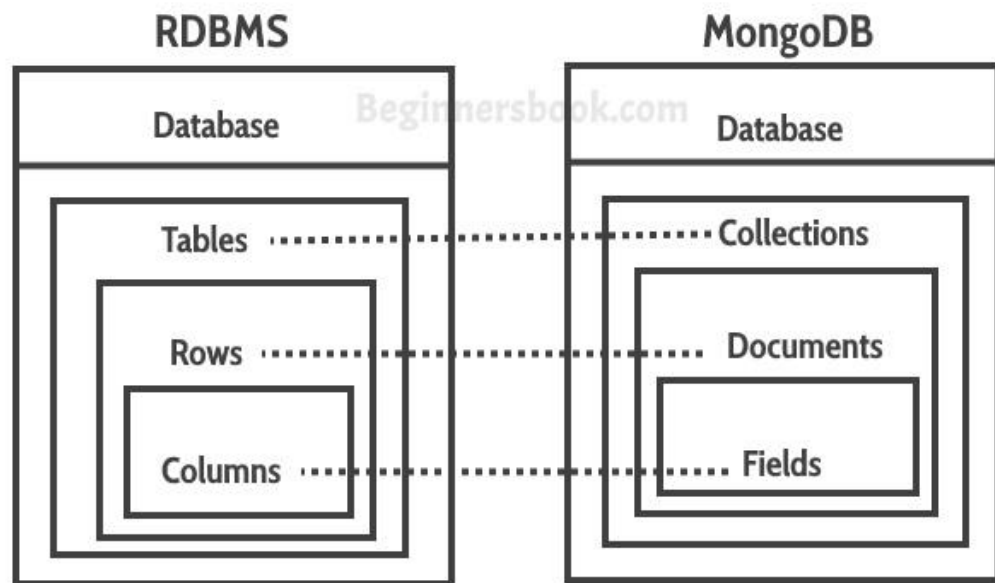


MongoDB es una base de datos orientada a documentos. Esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos. Estos documentos son almacenados en BSON, que es una representación binaria de JSON.

Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que **no es necesario seguir un esquema**.

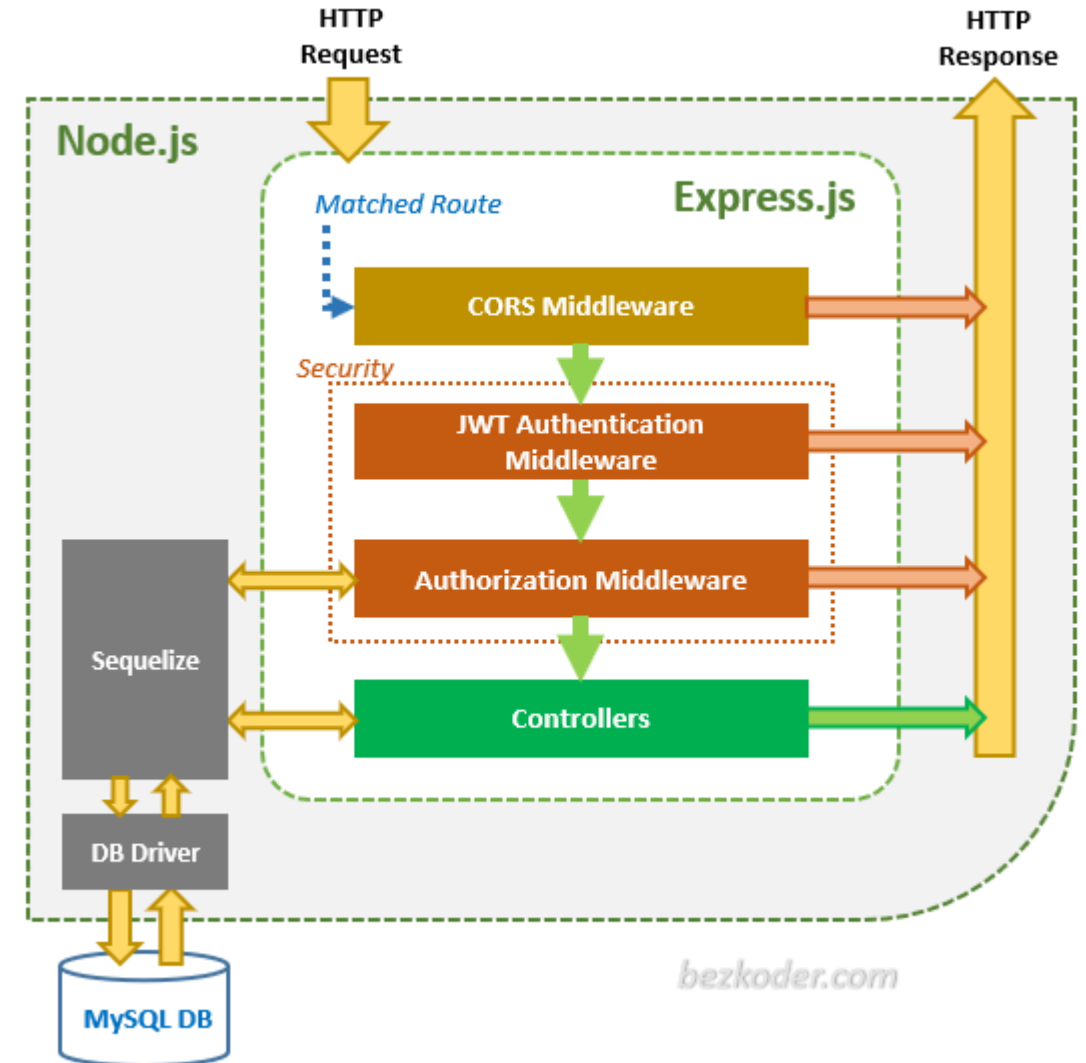
```
{id: 1,  
  name: Joe,  
  url: '...',  
  stream:  
    [{  
      user: 2,  
      title: 'today',  
      body: 'go fly a kite',  
      likes: [  
        {user: 3},  
        {user: 1},  
      ],  
    }]  
}
```


Mongo DB



EXPRESS JS

Express es un framework de Node.js que nos permitirá crear una aplicación web de Backend utilizando código ya preparado, con esto nos referimos a que se encarga de todos los subprocesos que se llevan a cabo al momento de, por ejemplo, realizar una petición http o solicitar un dato de la base de datos, dándonos así un conjunto de métodos y funciones ya preestablecidos para que nosotros podamos enfocarnos solamente en el desarrollo y funcionalidad de nuestra aplicación. Express se instala junto a Node.js para poder crear la aplicación de servidor.

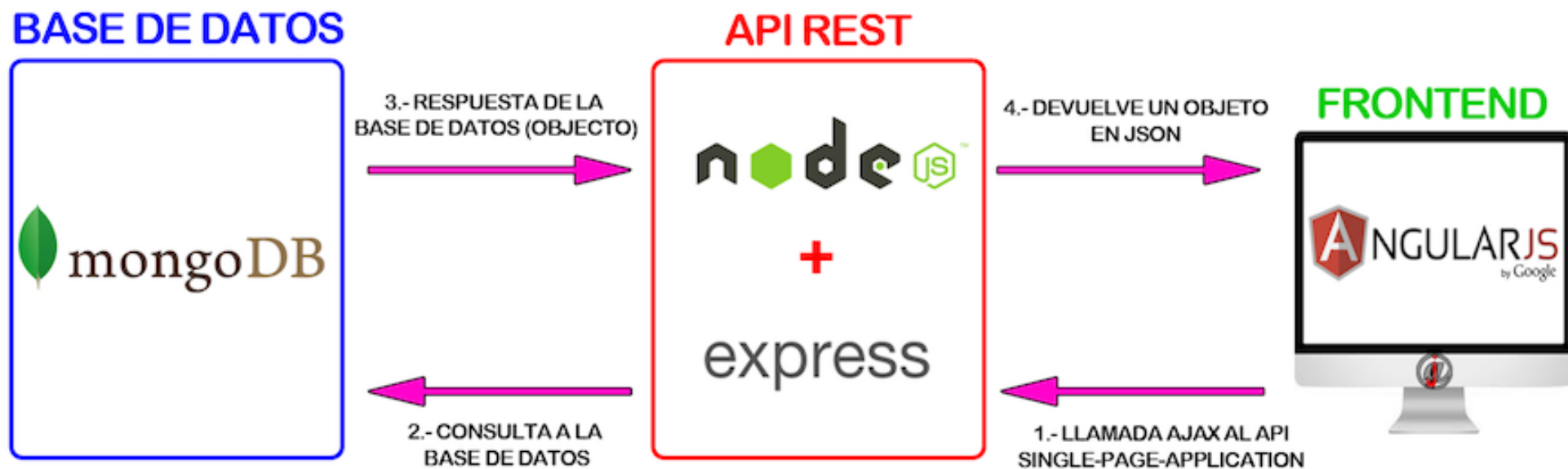


Ventajas

Una de sus mayores ventajas es ser rápido, escalable y eficiente.

Tiene un amplio conjunto de APIs de las que nos podemos servir.

Promueve la reutilización del código con su router incorporado, lo cual nos hace ahorrar tiempo y trabajo.



React

Sirve para desarrollar aplicaciones web de una manera más ordenada y con menos código que si usas Javascript puro o librerías como jQuery centradas en la manipulación del DOM. Permite que las vistas se asocien con los datos, de modo que si cambian los datos, también cambian las vistas.

```
JS App.js x
1  import React, { Component } from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4  import simpsons from './simpsons.js';
5
6  var personaje = simpsons.name;
7
8  class App extends Component {
9    render() {
10     return (
11       <div className="App">
12         <header className="App-header">
13           <img src={logo} className="App-logo" alt="logo" />
14           <h1 className="App-title">Welcome to React</h1>
15         </header>
16         <p className="App-intro">
17           To get started, edit <code>src/App.js</code> and save to reload.
18         </p>
19       </div>
20     );
21   }
22 }
23
24 export default App;
25
```

Ventajas

- Virtual DOM y JSX hacen que React sea mucho más rápido que el resto de los frameworks actuales, aumentando así su rendimiento.
- Permite componentes que promueven la reutilización del código y hacen que la aplicación web en general sea más fácil de entender.
- Con React Native puedes codificar fácilmente aplicaciones basadas en Android o iOS con sólo el conocimiento de JavaScript y React.JS.

JS Identity.ios.js x

```
1  import React from 'react';
2  import { View, Text } from 'react-native';
3  import PropTypes from 'prop-types';
4
5  const Identity = ({ name: { firstName, lastName } }) => (
6
7    <View style={{position: 'absolute', bottom: 100}}>
8      <Text style={{color: 'white', fontSize: 20,
9        textAlign: 'center'}}>Hi {firstName}{lastName}!
10     This app is running on an ios Simulator.</Text>
11   </View>
12
13 );
14
15 Identity.propTypes = {
16   name: PropTypes.object
17 }
18
19 export default Identity;
20
```

Node.js

Node.js es un entorno de ejecución o tecnología de JavaScript que nos va a permitir que el código JS, que usualmente se ejecuta en el navegador, se pueda ejecutar directamente desde el servidor y así podamos trabajar con JS tanto en el Backend como en el Frontend. Y

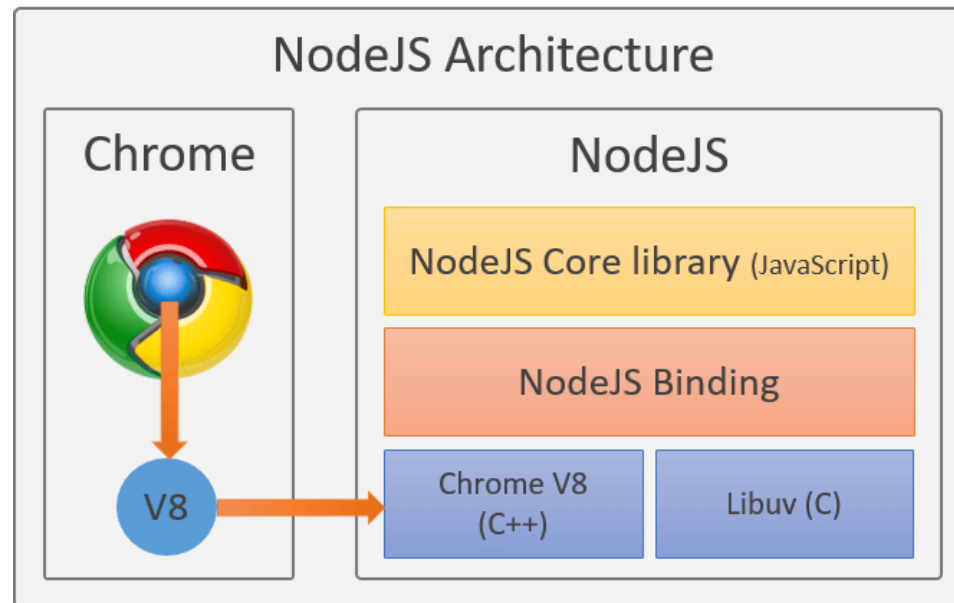
¿cómo se ejecuta JS en el servidor?

Para ello se tiene el motor V8 JavaScript que a la vez es el motor que Google implementa en su navegador Chrome, el cual interpreta el código y después lo ejecuta.



Ventajas

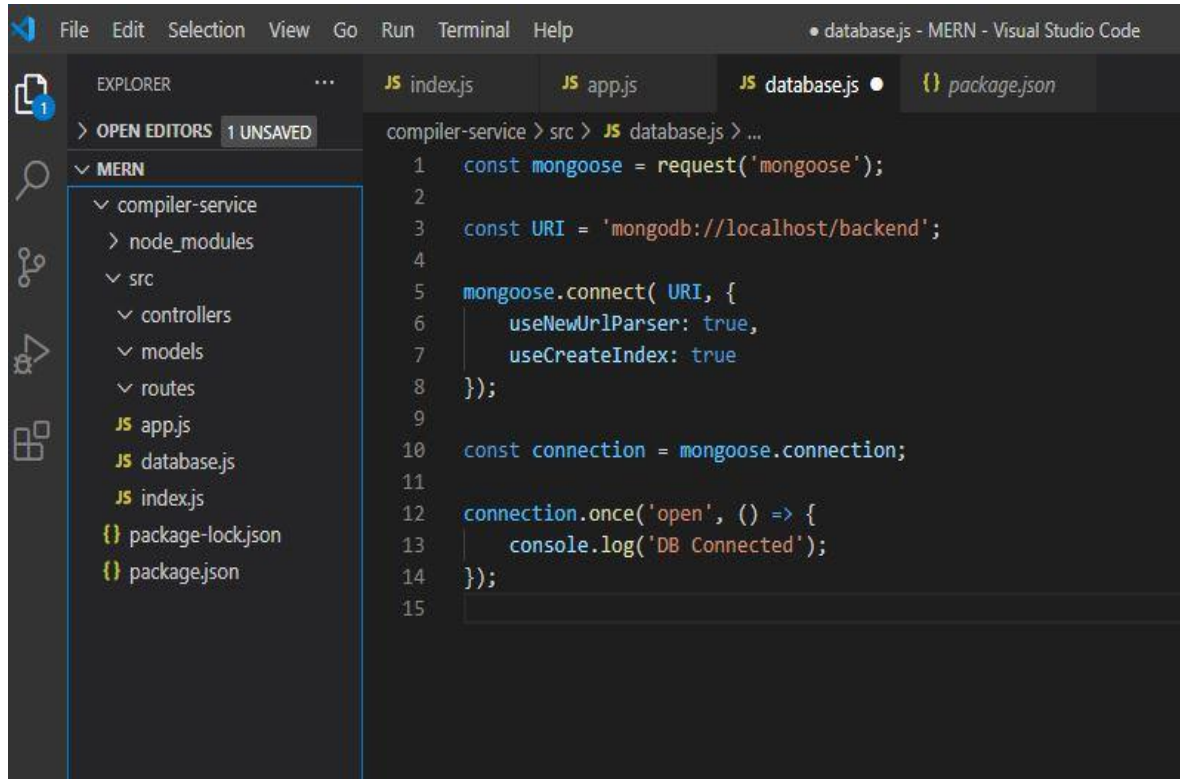
- Está construido sobre el motor JS de Google Chrome, por lo que tiene una gran velocidad de ejecución.
- Node.js también cuenta con un alto grado de escalabilidad.
- Cuenta con un gestor de paquetes de nodos, es decir, npm, que nos permite elegir entre miles de paquetes gratuitos (módulos de nodos) a descargar.



Comandos Iniciales

- **npm init – y:** Crea un archivo package.json el cual es un archivo de meta información donde viene toda la información relacionada al proyecto. Nos permitirá instalar módulos relacionados con el servidor.
- **npm install express:** Framework del servidor
- **npm install mongoose:** Modulo para conectarnos a la base de datos. ORM -> biblioteca para modelar los datos antes de guardarlos. Nos permite validar y crear esquemas.
- **npm install cors:** Permite la comunicación de 2 servidores para evitar errores al enviar y recibir datos.
- **npm install dotenv:** Variables de entorno en el que se guardara variables en el sistema operativo y no queremos que este dentro del código.
- **npm install nodemon –D:** Es para que el servidor se reinicie solo después de cada cambio.

Estructura



- src: permite ordenar la arquitectura
- Index.js: inicia el servidor en el cual se importa los modulos
- App.js: tiene el código del framework
- Database.js: tiene la conncecion a la base de datos
- Controller: funciones que se ejecutaran cuando se llamen a los endpoint
- Model: donde se modela los datos antes de ser guardados
- Routes: define las url del servidor.

Convención de código



Por qué convenciones de código

El 80% del coste del código de un programa va a su mantenimiento.

Casi ningún software lo mantiene toda su vida el autor original

Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo mucho mas rápidamente y mas a fondo

Si distribuyes tu código fuente como un producto, necesitas asegurarte de que esta bien hecho y presentando como cualquier otro producto,

Convención de código

- **Comentario de inicio:** Todos los archivos fuente deben comenzar con un comentario en el que indican copyright, información de la versión y autor.
- **Comentarios de código:** Los comentarios se utilizan para añadir información en el código. Hay 2 formas de comentarios **Por línea** y **Por Bloque**.
 - **Por línea:** El código después de `//` barras no será ejecutado.
 - **Por bloque:** Un slash seguido de un asterisco y termina con un asterisco seguido de un slash.

```
1  /*
2  @command.js Copyright (c) 2021 Jalasoft
3  2643 Av Melchor Perez de Olguin , Colquiri Sud, Cochabamba, Bolivia.
4  Av. General Inofuentes esquina Calle 20,Edificio Union Nº 1376, La Paz, Bolivia
5  All rights reserved
6  This software is the confidential and proprietary information of
7  Jalasoft , Confidential Information "). You shall not
8  disclose such Confidential Information and shall use it only in
9  accordance with the terms of the license agreement you entered into
10 with Jalasoft
11 */
12
13
14 const CompilersServiceError = require('../../common/errors/compilers_service_error');
15
16
17 // Builds the Command class
18 class Command {
19
20     // Defines the constructor for its children
21     constructor() {
22         if (this.constructor == Command) {
23             throw new CompilersServiceError("Abstract classes can't be instantiated.");
24         }
25     }
26
27     // Abstract method for its children that returns an array with the necessary commands to project run.
28     builder(parameters) {
29         throw new CompilersServiceError("Method 'builder()' must be implemented.");
30     }
31 }
```

```
// Returns the project name
get_name_project(){
    return this._name_project
}
```

```
/*
Builds the Command class
Exports Command class
Defines the constructor for its children
*/
```

Convención de código

```
1  /** @module EmployeeAddress */
2
3  /**
4   * @Class
5   * Represents the address of an employee
6   * @extends Address
7   */
8  class EmployeeAddress extends Address{
9
10     /**
11      * @param {string} address1 The primary address line.
12      * @param {string} address2 The secondary address line.
13      * @param {string} city The city where the address resides.
14      * @param {string} state The state where the address resides.
15      * @param {string} zipCode The zip code where the address resides.
16      * @param {number} employeeAddressId The id of the employee address.
17      */
18     constructor(address1, address2, city, state, zipCode, employeeAddressId){
19         super(address1, address2, city, state, zipCode);
20         this.employeeAddressId = employeeAddressId;
21     }
22
23     /**
24      * Calls the employee address save api.
25      * @returns {Promise} A promise to save the employee address in the database.
26      */
27     save(){
28         return new Promise((resolve, reject) => {
29             resolve();
30         })
31     }
32 }
```

Las convenciones de codificación de JavaScript

Las convenciones de codificación son las guías de estilo para la programación. Por lo general se refieren a:

- reglas de denominación y de declaración de variables y funciones.
- Reglas para el uso de espacios en blanco, la sangría, y comentarios.
- Programación de las prácticas y los principios

Las convenciones de codificación segura de calidad:

- Mejora la legibilidad del código
- Hacer más fácil el mantenimiento del código

Convención de código

Los nombres de variables

- En OOP utilizamos camelCase de nombres de identificadores (variables y funciones).
- Todos los nombres comienzan con una letra.

Espacios alrededor de los operadores

- Siempre ponga espacios alrededor de los operadores (= + - * /) , y después de comas:

Código sangría

- Siempre use 4 espacios para el sangrado de bloques de código:

```
firstName = "John";
lastName = "Doe";

price = 19.90;
tax = 0.20;

fullPrice = price + (price * tax);
```

```
const x = y + z;
let values = ["Test1", "Test2", "Test3"];
```

```
// Returns a array with the necessary commands tu run project
builder(parameters) {
  const validate = new ValidationCore();
  validate.command_parameters(parameters);
  let compiler_command = {
    main_command: `${parameters.get_path_binary()}`,
    arguments_list: ['/t:exe', '/out:${parameters.get_path_projects()}'+`${parameters.get_name_project()}`+`${parameters.get_name_project()}.exe', `${parameters.get_path_projects()}'+`${parameters.get_name_project()}`+`${parameters.get_name_project()}.exe',
  };
  let runner_command = {
    main_command: `${parameters.get_path_projects()}'+`${parameters.get_name_project()}`+`${parameters.get_name_project()}.exe',
    arguments_list: ['`']
  };
  return [compiler_command, runner_command];
}
```

Reglas de los estados

Reglas generales para declaraciones simples:

- Siempre termine una declaración simple con un punto y coma.

```
var values = ["Volvo", "Saab", "Fiat"];
```

```
var person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};
```


Reglas de los estados

Las reglas generales para instrucciones complejas (compuestos):

- Coloque el soporte de abertura en el extremo de la primera línea.
- Utilice un espacio antes del corchete de apertura.
- Coloque el soporte de cierre en una nueva línea, sin espacios iniciales.
- No finalice un comunicado complejo con un punto y coma.

Funciones:

```
function toCelsius(fahrenheit) {  
    return (5 / 9) * (fahrenheit - 32);  
}
```

Bucles:

```
for (i = 0; i < 5; i++) {  
    x += i;  
}
```

Condicionales:

```
if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

Reglas de Objetos

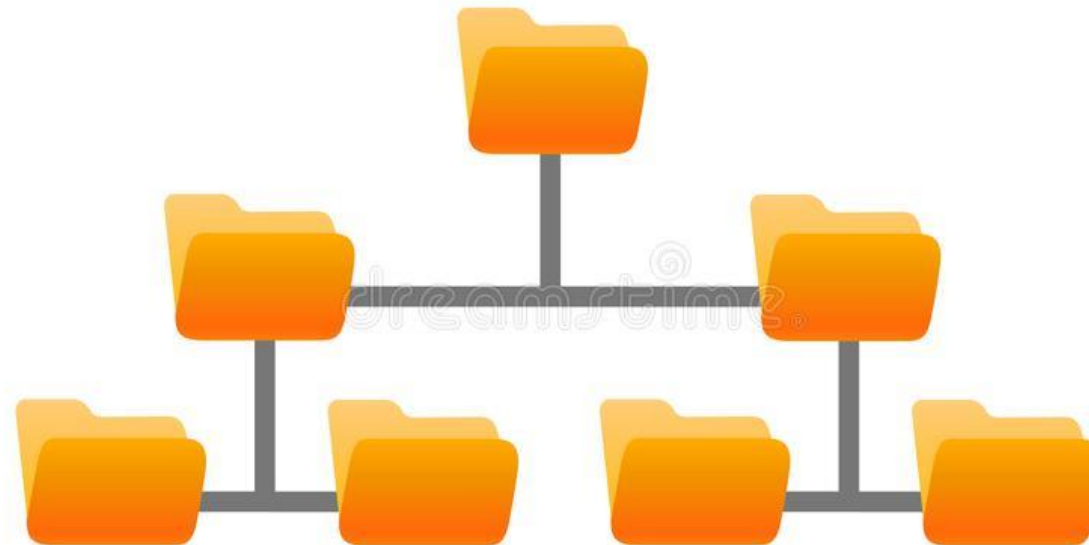
Reglas generales para las definiciones de objeto:

- Coloque el la llave apertura en la misma línea que el nombre del objeto.
- Utilice dos puntos más un espacio entre cada propiedad y su valor.
- Utilice comillas alrededor de los valores de cadena, no en torno a valores numéricos.
- No añadir una coma después de la última pareja propiedad-valor.
- Coloque la llave de cierre en una nueva línea, sin espacios iniciales.
- Siempre termine una definición de objeto con un punto y coma.

```
const person = {  
  firstName: "Juan",  
  lastName: "Perez",  
  age: 30,  
  eyeColor: "black"  
};
```

```
/*  
@node_command.js Copyright (c) 2021 Jalasoft  
2643 Av Melchor Perez de Olguin , Colquiri Sud, Cochabamba, Bolivia.  
Av. General Inofuentes esquina Calle 20,Edificio Union Nº 1376, La Paz, Bolivia  
All rights reserved  
This software is the confidential and proprietary information of  
Jalasoft , Confidential Information "). You shall not  
disclose such Confidential Information and shall use it only in  
accordance with the terms of the license agreement you entered into  
with Jalasoft  
*/
```

Paquetes – Estructura de folder



Carpetas

Una **carpeta** en JavaScript es un contenedor de clases que permite agrupar las distintas partes de un programa y que por lo general tiene una funcionalidad y elementos comunes, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica.

Carpeta es un mecanismo para encapsular un grupo de clases, sub-carpetas e interfaces. Las carpetas se usan para: Prevención de conflictos de nombres. Facilitar la búsqueda / localización y el uso de clases. Proporcionar acceso controlado.

Las **carpetas** son la forma mediante la cual JavaScript permite agrupar clases, interfaces, excepciones, constantes, etc. De esta forma, se agrupan conjuntos de estructuras de datos y de clases con algún tipo de relación en común.

Sin carpetas



Con carpetas



Práctica 2

Crear una aplicación(web service) que realice los siguiente.

1. Utilizando postman la aplicación debe retornar “Hello World” usando el siguiente endpoint
 - Post: <http://localhost:8080/hello>
2. Utilizando postman la aplicación debe recibir un parámetro del tipo String por ejemplo: name: “Juan Perez”
Y usando este valor debe retornar el siguiente mensaje “Hello Juan Perez” usando el siguiente endpoint
 - Post: <http://localhost:8080/hello>

Práctica 2

POST

localhost:8080/hello

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

	KEY	VALUE	DESCRIPTION	..
<input checked="" type="checkbox"/>	name	Juan		
<input checked="" type="checkbox"/>	lastName	Perez		
	Key	Value	Description	

Body

Cookies

Headers (3)

Test Results

Status: 200 OK Time: 98 ms Size: 132 B Save

Pretty

Raw

Preview

Visualize

Hello Juan Perez

Sintaxis Básica

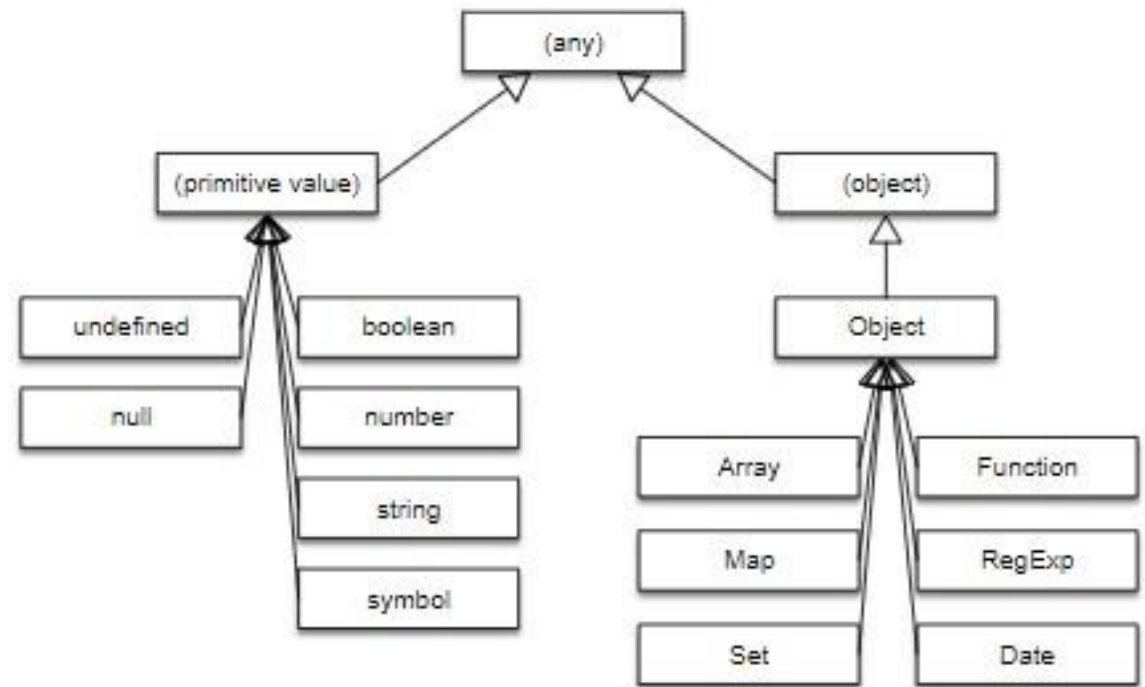
- Tipo de datos, expresiones y variables
- For, for-each y while
- Flow control

Tipos de datos en javascript

JavaScript es un lenguaje de tipado débil o dinámico. Esto significa que no es necesario declarar el tipo de variable antes de usarla.

En los lenguajes de tipado dinámico como JavaScript o Python la comprobación de los tipos se realiza durante la ejecución del programa en vez de durante la compilación.

En los lenguajes de tipado estático o fuertemente tipados como C#, Go, Java o C++ la comprobación de los tipos se realiza durante la compilación, y no durante la ejecución.

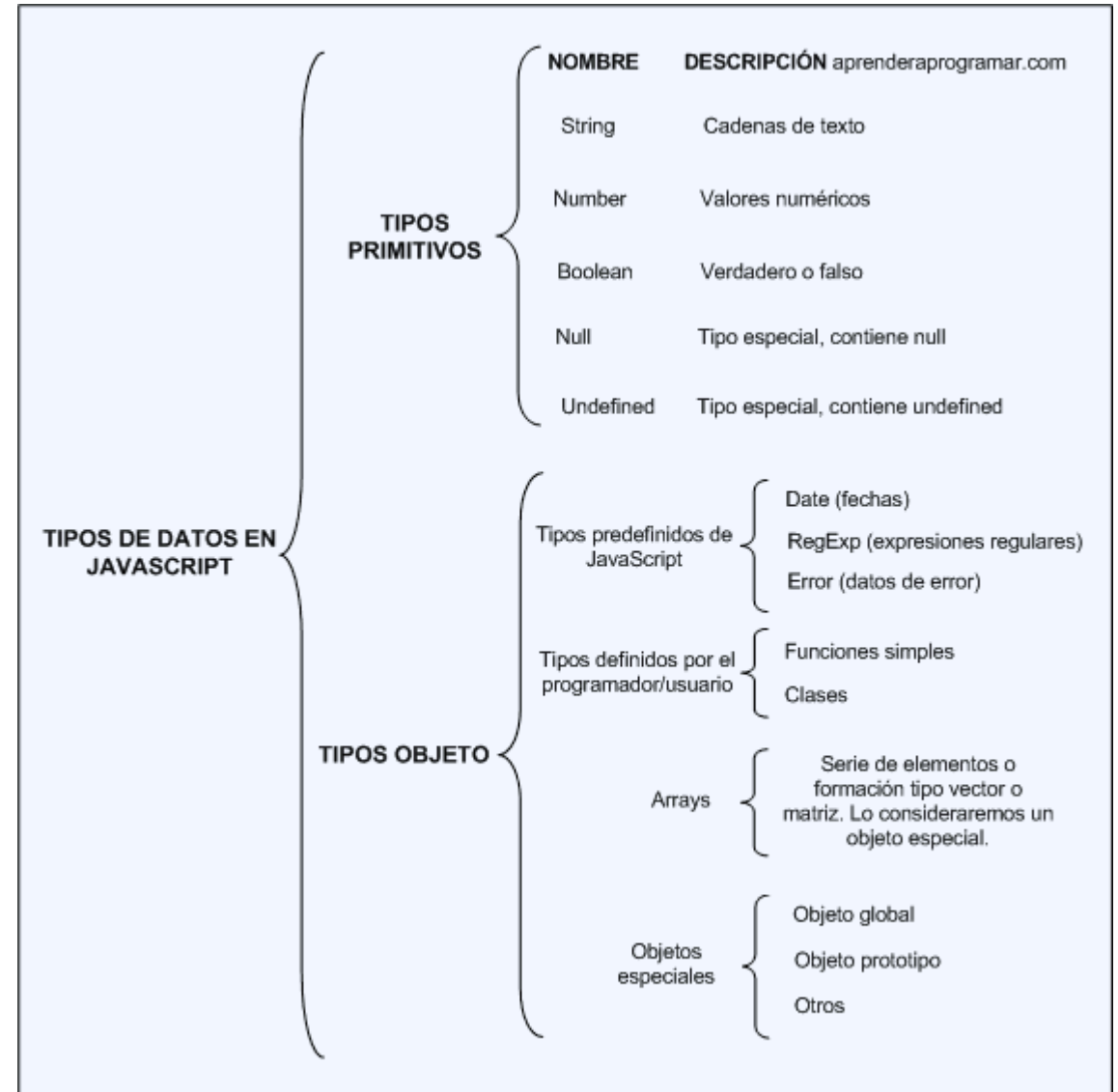


Tipo de datos

En Javascript existen 7 tipos de datos

- **String** Cadenas de texto.
- **Number** Valores numéricos.
- **Boolean** Representa una entidad lógica y puede tener dos valores: true y false.
- **null** Es un valor asignado tiene el valor de “no valor”.
- **undefined** Una variable a la que no se le ha asignado ningún valor tiene el valor undefined.
- **Symbol** Nuevo en ECMAScript 2015.
- **Object** Un valor en memoria al que podemos acceder por un identificador.

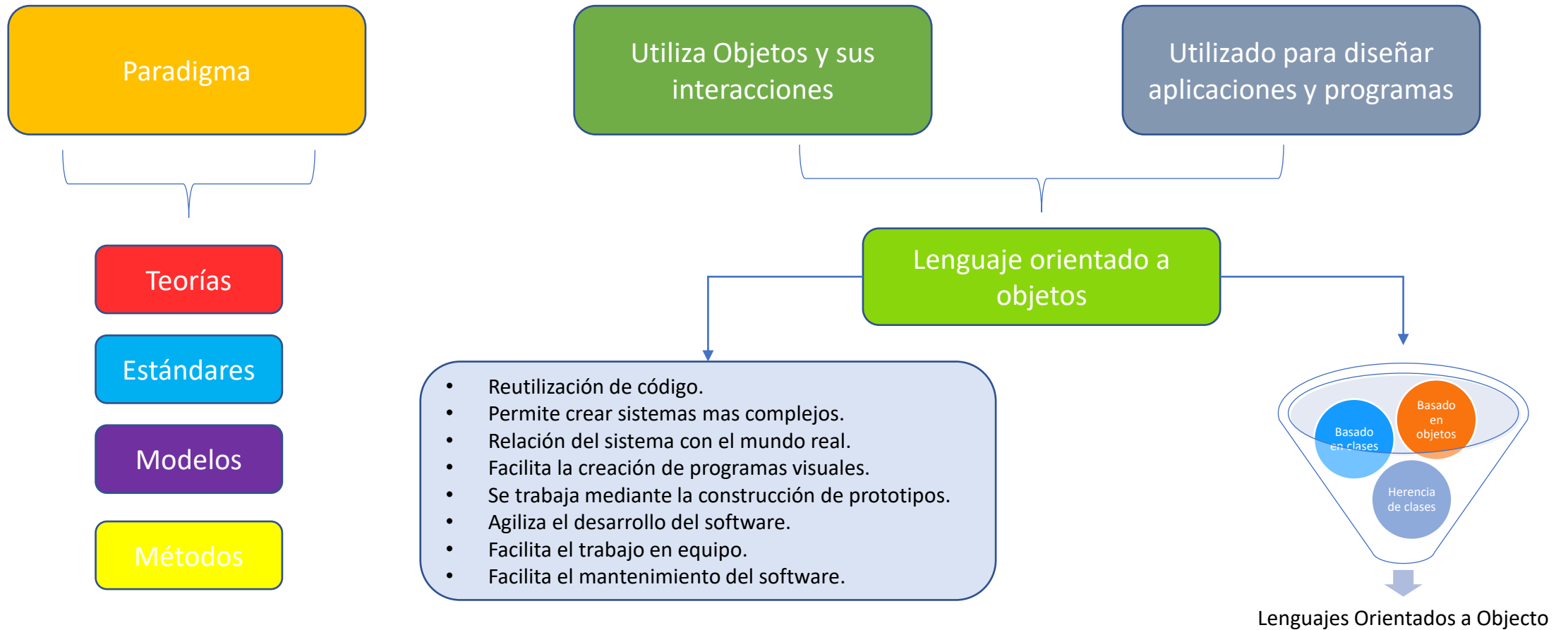
Estos tipos se dividen en dos grupos, Primitivos y de Objeto.



Programación Orientada a Objetos



Definición POO



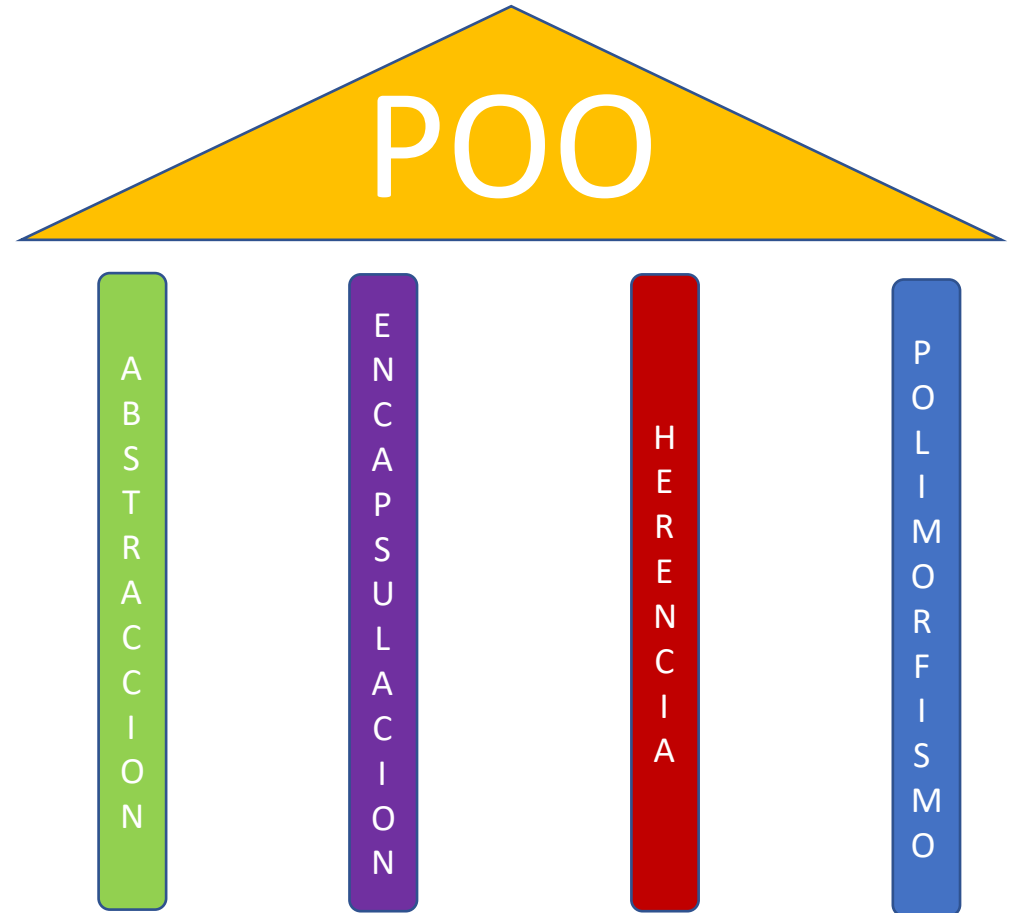
La **programación orientada a objetos (POO)** es un paradigma de programación que usa objetos para crear aplicaciones. Está basada en pilares fundamentales: Abstraccion herencia, polimorfismo, encapsulación.

POO

En la programación orientada a objetos, podemos encontrarnos con los siguientes elementos:

- Clases
- Objetos
- Métodos
- Atributos
- Mensajes
- **Abstracción**
- **Encapsulamiento**
- **Herencia**
- **Polimorfismo**

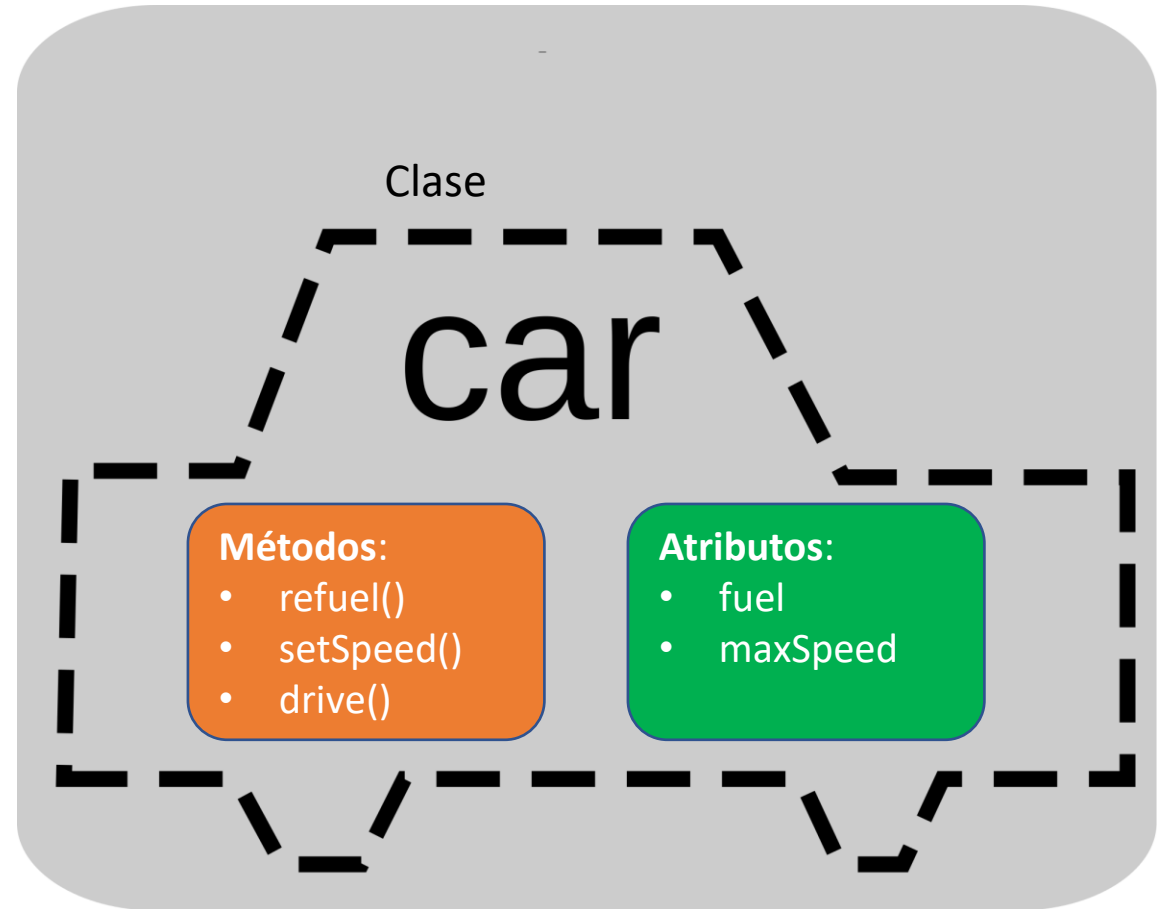
Pilares fundamentales de
la programación orientada
a objetos



QUÉ ES UNA CLASE?

Las clases son la base de la programación orientada a objetos, una clase es una **plantilla, molde** o modelo **para crear objetos**.

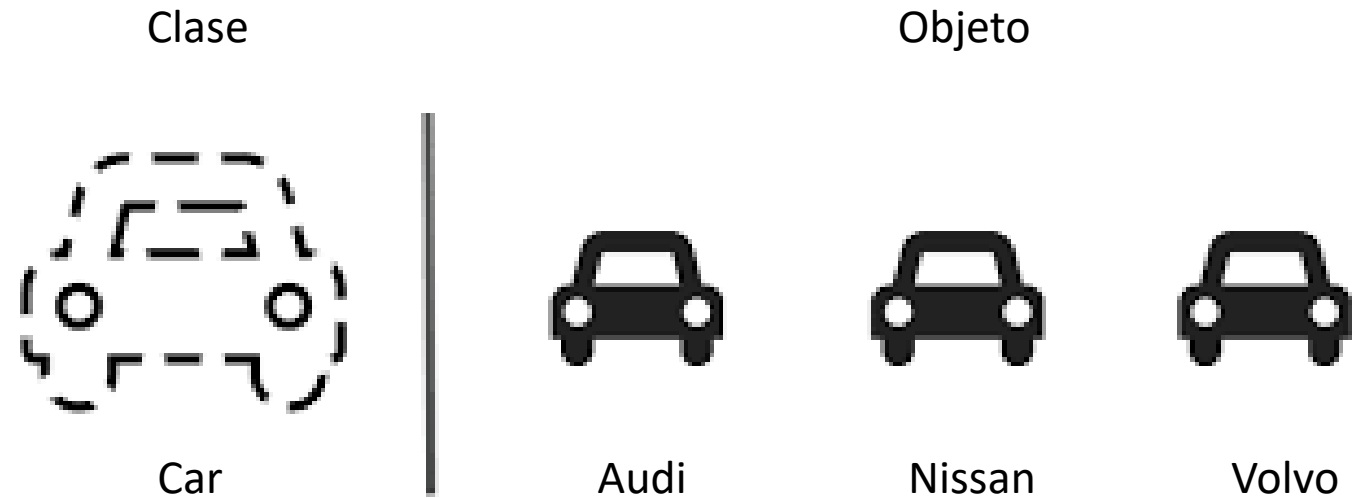
Una clase está compuesta por características, propiedades o atributos (**variables**) y por su comportamiento (**métodos**) que trabajan sobre las propiedades.



QUÉ ES UN OBJETO?

Un objeto es una instancia de una clase, una instancia es un ejemplar, un caso específico de una clase, por ejemplo yo puedo crear varios objetos de tipo clase y cada uno es diferente, por ejemplo el primero objeto es de color negro, el segundo objeto es de color blanco.

Como cada objeto es diferente se le conoce como instancia, entonces cuando escuches el término “hay que instanciar “, se refiere a crear un nuevo objeto y que cada objeto es diferente.



Métodos



- Encender
- Acelerar
- Frenar



- Correr
- Caminar
- Dormir



En una aplicación para que realice una tarea se requiere un método que vendría siendo las funcionalidades de un objeto, el cual contiene todas las instrucciones necesarias del programa para realizar la tarea, si lo vemos desde el lado del usuario los métodos son las acciones que se ejecutan realizando alguna acción internamente, en java estos métodos están alojados en una clase.

Atributos



- Matricula
- Color
- velocidad



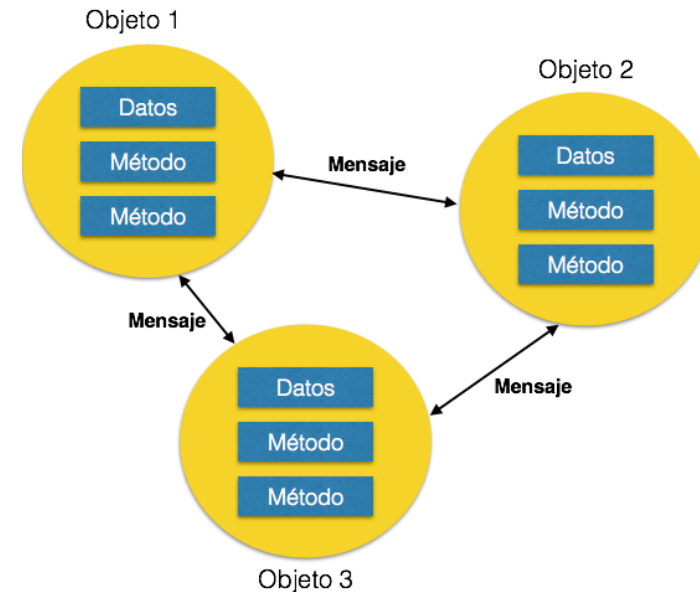
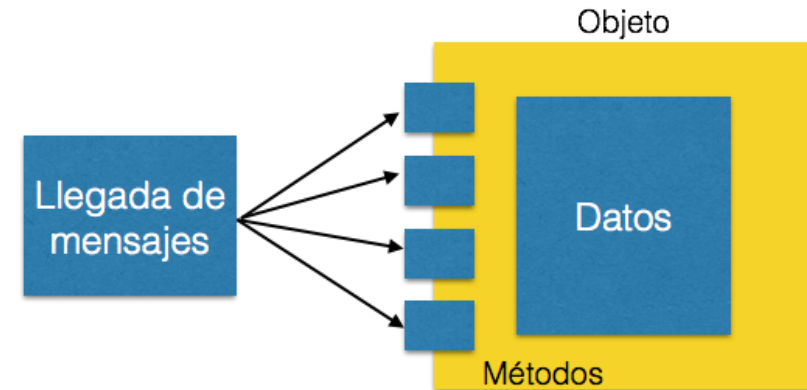
- Especie
- Edad
- color



Si tomamos el típico ejemplo del automóvil como un objeto, sus atributos serían el color, su matricula, su velocidad, el tamaño, etc., es decir todo aquello que lo puede describir de manera mas detallada.

Mensajes

- Un objeto envía un mensaje a otro.
 - Esto lo hace mediante una llamada a sus atributos o métodos.
- Los mensajes son tratados por la interfaz pública del objeto que los recibe.
 - Eso quiere decir que sólo podemos hacer llamadas a aquellos atributos o métodos de otro objeto que sean públicos o accesibles desde el objeto que hace la llamada.
- El objeto receptor reaccionará.
 - **Cambiando su estado:** es decir modificando sus atributos.
 - **Enviando otros mensajes:** es decir llamando a otros atributos o métodos del mismo objeto (públicos o privados) o de otros objetos (públicos o accesibles desde ese objeto)

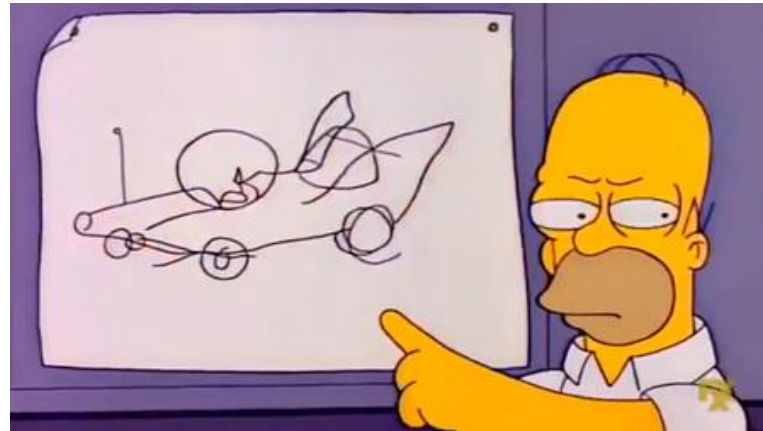


Abstracción

El término abstracción consiste en **ver a algo como un todo sin saber cómo está formado internamente.**

Se dice que las personas gestionan la complejidad a través de la abstracción, que quiere decir esto, por ejemplo para alguien es difícil entender todos los componentes, circuitos de un televisor y como trabajan, sin embargo es más fácil conocerlo como un todo, como un televisor, sin pensar en sus detalles o partes internas.

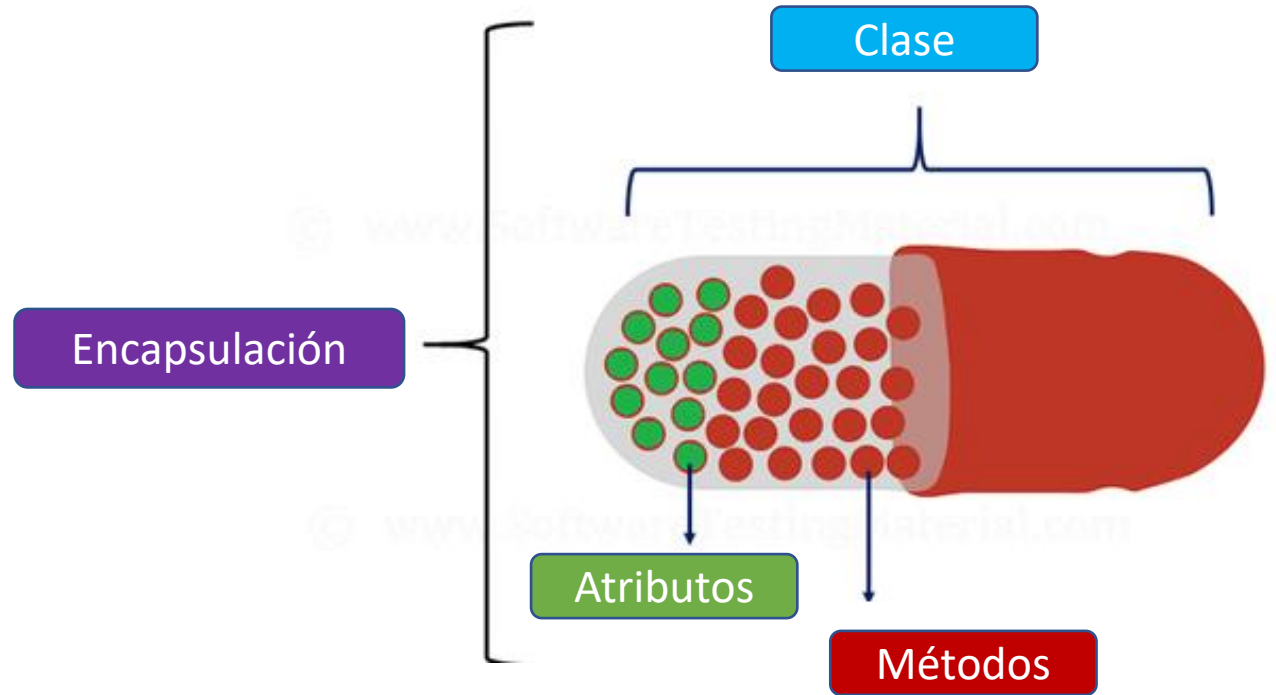
Abstracción



Énfasis en el ¿Qué hace?
mas que el ¿Cómo lo
hace?

Encapsulación

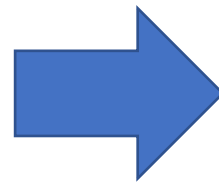
- Cuando se habla de encapsulamiento, hablamos de **ocultar la información**. Esto significa que sólo se debe mostrar los detalles esenciales de un objeto, mientras que los detalles no esenciales se los debe ocultar.



Clases

Con la salida de ES6, se agregó el concepto de clases en JavaScript, ¿esto quiere decir que antes de ES6 no existían clases?. Básicamente no, no existían clases antes de ES6, al menos no semánticamente hablando

A simple vista se nota que va mucho más acorde a la creación de clases de otros lenguajes de programación, haciendo así que sea mucho mas semántico a la hora de definir clases.



```
class Persona {  
  constructor(nombre, edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
  }  
  hablar () {  
    console.log('Hola!');  
  }  
}
```

¿Que se puede usar dentro de una clase?

- **Constructor:** En el ejemplo de la sintaxis de ES6 para clases, creamos una clase vacía. Pero ¿que sucede si necesitamos recibir por parámetros los datos necesarios para instanciar una clase? Es aquí donde entra el *constructor*, que es el encargado de inicializar las instancias de una clase.
- **Setters y Getters:** Otra de las herramientas atribuidas a las con ES6 son los setters y getters, que son de alguna manera, variaciones de una función, que te permiten cambiar valores y obtener valores.

```
PRACTICE > src > gettersetter > JS Person.js > ...
1  class Person {
2      constructor(firstName) {
3          this.__firstName = firstName;
4      }
5
6      get firstName() {
7          return this.__firstName;
8      }
9
10     set firstName(value) {
11         this.__firstName = value;
12     }
13 }
14
15 module.exports = Person;
16
```

¿Que se puede usar dentro de una clase?

- **Métodos estáticos:** Los métodos estáticos nos son mas que aquellos métodos que solo pueden ser ejecutados desde la clase y no desde una instancia. Por lo general se usar para cosas que tengan un aporte a la clase como tal y que no varia según la instancia.
- **Métodos públicos:** Los métodos públicos son todos aquellos métodos que se usan para hacer procedimientos y que no necesitas que devuelvan algo en específico. Estos métodos son llamados desde las instancias..

```
1 class StaticMethodCall {
2   static staticMethod() {
3     return 'Haciendo algo desde un método estático';
4   }
5 }
6
7 StaticMethodCall.staticMethod()
8 // Haciendo algo desde un método estático
9
10
11 let staticMethodCall = new StaticMethodCall
12 staticMethodCall.staticMethod()
13 // TypeError: staticMethodCall.staticMethod is not a function
```

```
1 class PublicMethodCall {
2   publicMethod() {
3     console.log('empezando algún procedimiento desde un método público');
4     console.log('terminando algún procedimiento desde un método público');
5   }
6 }
```

Herencia

Parte de lo que implica la Programación Orientada a Objetos, implica la herencia entre clases, es aquí donde usamos la nueva palabra reservada ***extends***

```
class Polygon {  
  constructor(height, width) {  
    this.name = 'Polygon';  
    this.height = height;  
    this.width = width;  
  }  
  
  sayName() {  
    console.log('Holaaaaa, yo soy ', this.name + '.');  
  }  
  
  static staticMethod() {  
    console.log('STATIIIIIC DESDE POLYGON')  
  }  
}
```

```
class Square extends Polygon {  
  constructor(length) {  
    super(length, length);  
    this.name = 'Square';  
  }  
  
  get area() {  
    return this.height * this.width;  
  }  
  
  set area(value) {  
    this.area = value;  
  }  
}
```

```
let s = new Square(5);  
console.log(s.sayName());  
// Holaaaaa, yo soy Square.  
console.log(s.name);  
// Square  
console.log(s.area);  
// 25  
console.log(s.height);  
// 5  
console.log(s.width);  
// 5  
console.log(Square.staticMethod());  
// STATIIIIIC DESDE POLYGON
```


Clases abstractas

- Las clases abstractas, como su nombre lo indica, son algo abstracto, no representan algo específico y las podemos usar para crear otras clases. No pueden ser instanciadas, por lo que no podemos crear nuevos objetos con ellas.
- Habrá ocasiones en las cuales necesitemos crear una clase padre donde únicamente coloquemos la estructura de una abstracción, una estructura muy general, dejando que sean las clases hijas quienes definan los detalles. En estos casos haremos uso de las clases abstractas.
- Una clase abstracta es prácticamente idéntica a una clase convencional; las clases abstractas pueden poseer atributos, métodos, constructores, etc ... La principal diferencia entre una clases convencional y una clase abstracta es que la clase abstracta debe poseer por lo menos **un** método abstracto.

Manejo de excepciones



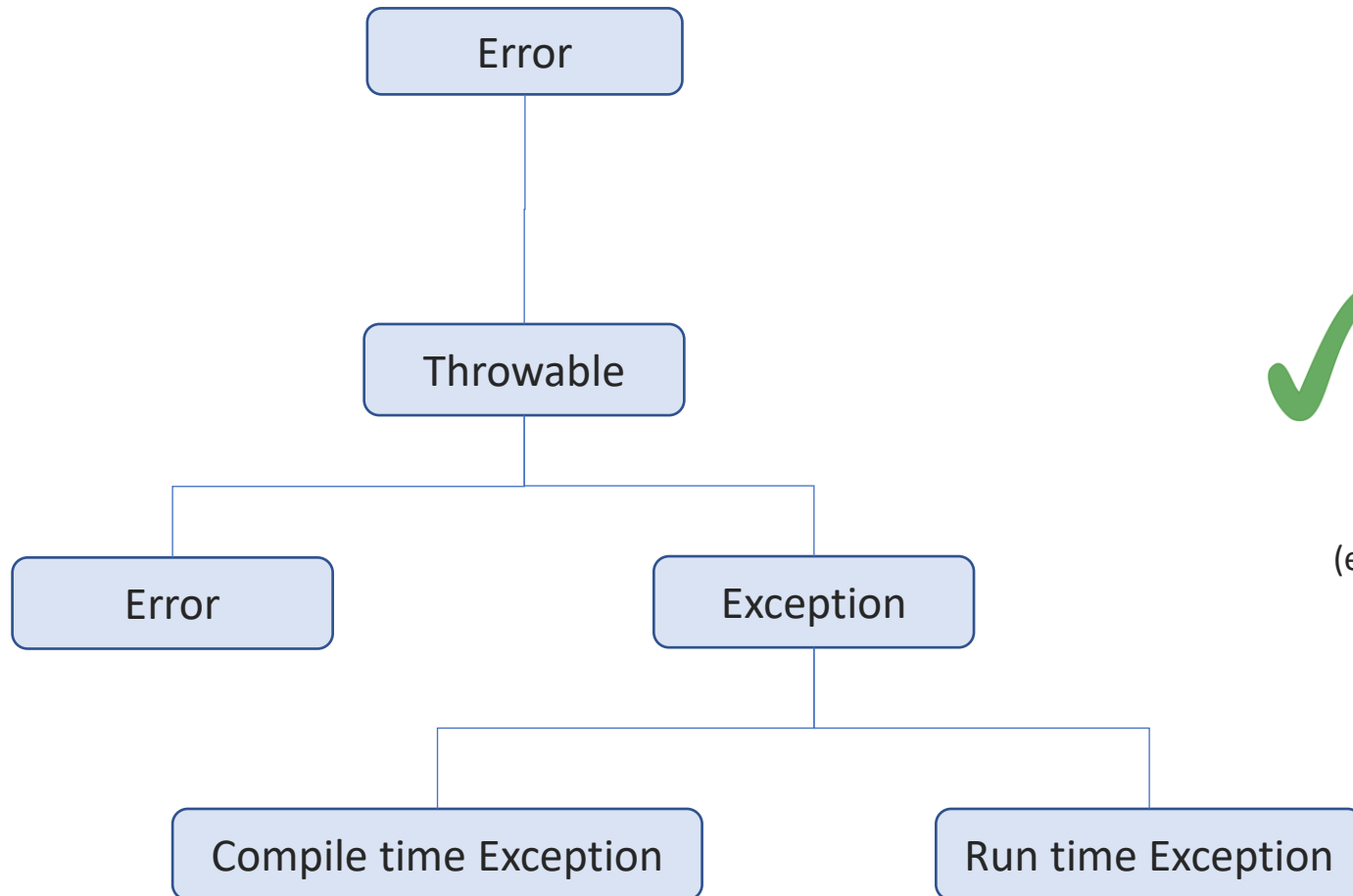
¿Qué es una excepción?

“Una excepción es la indicación de un problema que ocurre durante la ejecución de un programa”

Se le llama excepción porque se trata de un error que no sucede con frecuencia, es decir “una excepción a la regla”. La “regla”, por supuesto, es la ejecución normal del programa.

Excepción

Jerarquía de Errores



Programadores



IOException
(excepciones comprobadas)



RuntimeException
(excepciones no comprobadas)

Captura de excepciones try - catch - finally

- **try** : Aquí vamos a escribir todo el bloque de código que posiblemente llegue a lanzar una excepción la cual queremos manejar, aquí va tanto el código como llamados a métodos que puedan arrojar la excepción.
- **catch** : en caso de que en el try se encuentre alguna excepción, se ingresará automáticamente al bloque catch donde se encontrará el código o proceso que queremos realizar para controlar la excepción.
- **finally** : Este bloque es opcional, lo podremos si queremos ejecutar otro proceso después del try o el catch, es decir, siempre se ejecutará sin importar que se encuentre o no una excepción.



```
try {  
    // Código que pueda generar Errores ("Exception's")  
} catch(Tipo1 id1) {  
    // Manejar "Exception's" para la Clase Tipo1  
}  
catch(Tipo2 id2) {  
    // Manejar "Exception's" para la Clase Tipo2  
}  
catch(Tipo3 id3) {  
    // Manejar "Exception's" para la Clase Tipo3  
}  
finally {  
    // Actividades que siempre ocurren  
}
```

Lanzamiento de excepciones: throw

- **La sentencia throw** : Se utiliza en javascript para lanzar objetos del tipo Throwable, al lanzarse una excepción se sale inmediatamente del bloque de código actual

Propagación de excepciones: throws

- Si en el cuerpo de un método se lanza una excepción (de un tipo derivado de la clase Error), este puede propagarse a través de todas las llamadas hasta llegar a la clase solicitante.

```
try {  
    var error = ...;  
    if (error) {  
        throw "An error";  
    }  
    return true;  
  
} catch (e) {  
    alert (e);  
    return false;  
  
} finally {  
    //do cleanup, etc here  
}
```

Excepción

- No es sencillo al momento de decidir como y que excepciones deben ser lanzadas o manejadas.
- Los equipo de desarrollo tienen sus propio conjunto de reglas sobre cuando utilizarlas.
- Existen buenas practicas que son utilizadas por la mayoría de los equipos.



Buenas Prácticas:

1. Limpia los recursos en el bloque Finally o utiliza la sentencia de Try con recursos.
2. Utilizar Excepciones específicas.
3. Documente las excepciones que especifique.
4. Lance excepciones con mensajes descriptivos.
5. Captura la excepción mas específica primero.
6. No capture throwable.
7. No ignore excepciones.
8. No registre una excepción para luego lanzarla.
9. Envuelva la excepción sin consumirla



Excepción

```
1 FactoryController.prototype.create = function (callback) {
2   //The throw is working, and the exception is returned.
3   throw new Error('An error occurred'); //outside callback
4   try {
5     this.check(function (check_result) {
6       callback(check_result);
7     });
8   } catch (ex) {
9     throw new Error(ex.toString());
10  }
11 }
12
13 FactoryController.prototype.create = function (callback) {
14   try {
15     this.check(function (check_result) {
16       //The throw is not working on this case to return the exception to the caller(parent)
17       throw new Error('An error occurred'); //inside callback
18     });
19   } catch (ex) {
20     throw new Error(ex.toString());
21   }
22 }
```

Promesas JS



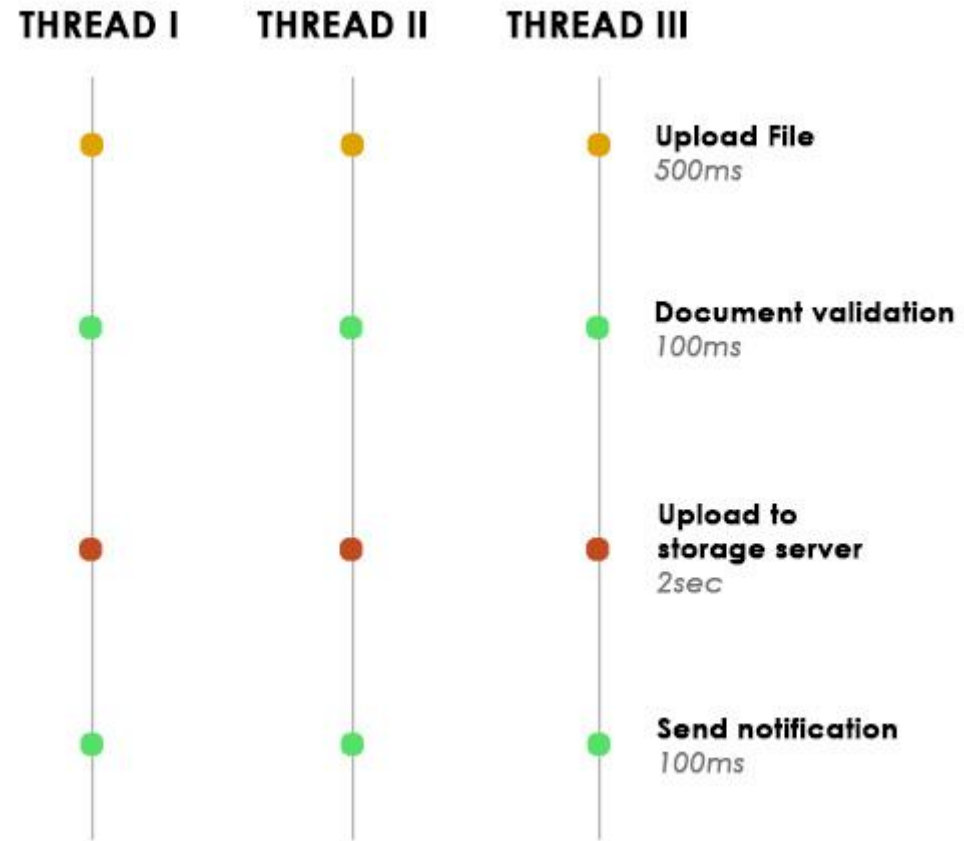
¿QUÉ ES UN PROCESO SÍNCRONO?

Por defecto nuestras aplicaciones tienen el comportamiento de ser síncrono, es decir que un proceso desencadenado no va a permitir ejecutar otro hasta que este haya finalizado. Por ende, esto genera una cola o tráfico dentro de nuestra aplicación



¿QUÉ ES UN PROCESO ASÍNCRONO?

Teniendo clara la explicación anterior, la asincronía nos va a permitir ejecutar nuestros procesos en varios hilos de ejecución o lo que se conoce como **multithreading**. Partiendo del caso que vimos anteriormente el bloqueo, cola o tráfico ya no va a existir porque los usuarios no van a tener que esperar a que un hilo se libere para que otro pueda entrar.



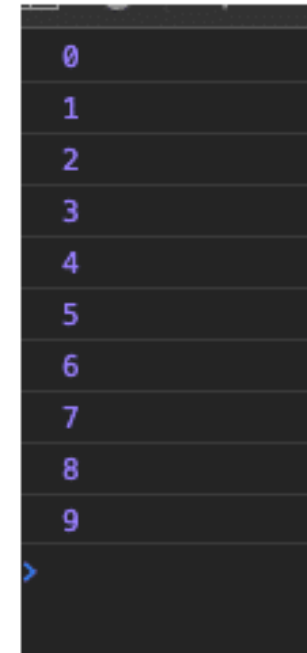
¿Javascript sincrónico o asíncrónico?

Vamos a echarle un vistazo a cómo JavaScript funciona realmente. Para ello es suficiente con construir un bucle for.

```
for (let i=0;i<10;i++) {
```

```
    console.log(i);
```

```
}
```



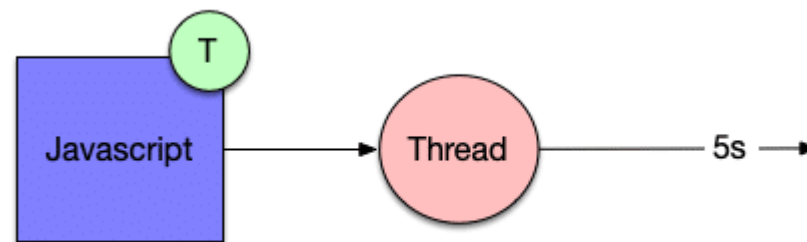
```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
>
```

¿Javascript sincrono o asincrono?

Podemos ahora construir un programa que nos da la sensación de ser asíncrono en el cual pulsamos un botón y solicitamos que después de un intervalo de tiempo se muestre un mensaje por la consola.

Muchas personas piensan que cuando este código se ejecuta hay un Thread de JavaScript que se abre para realizar esta tarea al cabo de 5 segundos .

```
function mensaje() {  
  
    setTimeout(function() {  
  
        console.log("hola desde javascript");  
  
    }, 5000);  
}
```



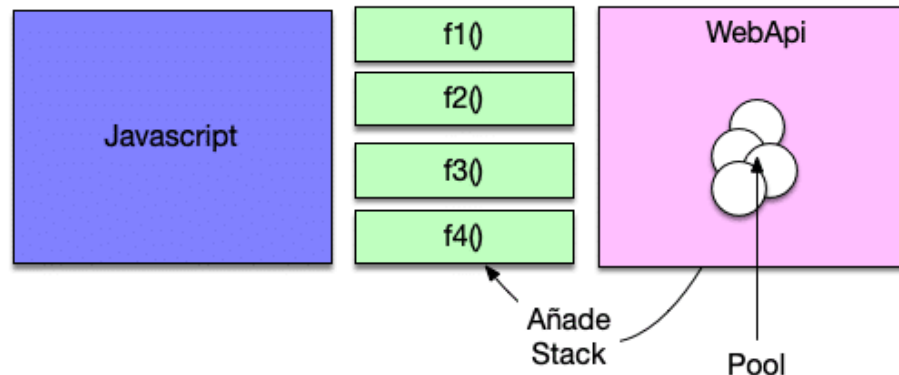
Promesas

¿Javascript sincrono o asincrono?

```
src > <> test.html > html
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Page Title</title>
5  <script type="text/javascript">
6  |   function times() {
7  |       |   setTimeout(() => {
8  |       |       |   console.info("hello");
9  |       |       |   }, 3000);
10 |       |   }
11 |   }
12 |   function alert2 (){
13 |       |   alert("hola");
14 |       |   }
15 |   }
16 </script>
17 </head>
18 <body>
19 |   <input type="button" value="time" onclick="times()" />
20 |   <input type="button" value="alert" onclick="alert2()" />
21 |   }
22 </body>
23 </html>
```

¿Javascript sincrónico o asíncrónico?

Ahora tenemos claro que el motor de JavaScript es sincrónico. Las cosas no son tan complicadas como parece. El navegador tiene un pool de Thread a nivel de WebAPI fuera del runtime de JavaScript que es el que se encarga de realizar estas tareas asíncronas cómo puede ser una petición Ajax . Cuando esa petición Ajax termine el pool registra en la pila de llamadas de JavaScript una nueva función con el resultado de la operación. Esta función será ejecutada por el motor de JavaScript en su event loop de forma sincrónica. Es decir en cuando no tenga otra cosa que hacer.

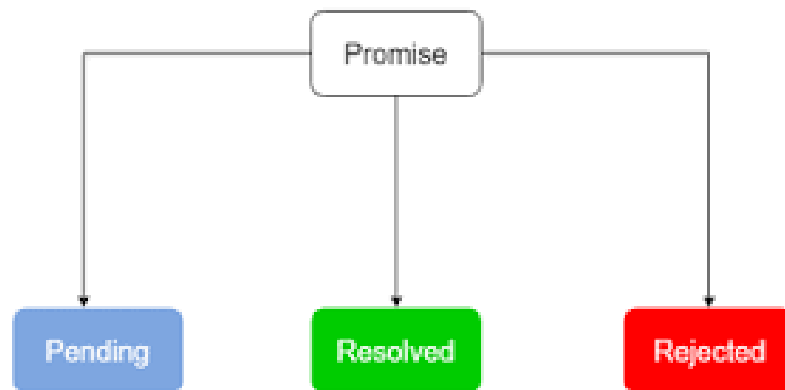


Promesas

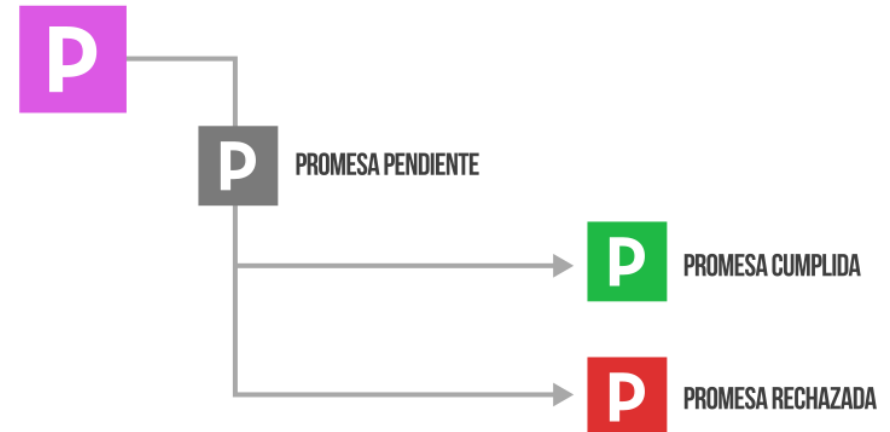
¿Javascript sincrónico o asíncrónico?

Una Promise es un objeto que representa la terminación o el fracaso de una operación asíncrona. Esencialmente, una promesa es un objeto devuelto al cuál se adjuntan funciones callback, en lugar de pasar callbacks a una función.

States of a Promise



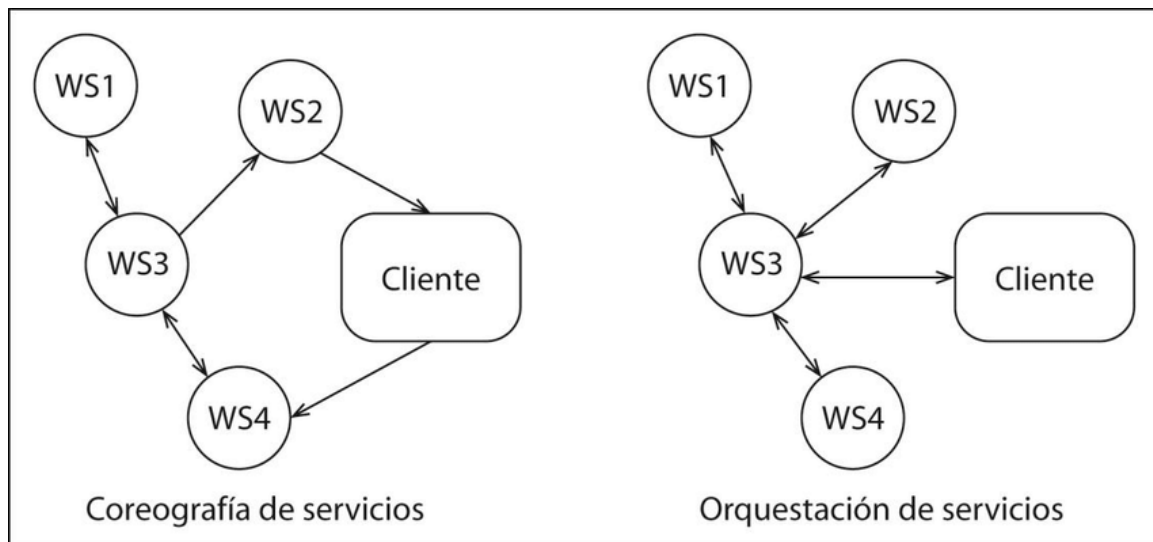
PROMESAS



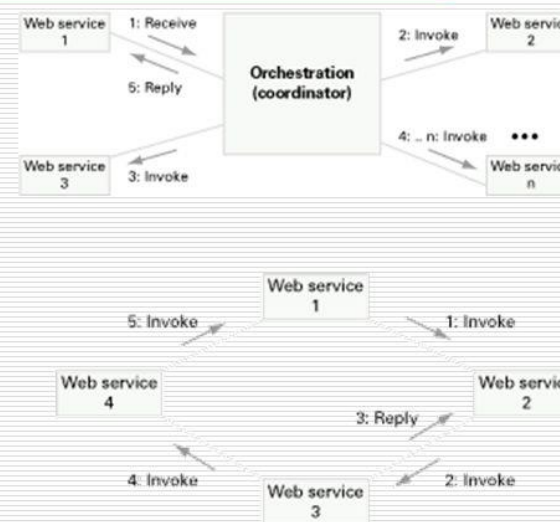


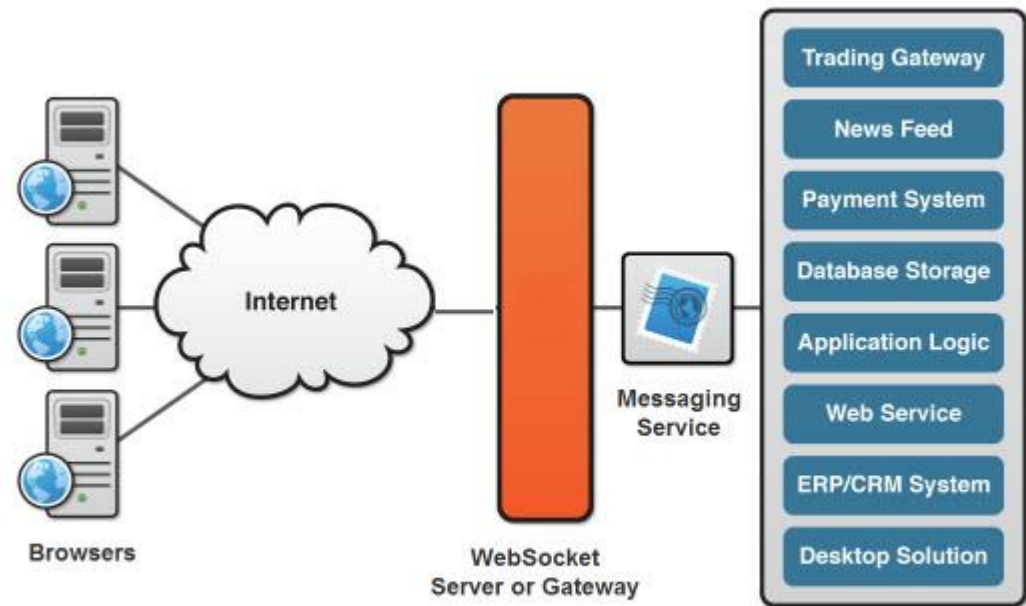
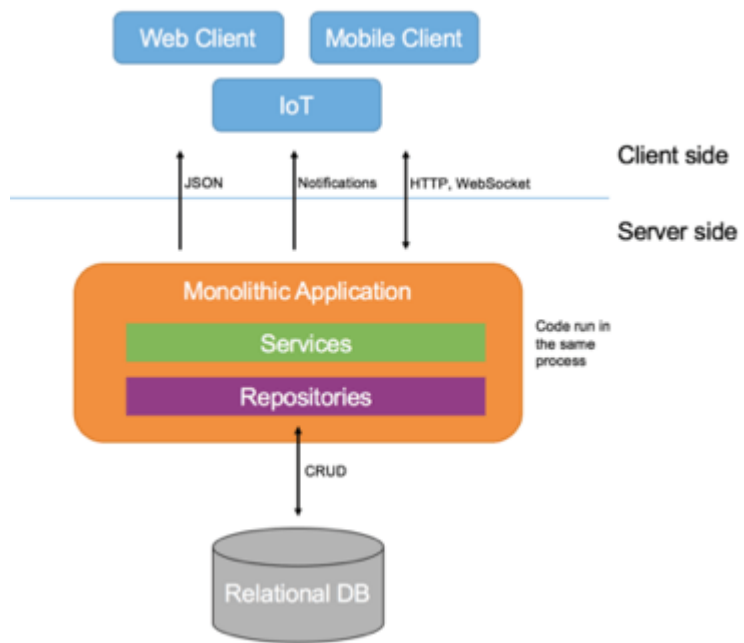
ARQUITECTURAS

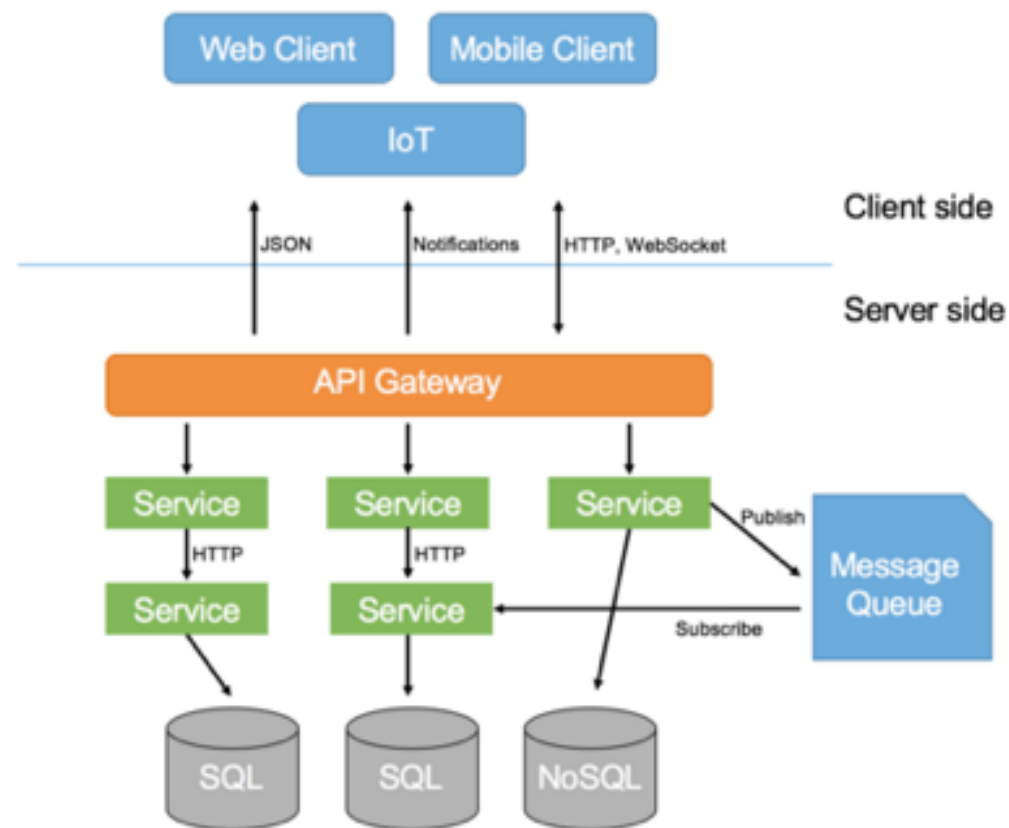
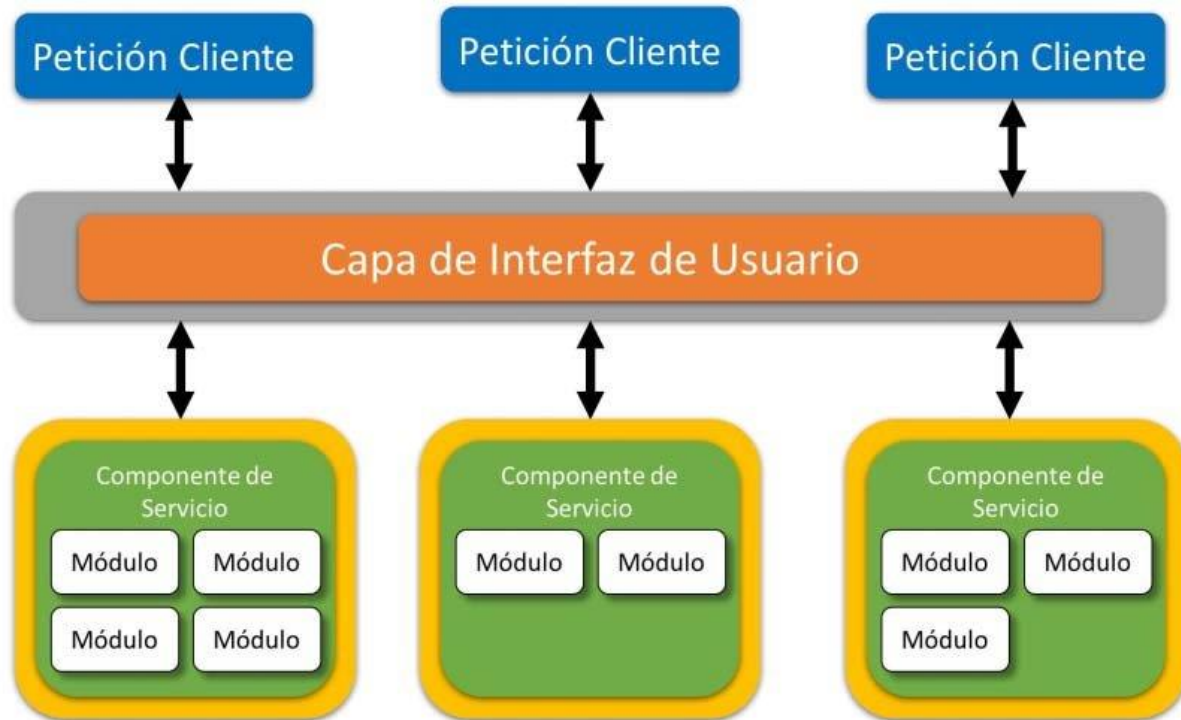





Orquestación vs Coreografía







- 
- Herramienta (base de datos, lenguaje, postman, etc)
 - Arquitectura global (micro servicio)
 - Servicios (endpoints resumen)
 - MERN
 - Diagrama de clases
 - SOLID
 - Unit test
 - Tareas en backlogs
 - Bichos conocidos
 - DEMO (explicar el caso de uso)
 - futuro
- 