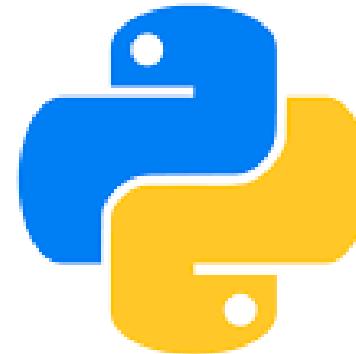


# AT Bootcamp Latam 02

---



**PYTHON**

**Paolo Sandoval Noel**

**Gmail:** [paolosandovaln@gmail.com](mailto:paolosandovaln@gmail.com)

**Email:** [paolo.sandoval@jalasoft.com](mailto:paolo.sandoval@jalasoft.com)

**Skype:** paolo.sandoval.jala

**Github:** jpsandovaln

## AGENDA

### Introducción

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

### Principios de código limpio

- Estilo de código / convenciones

### Paquetes

### Sintaxis Básica

- Variables, expresiones y sentencias
- Condicionales
- Funciones
- Iteraciones
- Cadenas
- Listas, Diccionarios y tuplas
- Flow control

### OOP

- Pilares de la programación Orientada a objetos

### • Clases, Objectos e Instancias

- Sobrecarga
- constructores
- Modificadores de acceso

### • 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**
  - Singleton
  - Factory
  - Builder
  - Composite
  - Decorator
  - Observer
  - Strategy

- **PROYECTO**

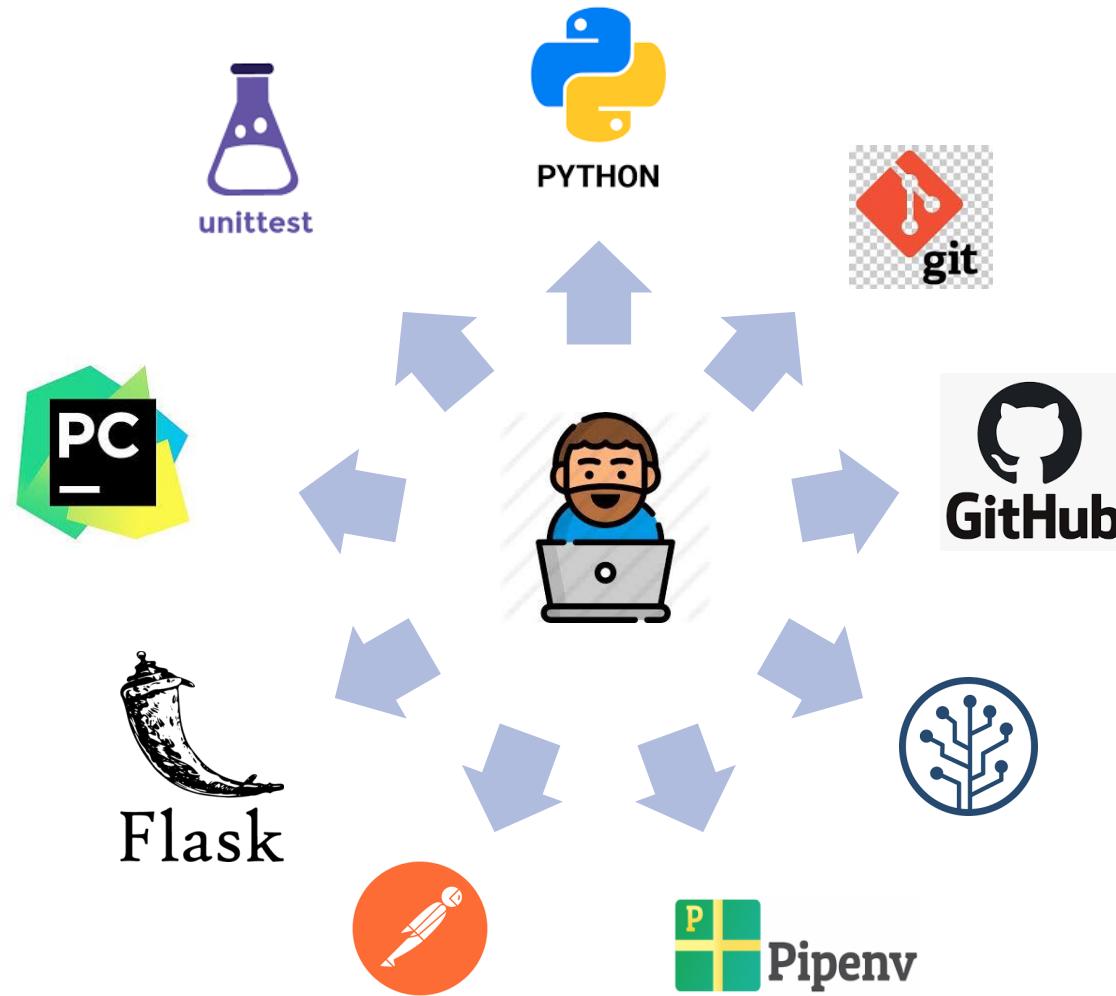
Implementar un proyecto donde los estudiantes puedan aplicar lo que aprendieron.

“Crear un Web service - Rest API para compilar y ejecutar código java, Python, javascript, etc”

“Crear un Web service - Rest API para buscar objetos dentro de un video (Machine Learning)”

## Tecnologías y Herramientas

- Python
- Git
- GitHub
- SourceTree
- ? Pipenv
- Postman
- Flask / Dianjo
- PyCharm
- Unit test
- Swagger



SourceTree <https://www.sourcetreeapp.com/>

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

- Instalar Git
- Instalar sourcetree
- Instalar Pycharm community
- 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%**

## Practica 1

# Realizar diagrama de clases para el siguiente caso de estudio

The screenshot shows a user interface for generating class diagrams. On the left, there is a sidebar with the following fields:

- Video Path: A text input field with a "Browse" button below it.
- Word: A text input field.
- Neural network Model: A dropdown menu showing "VGG16" as the selected option.
- Percentage: A dropdown menu showing "0" as the selected option.
- Search: A blue button at the bottom of the sidebar.

The main area contains a table with the following columns:

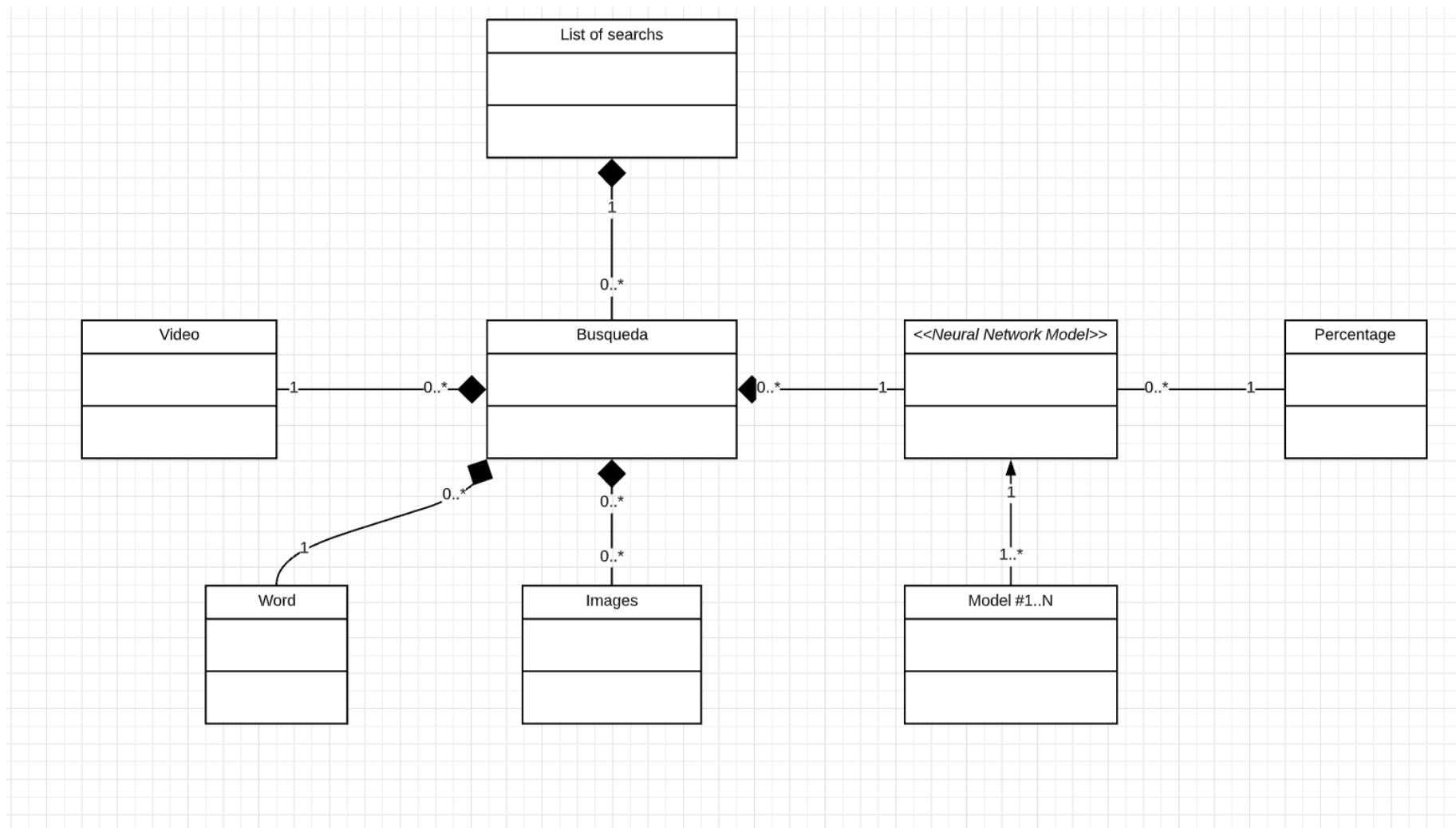
Algorithm	Word	Percentage	Second	Time

At the bottom right of the main area, there is a message: "Activar Windows" and "Ve a Configuración para activar Windows." Below the message is a "Show Image" button.

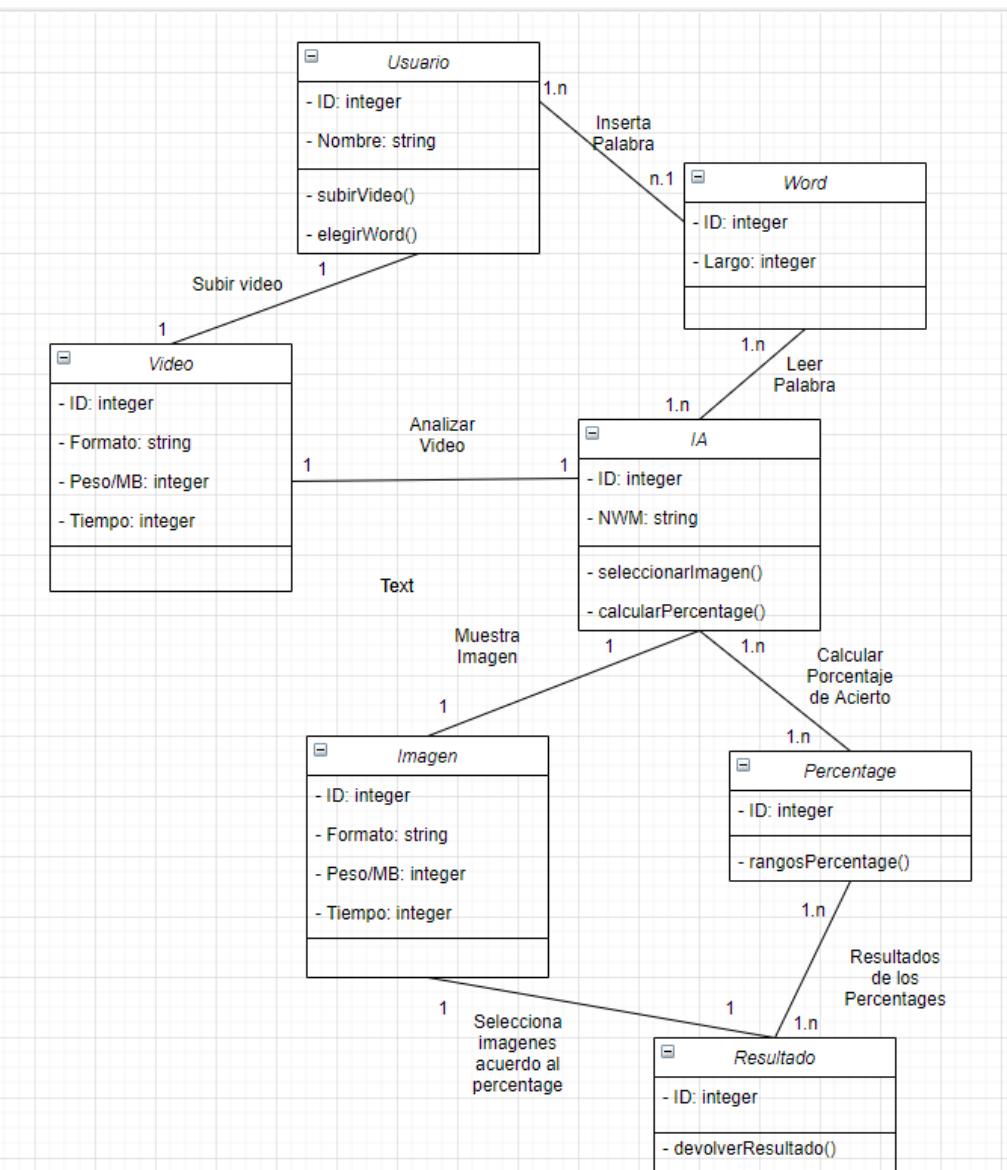
### Objetivo:

- Medir el grado de Abstracción del problema

## Practica 1



## Practica 1

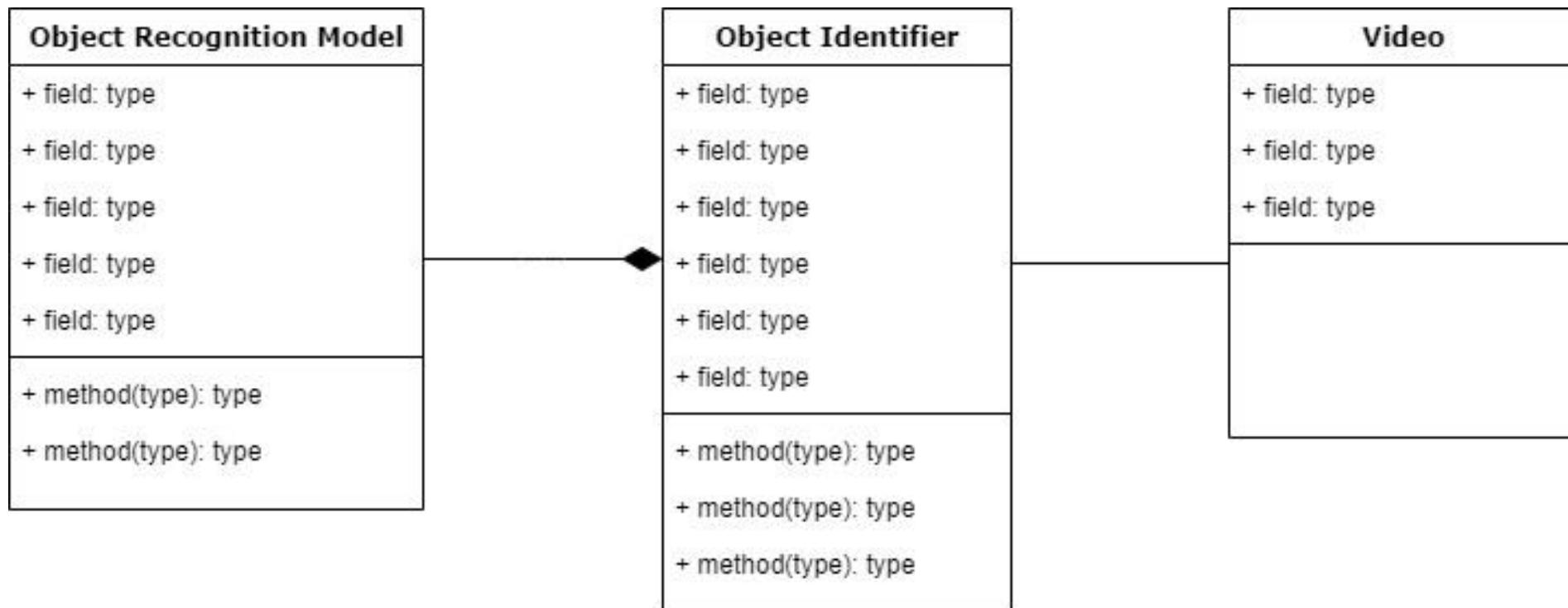


Practica 1

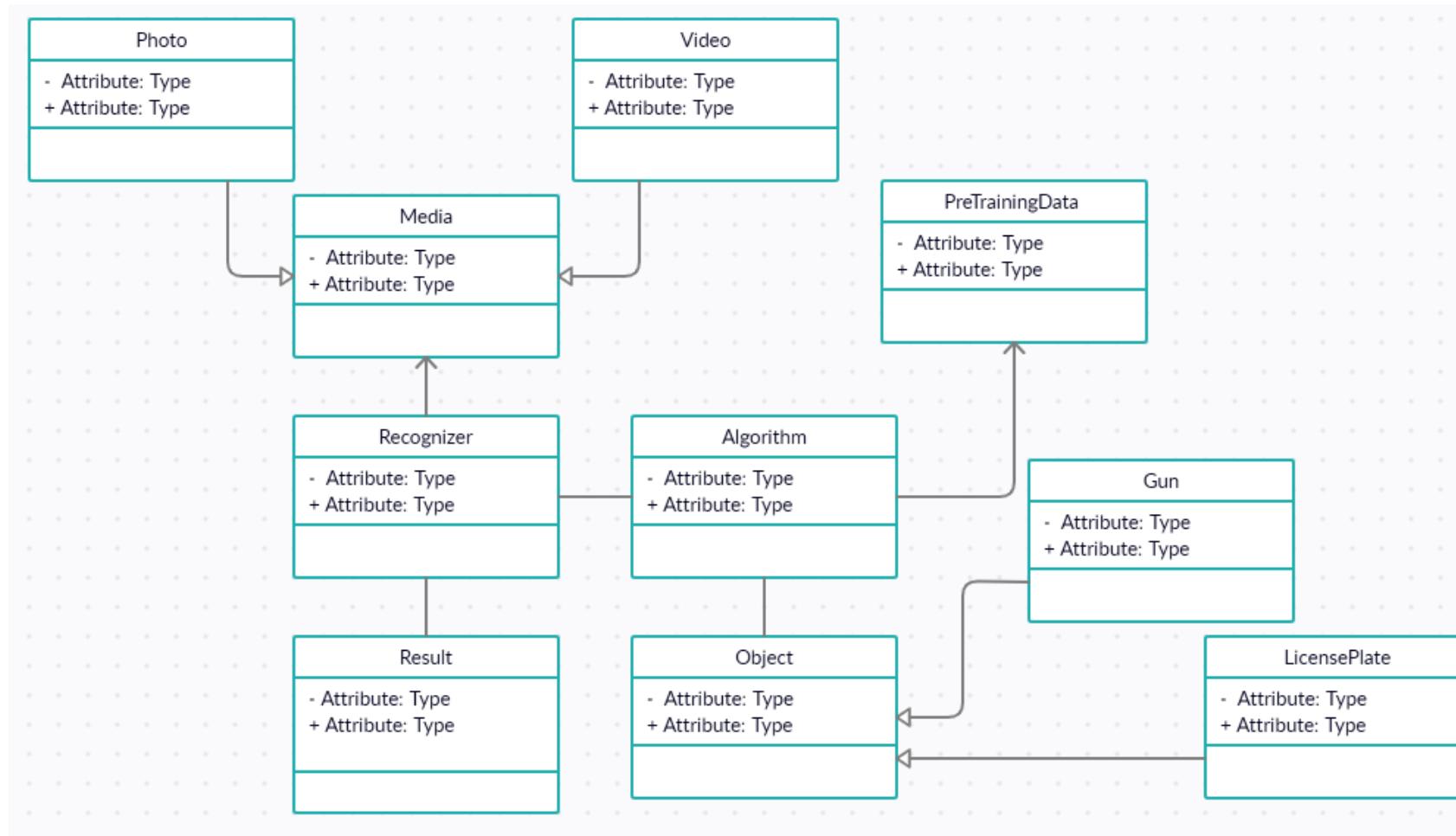


Alejandra

## Practica 1



## Practica 1



Christian

## Research 1

1. Investigar sobre Building Tools – Python / Definir la estructura de folder del proyecto con django.
2. Convertir video a imágenes
3. Django – crear endpoint (<http://localhost:8080/hello>) - hello world
4. Reconocimiento de imagen (ResNet50)
5. Reconocimiento de imagen (VGG16)

# Sistemas de control de versiones

---

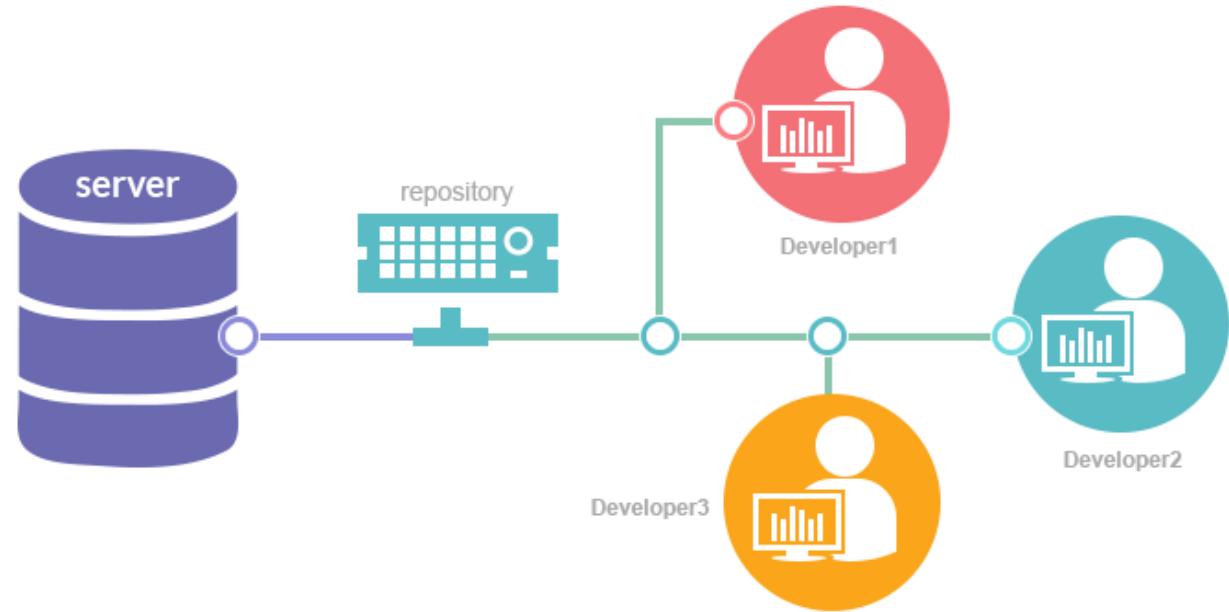


SourceTree <https://www.sourcetreeapp.com/>

Git: <https://git-scm.com/downloads>

## Sistema de control de versiones

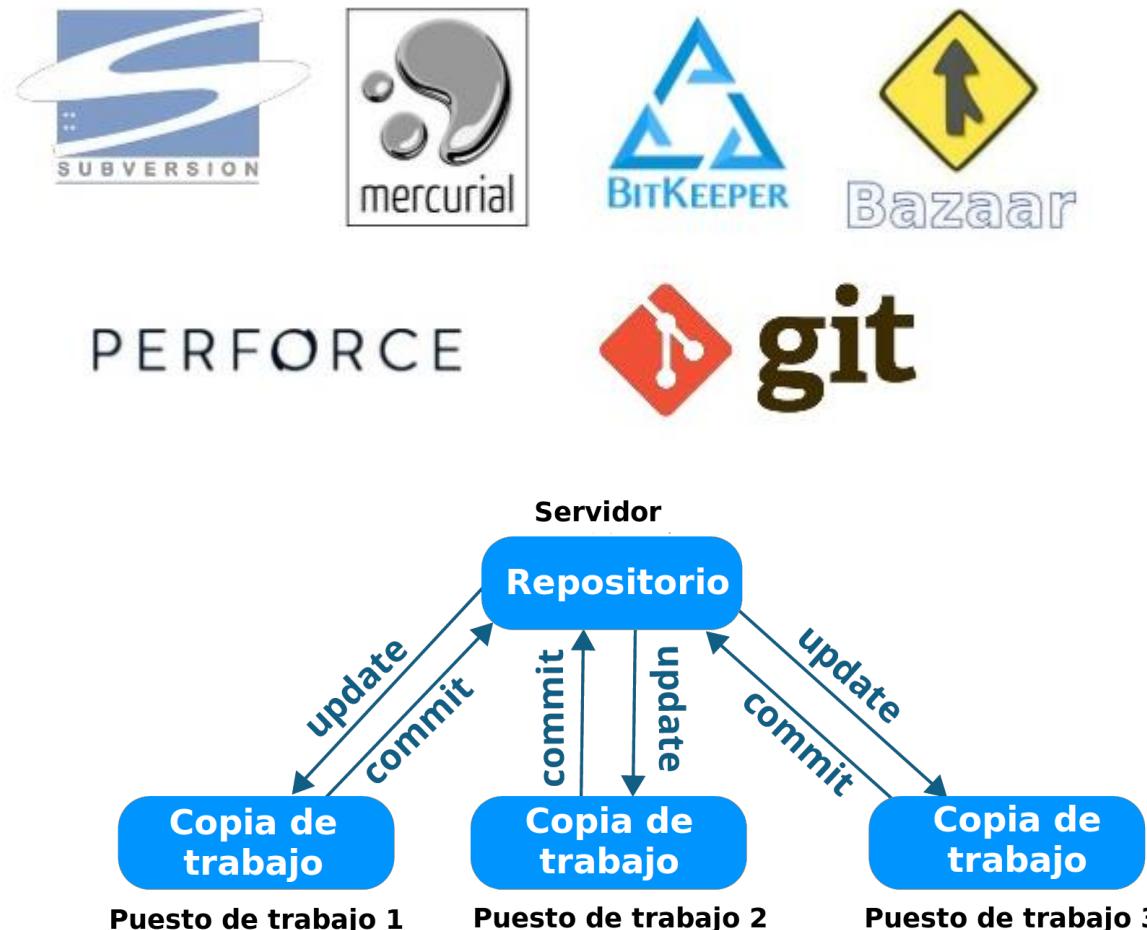
Los sistemas de control de versiones son programas que tienen como objetivo controlar los cambios en el desarrollo de cualquier tipo de software, permitiendo conocer el estado actual de un proyecto, los cambios que se le han realizado a cualquiera de sus piezas, las personas que intervinieron en ellos, etc.



## Sistema de control de versiones

Para facilitarnos la vida existen sistemas como Git, Subversion, CVS, etc. que sirven para controlar las versiones de un software y que deberían ser una obligatoriedad en cualquier desarrollo. Nos ayudan en muchos ámbitos fundamentales, como podrían ser:

- Comparar el código de un archivo, de modo que podamos ver las diferencias entre versiones.
- Restaurar versiones antiguas.
- Fusionar cambios entre distintas versiones
- Trabajar con distintas ramas de un proyecto, por ejemplo la de producción y desarrollo.



# Git

---



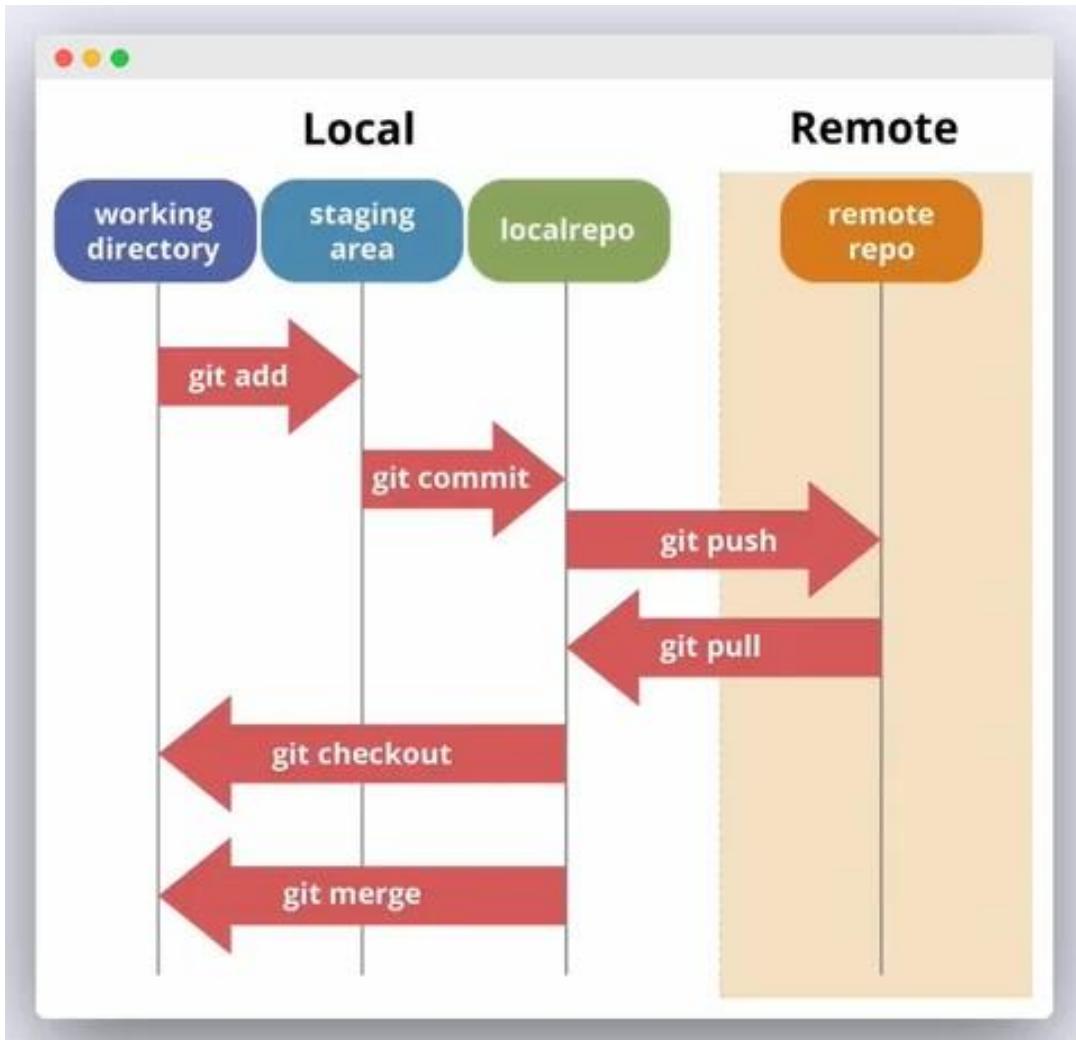
SourceTree <https://www.sourcetreeapp.com/>

Git: <https://git-scm.com/downloads>

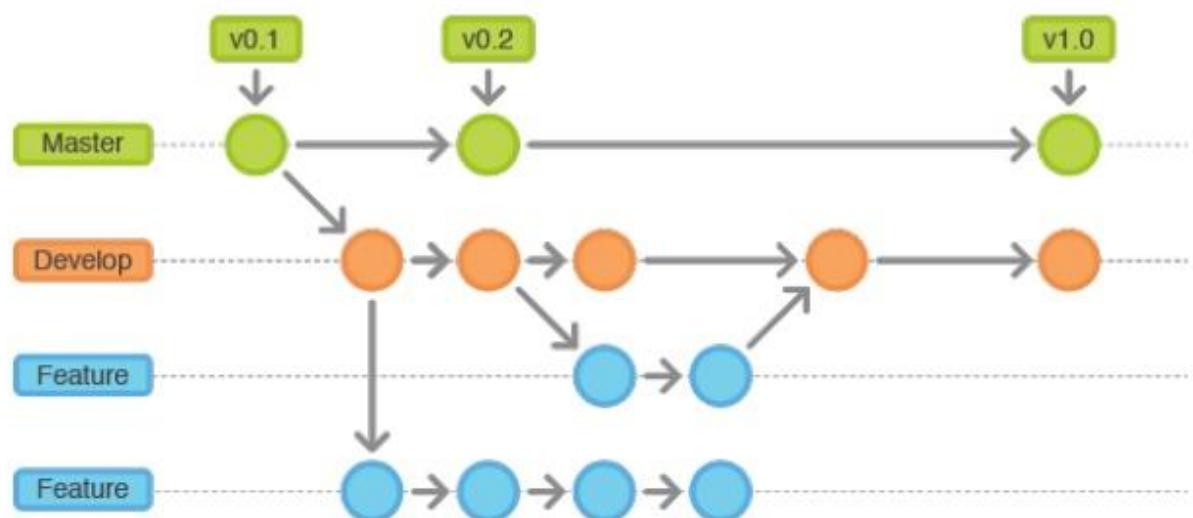
Git fue creado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente, es decir Git nos proporciona las herramientas para desarrollar un trabajo en equipo de manera inteligente y rápida y por trabajo nos referimos a algún software o página que implique código el cual necesitemos hacerlo con un grupo de personas.

Algunas de las características más importantes de Git son:

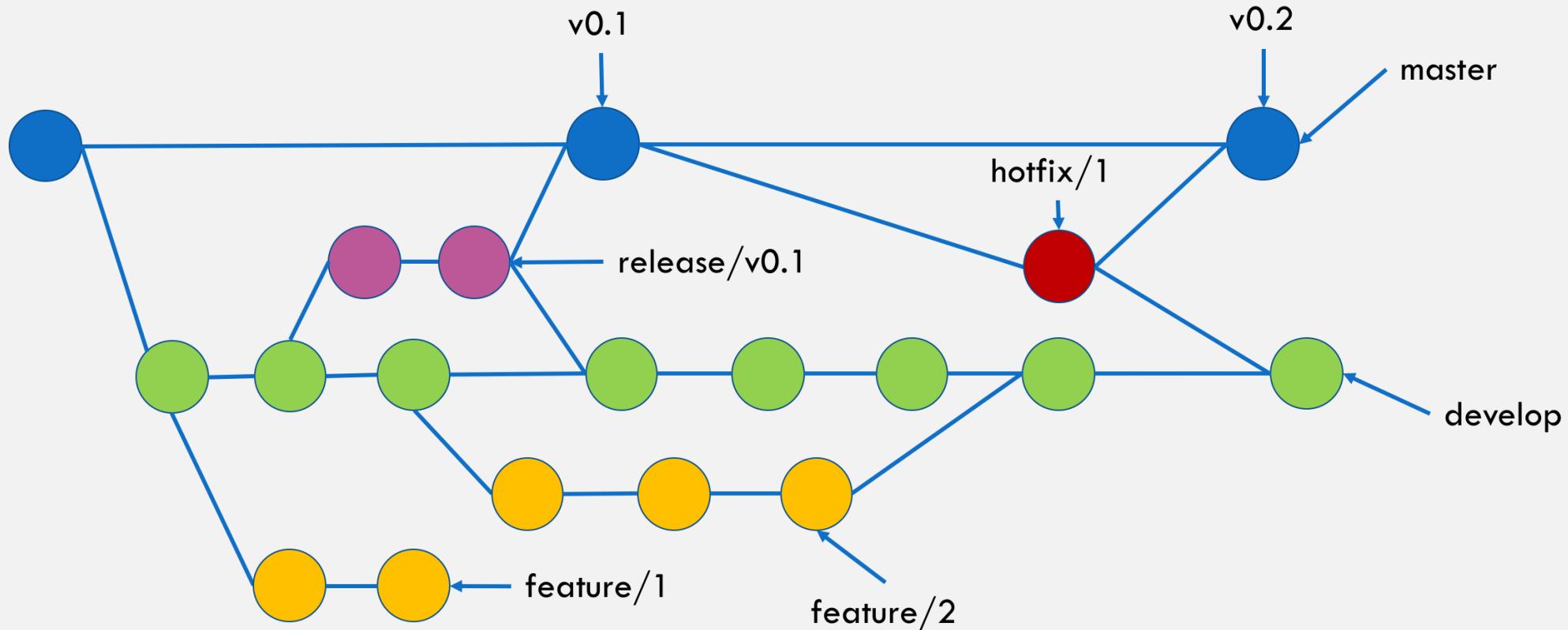
- Rapidez en la gestión de ramas, debido a que Git nos dice que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente.
- Gestión distribuida; Los cambios se importan como ramas adicionales y pueden ser fusionados de la misma manera como se hace en la rama local.
- Gestión eficiente de proyectos grandes.
- Realmacenamiento periódico en paquetes.



## Version Control: Development Branches

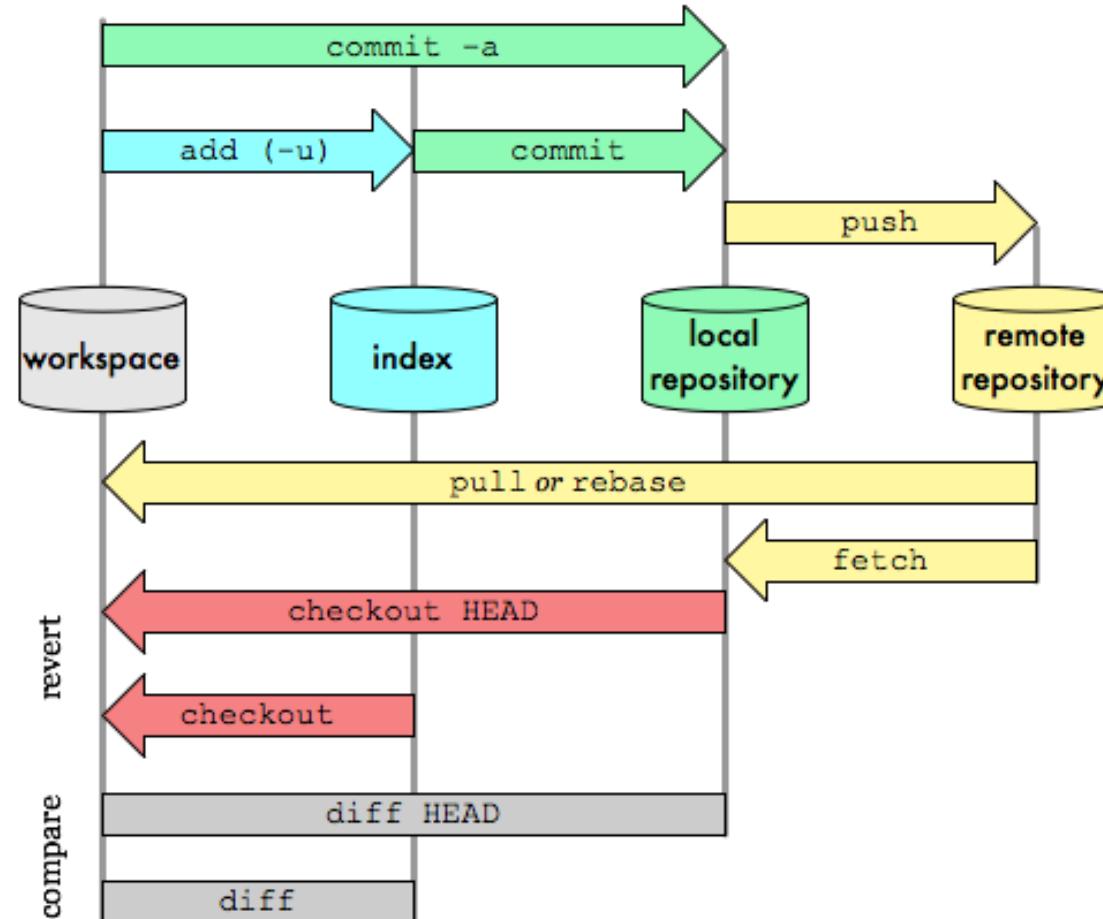


Git



## Git Data Transport Commands

<http://osteelle.com>



## Configuración Básica

|

Configurar Nombre que salen en los commits (autor)

```
git config --global user.name "AT-03"
```

Configurar Email (auto)

```
git config --global user.email at-03@gmail.com
```

Date	Author	
7 sep. 2020 20:46	maryangela23 <angelamdp16@gmail.com>	
7 sep. 2020 20:34	maryangela23 <angelamdp16@gmail.com>	
7 sep. 2020 20:10	julia-jpg <julia.escalante@fundacion-jala.org>	
7 sep. 2020 20:08	julia-jpg <julia.escalante@fundacion-jala.org>	
7 sep. 2020 18:35	laura helen montano toro <hlaura.toro@gmail.com>	
7 sep. 2020 17:11	laura helen montano toro <hlaura.toro@gmail.com>	
7 sep. 2020 11:35	Samuel Loza <starsaminf@gmail.com>	
7 sep. 2020 11:13	laura helen montano toro <hlaura.toro@gmail.com>	
7 sep. 2020 10:49	maryangela23 <angelamdp16@gmail.com>	
7 sep. 2020 1:27	maryangela23 <angelamdp16@gmail.com>	
7 sep. 2020 1:04	maryangela23 <angelamdp16@gmail.com>	
6 sep. 2020 23:28	Mirko Romay Ramos <mirko.romay@fundacion-jala.org>	
6 sep. 2020 22:47	Samuel Loza <starsaminf@gmail.com>	

## Iniciando repositorio

Iniciamos GIT en la carpeta donde esta el proyecto

`git init`

Clonamos el repositorio de github o bitbucket

`git clone <url>`

Añadimos todos los archivos para el commit

`git add .`

Hacemos el primer commit

`git commit -m "Added initial changes"`

subimos al repositorio

`git push origin master`

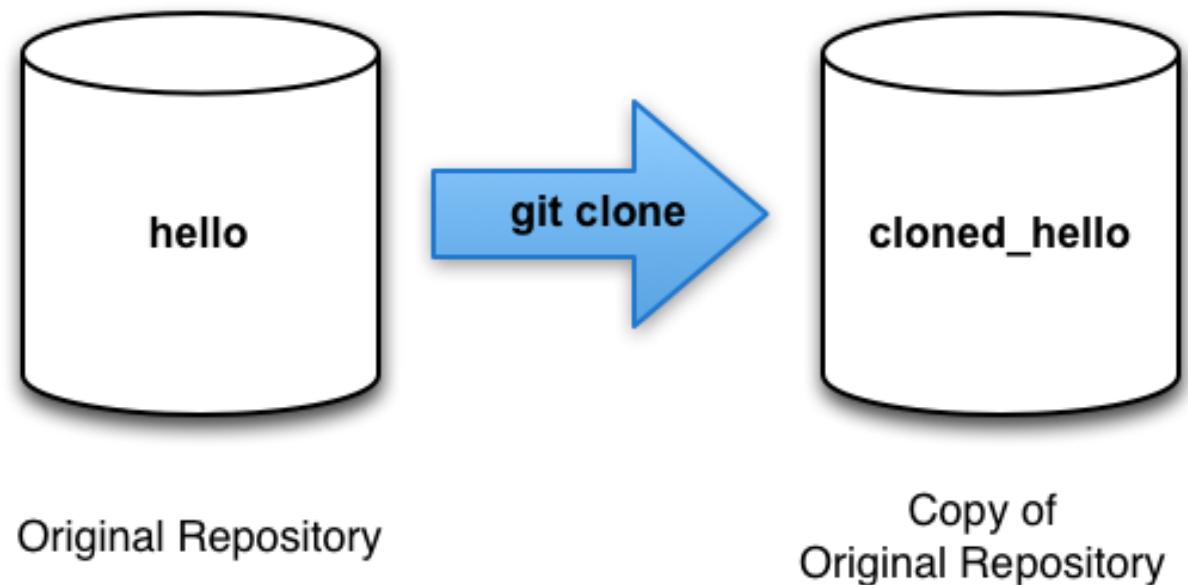
## GIT CLONE

Clonamos el repositorio de github o bitbucket

`git clone <url>`

Clonamos el repositorio de github o bitbucket

`git clone <url> git-demo`



## GIT ADD

Añadimos todos los archivos para el commit

`git add .`

Añadimos el archivo para el commit

`git add <archivo>`

Añadimos todos los archivos para el commit omitiendo los nuevos

`git add --all`

Añadimos todos los archivos con la extensión especificada

`git add *.txt`

Añadimos todos los archivos dentro de un directorio y de una extensión específica

`git add docs/*.txt`

Añadimos todos los archivos dentro de un directorios

`git add docs/`

## GIT COMMIT

Cargar en el HEAD los cambios realizados

```
git commit -m "Texto que identifique por que se hizo el commit"
```

Agregar y Cargar en el HEAD los cambios realizados

```
git commit -a -m "Texto que identifique por que se hizo el commit"
```

De haber conflictos los muestra

```
git commit -a
```

Agregar al ultimo commit, este no se muestra como un nuevo commit en los logs. Se puede especificar un nuevo mensaje

```
git commit --amend -m "Texto que identifique por que se hizo el commit"
```

Git

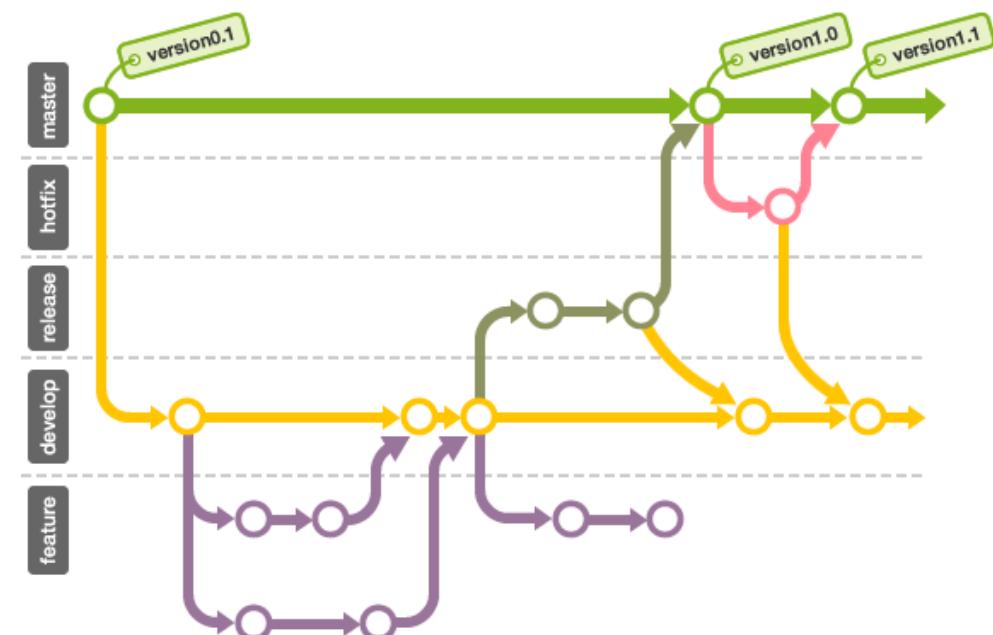
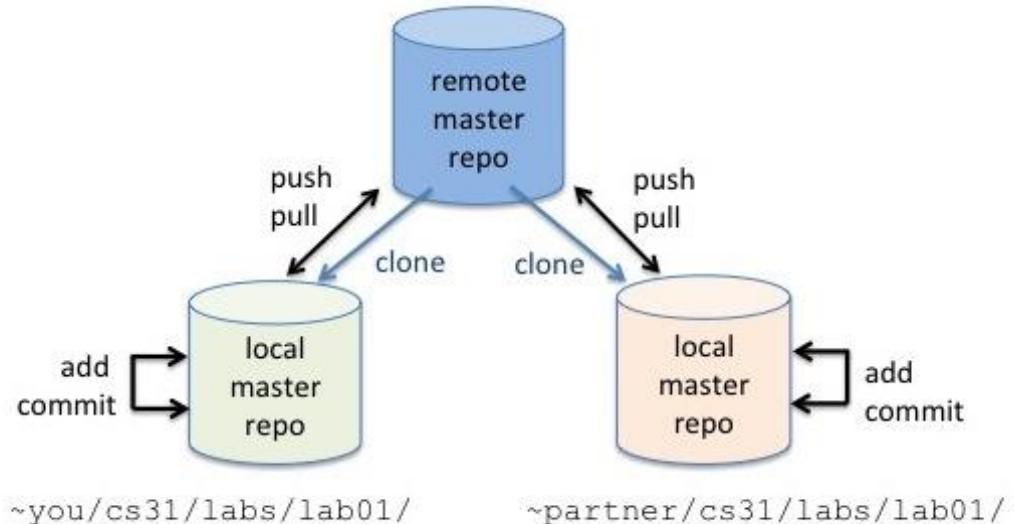
## GIT PUSH

Subimos al repositorio

`git push <origin> <branch>`

Subimos un tag

`git push --tags`



## GIT LOG

Muestra los logs de los commits

git log

Muestras los cambios en los commits

git log --oneline --stat

Muestra graficos de los commits

git log --oneline --graph

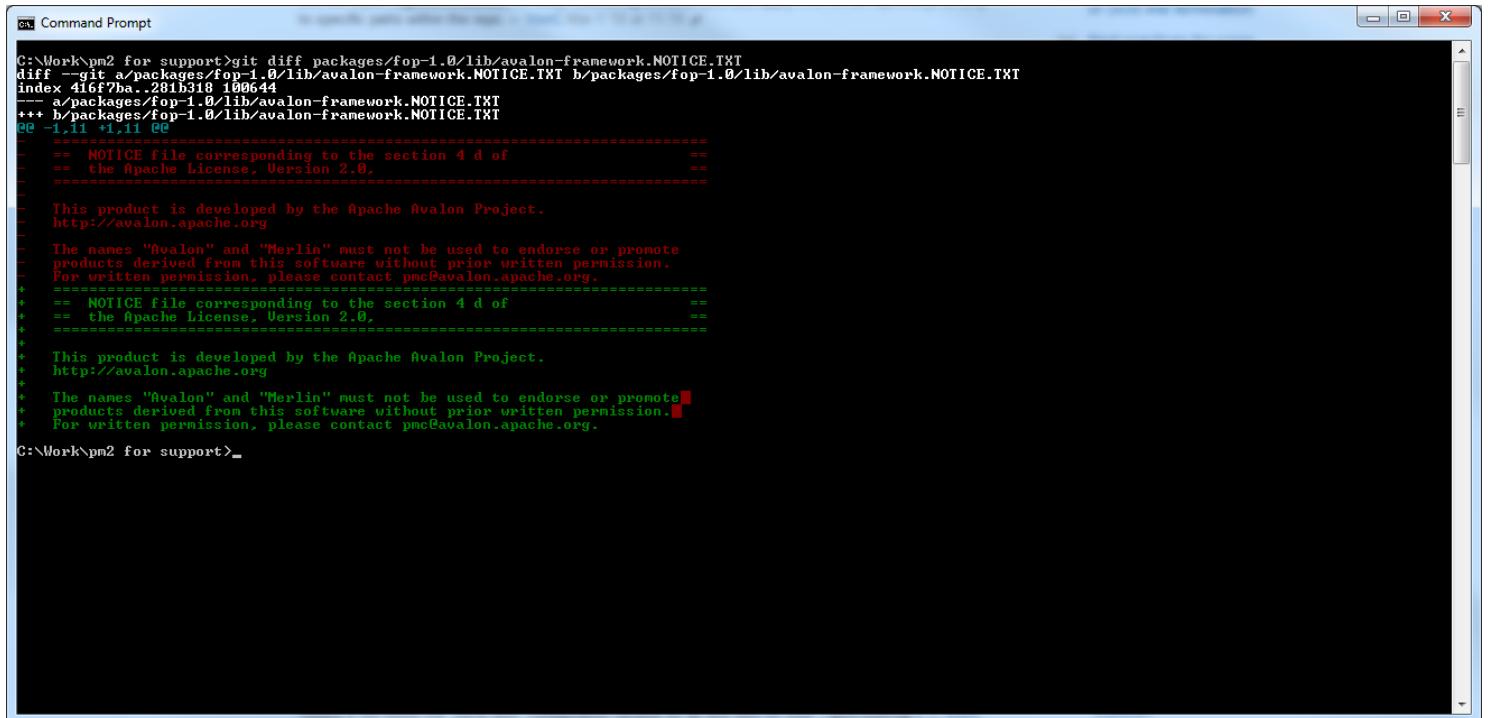
```
$ TZ=PST8PDT git log-compact --decorate --graph --no-merges -n 13 v2.6.1
== 2015-09-28 ==
* 22f698cb 19:19 jch (tag: v2.6.1) Git 2.6.1
  .....
* 24358560 15:34 jch (tag: v2.5.4) Git 2.5.4
  .....
* a2558fb8 15:30 jch (tag: v2.4.10) Git 2.4.10
  .....
* 18b58f70 15:26 jch (tag: v2.3.10, maint-2.3) Git 2.3.10
  .....
* 83c4d380 14:58 jk merge-file: enforce MAX_XDIFF_SIZE on incoming files
* dcd1742e 14:57 jk xdiff: reject files larger than ~1GB
* 3efb9880 14:57 jk react to errors in xdi_diff
== 2015-09-25 ==
* b2581164 15:32 bb http: limit redirection depth
* f4113cac 15:30 bb http: limit redirection to protocol-whitelist
* 5088d3b3 15:28 jk transport: refactor protocol whitelist code
== 2015-09-23 ==
* 33cfccbb 11:35 jk submodule: allow only certain protocols for submodule fetches
* a5adaced 11:35 jk transport: add a protocol-whitelist environment variable
/
== 2015-09-28 ==
* be08dee9 13:18 jch (tag: v2.6.0) Git 2.6
```

## GIT DIFF

Muestra los cambios realizados a un archivo

git diff

git diff --staged



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "git diff packages/fop-1.0/lib/avalon-framework.NOTICE.TXT". The output displays a diff between two versions of the file. The top section shows the commit details: index 416f7ba..201b319 100644, followed by the standard triple-dash separator. Below this, the changes are listed with color-coded markers: red for deleted text and green for added text. The text itself is the Apache License, Version 2.0, which remains largely unchanged except for the addition of a copyright notice at the bottom.

```
C:\Work\pm2 for support>git diff packages/fop-1.0/lib/avalon-framework.NOTICE.TXT
diff --git a/packages/fop-1.0/lib/avalon-framework.NOTICE.TXT b/packages/fop-1.0/lib/avalon-framework.NOTICE.TXT
index 416f7ba..201b319 100644
--- a/packages/fop-1.0/lib/avalon-framework.NOTICE.TXT
+++ b/packages/fop-1.0/lib/avalon-framework.NOTICE.TXT
@@ -1,11 +1,11 @@
-- NOTICE file corresponding to the section 4 d of
-- the Apache License, Version 2.0.
--
This product is developed by the Apache Avalon Project.
http://avalon.apache.org

The names "Avalon" and "Merlin" must not be used to endorse or promote
products derived from this software without prior written permission.
For written permission, please contact pmc@avalon.apache.org.
=====
-- NOTICE file corresponding to the section 4 d of
-- the Apache License, Version 2.0.
--

This product is developed by the Apache Avalon Project.
http://avalon.apache.org

The names "Avalon" and "Merlin" must not be used to endorse or promote
products derived from this software without prior written permission.
For written permission, please contact pmc@avalon.apache.org.

C:\Work\pm2 for support>
```

## GIT HEAD

Saca un archivo del commit

`git reset HEAD <archivo>`

Devuelve el ultimo commit que se hizo y pone los cambios en staging

`git reset --soft HEAD^`

Devuelve el ultimo commit y todos los cambios

`git reset --hard HEAD^`

Devuelve los 2 ultimo commit y todos los cambios

`git reset --hard HEAD^^`

Rollback merge/commit

`git log git reset --hard <commit_sha>`

## GIT BRANCH

Crea un branch

```
git branch <nameBranch>
```

Lista los branches

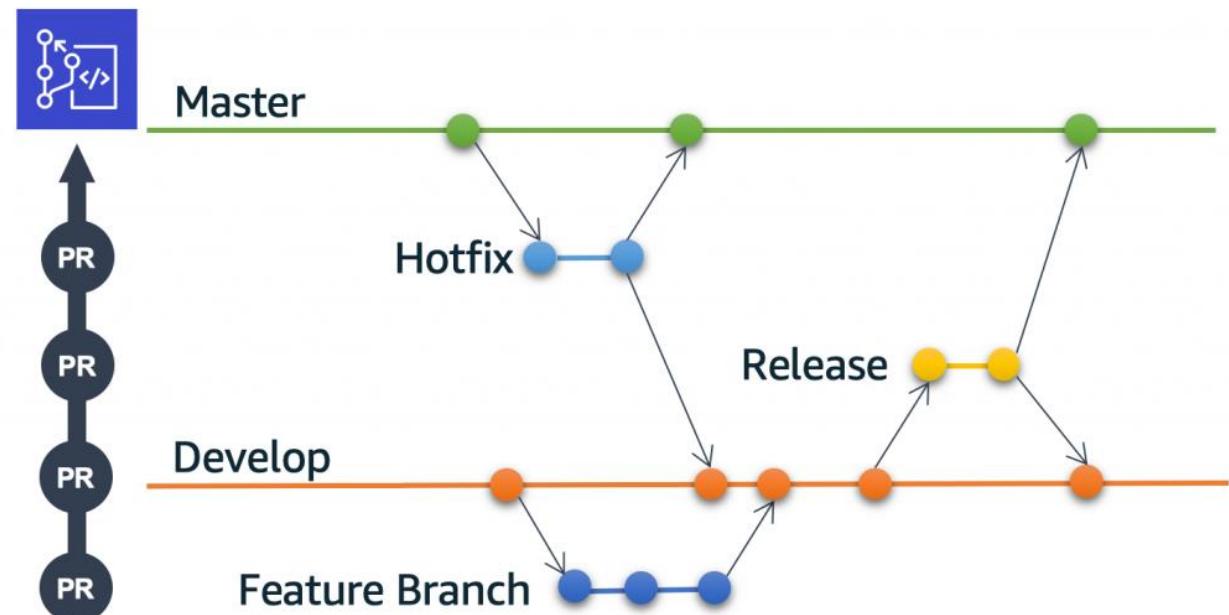
```
git branch
```

Comando -d elimina el branch y lo une al master

```
git branch -d <nameBranch>
```

Elimina sin preguntar

```
git branch -D <nameBranch>
```



## OTROS COMANDOS

Lista un estado actual del repositorio con lista de archivos modificados o agregados

`git status`

Quita del HEAD un archivo y le pone el estado de no trabajado

`git checkout -- <file>`

Busca los cambios nuevos y actualiza el repositorio

`git pull origin <nameBranch>`

Cambiar de branch

`git checkout <nameBranch/tagname>`

Une el branch actual con el especificado

`git merge <nameBranch>`

Verifica cambios en el repositorio online con el local

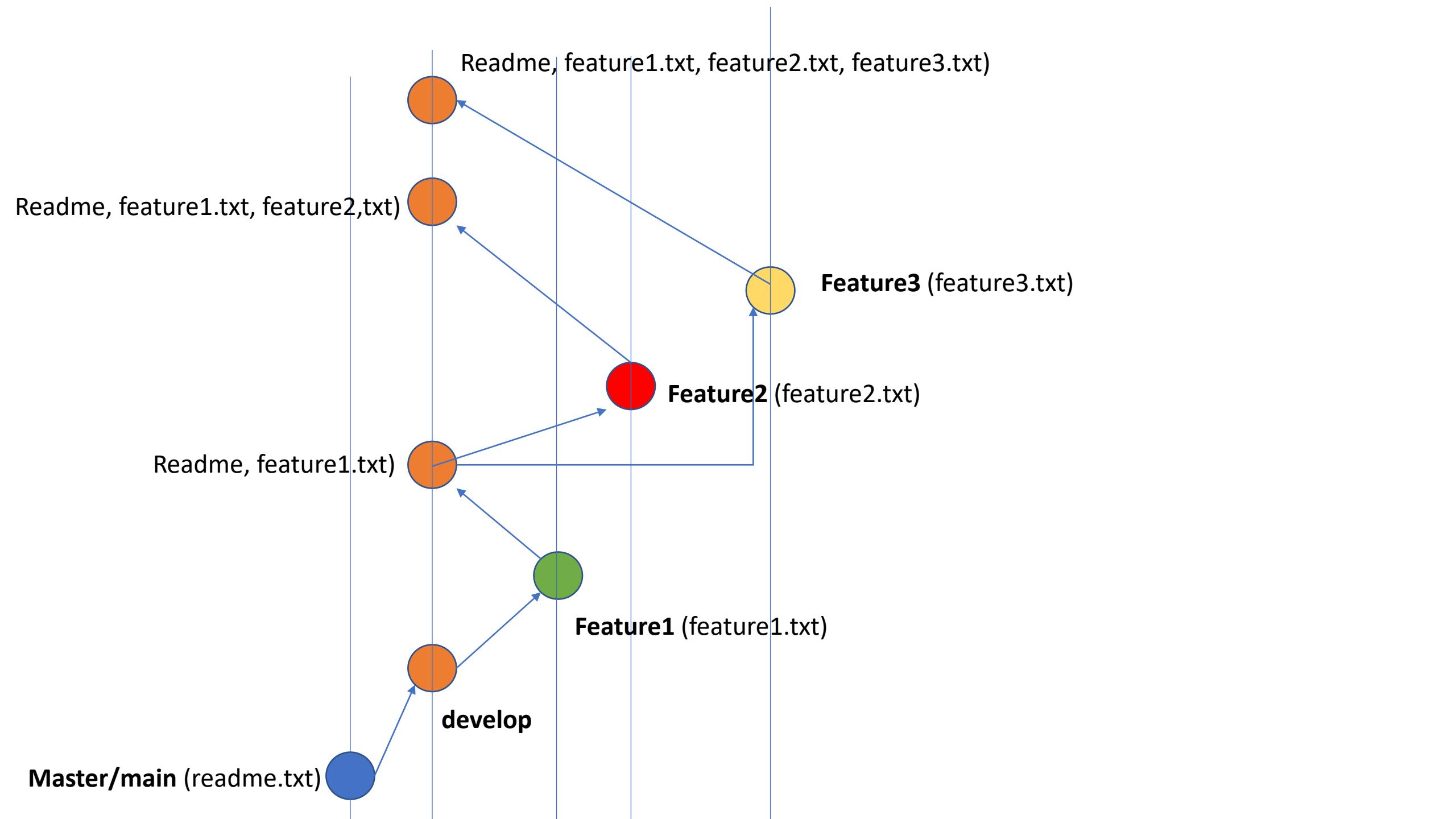
`git fetch`

Borrar un archivo del repositorio

`git rm <archivo>`

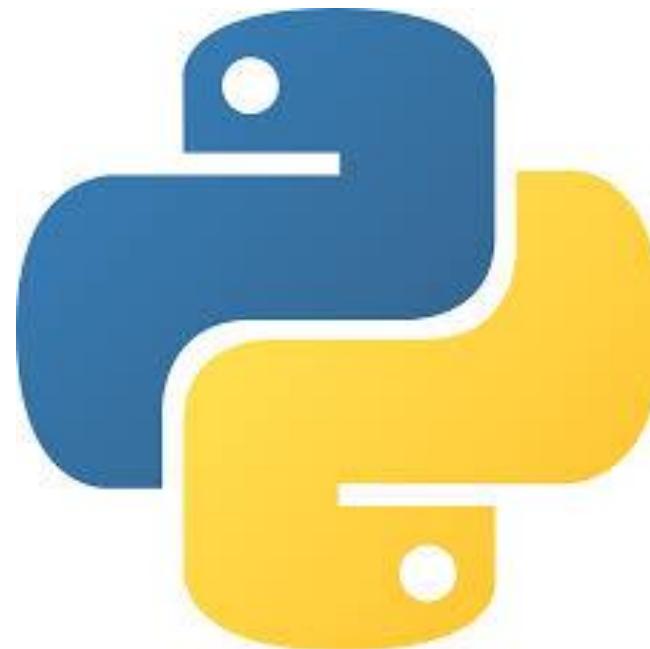
## Practica 2

- Crear / iniciar un repositorio local llamado <Practice2>
  - Agregar un archivo readme a la rama master/main
  - Crear una rama develop
  - Desde la rama develop crear una nueva rama llama <feature1>
  - y crear un archivo <feature1.txt>
  - Mergear la rama <feature1> a develop
  - Desde la rama develop crear 2 nuevas ramas: <feature2> <feature3>
  - Moverse a la rama feature2 y crear un archivo llamado <feature2.tx>
  - Mergear la rama <feature2> a develop
  - Moverse a la rama feature3 y crear un archivo llamado <feature3.tx>
  - Mergear la rama <feature3> a develop
- 
- **Expereado:** La rama develop deberá contener: Readme.txt, feature1.txt, feature2.txt, feature3.txt



# Python

---



Python: <https://www.python.org/downloads/>



- **Python** es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiplataforma, ya que soporta orientado objetos, programación imperativa y, en menor medida, programación funcional.
- Lenguaje de programación de propósito general, orientado a objetos, que también puede utilizarse para el desarrollo web.
- Python es un lenguaje del tipo interpretado, multiparadigma: Soporta orientación a objetos (oop), Programación imperativa y funcional, Es de tipado dinámico, multiplataforma y multipropósito.

# Convención de código

---



**PEP 8**  
**Coding style in Python**

*Coding style in Python*



Meccanismo  
Complesso

- <https://www.python.org/dev/peps/pep-0008/>
- <https://www.python.org/dev/peps/pep-0257/>

## 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,

## Diseño del código

- Usa 4 (cuatro) espacios por indentación.
- ¿Tabulaciones o espacios? Nunca mezcles tabulaciones y espacios. El método de indentación más popular en Python es con espacios.
- Máxima longitud de las líneas Limita todas las líneas a un máximo de 79 caracteres. (definido por el equipo)
- Líneas en blanco Separa funciones de alto nivel y definiciones de clase con dos líneas en blanco.

```
# Alineado con el paréntesis que abre la función
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

```
# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

## Importaciones

- Las importaciones deben estar en líneas separadas.

Sí: `import os  
import sys`

No: `import sys, os`

## Espacios en blanco en Expresiones y Sentencias

Evita usar espacios en blanco extraños en las siguientes situaciones:

- Inmediatamente dentro de paréntesis, corchetes o llaves
- Inmediatamente antes de una coma, un punto y coma o dos puntos
- Inmediatamente antes del paréntesis que comienza la lista de argumentos en la llamada a una función

Sí: `spam(ham[1], {eggs: 2})`  
No: `spam( ham[ 1 ], { eggs: 2 } )`

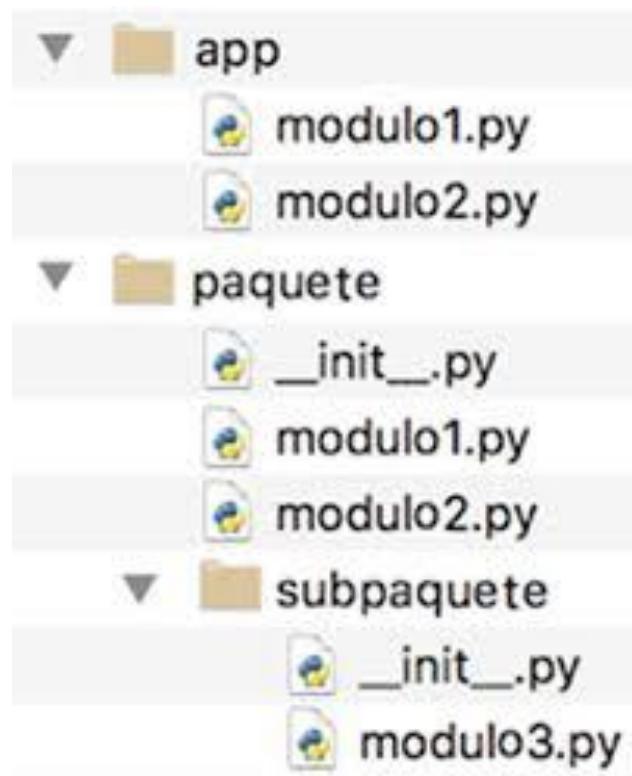
Sí: `if x == 4: print x, y; x, y = y, x`  
No: `if x == 4 : print x , y ; x , y = y , x`

Sí: `spam(1)`  
No: `spam (1)`

Sí: `dict['key'] = list[index]`  
No: `dict ['key'] = list [index]`

## Nombres de paquetes y módulos

- Los módulos deben tener un nombre corto y en minúscula. Guiones bajos pueden utilizarse si mejora la legibilidad.
- Los paquetes en Python también deberían tener un nombre corto y en minúscula, aunque el uso de guiones bajos es desalentado (poco recomendado).

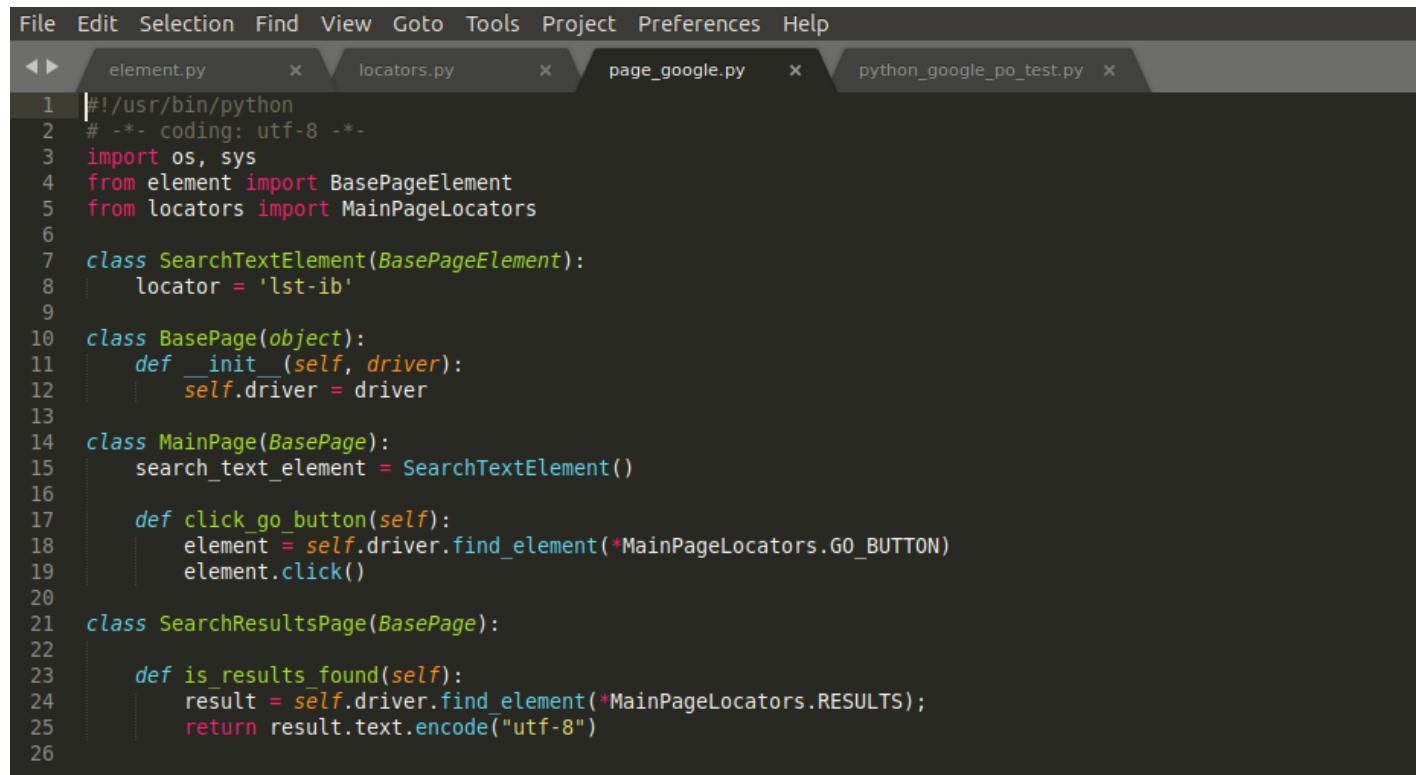


## Nombres de clases

- Casi sin excepción, los nombres de clases deben utilizar la convención “CapWords” (palabras que comienzan con mayúsculas). Clases para uso interno tienen un guión bajo como prefijo.

## Nombres de excepciones

- Debido a que las excepciones deben ser clases, se aplica la convención anterior.



```
File Edit Selection Find View Goto Tools Project Preferences Help
element.py      locators.py      page_google.py      python_google_po_test.py
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3 import os, sys
4 from element import BasePageElement
5 from locators import MainPageLocators
6
7 class SearchTextElement(BasePageElement):
8     locator = 'lst-ib'
9
10 class BasePage(object):
11     def __init__(self, driver):
12         self.driver = driver
13
14 class MainPage(BasePage):
15     search_text_element = SearchTextElement()
16
17     def click_go_button(self):
18         element = self.driver.find_element(*MainPageLocators.GO_BUTTON)
19         element.click()
20
21 class SearchResultsPage(BasePage):
22
23     def is_results_found(self):
24         result = self.driver.find_element(*MainPageLocators.RESULTS);
25         return result.text.encode("utf-8")
26
```

## Nombres de funciones

- Las funciones deben ser en minúscula, con las palabras separadas por un guión bajo, aplicándose éstos tanto como sea necesario para mejorar la legibilidad.

```
samepairs.py
1  from collections import defaultdict
2
3
4  def get_distance(l):
5      assert len(l) > 0
6      return max(l) - min(l) # takes O(n) time
7
8
9  def solution(A):
10     D = defaultdict(list)
11     for i, item in enumerate(A):
12         D[item].append(i)
13     D = {k: get_distance(v) for k, v in D.items() if len(v) > 1}
14     # dict set item takes O(1) in average
15     return max(D.values()) if len(D) > 0 else 0
16
17 print solution([4, 6, 2, 2, 6, 6, 1]) # should return 4
18 print solution([1, 2, 3, 4, 5]) # should return 0
19 print solution([1, 2, 2, 4, 5]) # should return 1
20
```

## Argumentos de funciones y métodos

- Siempre usa `self` para el primer argumento de los métodos de instancia.

```
# Recommended
class Foo:
    def __private_method(self):
        return 'private'

    def public_method(self):
        return 'public'
```

## Constantes

- Las constantes son generalmente definidas a nivel módulo, escritas con todas las letras en mayúscula y con guiones bajos separando palabras. Por ejemplo, MAX\_OVERFLOW y TOTAL.

Create a **constant.py**:

```
PI = 3.14
GRAVITY = 9.8
```

Create a **main.py**:

```
import constant

print(constant.PI)
print(constant.GRAVITY)
```

## Copyright

```
#  
  
# @video_converter.py Copyright (c) 2021 Jalasoft.  
  
# 2643 Av Melchor Perez de Olguin, Colquiri Sud, Cochabamba, Bolivia.  
  
# <add dirección de jala la paz>  
  
# 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.  
  
#
```

## Convención de código

### Ejemplo:

```
1  #  
2  # @video_converter.py Copyright (c) 2020 Jalasoft.  
3  # 2643 Av Melchor Perez de Olguin, Colquiri Sud, Cochabamba, Bolivia.  
4  # <add dirección de jala la paz>  
5  # All rights reserved.  
6  #  
7  # This software is the confidential and proprietary information of  
8  # Jalasoft, ("Confidential Information"). You shall not  
9  # disclose such Confidential Information and shall use it only in  
10 # accordance with the terms of the license agreement you entered into  
11 # with Jalasoft.  
12 #  
13  
14 from src.main.com.fundation.ChuchusmoteConverter.model.converter.\  
    converter_types.audio_converter import ConverterCriteriaAudio  
16  
17  
18 # This class inherits audio converter criteria and adds specific fields for video  
19 class ConverterCriteriaVideo(ConverterCriteriaAudio):  
20     def __init__(self):  
21         super().__init__()  
22         self.__video_codec = ''  
23         self.__frame_rate = ''  
24         self.__width = ''  
25         self.__height = ''  
26  
27     # This method sets video_codec field  
28     def set_video_codec(self, value):  
29         self.__video_codec = value  
30  
31     # This method gets video_codec field  
32     def get_video_codec(self):  
33         return self.__video_codec  
34  
35     # This method sets frame_rate field  
36     def set_frame_rate(self, value):  
37         self.__frame_rate = value  
38  
39     # This method gets frame_rate field  
40     def get_frame_rate(self):  
41         return self.__frame_rate  
42  
43     # This method sets width field  
44     def set_width(self, value):  
45         self.__width = value  
46
```

# Paquetes

---



## Paquetes

Un **Paquete** en python es un contenedor de módulos/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.

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

Los **paquetes** son la forma mediante la cual python 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.

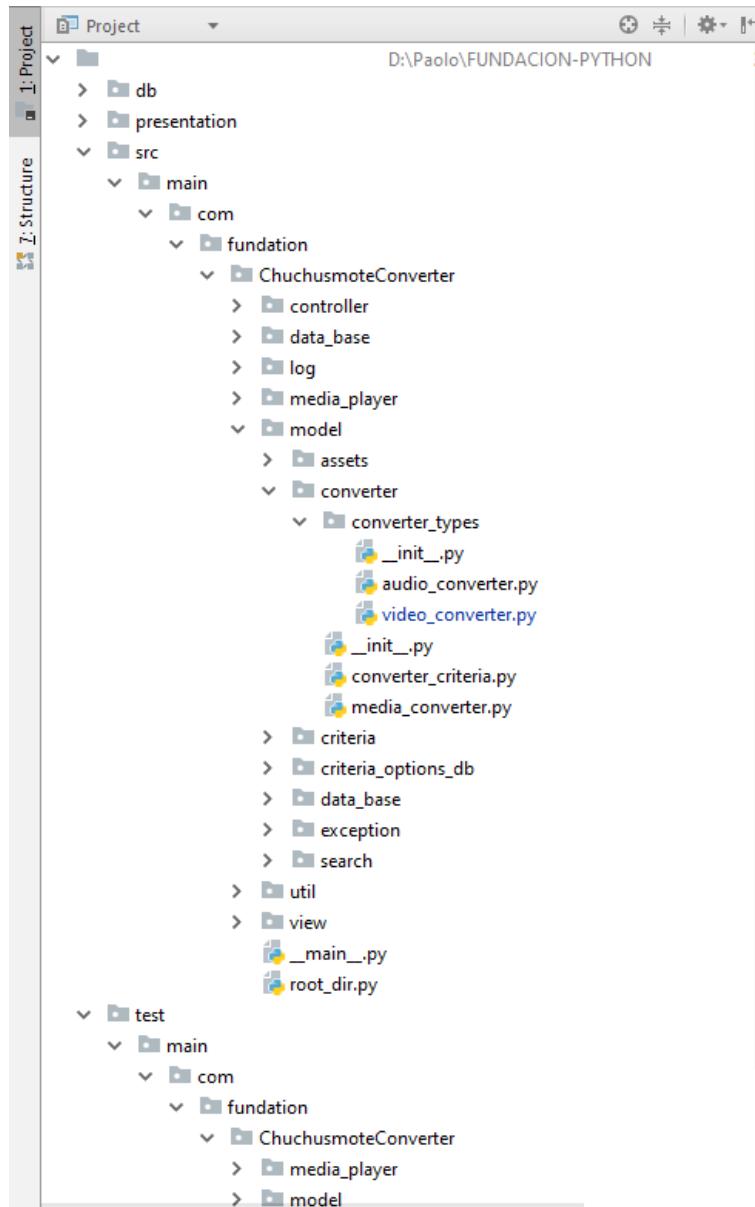
Sin paquetes



Con paquetes



## Paquetes



# Sintaxis

---

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



## Variables

- Las variables son contenedores para almacenar valores de datos. A diferencia de otros lenguajes de programación, Python no tiene ningún comando para declarar una variable.
- Las variables se definen de forma dinámica, lo que significa que no se tiene que especificar cuál es su tipo de antemano y puede tomar distintos valores en otro momento, incluso de un tipo diferente al que tenía previamente. Se usa el símbolo = para asignar valores.

```
x = 5  
y = "John"  
print(x)  
print(y)
```

The screenshot shows a Python code editor with the file Python5.2.py open. The code is as follows:

```
# Declare a variable and initialize it  
f = 101 # 1  
print(f)  
  
# Global vs. local variables in functions  
def someFunction():  
    # global f  
    f = 'I am learning Python' # 2  
    print(f)  
  
someFunction()  
print(f) # 3
```

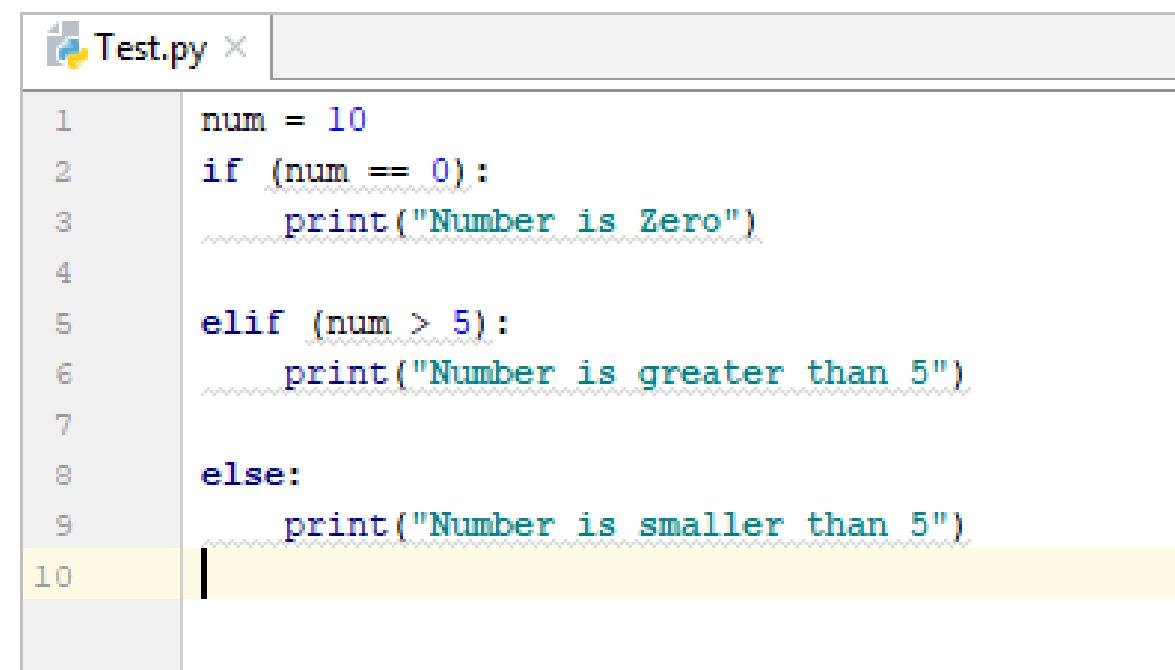
Annotations with red numbers 1, 2, and 3 point to specific lines of code: 1 points to the global variable assignment, 2 points to the local variable assignment inside the function, and 3 points to the print statement that outputs the value of the local variable. A callout box on the right states: "f is a local variable declared inside the function."

```
X = 1  
X = "texto" # Esto es posible porque los tipos son asignados dinámicamente
```

## Variables

Tipo	Clase	Notas	Ejemplo
<code>str</code>	Cadena	Inmutable	<code>'Cadena'</code>
<code>unicode</code>	Cadena	Versión Unicode de <code>str</code>	<code>u'Cadena'</code>
<code>list</code>	Secuencia	Mutable, puede contener objetos de diversos tipos	<code>[4.0, 'Cadena', True]</code>
<code>tuple</code>	Secuencia	Inmutable, puede contener objetos de diversos tipos	<code>(4.0, 'Cadena', True)</code>
<code>set</code>	Conjunto	Mutable, sin orden, no contiene duplicados	<code>set([4.0, 'Cadena', True])</code>
<code>frozenset</code>	Conjunto	Inmutable, sin orden, no contiene duplicados	<code>frozenset([4.0, 'Cadena', True])</code>
<code>dict</code>	Mapping	Grupo de pares clave:valor	<code>{'key1': 1.0, 'key2': False}</code>
<code>int</code>	Número entero	Precisión fija, convertido en <code>long</code> en caso de overflow.	<code>42</code>
<code>long</code>	Número entero	Precisión arbitraria	<code>42L ó 456966786151987643L</code>
<code>float</code>	Número decimal	Coma flotante de doble precisión	<code>3.1415927</code>
<code>complex</code>	Número complejo	Parte real y parte imaginaria <code>j</code> .	<code>(4.5 + 3j)</code>
<code>bool</code>	Booleano	Valor booleano verdadero o falso	<code>True o False</code>

- Una sentencia condicional (if) ejecuta su bloque de código interno solo si se cumple cierta condición.
- Se define usando la palabra clave if seguida de la condición, y el bloque de código.
- Condiciones adicionales, si las hay, se introducen usando elif seguida de la condición y su bloque de código.
- Todas las condiciones se evalúan secuencialmente hasta encontrar la primera que sea verdadera, y su bloque de código asociado es el único que se ejecuta.



```
Test.py x
1 num = 10
2 if (num == 0):
3     print("Number is Zero")
4
5 elif (num > 5):
6     print("Number is greater than 5")
7
8 else:
9     print("Number is smaller than 5")
10
```

The image shows a screenshot of a Python code editor with a file named 'Test.py'. The code contains a series of conditional statements using 'if', 'elif', and 'else'. It defines a variable 'num' and prints different messages based on its value. The code is color-coded, with keywords in blue and strings in green. Line numbers are visible on the left, and a cursor is at the end of line 10.

## Ciclo for

- El bucle for es similar a foreach en otros lenguajes.
- Recorre un objeto iterable, como una lista, una tupla o un generador, y por cada elemento del iterable ejecuta el bloque de código interno.
- Se define con la palabra clave for seguida de un nombre de variable, seguido de in, seguido del iterable, y finalmente el bloque de código interno.

The screenshot shows a code editor window titled "EjemploCicloForItem.py" containing the following Python code:

```
# decodigo.com
for i in (3, 5, 3, 6, 100):
    print(i)
else:
    print("Finalizado")
```

Below the code editor is a "Run" history window titled "EjemploCicloForItem" showing the execution results:

Run:	EjemploCicloForItem
▶	/home/decodigo/Documentos/python/venv/bin/python
↑	3
↓	5
☰	3
⟳	6
⠇	100
🖨️	Finalizado
🖨️	Process finished with exit code 0

- Para declarar una lista se usan los corchetes [], en cambio, para declarar una tupla se usan los paréntesis ().
- En ambas los elementos se separan por comas, y en el caso de las tuplas es necesario que tengan como mínimo una coma.
- Tanto las listas como las tuplas pueden contener elementos de diferentes tipos. No obstante las listas suelen usarse para elementos del mismo tipo en cantidad variable mientras que las tuplas se reservan para elementos distintos en cantidad fija.
- Para acceder a los elementos de una lista o tupla se utiliza un índice entero (empezando por "0", no por "1").
- Se pueden utilizar índices negativos para acceder elementos a partir del final.
- Las listas se caracterizan por ser mutables, es decir, se puede cambiar su contenido en tiempo de ejecución, mientras que las tuplas son inmutables ya que no es posible modificar el contenido una vez creada.

## Listas y Tuplas

```
>>> lista = ["abc", 42, 3.1415]
>>> lista[0] # Acceder a un elemento por su índice
'abc'
>>> lista[-1] # Acceder a un elemento usando un índice negativo
3.1415
>>> lista.append(True) # Añadir un elemento al final de la lista
>>> lista
['abc', 42, 3.1415, True]
>>> del lista[3] # Borra un elemento de la lista usando un índice (en este caso: True)
>>> lista[0] = "xyz" # Re-assignar el valor del primer elemento de la lista
>>> lista[0:2] # Mostrar los elementos de la lista del índice "0" al "2" (sin incluir este último)
['xyz', 42]
>>> lista_anidada = [lista, [True, 42L]] # Es posible anidar listas
>>> lista_anidada
[['xyz', 42, 3.1415], [True, 42L]]
>>> lista_anidada[1][0] # Acceder a un elemento de una lista dentro de otra lista (del segundo elemento, mostrar el primer elemento)
True
```

```
>>> tupla = ("abc", 42, 3.1415)
>>> tupla[0] # Acceder a un elemento por su índice
'abc'
>>> del tupla[0] # No es posible borrar (ni añadir) un elemento en una tupla, lo que provocará una excepción
( Excepción )
>>> tupla[0] = "xyz" # Tampoco es posible re-assignar el valor de un elemento en una tupla, lo que también provocará una excepción
( Excepción )
>>> tupla[0:2] # Mostrar los elementos de la tupla del índice "0" al "2" (sin incluir este último)
('abc', 42)
>>> tupla_anidada = (tupla, (True, 3.1415)) # También es posible anidar tuplas
>>> 1, 2, 3, "abc" # Esto también es una tupla, aunque es recomendable ponerla entre paréntesis (recuerda que requiere, al menos, una coma)
(1, 2, 3, 'abc')
>>> (1) # Aunque entre paréntesis, esto no es una tupla, ya que no posee al menos una coma, por lo que únicamente aparecerá el valor
1
>>> (1,) # En cambio, en este otro caso, sí es una tupla
(1,)
>>> (1, 2) # Con más de un elemento no es necesaria la coma final
(1, 2)
>>> (1, 2,) # Aunque agregarla no modifica el resultado
(1, 2)
```

## **Exposición sobre patrones de diseño**

- Patrón strategy
- Patrón Factory
- Patrón Facade
- Patrón Builder
- Patron Singleton

**Primera Tarea Proyecto final (El progreso deberá estar ya sus respectivos Branch del proyecto creado en Github)**

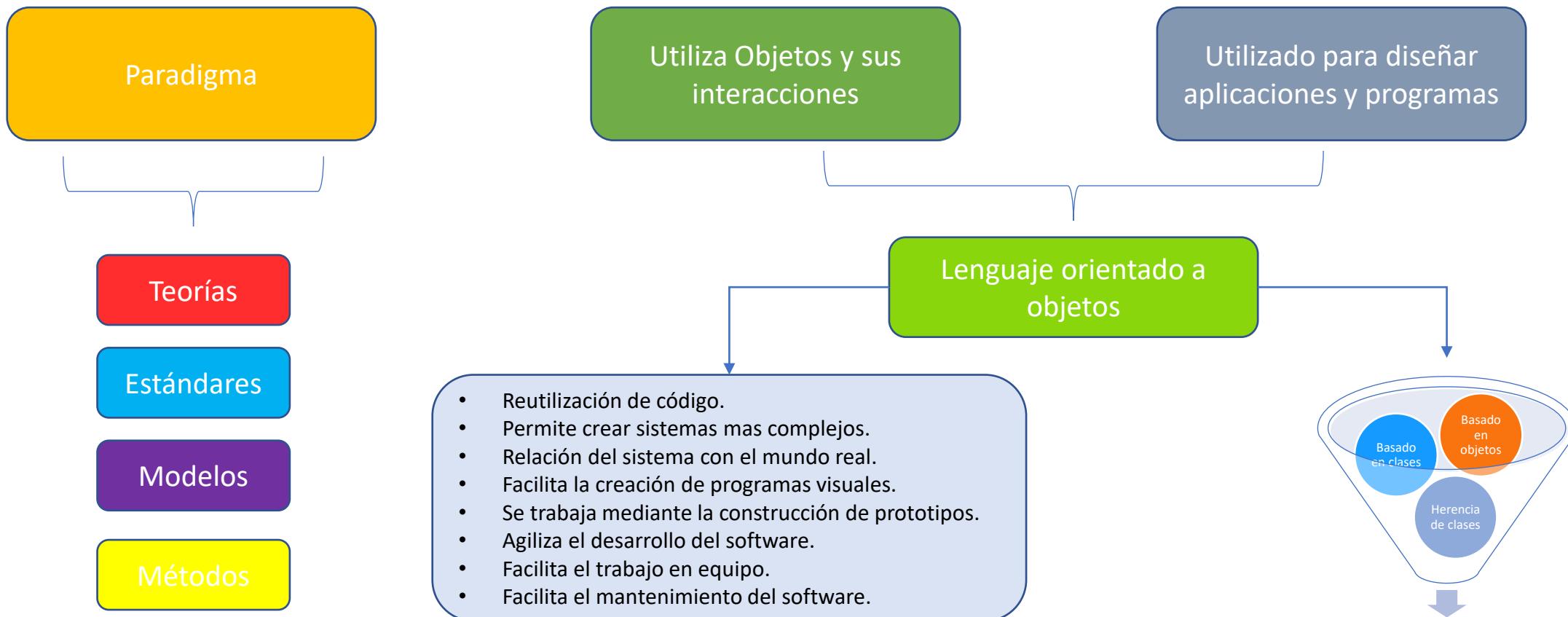
- Crear Hello world usando Flask y pipenv (i.e: get/post = <http://localhost:9090/hello>) - (Sebastian Ontiveros)
- Compilar clase java desde consola – ejecutar comandos desde Python (java) - (Alvaro Cruz)
- Complilar código Python desde consola - ejecutar comandos desde Python (python) - (Andres Cox)
- Bases de datos con Python - (Gonzalo Alarcon)

# Programación Orientada a Objetos

---



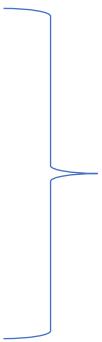
## Definición POO



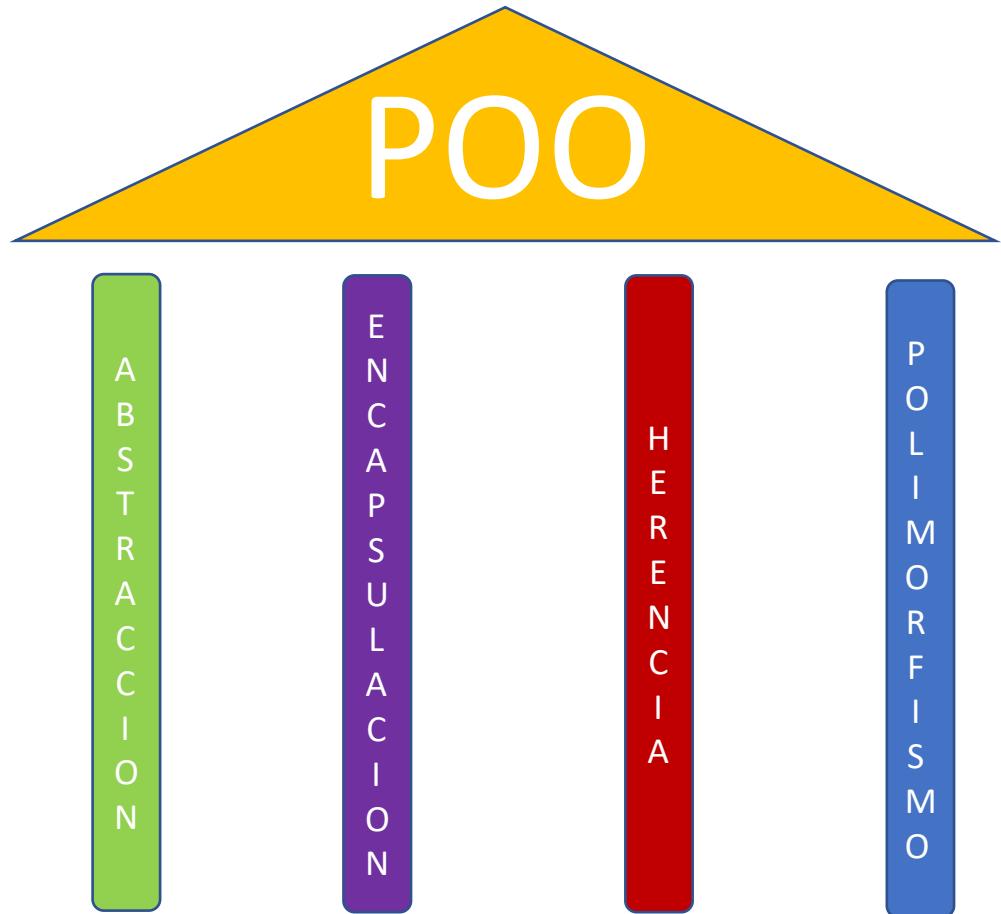
La **programación orientada a objetos (POO)** es un paradigma de programación que usa objetos para crear aplicaciones. Está basada en pilares fundamentales: Abstracción, herencia, polimorfismo, encapsulación.

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

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



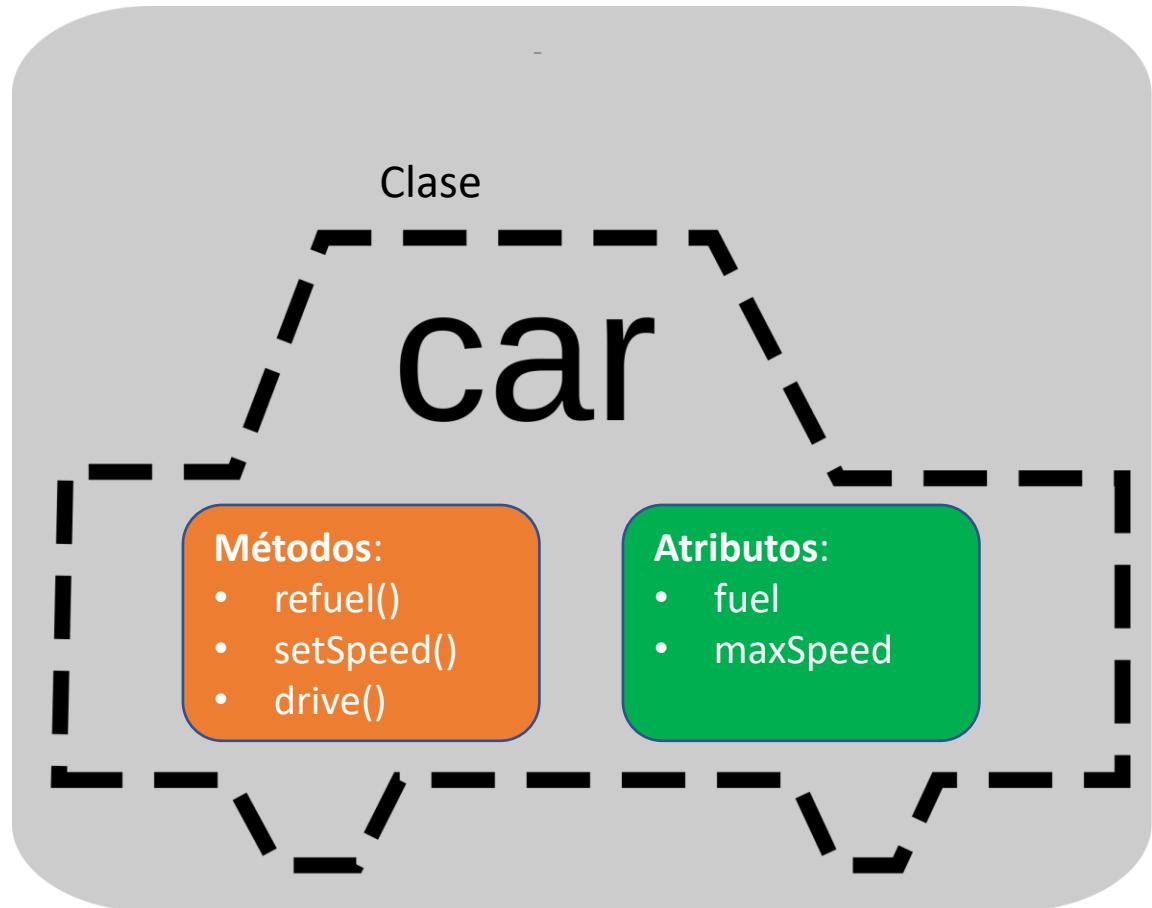
Pilares fundamentales de  
la programación orientada  
a objetos



## 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.

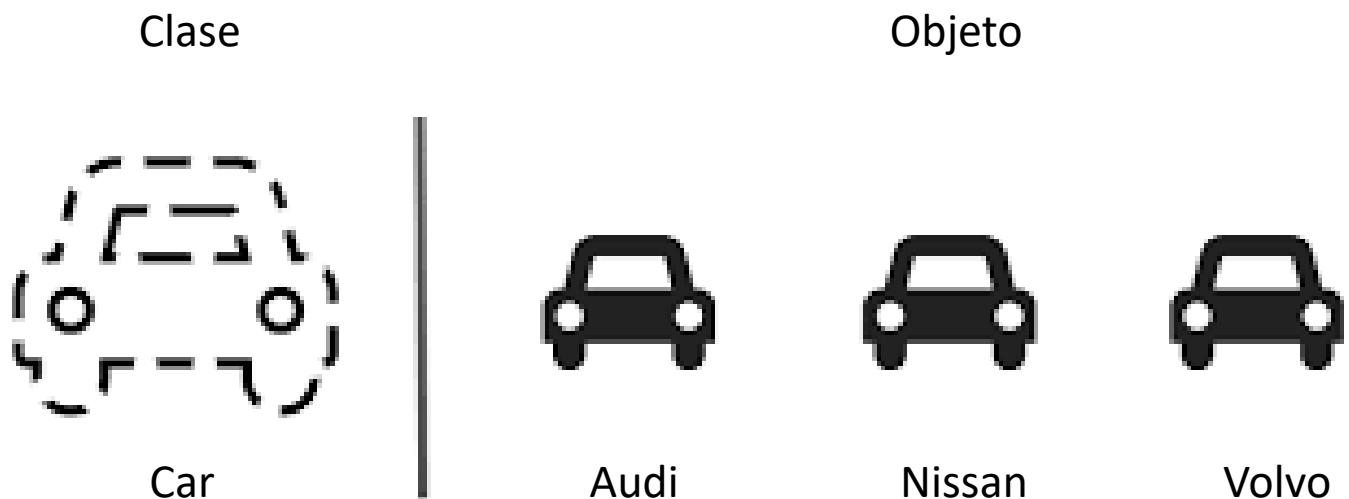


Objeto

## QUÉ ES UN OBJETO?

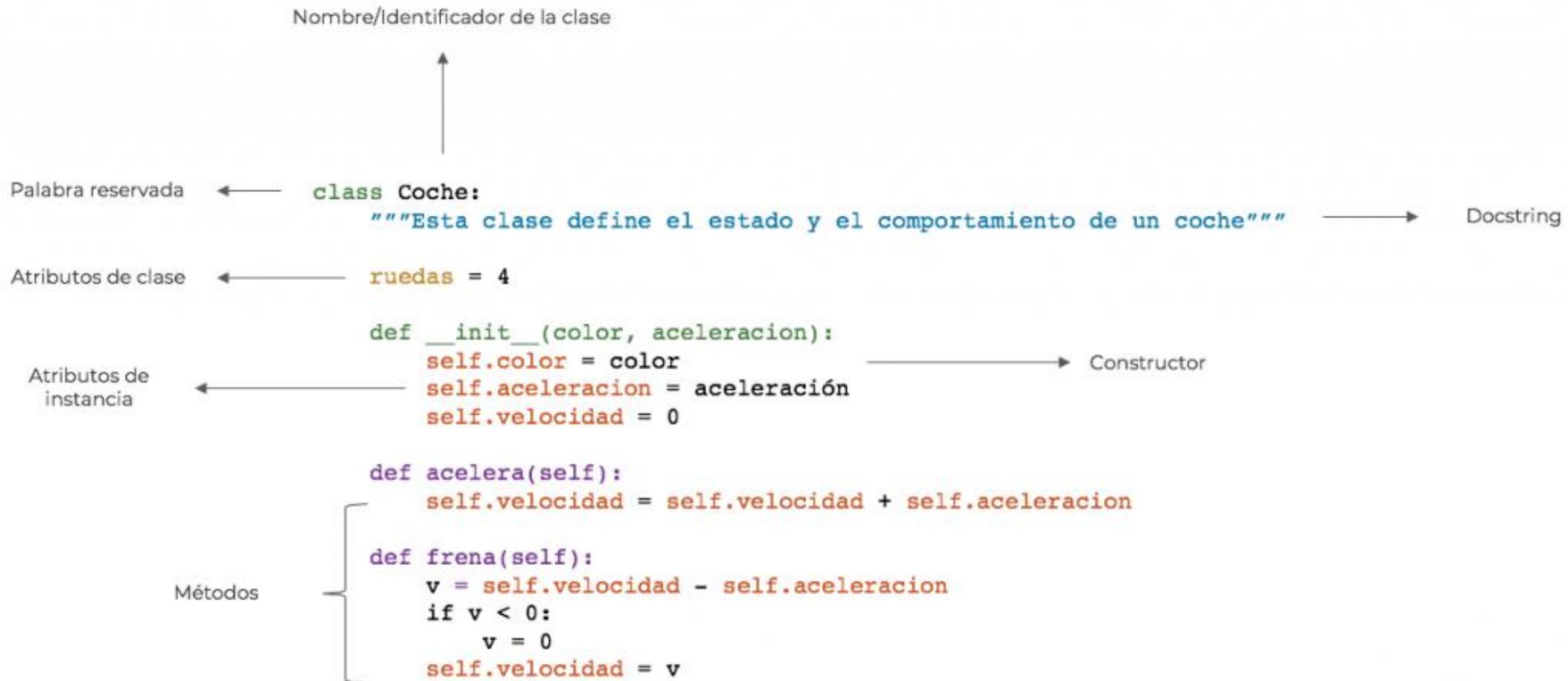
Un objeto es una instancia de una clase, una instancia es un ejemplar, un caso específico de una clase, por ejemplo yo puedo crear varios objetos de tipo clase y cada uno es diferente, por ejemplo el primero objeto es de color negro, el segundo objeto es de color blanco.

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.



## Ejemplo: clase y objeto

### Clase



## Ejemplo: clase y objeto

### Objeto

```
1. >>> c1 = Coche('rojo', 20)
2. >>> print(c1.color)
3. rojo
4. >>> print(c1.ruedas)
5. 4
6. >>> c2 = Coche('azul', 30)
7. >>> print(c2.color)
8. azul
9. >>> print(c2.ruedas)
10. 4
```

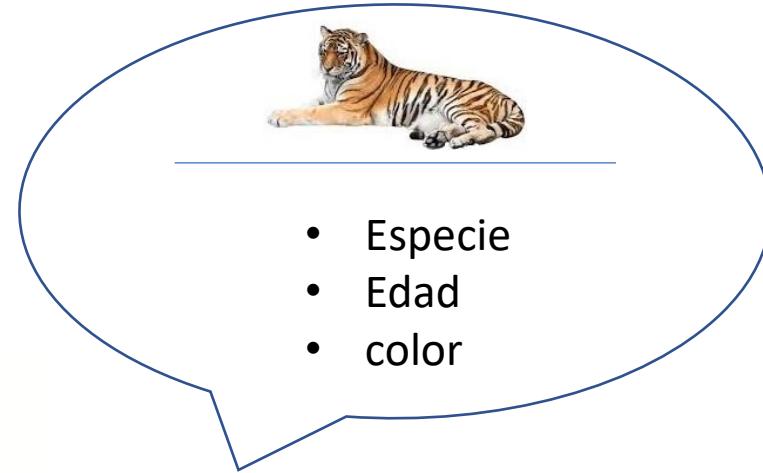
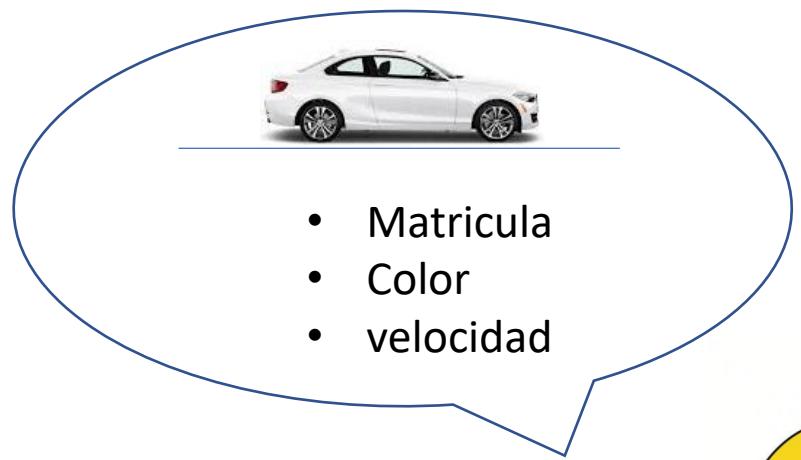


## Métodos



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 PYTHON estos métodos están alojados en una clase.

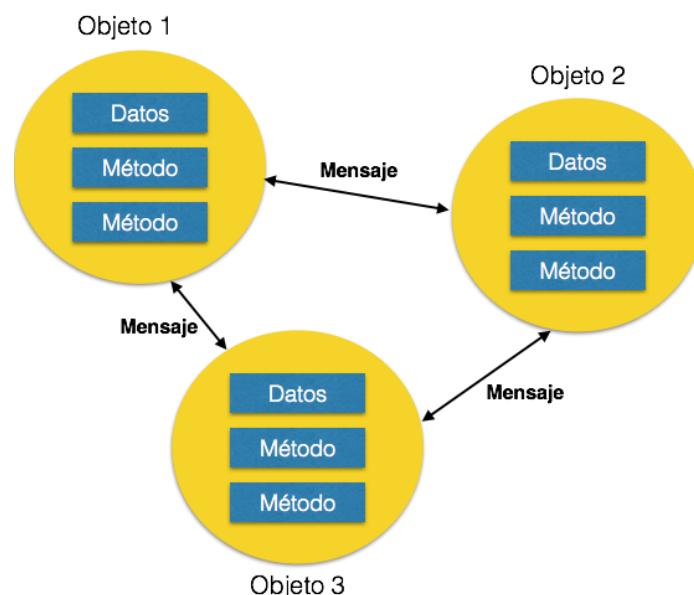
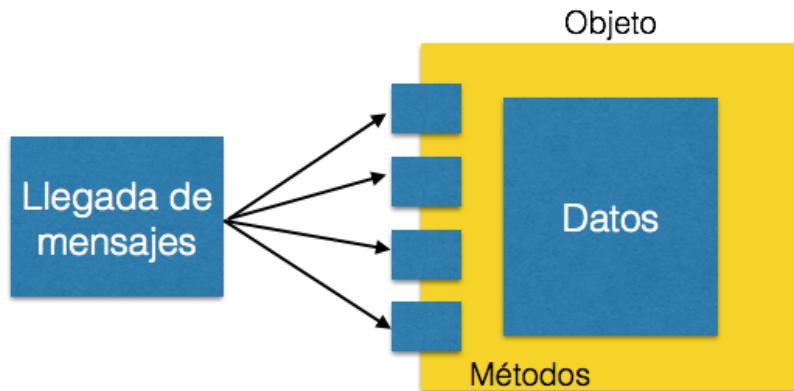
## Atributos



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)

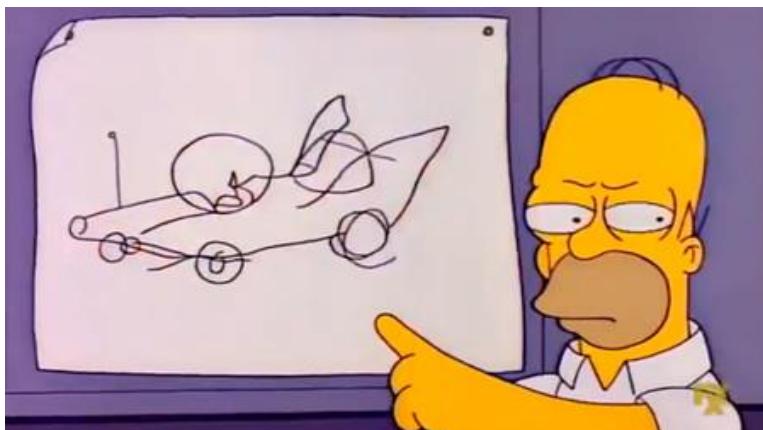


## Abstracción

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.

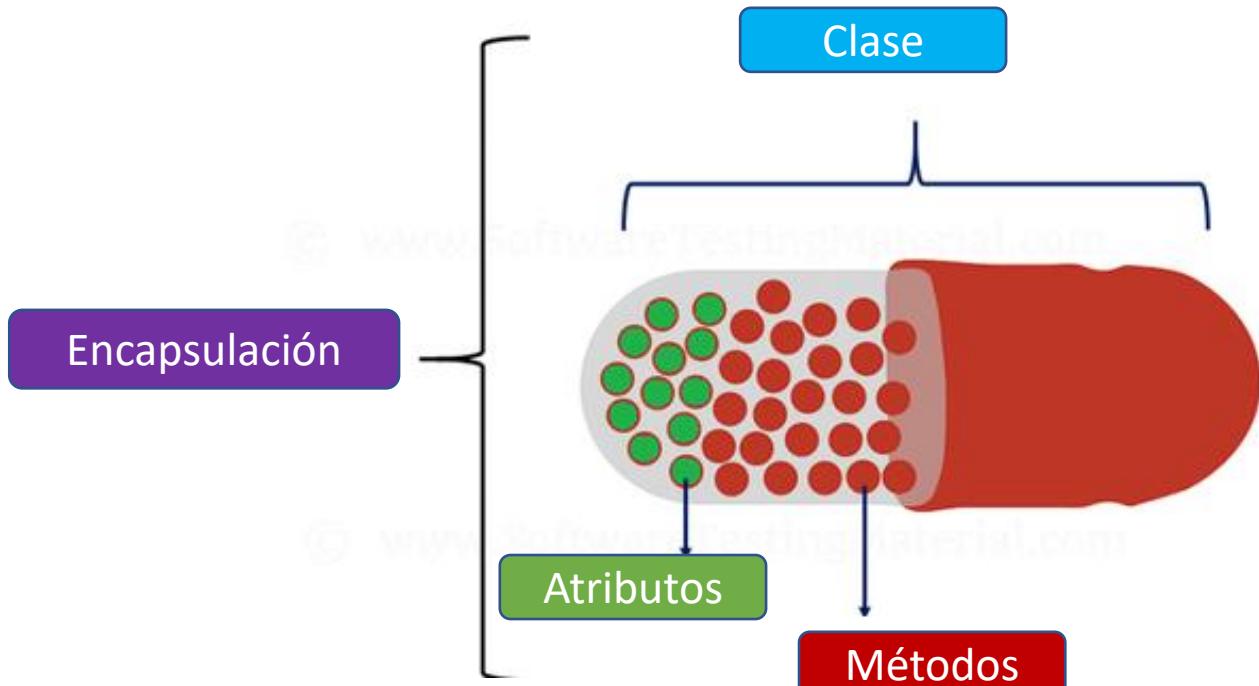
# Abstracción



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.
- Python no distingue entre métodos o atributos públicos y privados, sino que todos los objetos dentro de una clase o módulo pueden ser accedidos por fuera de ellos. No obstante, como convención se prefija un guión bajo para indicar que un objeto *debería* ser interpretado –por el programador– como privado.



```
class MiClase:

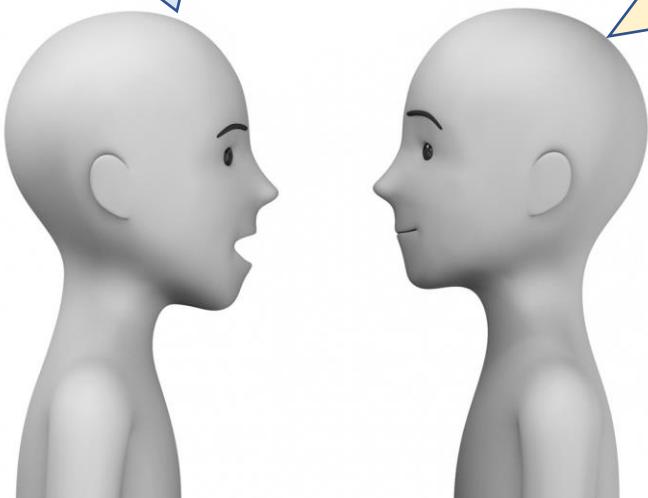
    def __init__(self):
        self._atributo_privado = 1

    def _metodo_privado(self):
        print("Hola mundo!")
```

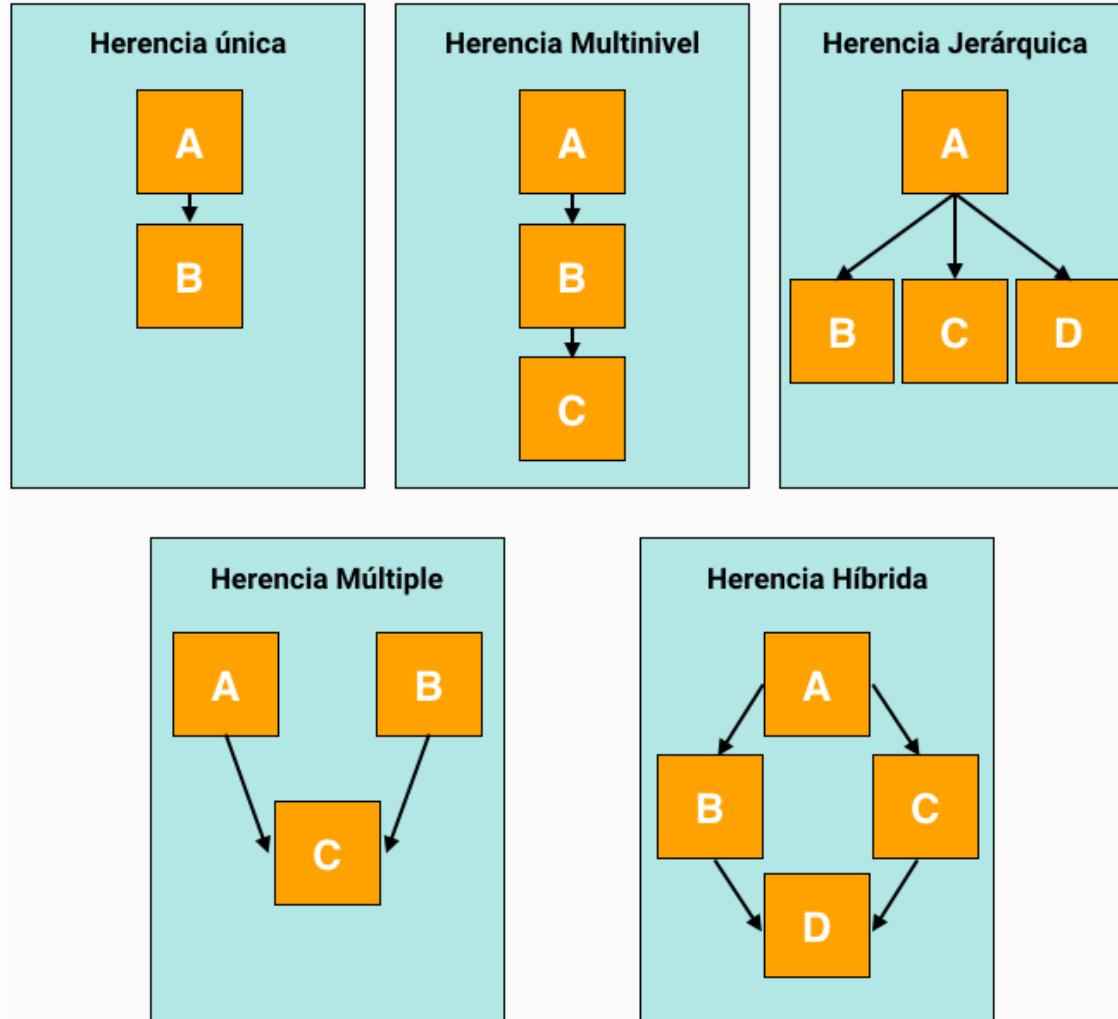
## Herencia

¿Qué es la Herencia en la programación Orientada a Objetos?

En POO, la herencia es un mecanismo que nos permite extender las funcionalidades de una clase ya existente. De este modo vamos a favorecer la **reutilización** de nuestro código.



Herencia



## Superclases

La idea de la herencia es identificar una clase base (la superclase) con los atributos comunes y luego crear las demás clases heredando de ella (las subclases) extendiendo sus campos específicos. En nuestro caso esa clase sería el Producto en sí mismo:

```
class Producto:  
    def __init__(self, referencia, nombre, pvp, descripcion):  
        self.referencia = referencia  
        self.nombre = nombre  
        self.pvp = pvp  
        self.descripcion = descripcion  
  
    def __str__(self):  
        return f"REFERENCIA\t {self.referencia}\n" \  
               f"NOMBRE\t\t {self.nombre}\n" \  
               f"PVP\t\t {self.pvp}\n" \  
               f"DESCRIPCIÓN\t {self.descripcion}\n"
```

## Subclases

Para heredar los atributos y métodos de una clase en otra sólo tenemos que pasarla entre paréntesis durante la definición:

```
class Adorno(Producto):
    pass

adorno = Adorno(2034, "Vaso adornado", 15, "Vaso de porcelana")
print(adorno)
```

Como se puede apreciar es posible utilizar el comportamiento de una superclase sin definir nada en la subclase.

```
class Alimento(Producto):
    productor = ""
    distribuidor = ""

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n" \
               f"PRODUCTOR\t\t {self.productor}\n" \
               f"DISTRIBUIDOR\t\t {self.distribuidor}\n"

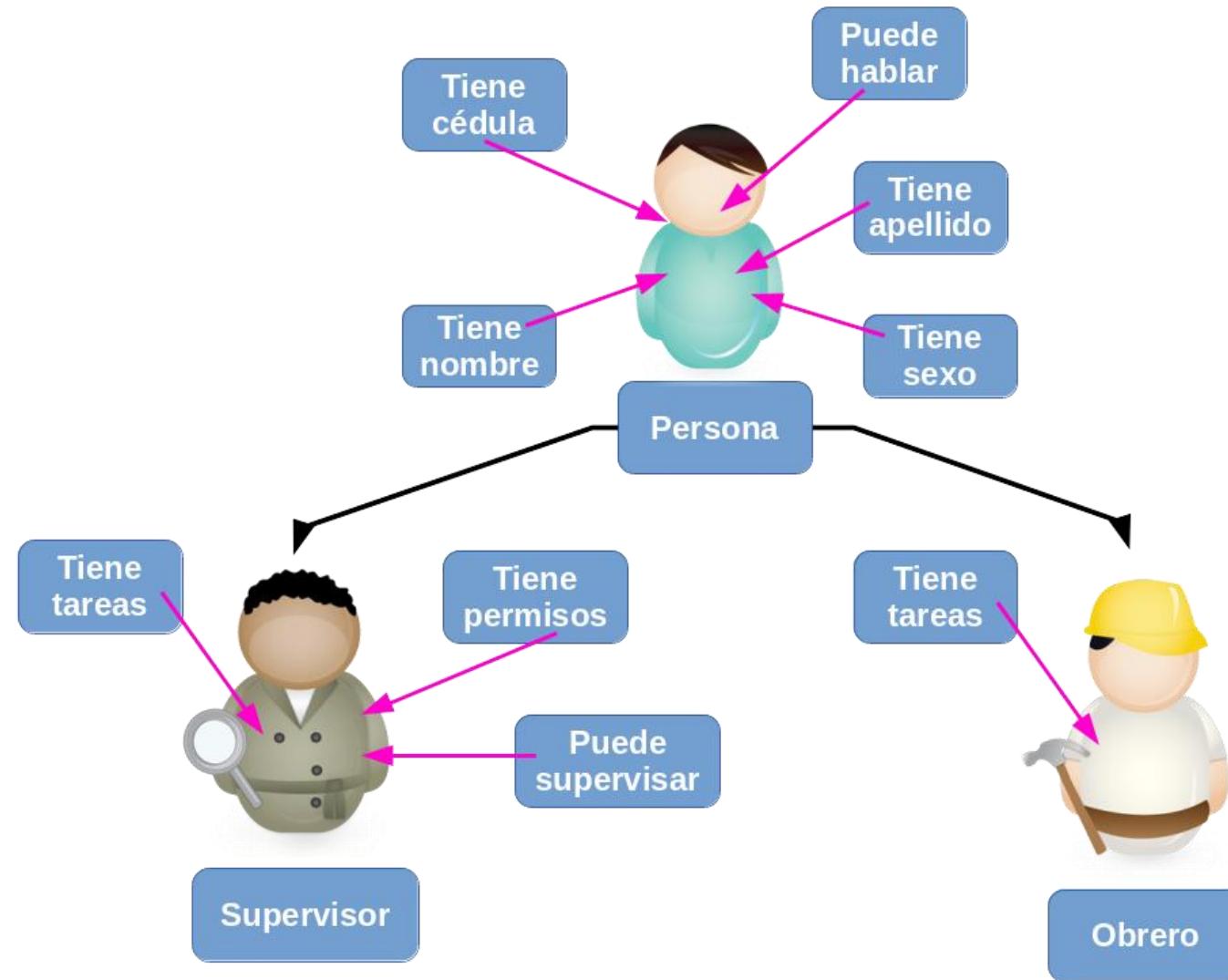
class Libro(Producto):
    isbn = ""
    autor = ""

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n" \
               f"ISBN\t\t {self.isbn}\n" \
               f"AUTOR\t\t {self.autor}\n"
```

```
alimento = Alimento(2035, "Botella de Aceite de Oliva", 5, "250 ML")
alimento.productor = "La Aceitera"
alimento.distribuidor = "Distribuciones SA"
print(alimento)

libro = Libro(2036, "Cocina Mediterránea", 9, "Recetas sanas y buenas")
libro.isbn = "0-123456-78-9"
libro.autor = "Doña Juana"
print(libro)
```

## Herencia- simple



## Herencia- simple

```
class Persona(object):
    """Clase que representa una Persona"""

    def __init__(self, cedula, nombre, apellido, sexo):
        """Constructor de clase Persona"""
        self.cedula = cedula
        self.nombre = nombre
        self.apellido = apellido
        self.sexo = sexo

    def __str__(self):
        """Devuelve una cadena representativa de Persona"""
        return "%s: %s, %s %s." % (
            self.__doc__[25:34], str(self.cedula), self.nombre,
            self.apellido, self.getGenero(self.sexo))

    def hablar(self, mensaje):
        """Mostrar mensaje de saludo de Persona"""
        return mensaje

    def getGenero(self, sexo):
        """Mostrar el genero de la Persona"""
        genero = ('Masculino', 'Femenino')
        if sexo == "M":
            return genero[0]
        elif sexo == "F":
            return genero[1]
        else:
```

```
class Supervisor(Persona):
    """Clase que representa a un Supervisor"""

    def __init__(self, cedula, nombre, apellido, sexo, rol):
        """Constructor de clase Supervisor"""
        # Invoca al constructor de clase Persona
        Persona.__init__(self, cedula, nombre, apellido, sexo)

        # Nuevos atributos
        self.rol = rol
        self.tareas = ['10', '11', '12', '13']

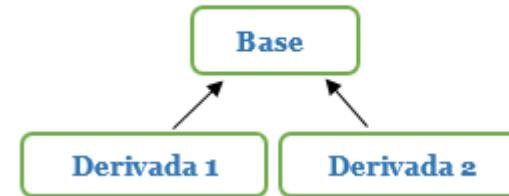
    def __str__(self):
        """Devuelve una cadena representativa al Supervisor"""
        return "%s: %s %s, rol: '%s', sus tareas: %s." % (
            self.__doc__[26:37], self.nombre, self.apellido,
            self.rol, self.consulta_tareas())

    def consulta_tareas(self):
        """Mostrar las tareas del Supervisor"""
        return ', '.join(self.tareas)
```

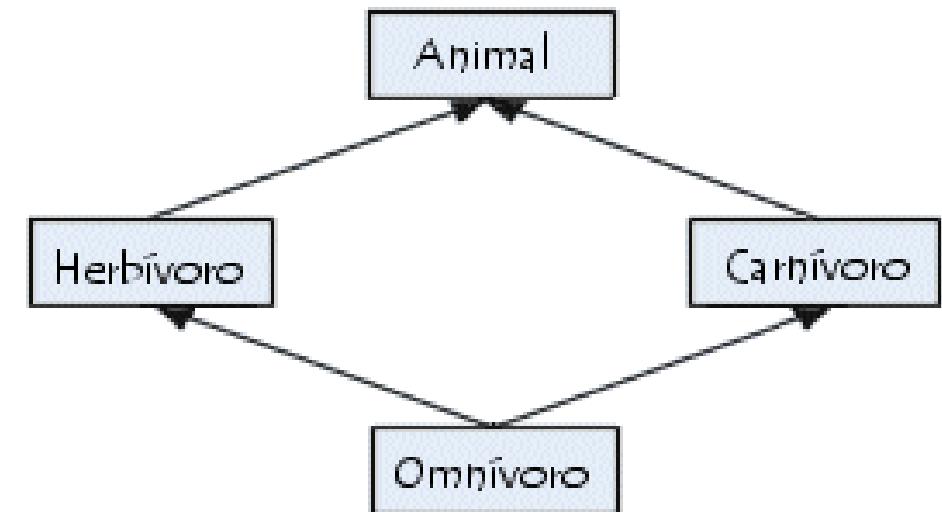
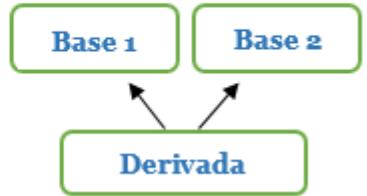
### Herencia Múltiple

- A diferencia de lenguajes como *Java* y *C#*, el lenguaje *Python* permite la herencia múltiple, es decir, se puede heredar de múltiples clases.
- La herencia múltiple es la capacidad de una subclase de heredar de múltiples súper clases.
- Esto conlleva un problema, y es que si varias súper clases tienen los mismos atributos o métodos, la subclase sólo podrá heredar de una de ellas.
- En estos casos Python dará prioridad a las clases más a la izquierda en el momento de la declaración de la subclase

**Herencia simple**



**Herencia múltiple**



## Herencia - multiple

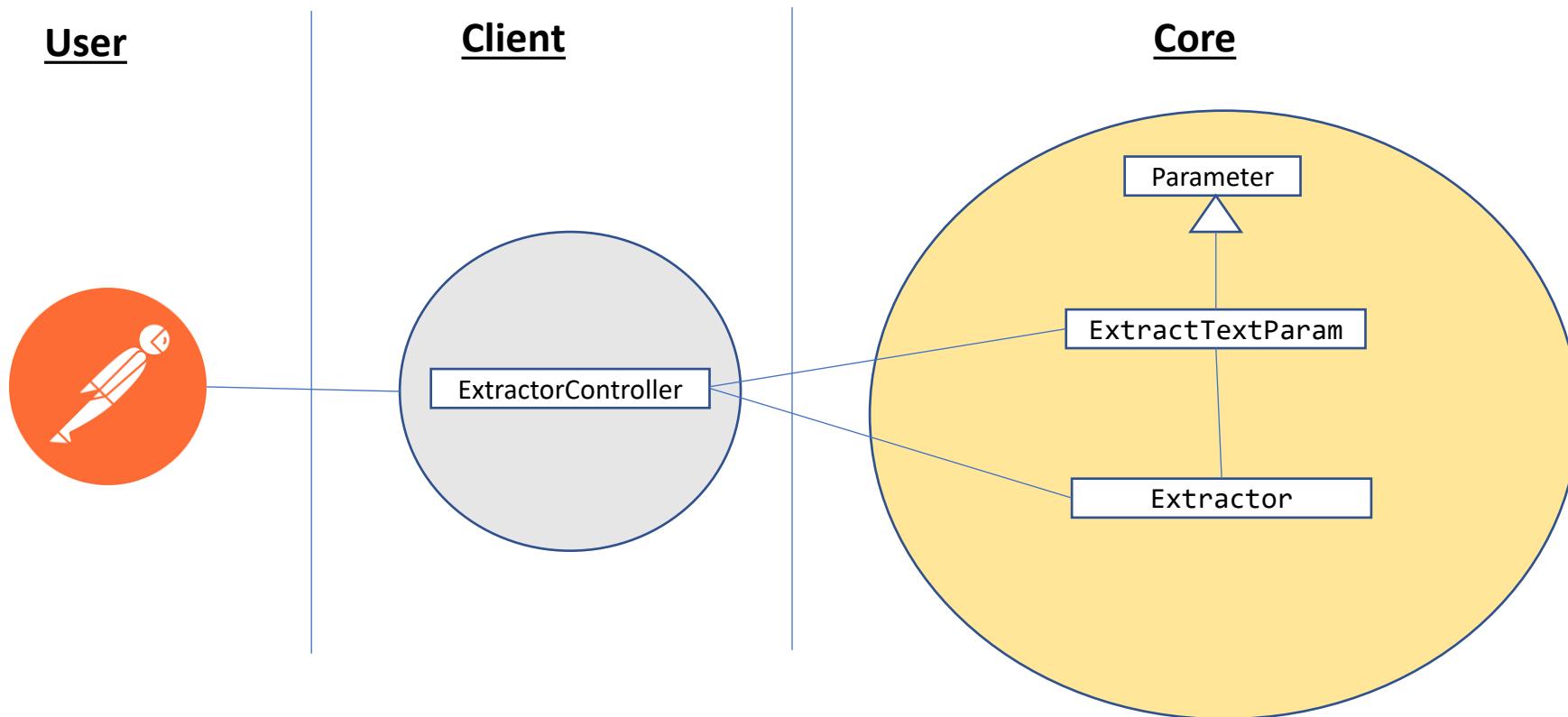
```
class A:
    def __init__(self):
        print("Soy de clase A")
    def a(self):
        print("Este método lo heredo de A")
from abc import ABCMeta, abstractmethod
class AbstractFoo:
    __metaclass__ = ABCMeta
    @abstractmethod
    def bar(self):
        pass
class Foo(object):
    @classmethod
    def __subclasshook__(cls, C):
        return NotImplemented
    class Foo(object):
        def bar(self):
            print "hola"
c = (
    c.a(
        AbstractFoo.register(Foo)
    ),
    c.b(
        foo = Foo()
    ),
    c.c(
        issubclass(Foo, AbstractFoo)
    )
)
```

True

```
Soy de clase B
Este método lo heredo de A
Este método lo heredo de B
Este método es de C
```

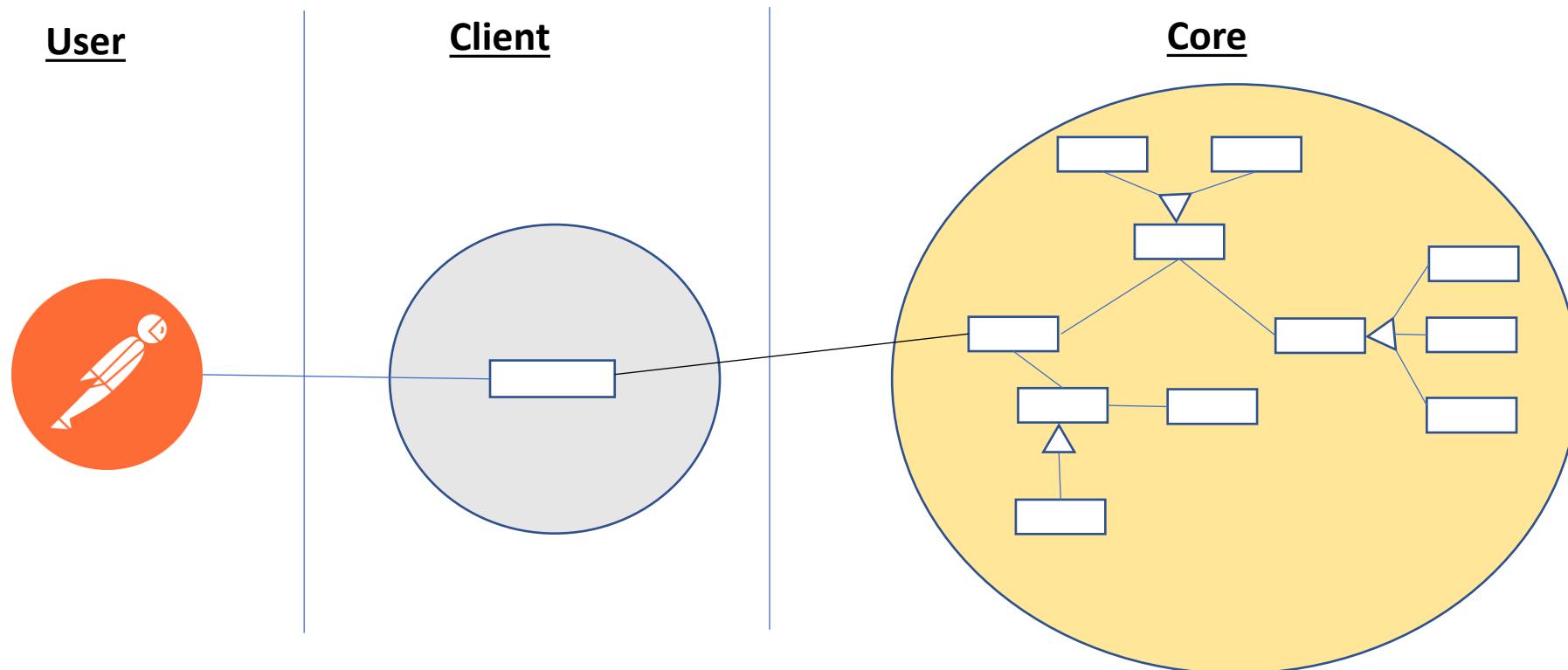
## Diagrama de Clase

## Diagrama de clases – Extractor



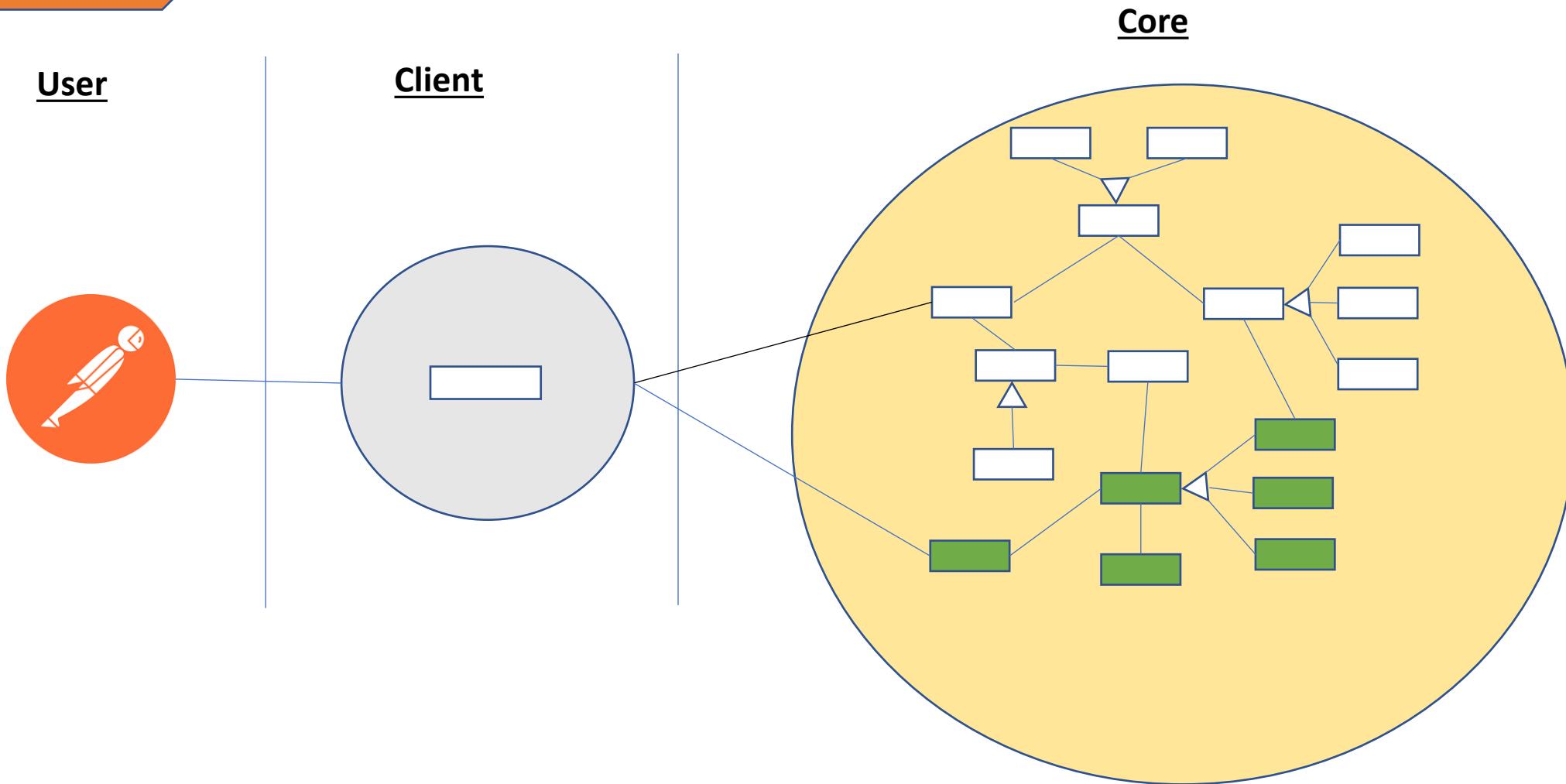
## Diagrama de Clase

## Diagrama de clases – Extractor



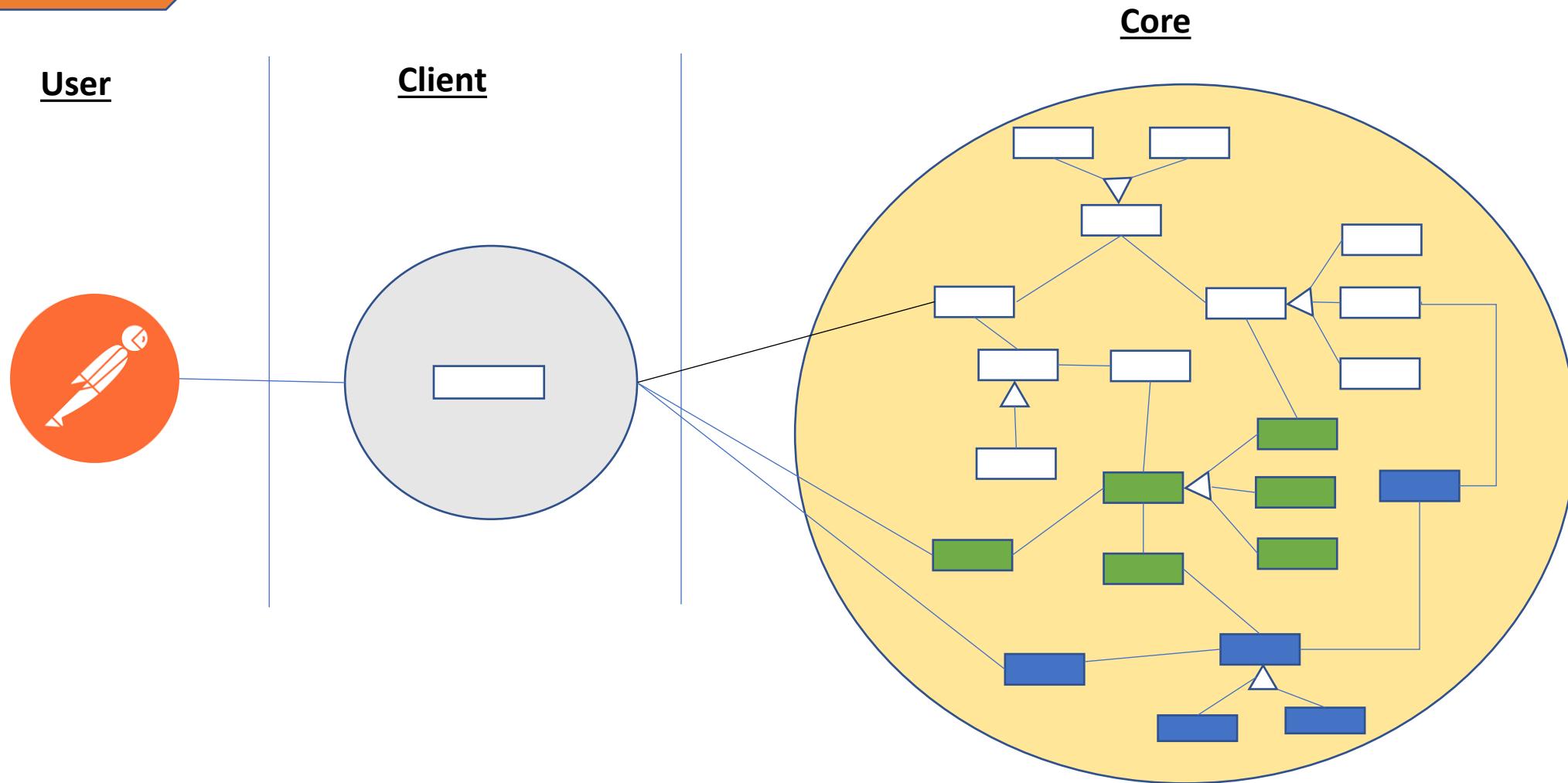
## Diagrama de Clase

## Diagrama de clases – Extractor y Usuario



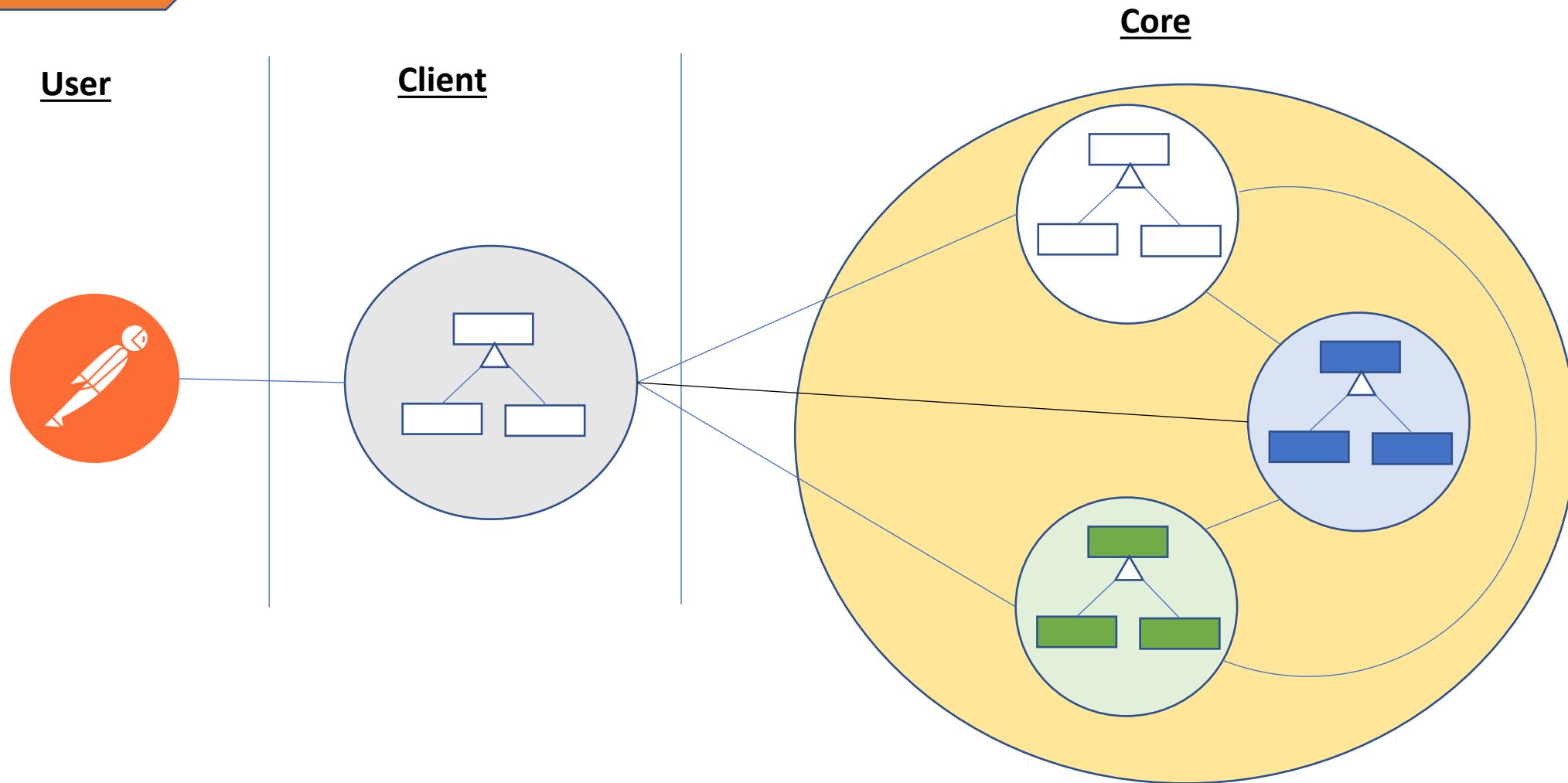
## Diagrama de Clase

## Diagrama de clases – Extractor, Usuario y Conversor



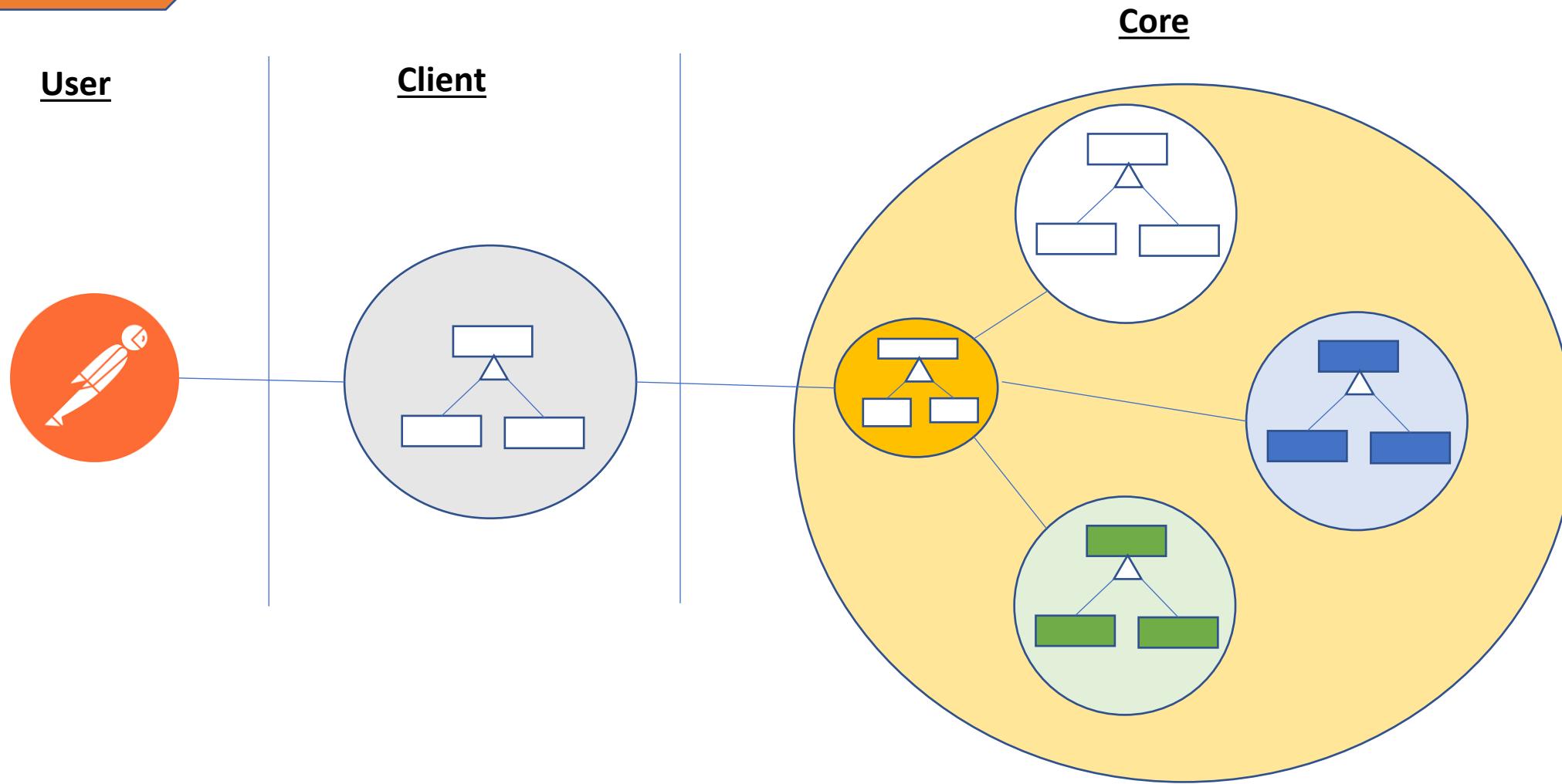
## Diagrama de Clase

## Diagrama de clases – Extractor, Usuario y Conversor



## Diagrama de Clase

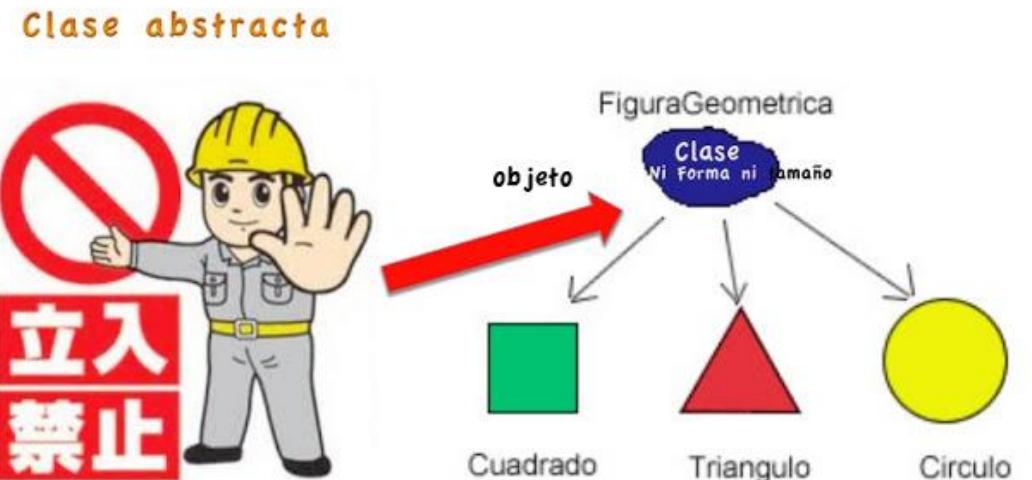
## Diagrama de clases – Extractor, Usuario y Conversor



## Clases Abstractas

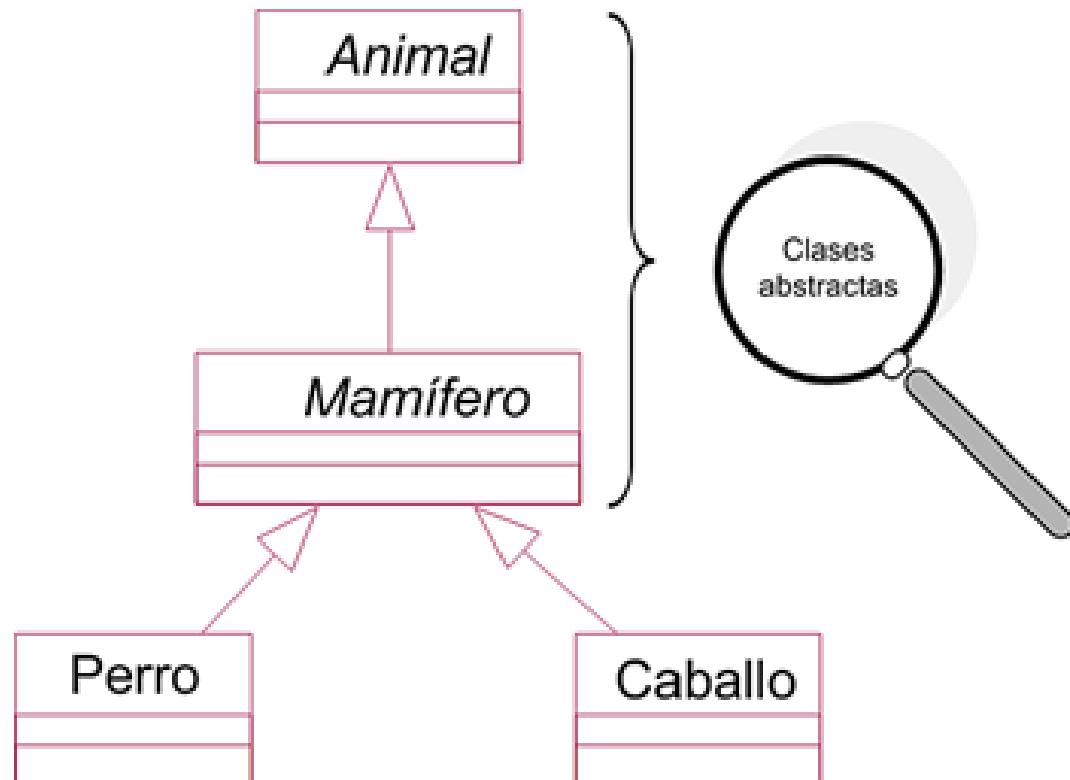
Una clase abstracta es una clase de la cual no se pueden definir instancias (u objetos). Por tanto, las clases abstractas tendrán dos utilidades principales:

1. En primer lugar, evitan que los usuarios de la clase puedan crear objetos de la misma, como dice la definición de clase abstracta.
2. En segundo lugar, permiten crear interfaces que luego deben ser implementados por las clases que hereden de la clase abstracta.



### ¿Por qué usar clases base abstractas?

- Las clases base abstractas son una forma de comprobación de interfaz más estricta que las comprobaciones individuales `hasattr()` para métodos particulares.
- Al definir una clase base abstracta, se puede establecer una interfaz de programación común para un conjunto de subclases.
- Esta capacidad es especialmente útil en situaciones en las que alguien menos familiarizado con el origen de una aplicación proporcionará extensiones de complementos, pero también puede ayudar cuando se trabaja en un equipo grande o con una base de código grande donde se realiza un seguimiento de todas las clases al mismo tiempo son difíciles o imposibles.



## Clases Abstractas

- En otros lenguajes se puede crear clases y métodos abstractos de forma explícita.
- Python no requiere de una definición explícita de una clase o método abstracto.
- El modulo abc permite el uso explícito de clases abstractas básicas tal como se especifica en el PEP 3119

```
from abc import ABCMeta, abstractmethod

class AbstractFoo:
    __metaclass__ = ABCMeta

    @abstractmethod
    def bar(self):
        pass

    @classmethod
    def __subclasshook__(cls, C):
        return NotImplemented

class Foo(object):
    def bar(self):
        print "hola"
```

```
AbstractFoo.register(Foo)
foo = Foo()
issubclass(Foo, AbstractFoo)
```

True

### Clase base auxiliar

Olvidar establecer la metaclasses correctamente significa que las implementaciones concretas no tienen sus interfaces de programación forzadas. Para facilitar la configuración adecuada de la clase abstracta, se proporciona una clase base que establece la metaclasses automáticamente.

```
import abc

class PluginBase(abc.ABC):

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source
        and return an object.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""

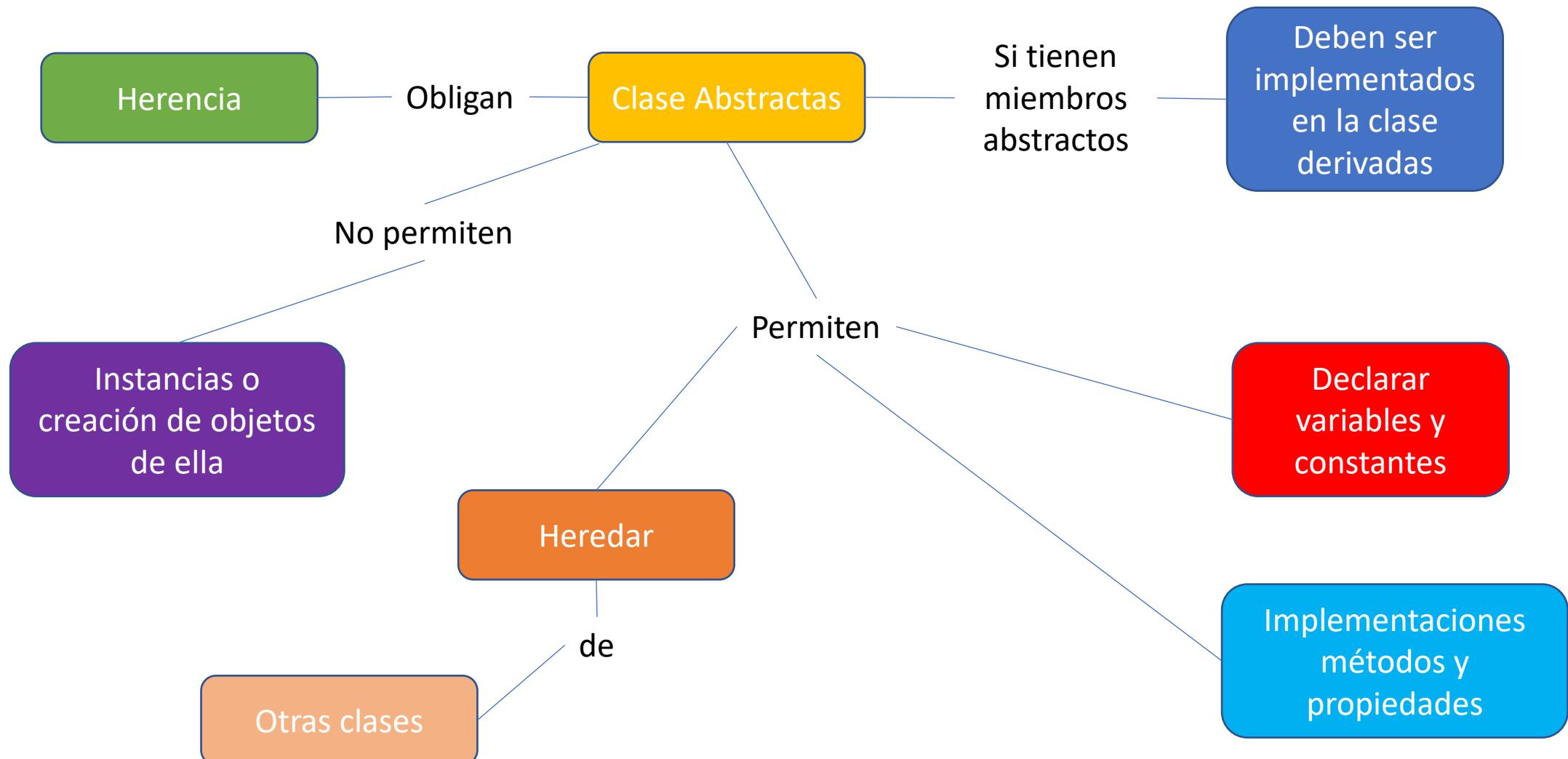
class SubclassImplementation(PluginBase):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)

if __name__ == '__main__':
    print('Subclass:', issubclass(SubclassImplementation,
                                 PluginBase))
    print('Instance:', isinstance(SubclassImplementation(),
                                 PluginBase))
```

## Clases Abstractas



## Clases Abstractas

```
import abc
from abc import ABC

# Declaramos nuestra clase
class Animal(ABC):

    # Primer método
    # Decorador para métodos abstractos
    @abc.abstractmethod
    def setName(self, name):
        self.name = name

    # Segundo método
    # Decorador para métodos abstractos
    @abc.abstractmethod
    def getName(self):
        return self.name
```

```
class Perro(Animal):

    def __init__(self):
        pass

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

perro = Perro()
perro.setName("Marly")
perro.getName()
'Marly'
```

# Manejo de excepciones

---



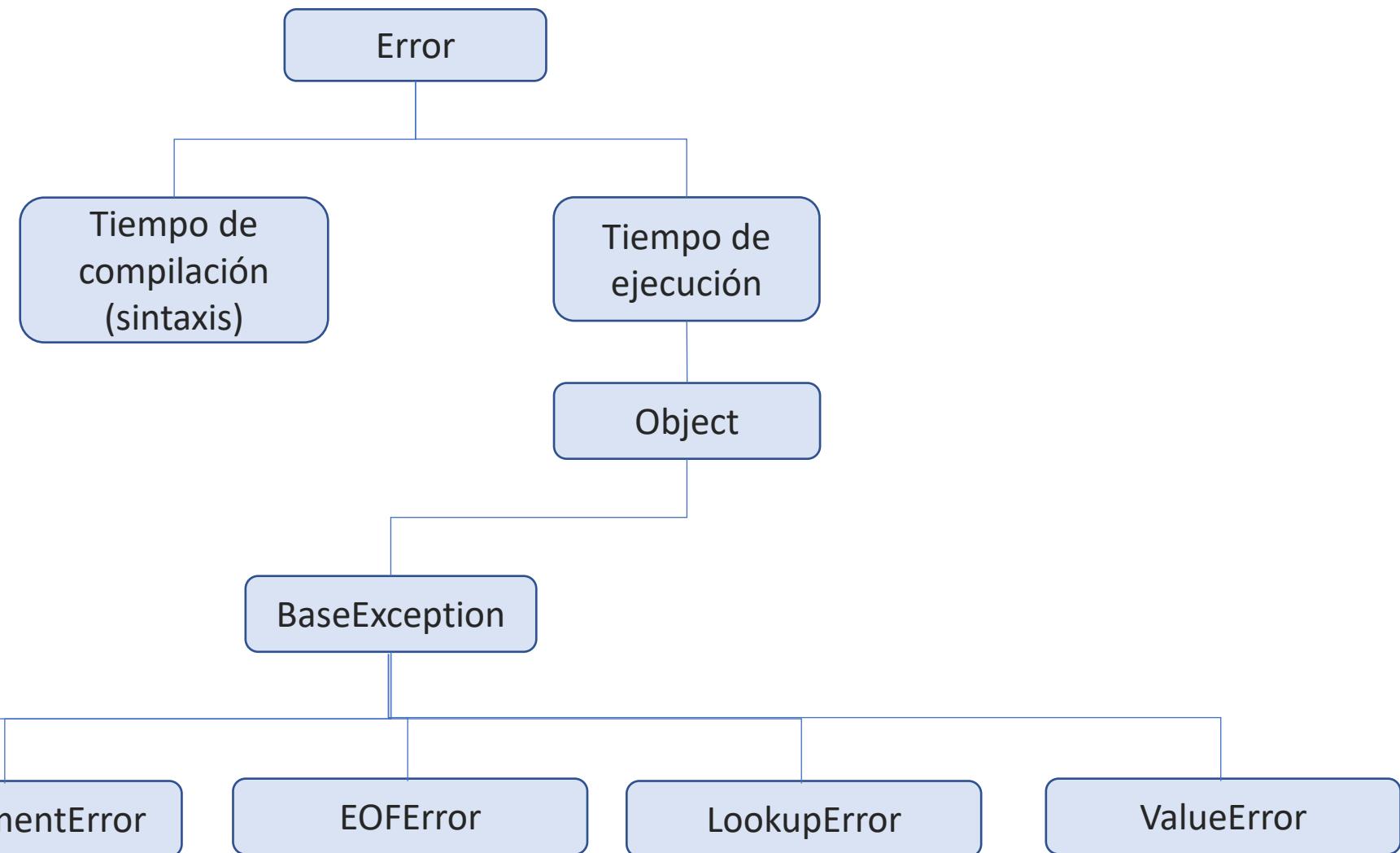
## ¿Qué es una excepción?

“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



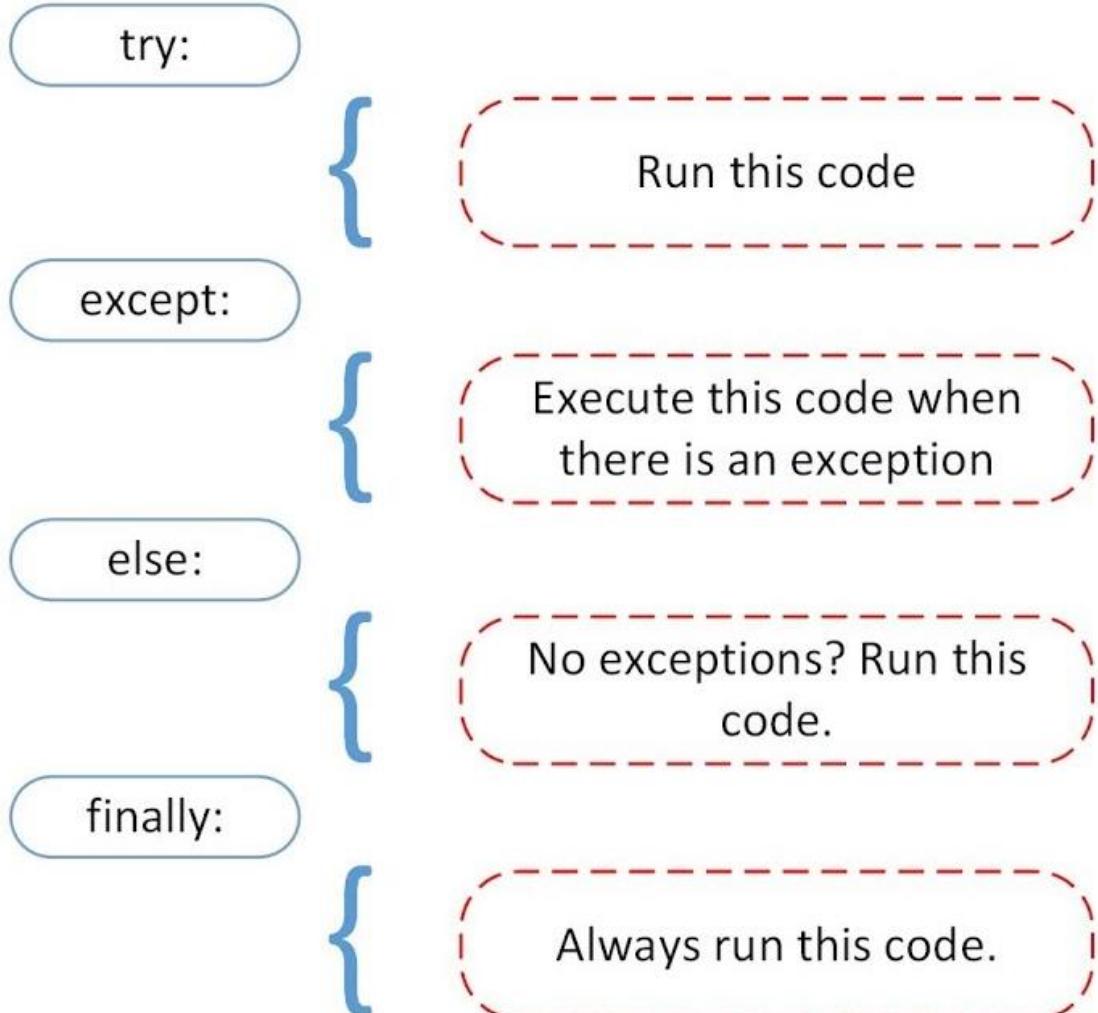
## Excepción

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        +-- BufferError
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- EnvironmentError
            +-- IOError
            +-- OSError
                +-- WindowsError (Windows)
                +-- VMSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            +-- NotImplementedError
        +-- SyntaxError
            +-- IndentationError
                +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            +-- UnicodeError
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
```

## Excepción

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.

En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias **try**, **except** y **finally**.



## Captura de excepciones try - except - finally

- **try** : Aquí vamos a escribir todo el bloque de código que posiblemente llegue a lanzar unas excepción la cual queremos manejar, aquí va tanto el código como llamados a métodos que puedan arrojar la excepción.
- **except** : en caso de que en el try se encuentre alguna excepción, se ingresara automáticamente al bloque except donde se encontrara 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 except, es decir, siempre se ejecutara sin importar que se encuentre o no una excepción.



```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

## Excepción

```
try:  
    archivo = open("miarchivo.txt")  
    # procesar el archivo  
except IOError:  
    print "Error de entrada/salida."  
    # realizar procesamiento adicional  
except:  
    # procesar la excepción  
finally:  
    # si el archivo no está cerrado hay que cerrarlo  
    if not(archivo.closed):  
        archivo.close()
```

```
try:  
    # aquí ponemos el código que puede Lanzar excepciones  
except:  
    # ERROR de sintaxis, esta sentencia no puede estar aquí,  
    # sino que debería estar luego del except IOError.  
except IOError:  
    # Manejo de la excepción de entrada/salida
```

## Procesamiento y propagación de excepciones

Hemos visto cómo atrapar excepciones, es necesario ahora que veamos qué se supone que hagamos al atrapar una excepción.

- Ejecutar alguna lógica particular como: cerrar un archivo, realizar una procesamiento alternativo al del bloque try, etc.
- Propagación de la excepción hacia la función que lo había invocado. Para hacer esto Python nos brinda la instrucción `raise`.

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError as e:
    print 'My exception occurred, value:', e.value
```

## Excepción

```
1 import sys
2
3 def f(a):
4     c=5/a
5     return c
6
7 def sub(d):
8     print("Entered sub(%d)" % d)
9     try:
10         print("Entered try @sub")
11         e = f(d)/(d-1)
12         return e
13     except:
14         print("Do some staff before re-raising the exception upwards")
15         raise
16
17 def main():
18     try:
19         print("Entered try @main")
20         d = int(sys.argv[1])
21         sub(d)
22     except:
23         print("Reached except block @main")
24         raise
25
26 if __name__ == '__main__':
27     main()
```

Console Tasks Problems Executables Search Git Repositories History Caught Exceptions

main.py [debug] [/usr/bin/python3.5]

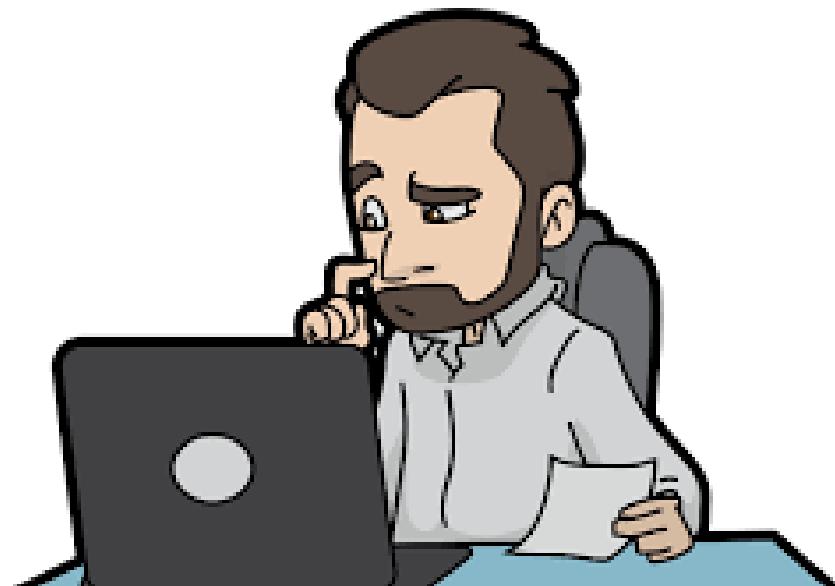
```
Entered try @main
Entered sub(1)
Entered try @sub
Do some staff before re-raising the exception upwards
Reached except block @main
Traceback (most recent call last):
  File "/home/moshe/eclipse/plugins/org.python.pydev.core_6.3.3.201805051638/pysrc/pydevd.py", line 1655, in <module>
    main()
  File "/home/moshe/eclipse/plugins/org.python.pydev.core_6.3.3.201805051638/pysrc/pydevd.py", line 1649, in main
    globals = debugger.run(setup['file'], None, None, is_module)
  File "/home/moshe/eclipse/plugins/org.python.pydev.core_6.3.3.201805051638/pysrc/pydevd.py", line 1055, in run
    pydev_imports.execfile(file, globals, locals) # execute the script
  File "/home/moshe/eclipse/plugins/org.python.pydev.core_6.3.3.201805051638/pysrc/_pydev_imps/_pydev_execfile.py", line 25, in execfile
    exec(compile(contents+"\n", file, 'exec'), glob, loc)
  File "/home/moshe/workspace/test_exceptions/main.py", line 27, in <module>
    main()
  File "/home/moshe/workspace/test_exceptions/main.py", line 21, in main
    sub(d)
  File "/home/moshe/workspace/test_exceptions/main.py", line 11, in sub
    e = f(d)/(d-1)
ZeroDivisionError: float division by zero
```

## Excepción

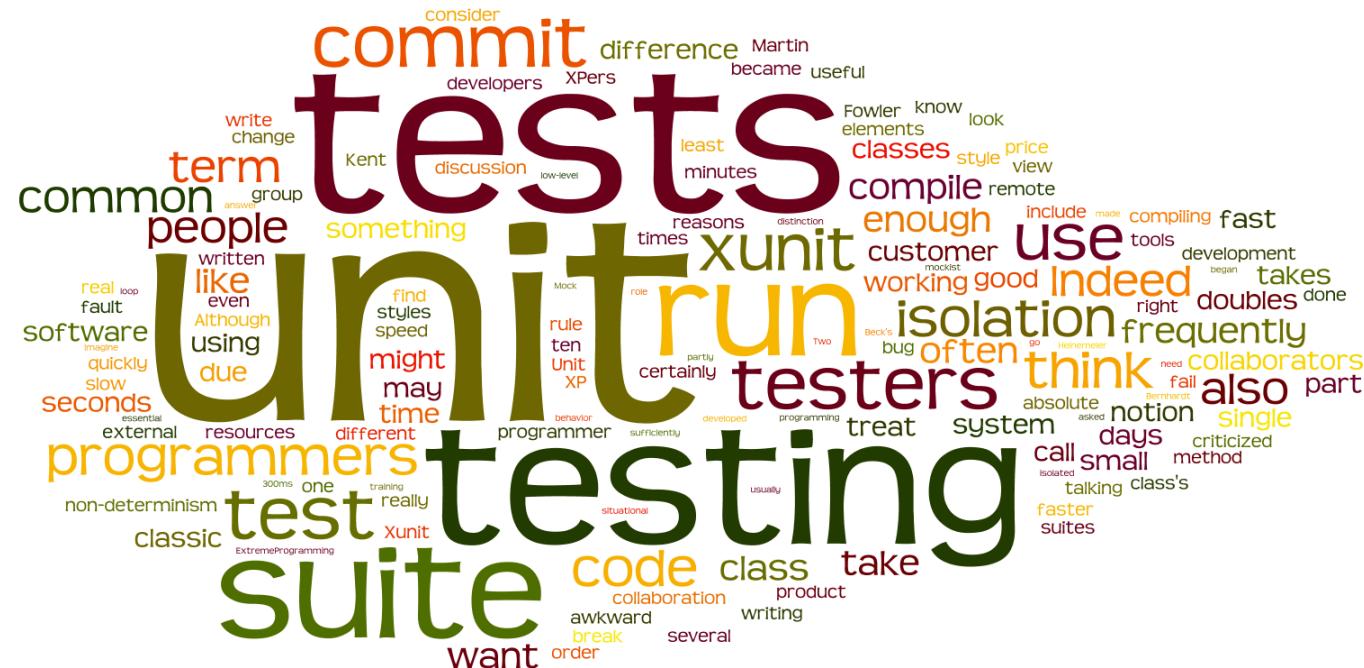
```
64      # Method to prepare a range of values 'min max'
65      def get_range_value_prepared(self, range_type):
66          value_range = ''
67          if self.left or self.right:
68              if self.left and self.right and (self.sign == 'in range'):
69                  if self.verify_type_range_value(self.left, self.right):
70                      if self.verify_range_value():
71                          result_left_value_convert = self.verify_value_converted(self.left, self.weight)
72                          result_right_value_convert = self.verify_value_converted(self.right, self.weight)
73                          value_range = str(result_left_value_convert) + ' ' + str(result_right_value_convert)
74                      else:
75                          raise RangeValueNotValidException(sms.range_value_not_valid.format(range_type))
76                  else:
77                      raise ValueTypeNotValidException(sms.value_type_not_valid.format(range_type))
78              elif self.left and not (self.sign == 'in range'):
79                  if self.verify_type_value(self.left):
80                      result_left_value_convert = self.verify_value_converted(self.left, self.weight)
81                      if self.sign == '>':
82                          value_range = str(result_left_value_convert) + ' ' + 'maxsize'
83                      elif self.sign == '<':
84                          value_range = '-1' + ' ' + str(result_left_value_convert)
85                      else:
86                          raise ValueTypeNotValidException(sms.value_type_not_valid.format(range_type))
87              elif (self.sign == 'in range') and (not self.left or not self.right):
88                  raise RangeValueEmptyValidException(sms.range_value_empty.format(range_type))
89          return value_range
90
```

## 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.

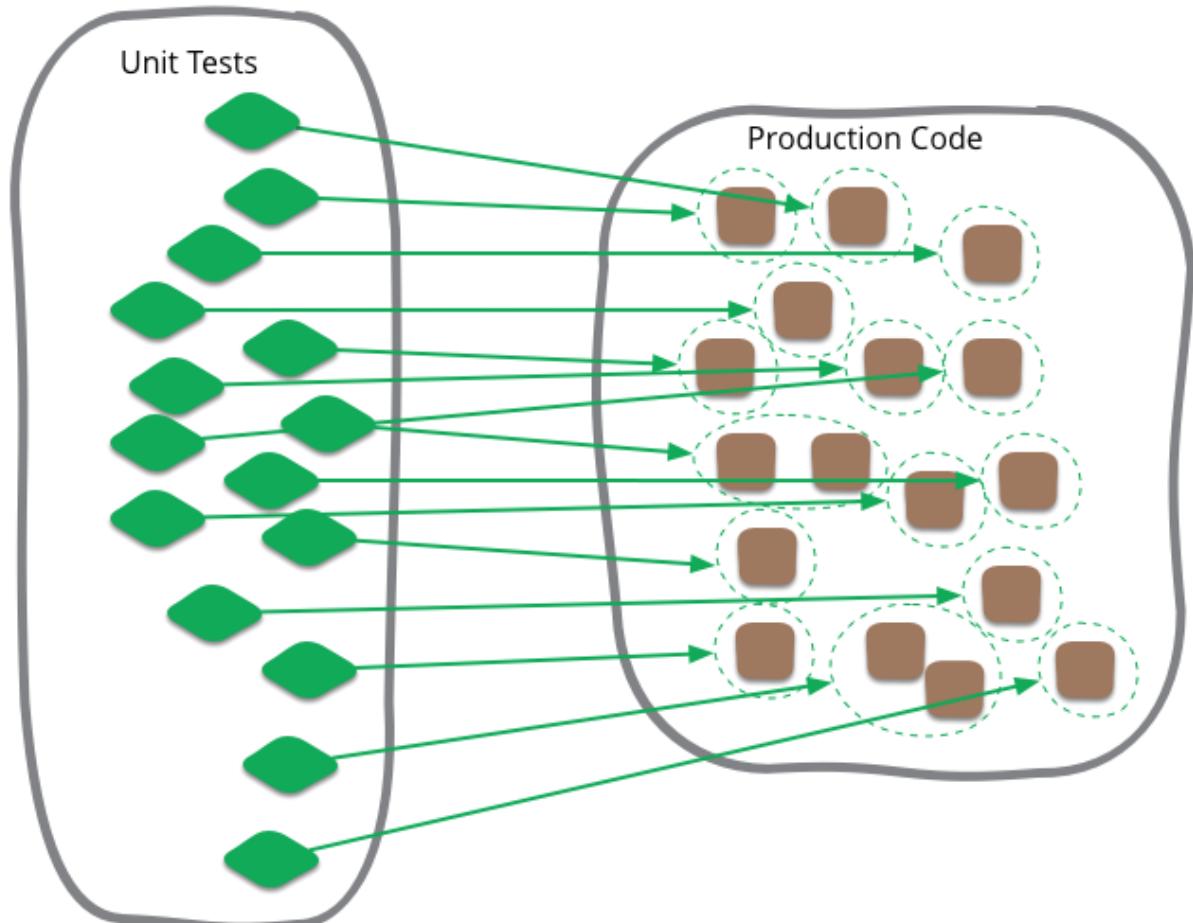


# Pruebas unitarias



### ¿Qué es una prueba unitaria?

- Es una prueba vinculada al código.
- Una clase cuyo métodos ejercitan los métodos de otra.
- Estas clases se integran en un proyecto diferente llamado pruebas(test).



### ¿Por qué probar las aplicaciones?

- Las pruebas no son el remedio de los errores, pero incrementan la calidad.
- Una prueba es un testigo de comprobación.
- Mucho más útiles según evoluciona el software.
- Debemos complementar con buenas prácticas y prácticas adquiridas



### El Módulo Unittest

El módulo unittest viene con la biblioteca estándar de Python. Proporciona una clase llamada `TestCase`, de la que se puede derivar su clase.

- **setUp()**: Método llamado para preparar el dispositivo de prueba. Esto se llama inmediatamente antes de llamar al método de prueba;
- **tearDown()**: Método llamado inmediatamente después de que se haya llamado al método de prueba y se haya registrado el resultado. Este método solo se llamará si `setUp ()` tiene éxito, independientemente del resultado del método de prueba.
- **setUpClass()**: Un método de clase llamado antes de que se ejecuten las pruebas en una clase individual. `setUpClass` se llama con la clase como único argumento y debe decorarse como un `classmethod()`
- **tearDownClass()**: Un método de clase llamado después de que se hayan ejecutado las pruebas en una clase individual. `tearDownClass` se llama con la clase como único argumento y debe decorarse como un `classmethod()`.

La clase TestCase proporciona varios métodos assert para verificar e informar fallas. La siguiente tabla lista los métodos más utilizados:

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assert IsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assert IsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

También existen otros métodos que se utilizan para realizar comprobaciones más específicas, como:

Method	Checks that	New in
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>	
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>	
<code>assertGreater(a, b)</code>	<code>a &gt; b</code>	3.1
<code>assertGreaterEqual(a, b)</code>	<code>a &gt;= b</code>	3.1
<code>assertLess(a, b)</code>	<code>a &lt; b</code>	3.1
<code>assertLessEqual(a, b)</code>	<code>a &lt;= b</code>	3.1
<code>assertRegex(s, r)</code>	<code>r.search(s)</code>	3.1
<code>assertNotRegex(s, r)</code>	<code>not r.search(s)</code>	3.2
<code>assertCountEqual(a, b)</code>	<code>a</code> and <code>b</code> have the same elements in the same number, regardless of their order.	3.2

La lista de métodos específicos de tipo utilizados automáticamente por `assertEqual()` se resume en la siguiente tabla. Tenga en cuenta que, por lo general, no es necesario invocar estos métodos directamente.

Method	Used to compare	New in
<code>assertMultiLineEqual(a, b)</code>	strings	3.1
<code>assertSequenceEqual(a, b)</code>	sequences	3.1
<code>assertListEqual(a, b)</code>	lists	3.1
<code>assertTupleEqual(a, b)</code>	tuples	3.1
<code>assertSetEqual(a, b)</code>	sets or frozensets	3.1
<code>assertDictEqual(a, b)</code>	dicts	3.1

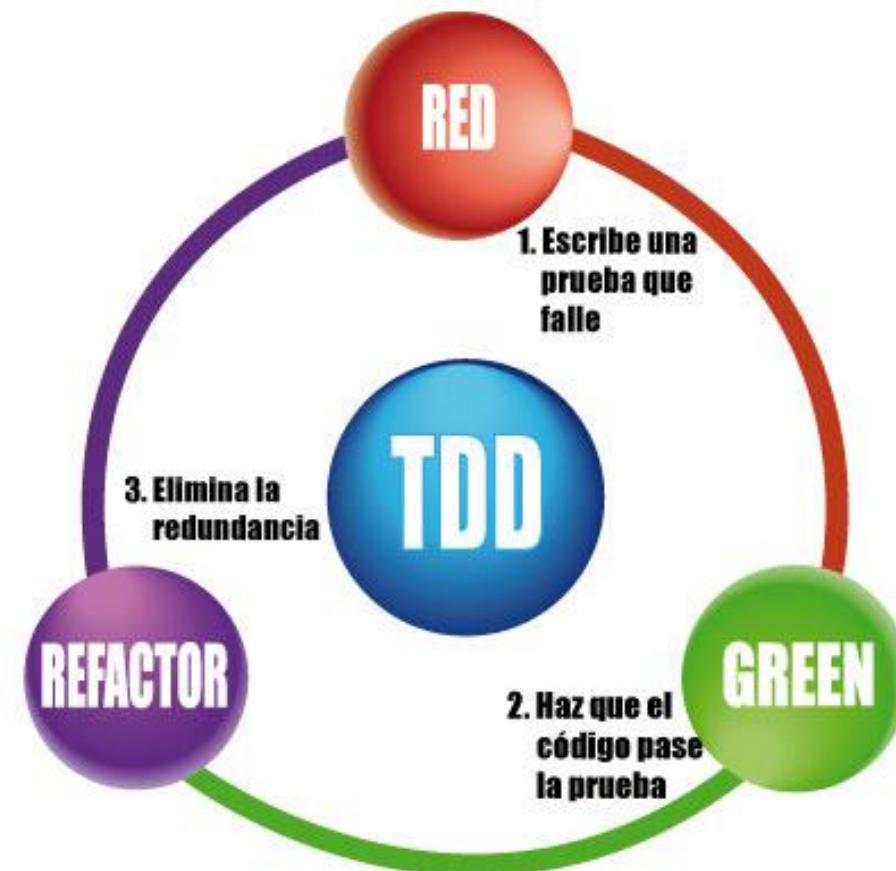
### Code Coverage:

El code coverage es una métrica utilizada en el desarrollo de software que determina el número de líneas de código que fueron ejecutadas durante nuestros test unitarios. Utilizando code coverage podemos determinar si nuestras aplicaciones se prueban de forma correcta y que porcentaje de escenarios no cuenta con pruebas automatizadas.



## ¿Qué es TDD?

- Test-driven development (TDD), Proceso iterativo en el cual el desarrollo esta guiado por test.
- Primero escribimos los test que expresan los requerimientos a cumplir luego desarrollamos para cumplir con dicho requerimientos



## **buenas practicas para las pruebas unitarias**

1. Una prueba unitaria no debe probar más que solo una cosa.
2. Ejecución rápida.
3. Resultado consistente (confiables).
4. Pruebas unitarias aisladas. la prueba unitaria no puede depender de otras pruebas.
5. Si falla debe ser fácilmente reconocible el fallo.
6. Repetible. Poder ejecutar las veces que fuese necesario y no requerir alguna configuración adicional.
7. Ejecutable por cualquier persona.
8. Veracidad de la prueba. El test debería asegurar que podría fallar si se cambia la lógica del objeto de prueba

## Pruebas Unitarias

- En las primeras dos líneas importamos el módulo unittest necesario para crear las pruebas unitarias y a continuación el propio módulo que queremos probar.
- Luego creamos la clase TestMyModule, una unidad de prueba que comprobara el comportamiento de nuestro modulo.
- Dentro de esta creamos tantos métodos que queramos probar.
- Todos los métodos que comiencen con el nombre **test** serán ejecutados.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def sum(a, b):
    for n in (a, b):
        if not isinstance(n, int) and not isinstance(n, float):
            raise TypeError
    return a + b

#!/usr/bin/env python
# -*- coding: utf-8 -*-

import unittest
import mymodule

class TestMyModule(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(mymodule.sum(5, 7), 12)

if __name__ == "__main__":
    unittest.main()
```

## Pruebas Unitarias

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def sum(a, b):
    for n in (a, b):
        if not isinstance(n, int) and not isinstance(n, float):
            raise TypeError
    return a + b

def get_prime_numbers(max_number):
    numbers = [True, True] + [True] * (max_number-1)
    last_prime_number = 2
    i = last_prime_number

    while last_prime_number**2 <= max_number:
        i += last_prime_number
        while i <= max_number:
            numbers[i] = False
            i += last_prime_number
        j = last_prime_number + 1
        while j < max_number:
            if numbers[j]:
                last_prime_number = j
                break
            j += 1
        i = last_prime_number

    return [i + 2 for i, not_crossed in enumerate(numbers[2:]) if not_crossed]

def is_prime(n):
    return n in get_prime_numbers(n)
```



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import unittest
import mymodule

class TestMyModule(unittest.TestCase):

    def test_get_prime_numbers(self):
        self.assertEqual(mymodule.get_prime_numbers(10), [2, 3, 5, 7])

    def test_is_prime(self):
        self.assertTrue(mymodule.is_prime(5))
        self.assertFalse(mymodule.is_prime(6))

    def test_sum(self):
        self.assertEqual(mymodule.sum(5, 7), 12)
        with self.assertRaises(TypeError):
            mymodule.sum(5, "Python")

if __name__ == "__main__":
    unittest.main()
```

# Patrones de Diseño

---



Se define los patrones como una solución ya probada a un problema en específico.

*“Un design pattern o patrón de diseño consiste en un diagrama de objetos que forma una solución a un problema conocido y frecuente”*

Laurent Debrauwer

## Categoría de Patrones

Según la escala o nivel de abstracción:

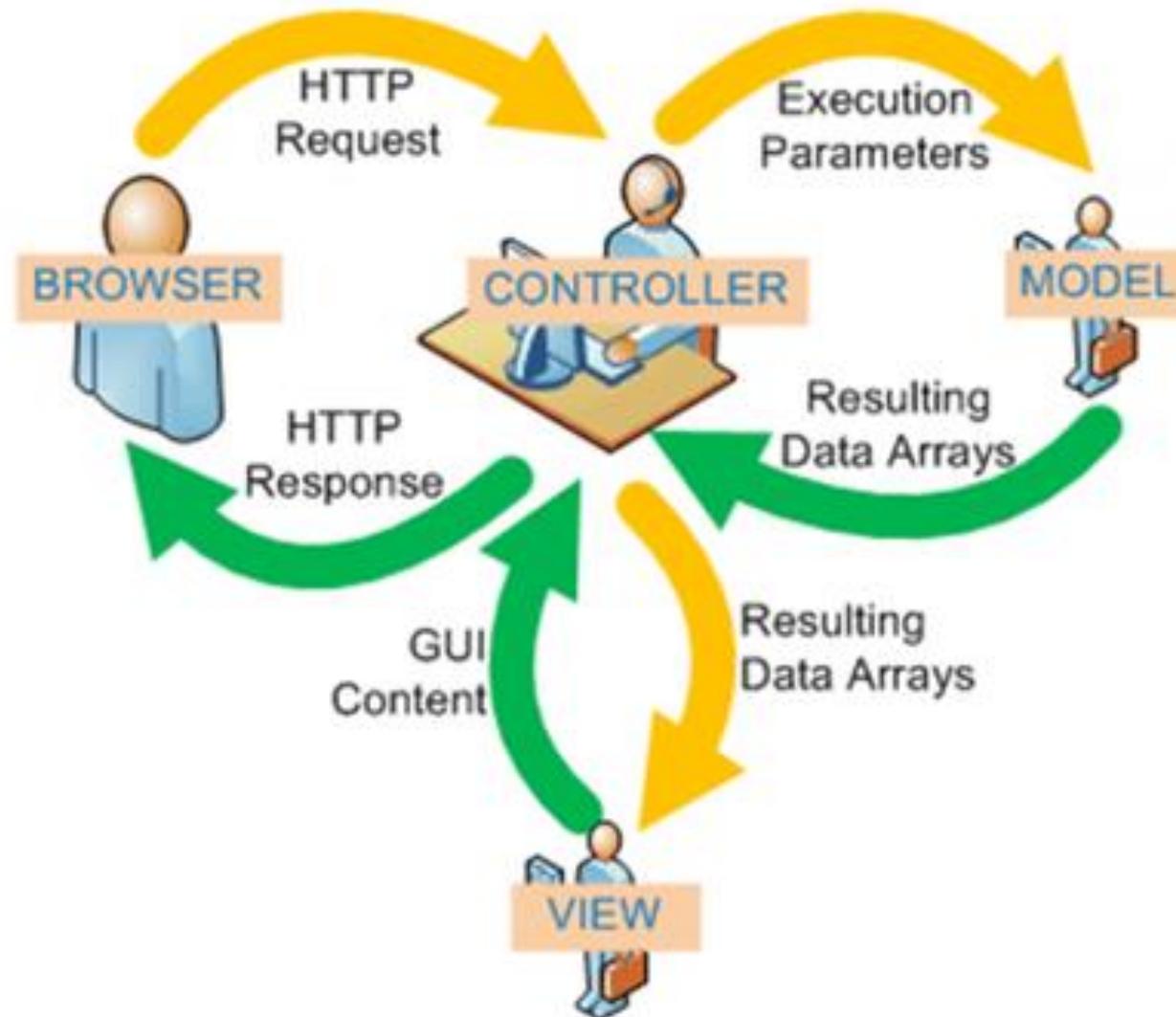
- **Patrones de arquitectura:** Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.
- **Patrones de diseño:** Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
- **Dialectos:** Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

## Patrones de arquitectura:

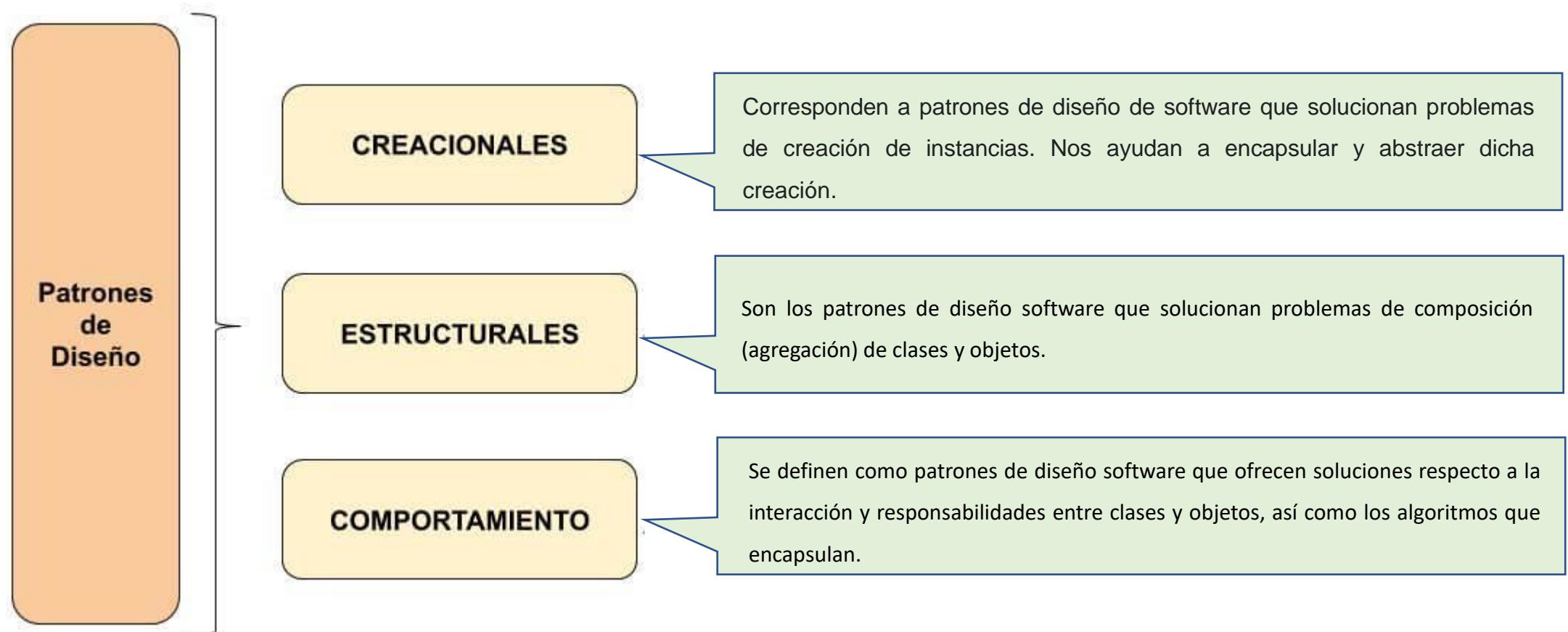
### Patrones de Arquitectura

- Programación por capas.
- Tres niveles.
- Arquitectura de microservicios.
- Arquitectura de microkernel.
- Pipeline.
- Invocación implícita.
- Arquitectura orientada a servicios
- **Modelo vista controlador**
- Peer to peer.

**Modelo Vista Controlador:**



### Tipos de Patrones de Diseño:



## Patrones de Diseño

### Patrones de Diseño

#### Creacionales

- Abstract Factory
- Factory Method
- Builder
- Singleton
- Prototype

#### Estructurales

- Adapter o Wrapper
- Bridge
- Composite
- Decorator
- Facade
- Proxy

#### Comportamiento

- Strategy
- Observer
- Command
- Chain of Responsibility
- State
- Iterator
- Visitor

# Patrones

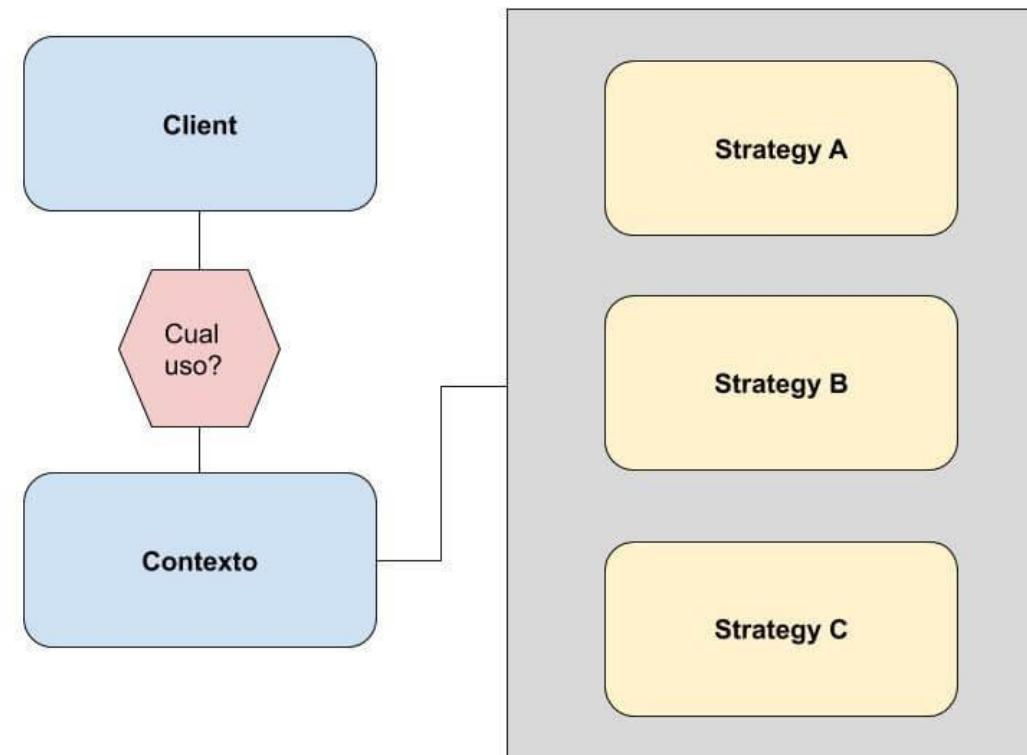
---



## Patrón de comportamiento - STRATEGY

El patrón de diseño Strategy en python ayuda a definir diferentes comportamientos o funcionalidades que pueden ser cambiadas en tiempo de ejecución.

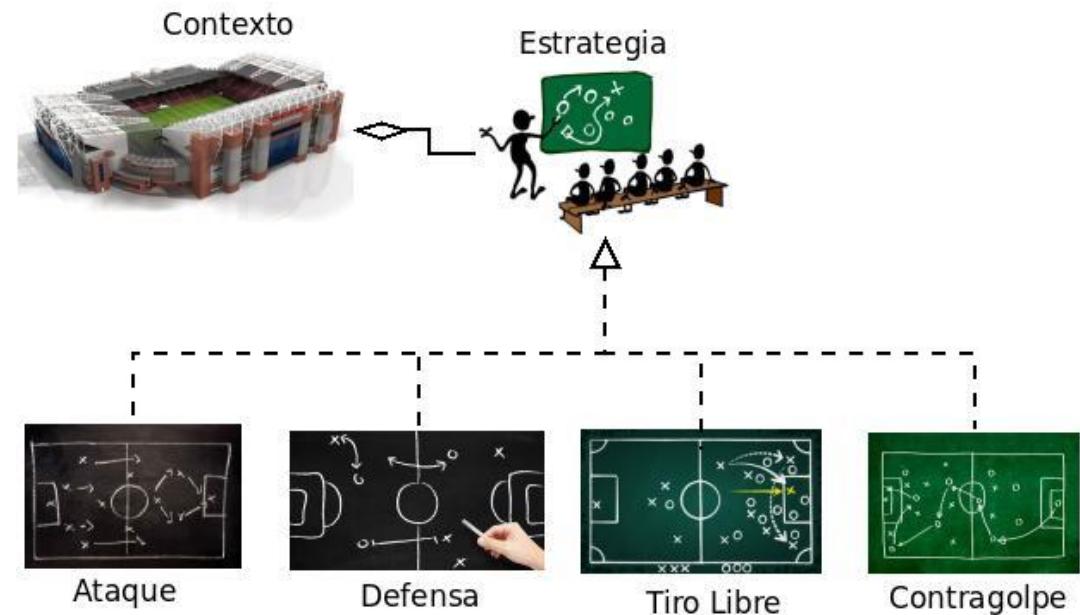
En el patrón Strategy creamos diferentes clases que representan estrategias y que podremos usar según alguna variación o input.



## Patrón de comportamiento – STRATEGY – componentes.

Los componentes del patrón Strategy son:

- **Interfaz Strategy:** es la interfaz que define cómo se conformará el contrato de la estrategia.
- **Clases Concretas Strategy:** son las clases que implementan la interfaz y donde se desarrolla la funcionalidad.
- **Contexto:** donde se establece que estrategia se usará.



## Patrón de comportamiento – STRATEGY

```
class Strategy(metaclass=abc.ABCMeta):
    """
    Declare an interface common to all supported algorithms. Context
    uses this interface to call the algorithm defined by a
    ConcreteStrategy.
    """

    @abc.abstractmethod
    def algorithm_interface(self):
        pass


class ConcreteStrategyA(Strategy):
    """
    Implement the algorithm using the Strategy interface.
    """

    def algorithm_interface(self):
        pass


class ConcreteStrategyB(Strategy):
    """
    Implement the algorithm using the Strategy interface.
    """

    def algorithm_interface(self):
        pass
```

## Patrón de comportamiento – STRATEGY

```
"""
Define a family of algorithms, encapsulate each one, and make them
interchangeable. Strategy lets the algorithm vary independently from
clients that use it.
"""

import abc

class Context:
    """
    Define the interface of interest to clients.
    Maintain a reference to a Strategy object.
    """

    def __init__(self, strategy):
        self._strategy = strategy

    def context_interface(self):
        self._strategy.algorithm_interface()
```

```
def main():
    concrete_strategy_a = ConcreteStrategyA()
    context = Context(concrete_strategy_a)
    context.context_interface()

if __name__ == "__main__":
    main()
```

# Patrones

---



## Patrón Adapter

## Patrón Estructural – ADAPTER

El patrón de diseño Adapter te sirve cuando tienes interfaces diferentes o incompatibles entre sí y necesitas que el cliente pueda usar ambas del mismo modo.

El patrón de diseño Adapter dice en su definición que convierte una interfaz o clase en otra interfaz que el cliente desea.

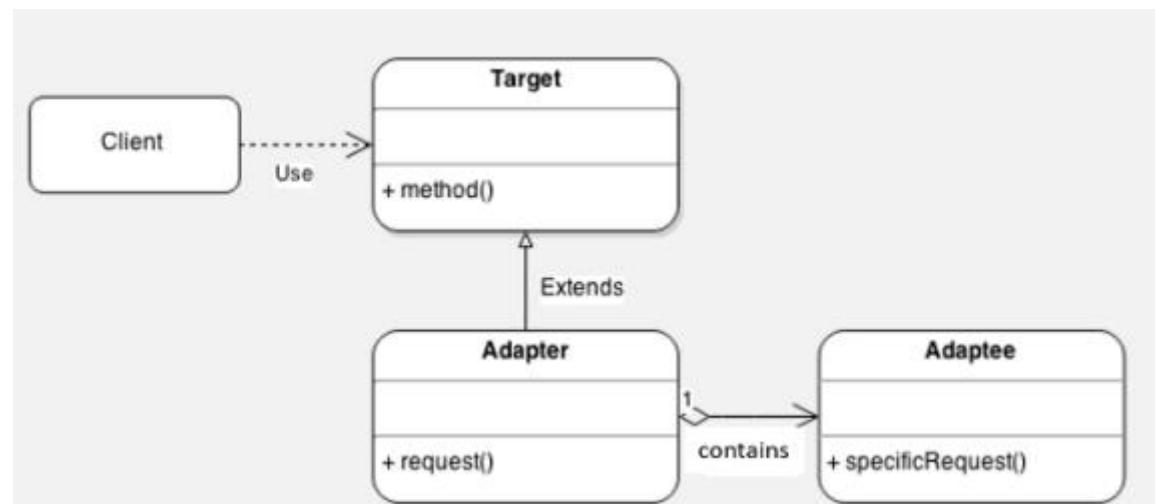


## Patrón Estructural – ADAPTER

### Partes del patrón de diseño Adapter

Las partes de patrón Adapter son:

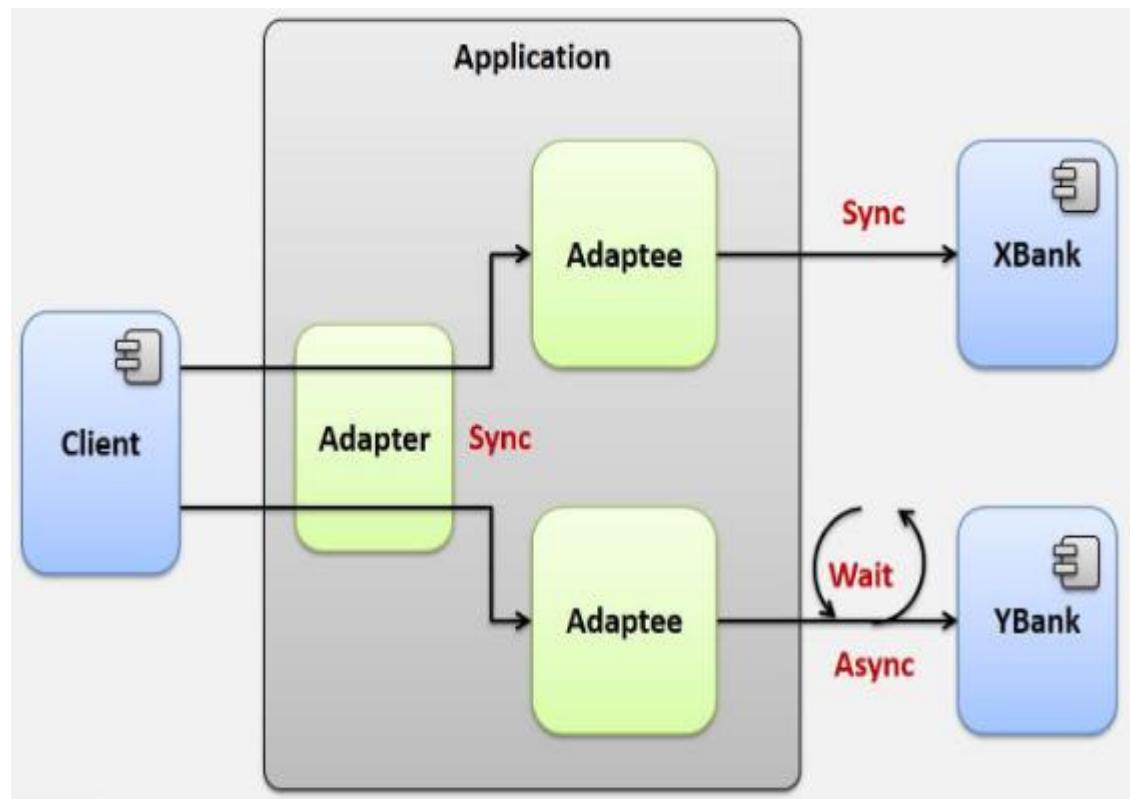
- **Target**: la interfaz que usamos para crear el adapter.
- **Adapter**: es la implementación del target y que se ocupará de realizar la adaptación.
- **Client**: es el que interactúa y usa el adapter.
- **Adaptee**: es la interfaz incompatible que necesitamos adaptar con el adapter.



## Patrón Estructural – ADAPTER

Este patrón se debe utilizar cuando:

- Se quiere utilizar una clase que llame a un método a través de una interface, pero se busca utilizarlo con una clase que no implementa ese interface.
- Se busca determinar dinámicamente qué métodos de otros objetos llama un objeto.
- No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.



## Patrón Estructural – ADAPTER

```
class EuropeanSocketInterface:  
    def voltage(self): pass  
  
    def live(self): pass  
    def neutral(self): pass  
    def earth(self): pass  
  
# Adaptee  
class Socket(EuropeanSocketInterface):  
    def voltage(self):  
        return 230  
  
        def live(self):  
            return 1  
  
    def neutral(self):  
        return -1  
  
    def earth(self):  
        return 0
```

```
# Target interface  
class USASocketInterface:  
    def voltage(self): pass  
    def live(self): pass  
    def neutral(self): pass  
  
# The Adapter  
class Adapter(USASocketInterface):  
    __socket = None  
    def __init__(self, socket):  
        self.__socket = socket  
  
    def voltage(self):  
        return 110  
  
    def live(self):  
        return self.__socket.live()  
  
    def neutral(self):  
        return self.__socket.neutral()
```

## Patrón Estructural – ADAPTER

```
# Client
class ElectricKettle:
    __power = None

    def __init__(self, power):
        self.__power = power

    def boil(self):
        if self.__power.voltage() > 110:
            print "Kettle on fire!"
        else:
            if self.__power.live() == 1 and \
                self.__power.neutral() == -1:
                print "Coffee time!"
            else:
                print "No power."

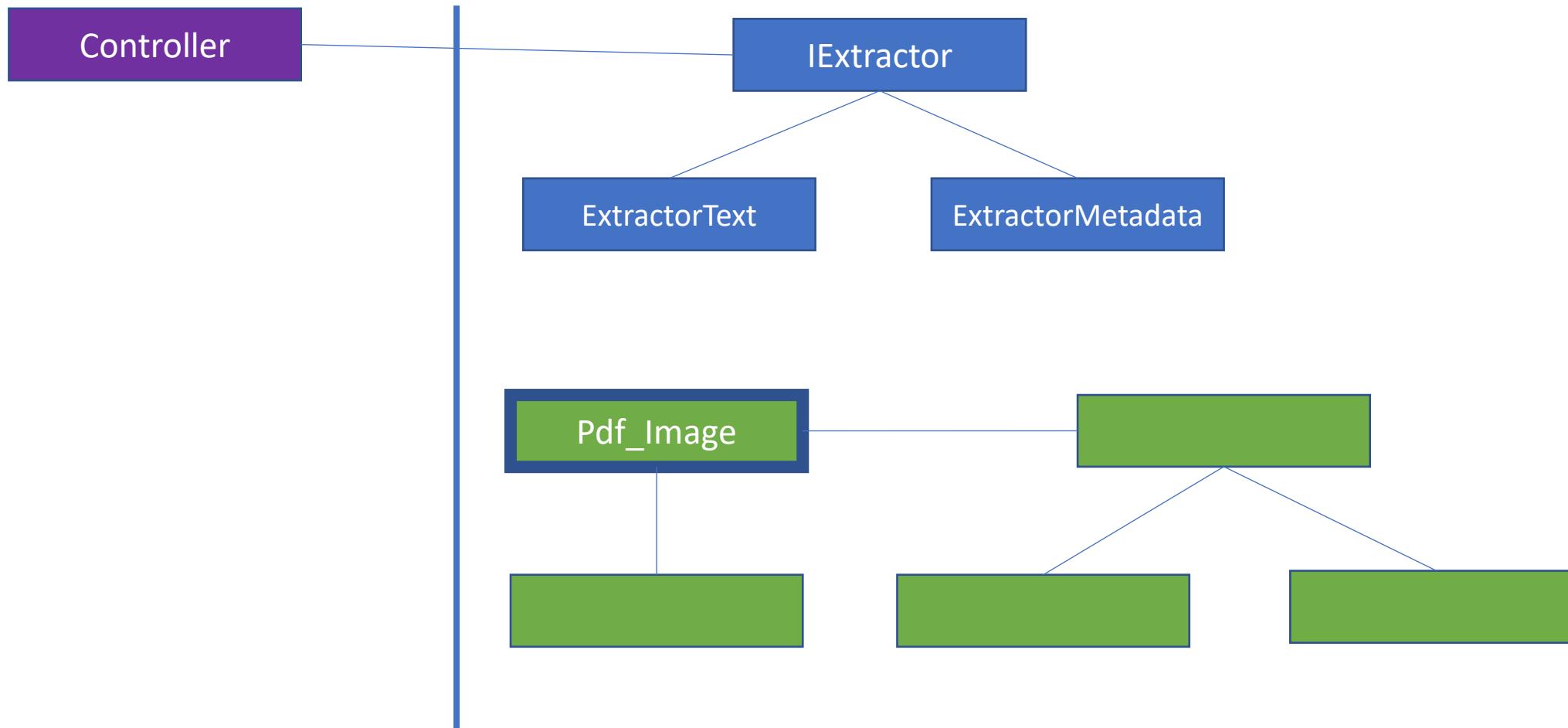
def main():
    # Plug in
    socket = Socket()
    adapter = Adapter(socket)
    kettle = ElectricKettle(adapter)

    # Make coffee
    kettle.boil()

return 0
```

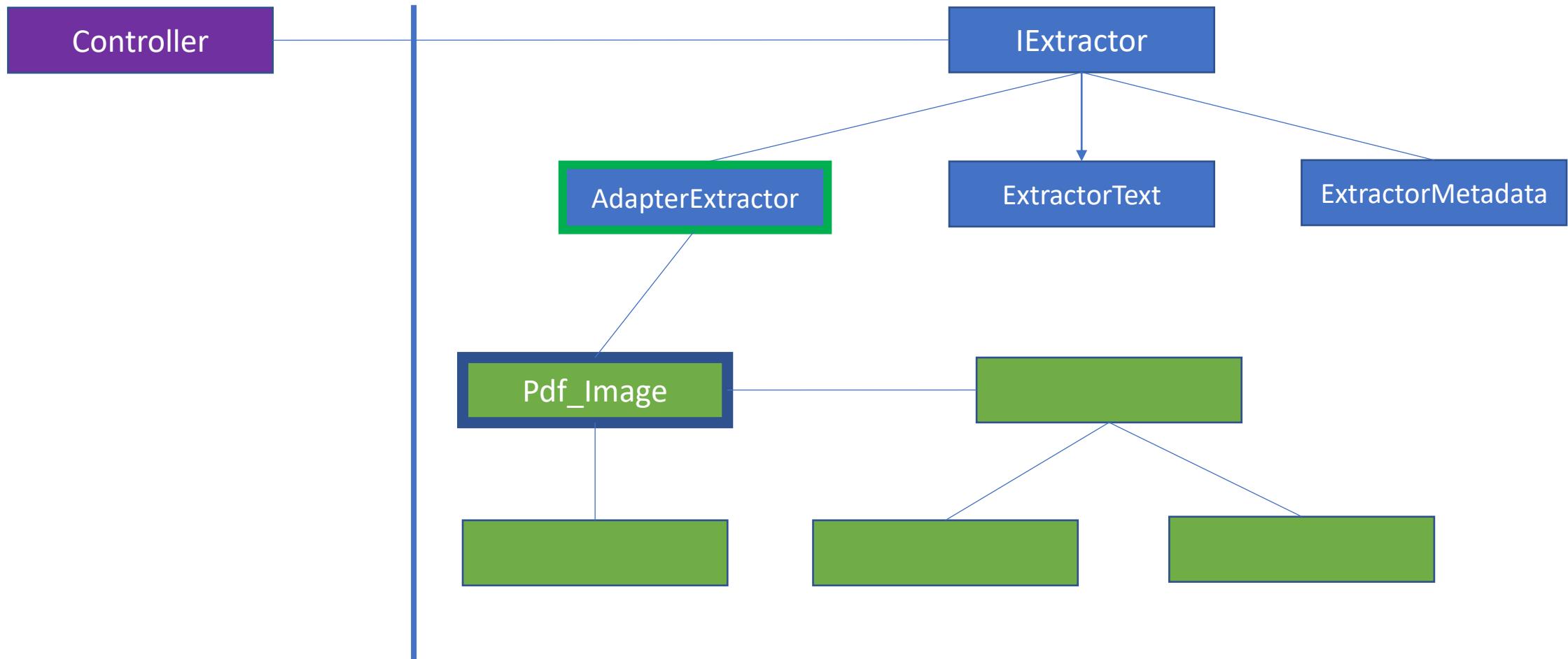
Patrones

## Patrón Estructural – ADAPTER



Patrones

## Patrón Estructural – ADAPTER



# Patrones

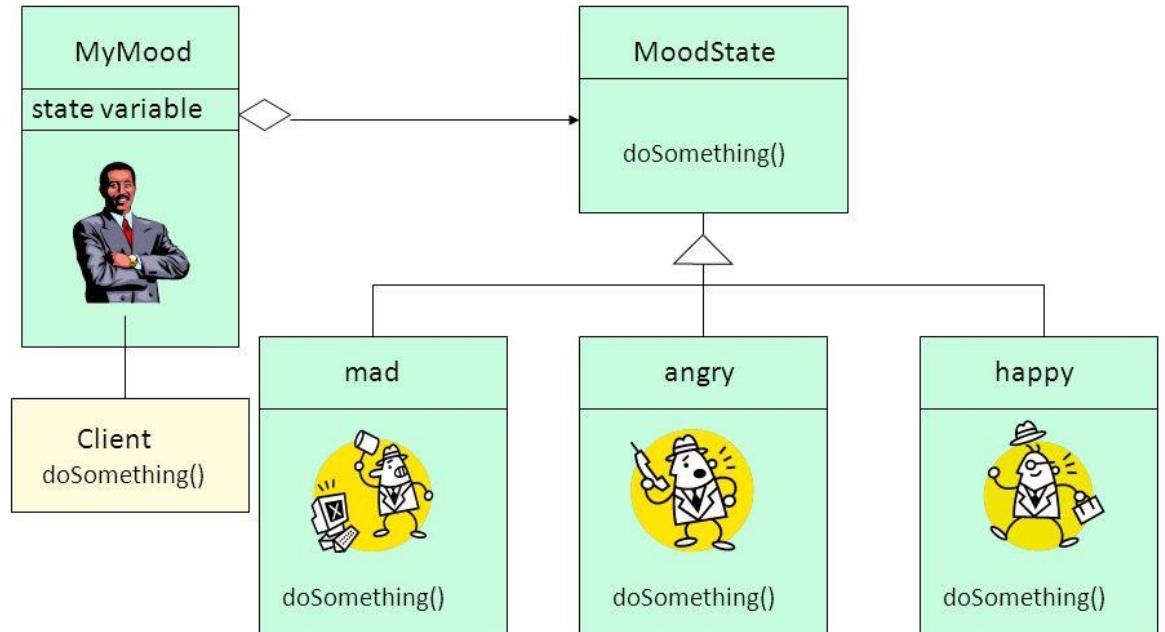
---



## Patrón de comportamiento – state

Propósito:

- Permitir a un objeto modificar su comportamiento cuando su estado interno cambia. El objeto aparecerá para cambiar su clase.



## Patrón de comportamiento – state

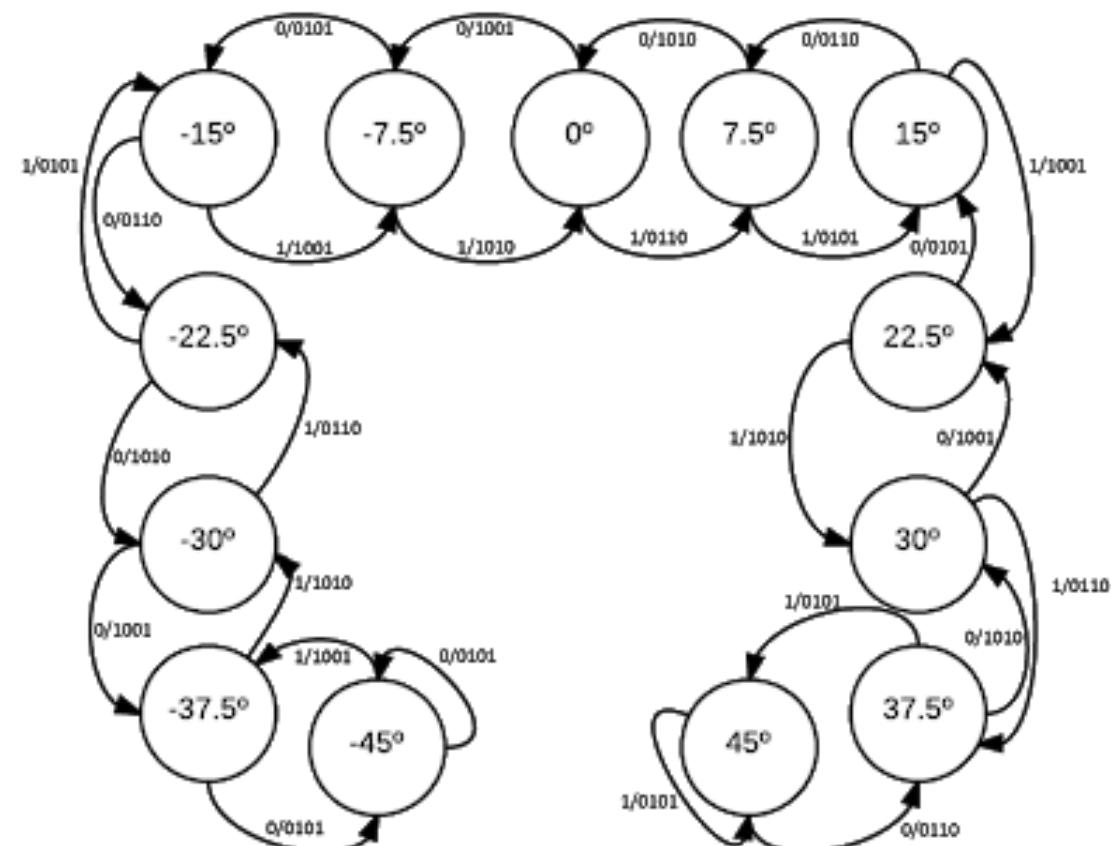
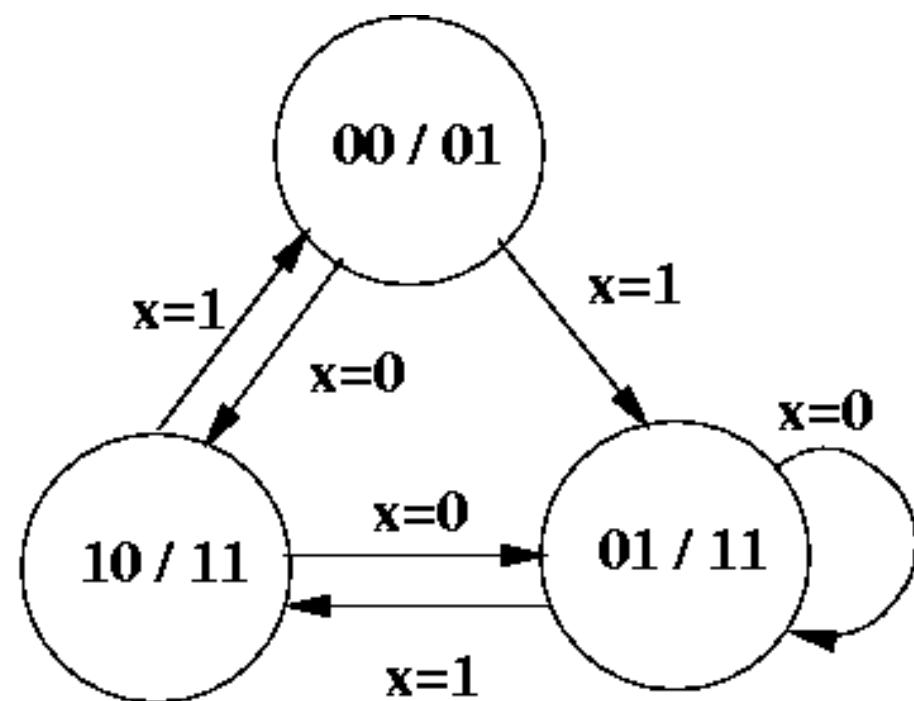
### Aplicabilidad:

El patrón State se usa en cualquiera de estos casos:

- El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen muchos condicionales que dependen del estado del objeto. Este estado está representado normalmente por uno o más enumerados. A menudo, varias operaciones contendrán la misma estructura condicional. El patrón State pone cada rama del condicional en una clase separada. Esto permite tratar el estado del objeto que puede variar independientemente de otros objetos.

Más concretamente, el patrón State suele usarse para implementar máquinas de estados

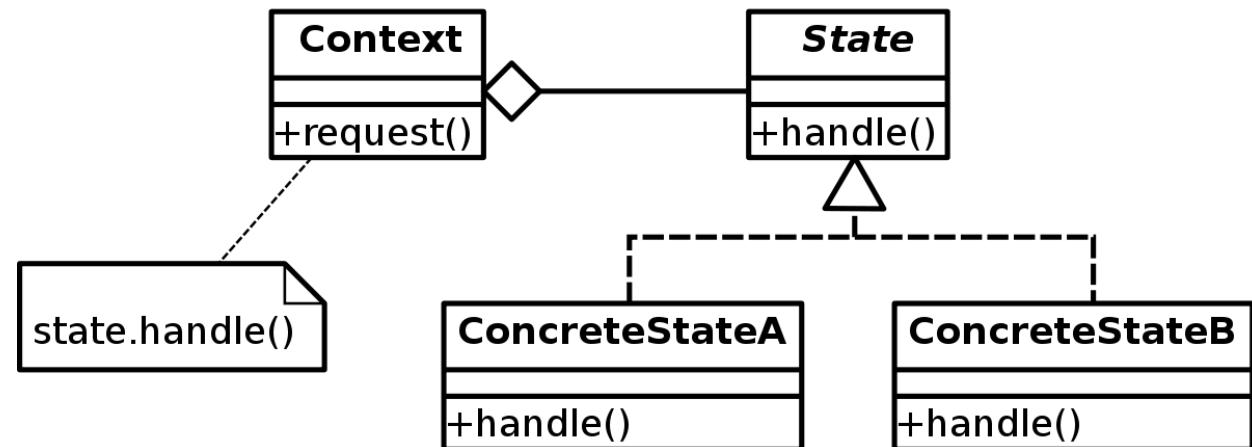
## Patrón de comportamiento – state



## Patrón de comportamiento – state

### Estructura:

La clase **Contexto** define la interfaz de interés para los clientes. La clase **State** define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto. Cada clase **ConcreteState** implementa un comportamiento asociado con un estado el contexto.



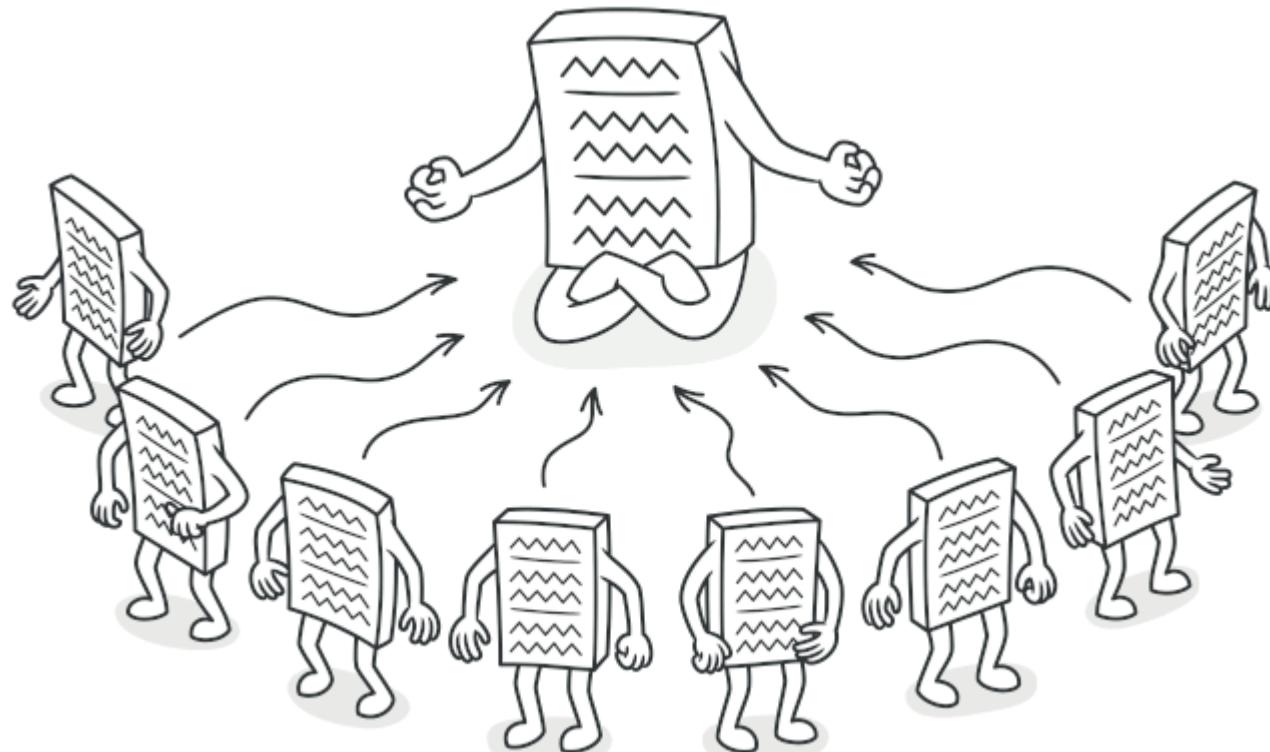
## Patrón de comportamiento – state

### Ventajas:

- **Localiza el comportamiento de un estado específico y divide el comportamiento para diferentes estados:** El patrón State pone todo el comportamiento asociado con un estado particular en un objeto. Debido a que todo el código de un estado específico está en una subclase de State, los nuevos estados y transiciones pueden ser añadidos fácilmente añadiendo nuevas subclases.
- **Hace las transiciones entre estados explícitas:** Cuando un objeto define su estado actual únicamente en términos de datos internos, sus transiciones no tienen una representación explícita. Introducir objetos separados para diferentes estados hace las transiciones más explícitas.
- **Los objetos de los estados pueden ser compartidos:** Si los objetos de los estados no tienen variables instanciadas, el estado que representan esta codificado completamente en su tipo, entonces el contexto puede compartir el objeto del estado.

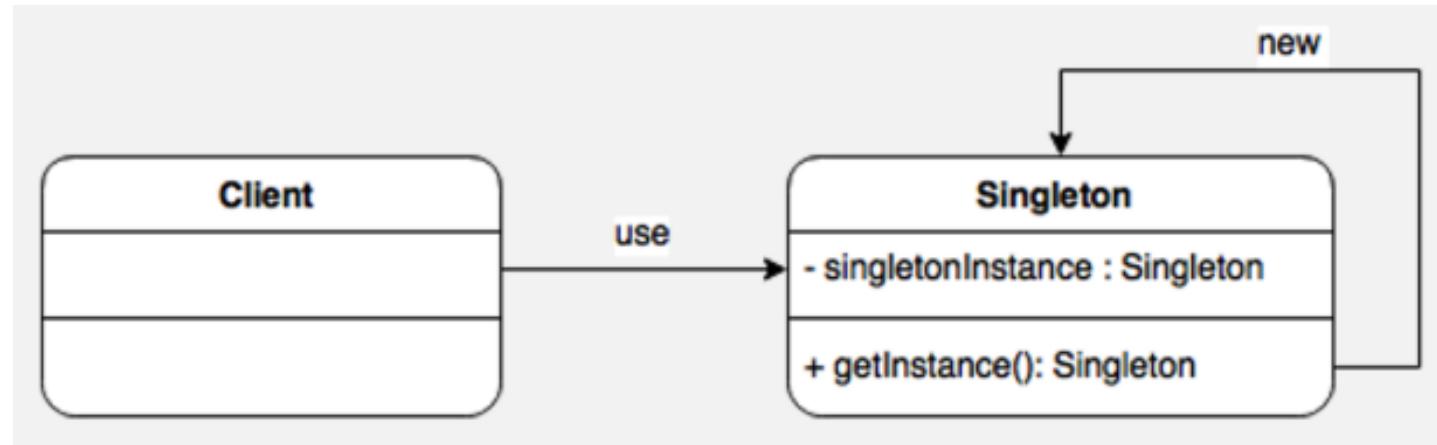
# Patron - Singleton

---



## Patrón de creación – SINGLETON

El patrón singleton es un patrón de creación (**Creational Design Pattern**) que tiene como objetivo garantizar que una clase sólo tenga una instancia y proporcionar un único punto de acceso a esa instancia. Este patrón suele ser utilizado en temas de logging (entre otros) donde solo queremos tener una una instancia del logger que centralice todo.

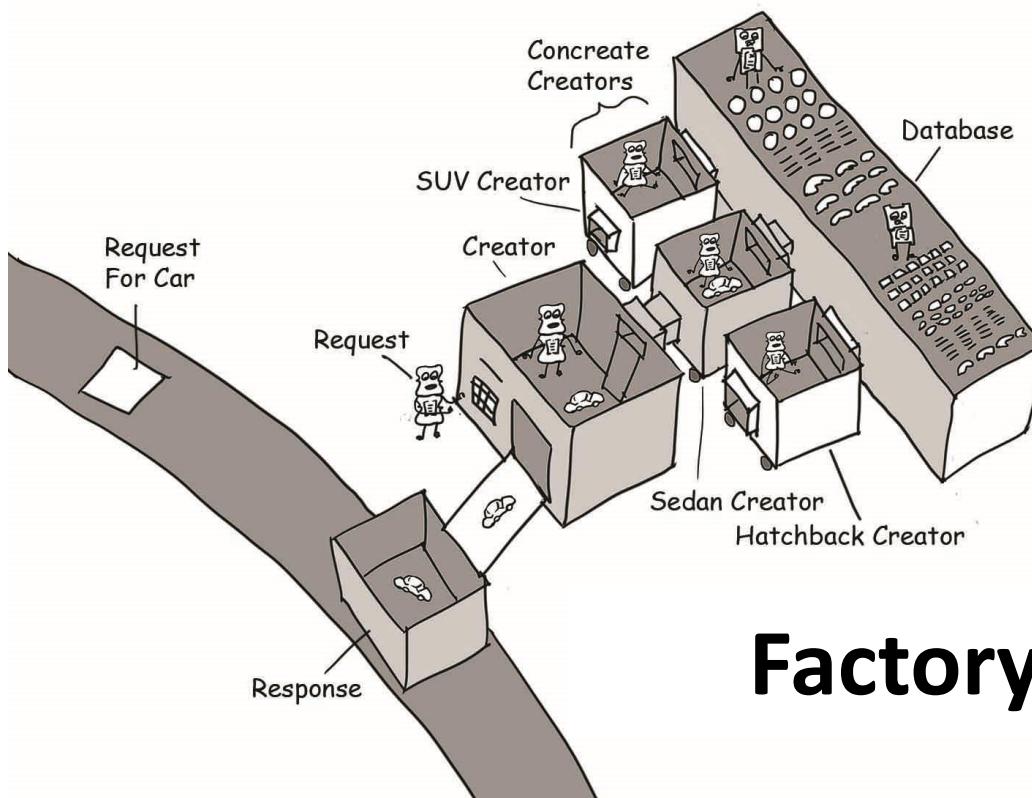


## Patrón de creación – SINGLETON

```
1  class Tool(object):
2      __instance = None
3      __firstset = False
4
5  ⚡ def __new__(cls, *args, **kwargs):
6      if cls.__instance is None:
7          cls.__instance = object.__new__(cls)
8      return cls.__instance
9
10 def __init__(self, name):
11     if self.__firstset is False:
12         self.__class__.__firstset = True
13     self.name = name
14
```

# Patrones

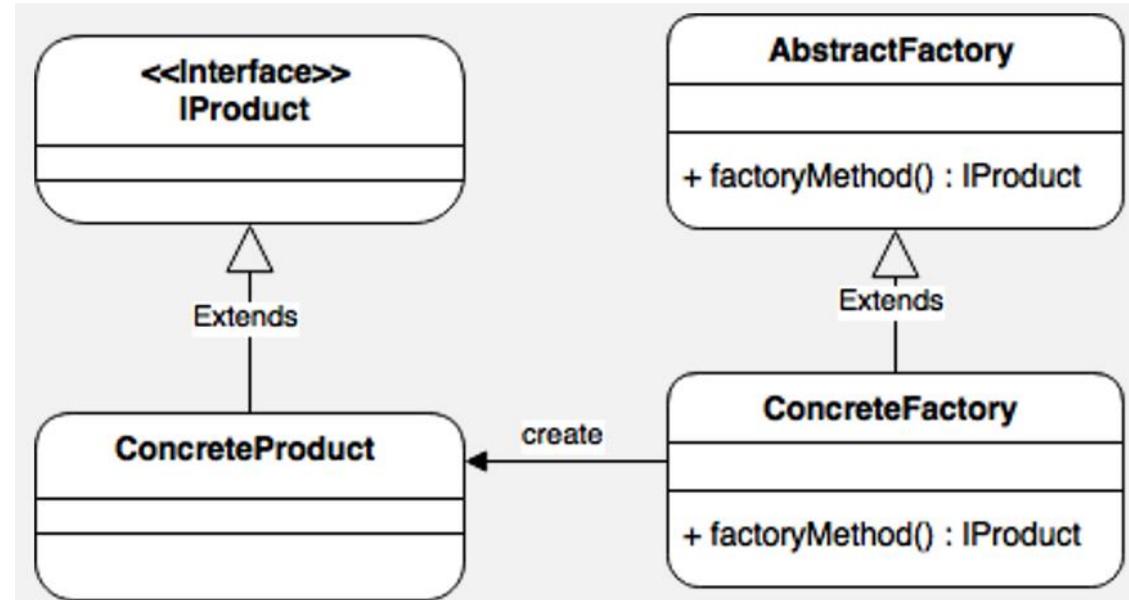
---



## Factory Method

## Patrón de creación – Factory method

- Es un patrón de diseño que define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
- Permite que una clase delegue en sus subclases la creación de objetos.
- Permite escribir aplicaciones que son más flexibles respecto de los tipos a utilizar.
- Permite también encapsular el conocimiento referente a la creación de objetos.

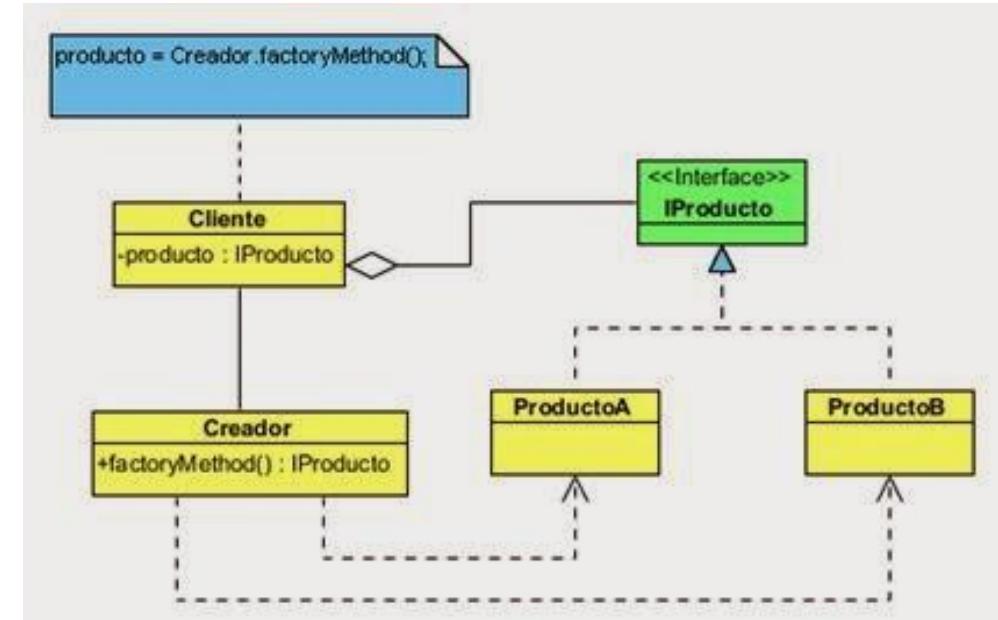
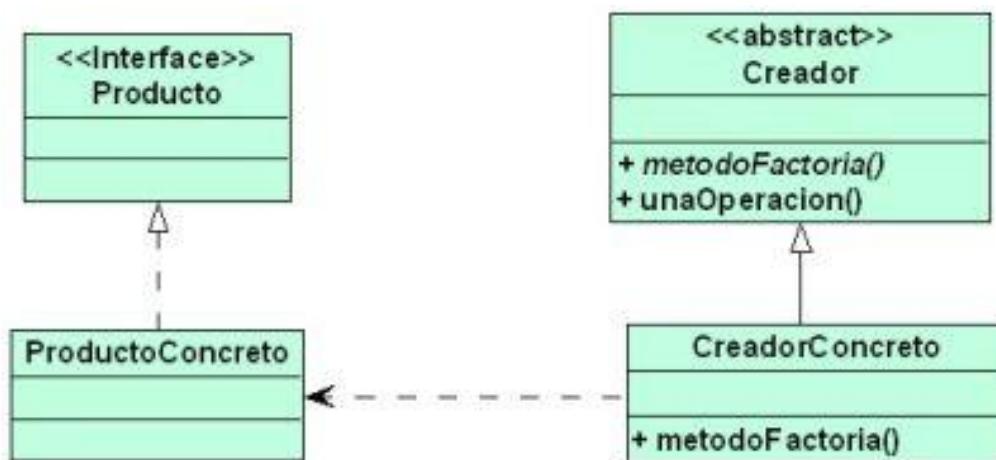


## Patrón de creación – Factory method

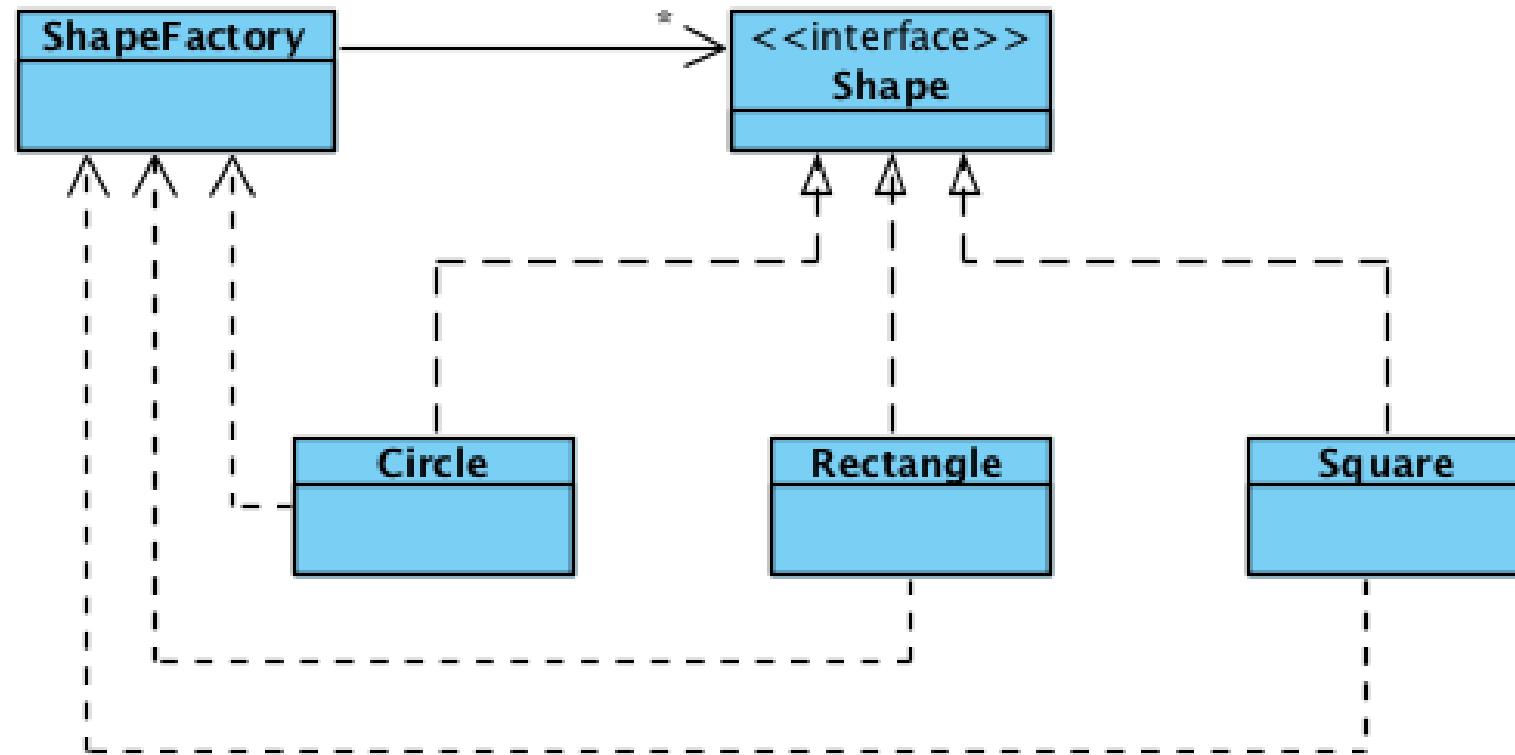
### Participantes:

- **Producto:** Define la interfaz de los objetos que crea el método de fabricación.
- **ProductoConcreto:** Implementa la interfaz Producto.
- **Creador:** Declara el método de fabricación, el cual devuelve un objeto del tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelve un objeto ProductoConcreto. Puede llamar al método de fabricación para crear un objeto Producto.
- **CreadorConcreto:** Redefine el método de fabricación para devolver una instancia de ProductoConcreto.

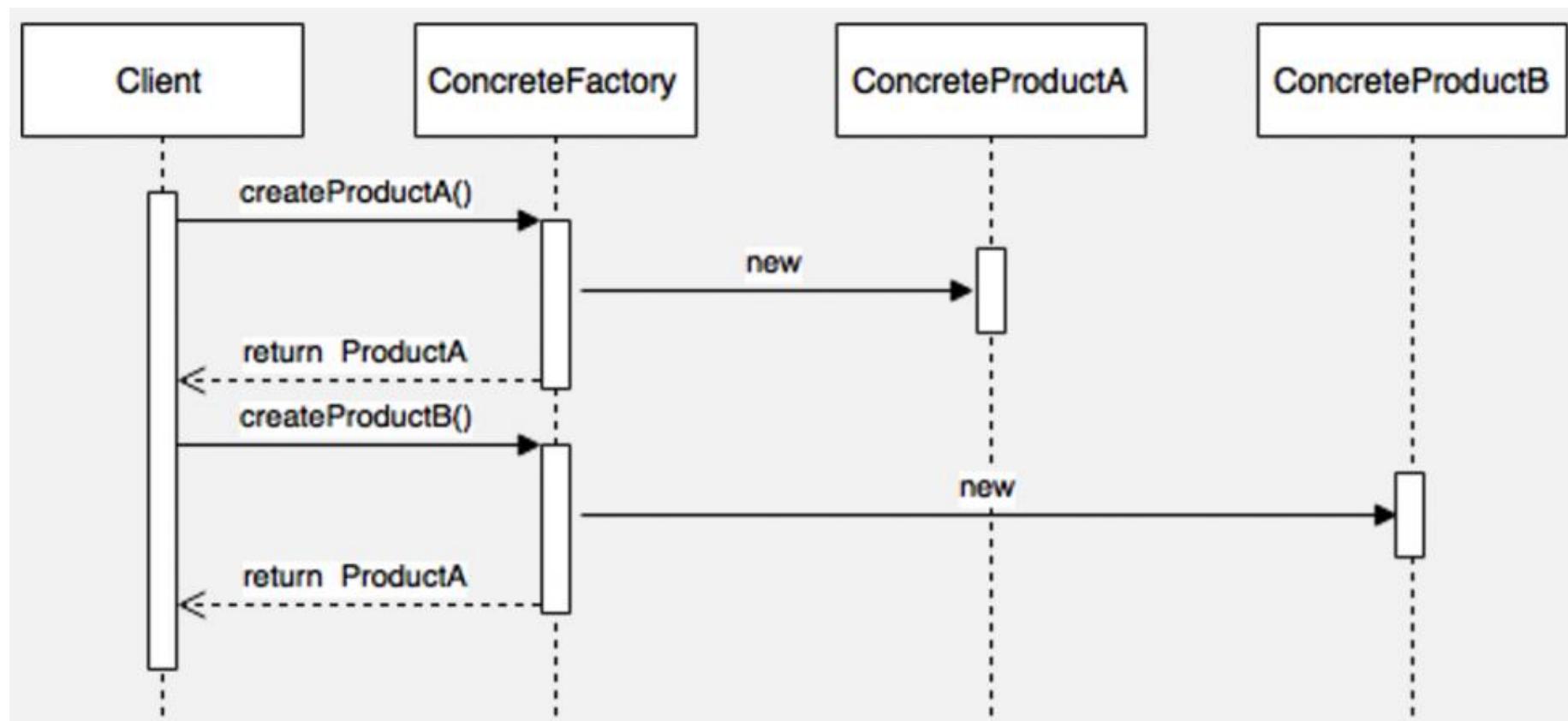
## Patrón de creación – Factory method



## Patrón de creación – Factory method



## Patrón de creación – Factory method



## Patrón de creación – Factory method

### Debe Usarse:

- Cuando una clase no puede adelantar las clases de objetos que debe crear.
- Cuando una clase pretende que sus subclases especifiquen los objetos que ella crea.
- Cuando una clase delega su responsabilidad hacia una de entre varias subclases auxiliares y queremos tener localizada a la subclase delegada.

## Patrón de creación – Factory method

### Ventajas

- **Se gana en flexibilidad**, pues crear los objetos dentro de una clase con un "Método de Fábrica" es siempre más flexible que hacerlo directamente, debido a que se elimina la necesidad de atar nuestra aplicación unas clases de productos concretos.
- **Se facilitan futuras ampliaciones**, puesto que se ofrece las subclases la posibilidad de proporcionar una versión extendida de un objeto, con sólo aplicar en los Productos la misma idea del "Método de Fábrica".

## Patrón de creación – Factory method

```
import abc
class Shape(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def calculate_area(self):
        pass

    @abc.abstractmethod
    def calculate_perimeter(self):
        pass
```

```
class Rectangle(Shape):
    def __init__(self, height, width):
        self.height = height
        self.width = width

    def calculate_area(self):
        return self.height * self.width

    def calculate_perimeter(self):
        return 2 * (self.height + self.width)

class Square(Shape):
    def __init__(self, width):
        self.width = width

    def calculate_area(self):
        return self.width ** 2

    def calculate_perimeter(self):
        return 4 * self.width

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius * self.radius

    def calculate_perimeter(self):
        return 2 * 3.14 * self.radius
```

## Patrón de creación – Factory method

```
class ShapeFactory:  
    def create_shape(self, name):  
        if name == 'circle':  
            radius = input("Enter the radius of the circle: ")  
            return Circle(float(radius))  
  
        elif name == 'rectangle':  
            height = input("Enter the height of the rectangle: ")  
            width = input("Enter the width of the rectangle: ")  
            return Rectangle(int(height), int(width))  
  
        elif name == 'square':  
            width = input("Enter the width of the square: ")  
            return Square(int(width))
```

```
def shapes_client():  
    shape_factory = ShapeFactory()  
    shape_name = input("Enter the name of the shape: ")  
  
    shape = shape_factory.create_shape(shape_name)  
  
    print(f"The type of object created: {type(shape)}")  
    print(f"The area of the {shape_name} is: {shape.calculate_area()}")  
    print(f"The perimeter of the {shape_name} is: {shape.calculate_perimeter()}")
```

# Patrones

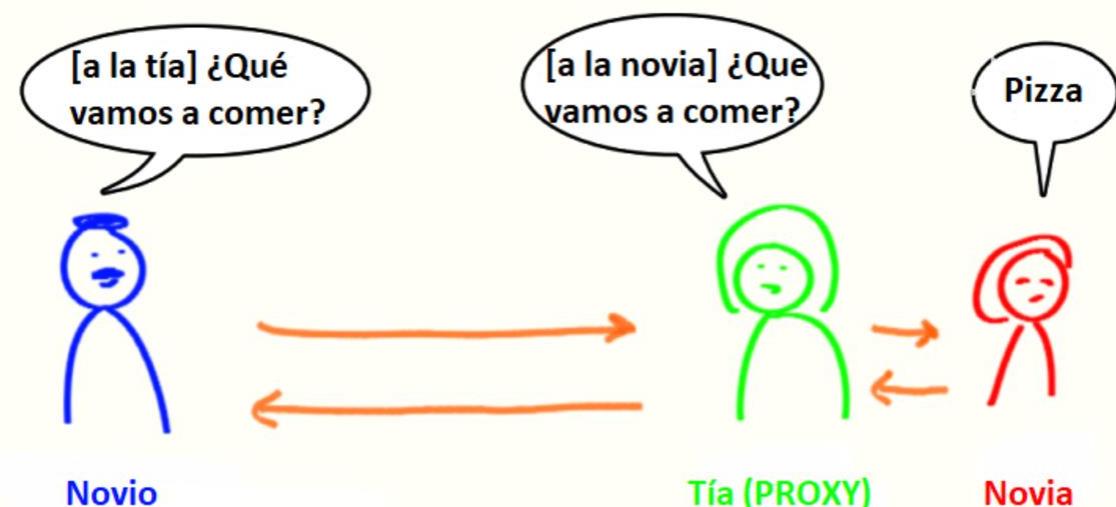
---



## Patrón Estructural – Proxy

El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

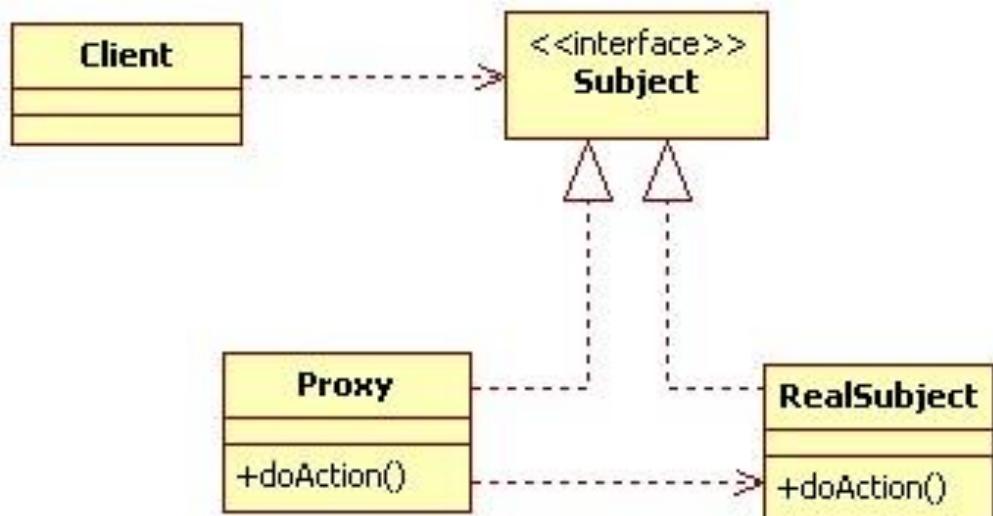
Para ello obliga que las llamadas a un objeto ocurran indirectamente a través de un objeto proxy, que actúa como un sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.



## Patrón Estructural – Proxy

### Participantes:

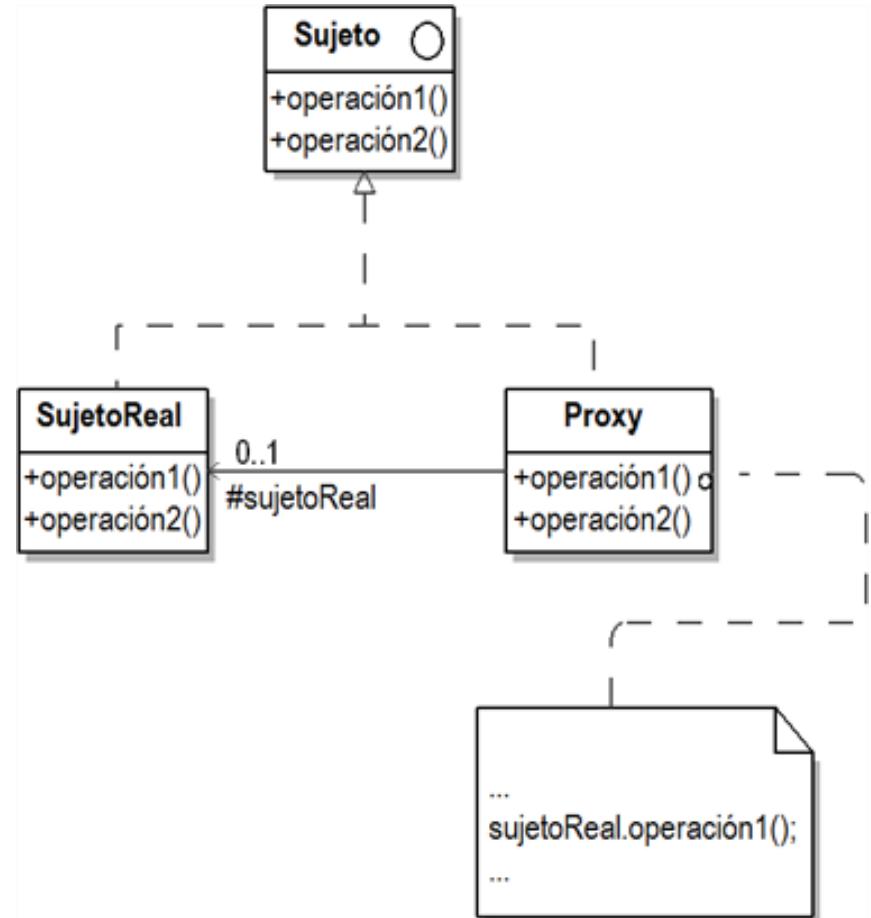
- **Subject:** interfaz o clase abstracta que proporciona un acceso común al objeto real y su representante (proxy).
- **Proxy:** mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.
- **RealSubject:** define el objeto real representado por el Proxy.
- **Client:** solicita el servicio a través del Proxy y es éste quién se comunica con el RealSubject.



## Patrón Estructural – Proxy

### Ventajas y desventajas de este patrón:

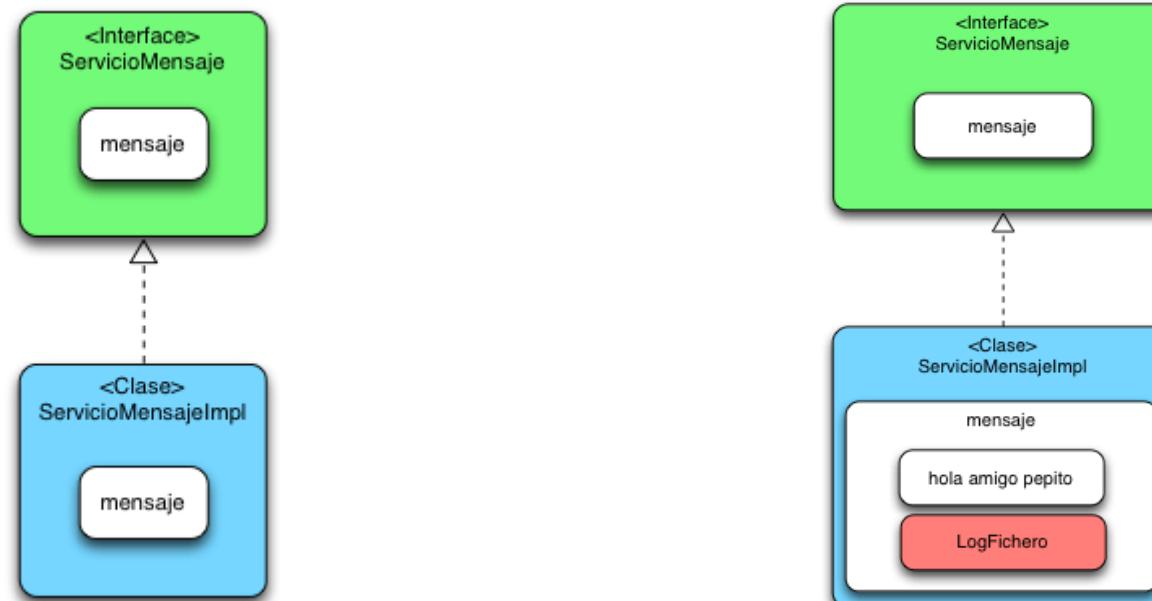
- Podemos modificar funcionalidad sin afectar a los clientes.
- El código se puede volver complicado si es necesario introducir gran cantidad de funcionalidad nueva.
- La respuesta real del servicio puede ser desconocida hasta que realmente se la invoca.



## Patrón Estructural – Proxy

### Ejemplo:

Vamos a suponer que tenemos una clase de servicio con un método que nos devuelve el mensaje “**hola amigo pepito**”, recibiendo como parámetro el nombre que deseemos. Para ello construiremos una interface y una clase que la implemente.



## Patrón Estructural – Proxy

### Ejemplo:

```
from abc import ABC, abstractmethod

class Subject(ABC):
    """
    The Subject interface declares common operations for both RealSubject and
    the Proxy. As long as the client works with RealSubject using this
    interface, you'll be able to pass it a proxy instead of a real subject.
    """

    @abstractmethod
    def request(self) -> None:
        pass

class RealSubject(Subject):
    """
    The RealSubject contains some core business logic. Usually, RealSubjects are
    capable of doing some useful work which may also be very slow or sensitive -
    e.g. correcting input data. A Proxy can solve these issues without any
    changes to the RealSubject's code.
    """

    def request(self) -> None:
        print("RealSubject: Handling request.")
```

```
class Proxy(Subject):
    """
    The Proxy has an interface identical to the RealSubject.
    """

    def __init__(self, real_subject: RealSubject) -> None:
        self._real_subject = real_subject

    def request(self) -> None:
        """
        The most common applications of the Proxy pattern are lazy loading,
        caching, controlling the access, logging, etc. A Proxy can perform one
        of these things and then, depending on the result, pass the execution to
        the same method in a linked RealSubject object.
        """

        if self.check_access():
            self._real_subject.request()
            self.log_access()

    def check_access(self) -> bool:
        print("Proxy: Checking access prior to firing a real request.")
        return True

    def log_access(self) -> None:
        print("Proxy: Logging the time of request.", end="")
```

## Patrón Estructural – Proxy

### Ejemplo:

```
def client_code(subject: Subject) -> None:
    """
    The client code is supposed to work with all objects (both subjects and
    proxies) via the Subject interface in order to support both real subjects
    and proxies. In real life, however, clients mostly work with their real
    subjects directly. In this case, to implement the pattern more easily, you
    can extend your proxy from the real subject's class.
    """

    # ...
    subject.request()
    # ...

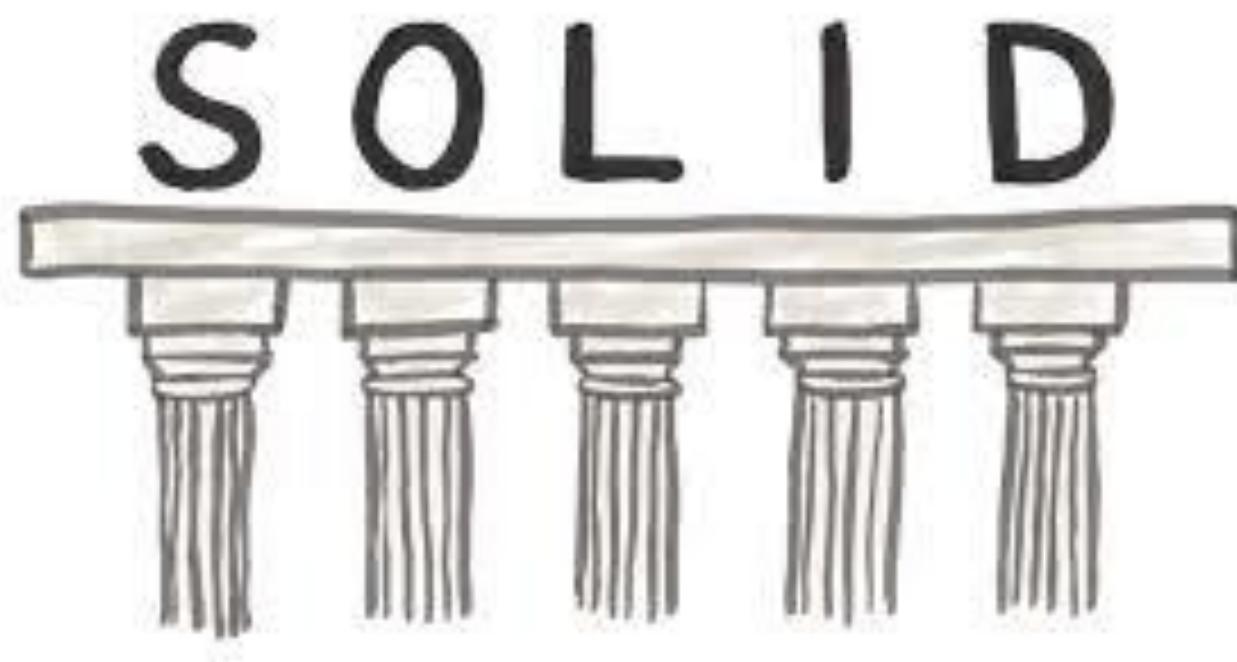
if __name__ == "__main__":
    print("Client: Executing the client code with a real subject:")
    real_subject = RealSubject()
    client_code(real_subject)

    print("")

    print("Client: Executing the same client code with a proxy:")
    proxy = Proxy(real_subject)
    client_code(proxy)
```

# Principios

---



- Como sabemos Programación Orientada a Objetos nos permite **agrupar entidades con funcionalidades parecidas o relacionadas entre sí**, pero esto no implica que los programas no se **vuelvan confusos o difíciles de mantener**.
- Muchos programas acaban volviéndose un monstruo al que se va alimentando según se añaden nuevas funcionalidades, se realiza mantenimiento, etc.
- Viendo este problema, **Robert C. Martin** estableció cinco directrices o principios para facilitarnos a los desarrolladores la labor de crear programas legibles y mantenibles.

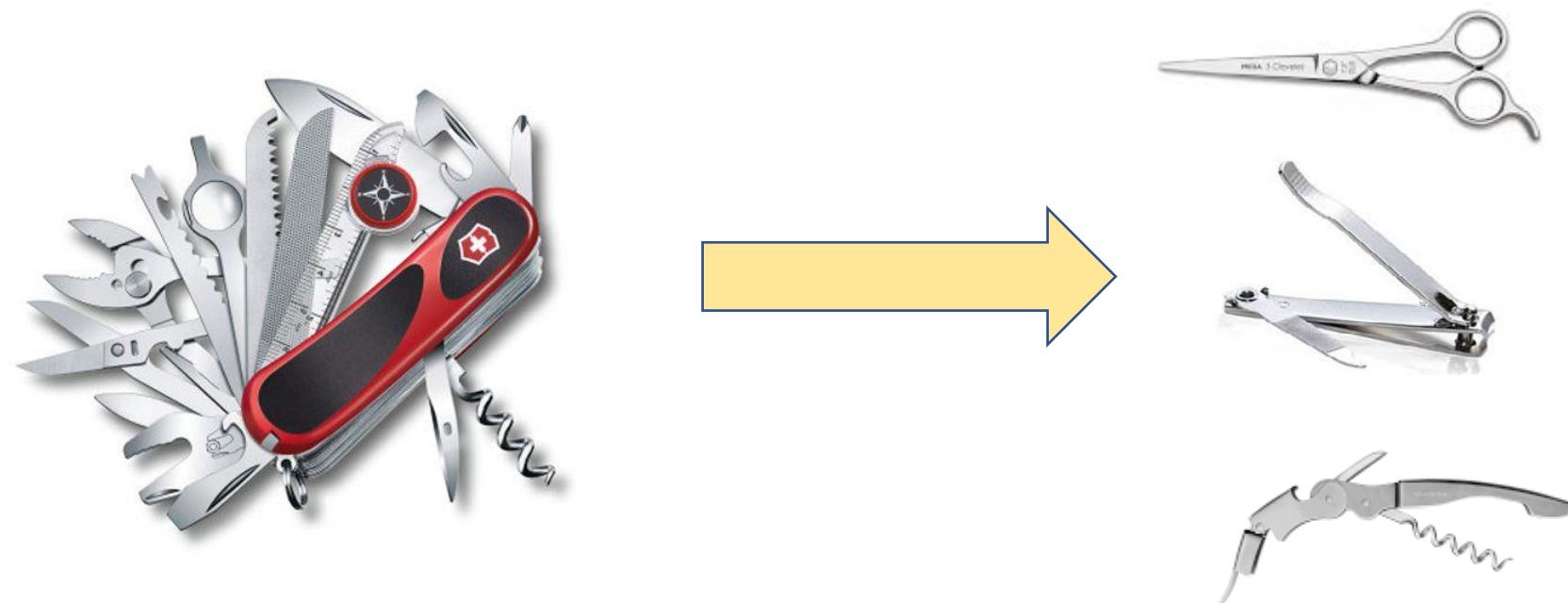


Estos principios se llamaron **S.O.L.I.D.** por sus siglas en inglés:



**S: Principio de responsabilidad única:**

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término “decoupled” en inglés). Si una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.



S.O.L.I.D

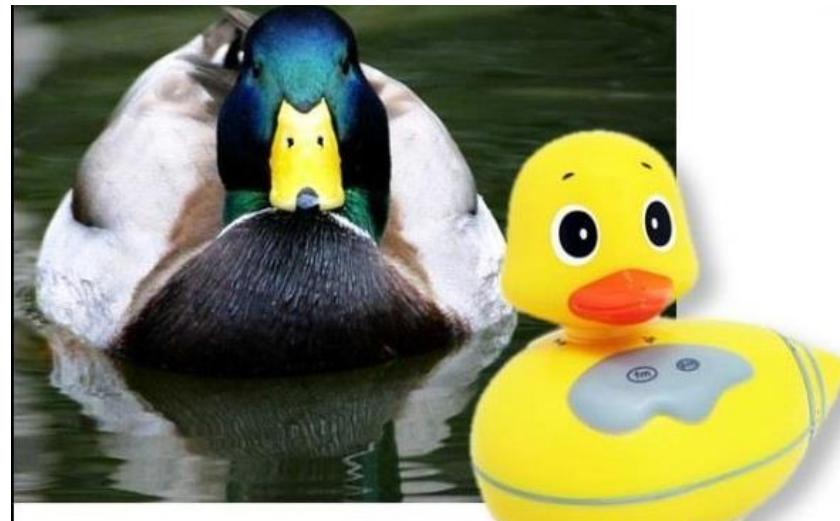
## O: Principio abierto/cerrado:

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.



**L: Principio de substitución de Liskov:**

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio. Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.



si parece un pato, flota como un pato, pero necesita baterías - probablemente tiene la abstracción equivocada.

## I: Principio de segregación de interfaz:

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

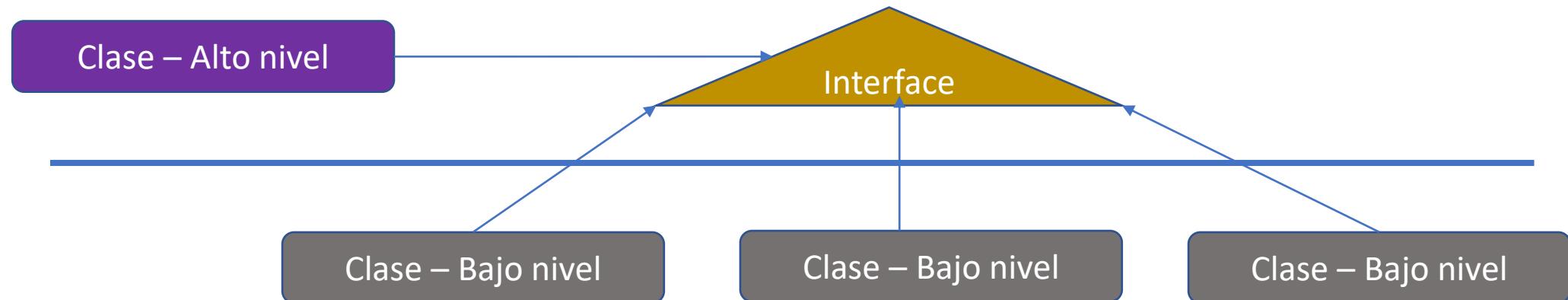
Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.



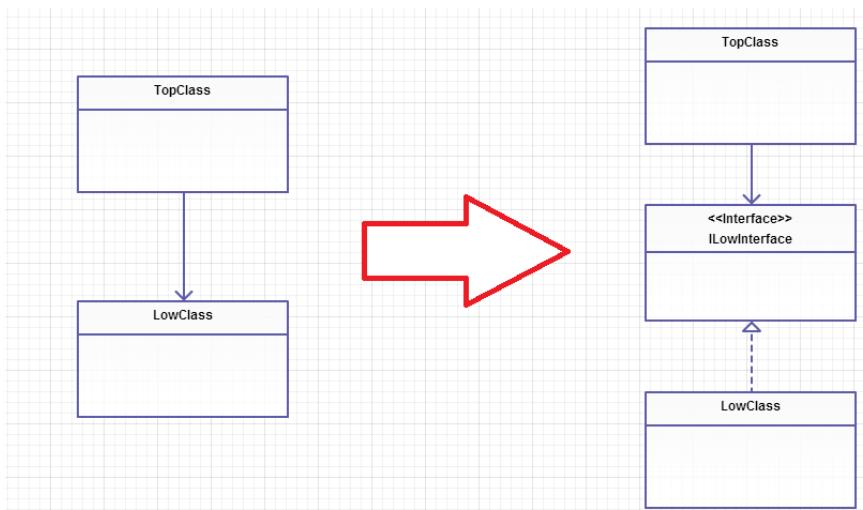
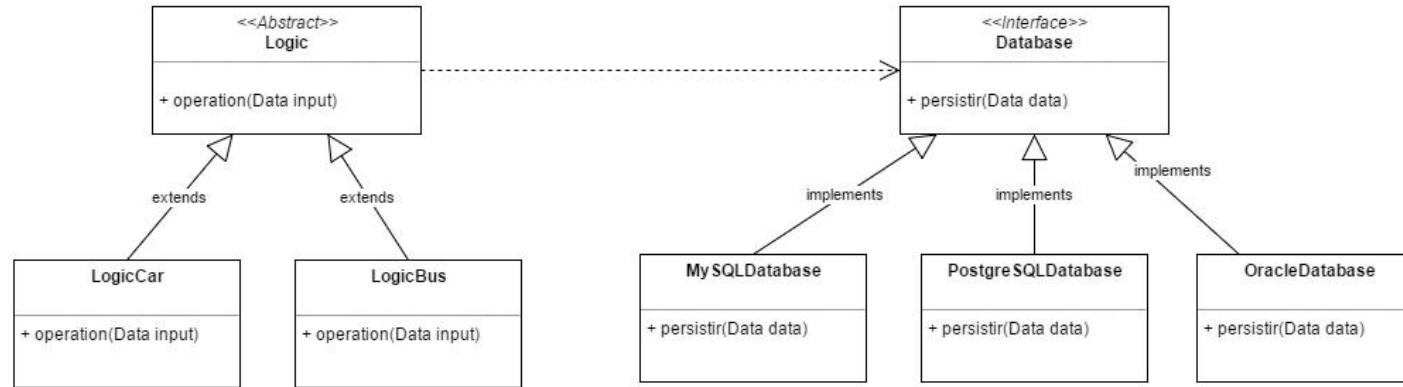
#### D: Principio de inversión de dependencias:

Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.



## D: Principio de inversión de dependencias:



## VENTAJAS

Mantenimiento del código más fácil y rápido

Permite añadir nuevas funcionalidades de forma más sencilla

Favorece una mayor reusabilidad y calidad del código, así como la encapsulación



# Anti patrones

---



## Anti patrones

- Los antipatrones muestran formas de enfrentarse a problemas con consecuencias negativas conocidas.
- Los antipatrones se basan en la idea de que puede resultar más fácil detectar a priori fallos en el desarrollo del proyecto que elegir el camino correcto, o lo que es lo mismo, descartar las alternativas incorrectas nos puede ayudar a la elección de la mejor alternativa.



- **Fabrica de combustible (gas factory)**: Diseñar de manera innecesariamente compleja.
- **Clase Gorda**: Dotar a una clase con demasiados atributos y/o métodos, haciéndola responsable de la mayoría de la lógica de negocio.
- **Gran bola de lodo (big ball of mud)**: Construir un sistema sin estructura definida.
- **Re-dependencia (re-coupling)**: Introducir dependencias innecesarias entre objetos.
- **Acoplamiento secuencial (sequential coupling)**: Construir una clase que necesita que sus métodos se invoquen en un orden determinado.
- **Llamar a super (callsuper)**: Obligar a las subclases a llamar a un método de la superclase que ha sido sobrescrito.
- **Objeto todopoderoso (god object)**: Concentrar demasiada funcionalidad en una única parte del diseño (clase).
- **Problema del yoyo (yo-yo problem)**: Construir estructuras (por ejemplo, de herencia) que son difíciles de comprender debido a su excesiva fragmentación.

- **Singletonitis:** Abuso de la utilización del patrón singleton.
- **Ancla del barco (*boat anchor*):** Retener partes del sistema que ya no tienen utilidad.
- **Comprobación de tipos en lugar de intefaz (*checking type instead of interface*):** Comprobar que un objeto es de un tipo concreto cuando lo único que se necesita es verificar si cumple un contrato determinado.
- **Manejo de excepciones inútil (*useless exception handling*):** Introducir condiciones para evitar que se produzcan excepciones en tiempo de ejecución.
- **Números mágicos (*magic numbers*):** Incluir en los algoritmos números concretos sin explicación aparente.
- **Reinventar la rueda (*reinventing the wheel*):** Enfrentarse a las situaciones buscando soluciones desde cero, sin tener en cuenta otras que puedan existir ya para afrontar los mismos problemas.

