Anuncios

1. Encuesta de carga académica
2. Lean el material y asistan a la ayudantía
3. Actividad de hoy es formativa. Deben subir una retroalimentación personal.

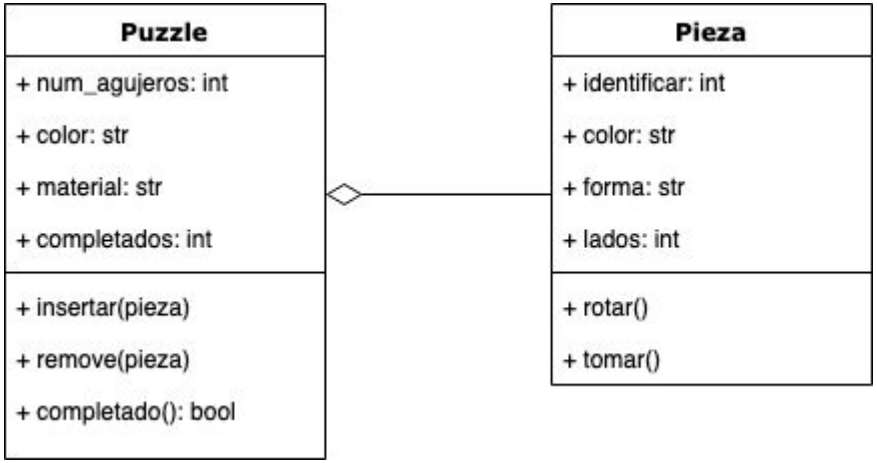
Programación Orientada a Objetos

Semana 02 - Jueves 22 de agosto 2019

¿Qué son los objetos?

- En el mundo real, los objetos son tangibles; se pueden manipular, sentir, y representan algo que tiene significado para nosotros.

- En el desarrollo de software, un objeto es una colección de **datos** que, además, tiene asociado ciertos **comportamientos**.
    - Datos: **describen** a los objetos
    - Comportamientos: **representan acciones** que ocurren en los objetos.



Ejemplo: datos y comportamiento

Objeto: Auto	
Datos	**Comportamiento**
Marca	Calcular próxima mantención
Modelo	Calcular distancia a alguna dirección
Color	Pintarlo de otro color
Año	Realizar nueva mantención
Motor	
Kilometraje	
Ubicación actual	
Mantenciones	

¿Qué es OOP?

- *Programación orientada a objetos* (OOP) es un **paradigma de programación** en el cual los programas modelan funcionalidades a través de la **interacción** entre **objetos** por medio de sus **datos** y **comportamiento**.

Clases versus instancia

- En OOP, los objetos son descritos por **clases**.

Cada objeto es una **instancia** de la clase Auto

Objeto 1

Objeto 2

Objeto 3

Clase Auto

Clase *versus* Instancia

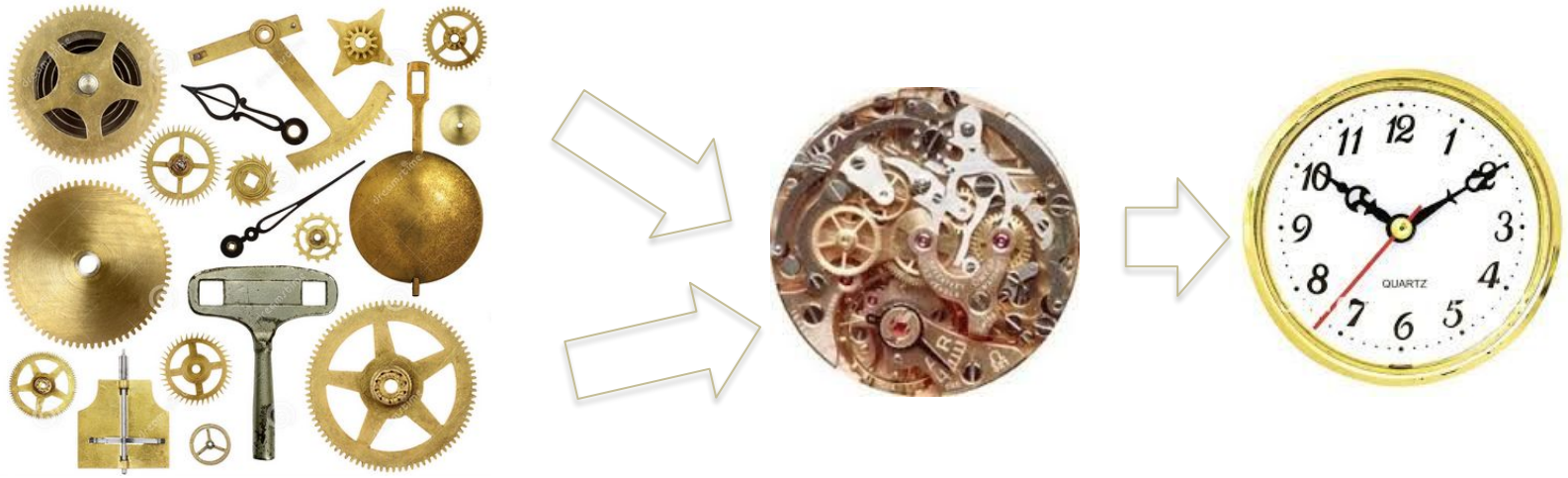
```
class A:
    atributo_clase = 1
    def __init__(self, parametro):
        self.atributo_instancia = parametro

    def metodo_instancia(self, parametro):
        self.atributo_instancia += parametro
        print(self.atributo_instancia)

objeto_1 = A(4)
objeto_1.metodo_instancia(3)
objeto_1.atributo_clase += 3
print(objeto_1.atributo_clase)
objeto_2 = A(5)
objeto_2.metodo_instancia(3)
print(objeto_2.atributo_clase)
```

Encapsulamiento e interfaces

- *Properties*
- *getters* y *setters*

Encapsulamiento

- Existen atributos de los objetos que no necesitan ser visualizados ni accedidos por los otros objetos con que se interactúa



Interfaz

Interface
Turn on Turn off Volume up Volume down Switch to next channel Switch to previous channel
Current channel Volume level

Conjunto de atributos y métodos que pueden ser usados por otros objetos para interactuar con este objeto.

Interfaz

Interface
Turn on Turn off Volume up Volume down Switch to next channel Switch to previous channel
Current channel Volume level

Conjunto de atributos y métodos que pueden ser usados por otros objetos
para interactuar con este objeto.

Interfaz

- Proceso de extracción de la interfaz de un objeto a partir de sus detalles internos.
- El nivel de detalle de la interfaz se denomina **abstracción**.

Vendedor
Nombre
Nº autos vendidos
Comisión asignada

Properties

Properties

- La modificación de ciertos atributos debería estar únicamente bajo el control de la clase: principio de encapsulamiento.

```
class Puente:

    def __init__(self, maximo):
        self.maximo = maximo
        self.personas = 0

puente = Puente(10)
puente.personas += 7
if puente.personas > puente.maximo:
    puente.personas = puente.maximo
print(f"Hay {puente.personas} personas")
```

La lógica de establecer límites al atributo `personas` debería estar encapsulada dentro del `Puente`.

Properties

- Añadir métodos que retornen o ajusten el atributo nos obliga a modificar todas las llamadas que ya están utilizando ese atributo y hace el código menos legible.

```
class Puente:

    def __init__(self, maximo):
        self.maximo = maximo
        self.__personas = 0

    def contar(self):
        return self.__personas

    def ingresar(self, p):
        if self.__personas + p > self.maximo:
            self.__personas = self.maximo
        elif self.__personas + p < 0:
            self.__personas = 0
        else:
            self.__personas = p

puente = Puente(10)
puente.ingresar(7)
print(f"Hay {puente.contar()} personas")
```

El atributo `personas` es "privado", y la lógica está encapsulada.

Pero la lectura (get) y modificación (set) es menos claro.

Si cambiamos el nombre de `contar` o `ingresar`, hay que propagar esos cambios en todo el código.

Properties

- Mecanismo de *properties* permite definir dos métodos especiales para manejar un atributo.
- Un método para leer el atributo: método *getter*, y uno para modificarlo: *setter*

```
class Puente:

    def __init__(self, maximo):
        self.maximo = maximo
        self.__personas = 0

    @property
    def personas(self):
        return self.__personas

    @personas.setter
    def personas(self, p):
        if p > self.maximo:
            self.__personas = self.maximo
        elif p < 0:
            self.__personas = 0
        else:
            self.__personas = p

puente = Puente(10)
puente.personas += 7
print(f"Hay {puente.personas} personas.")
```

Método getter y setter se usan como si fueran atributos

Invoca al setter: puente.personas(7)

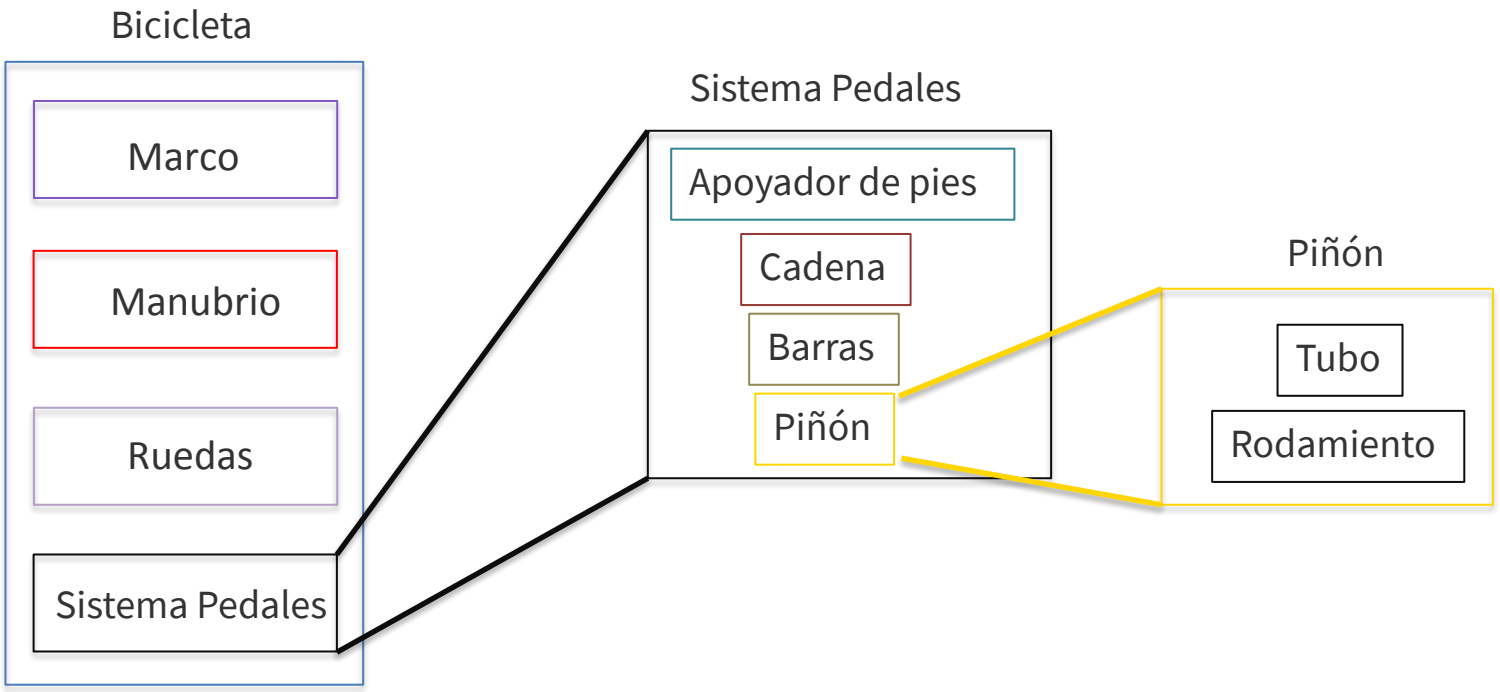
Invoca al getter: puente.personas()

Relaciones entre clases

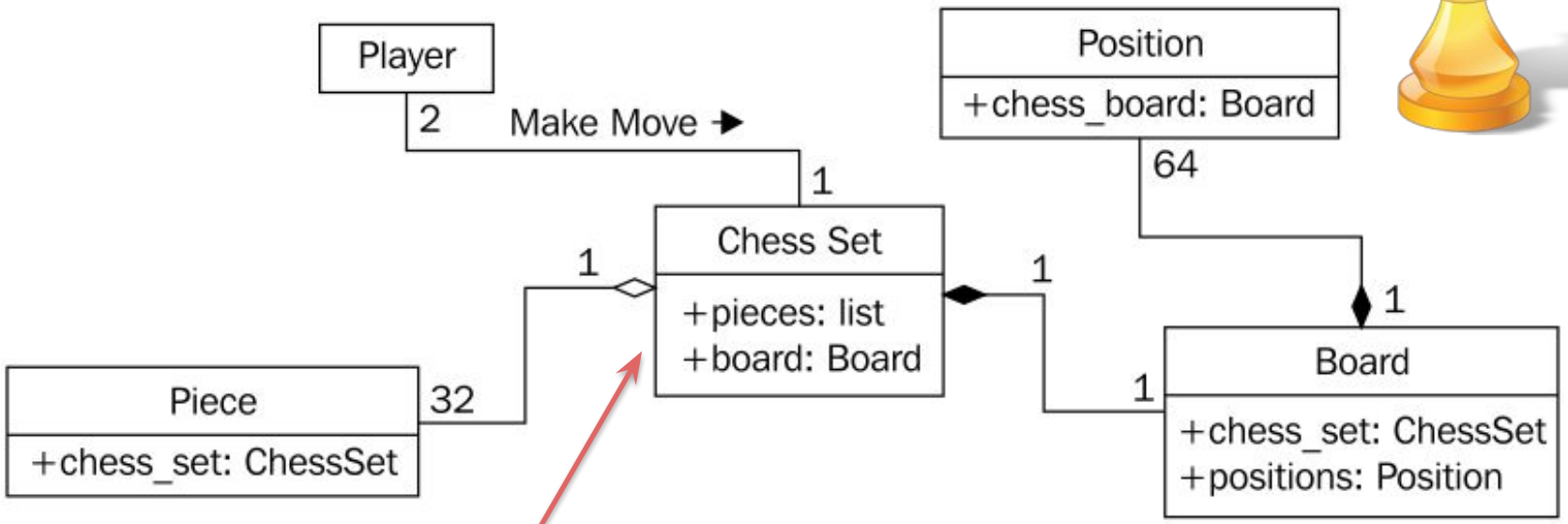
- Composición
- Agregación
- Herencia

Composición

- Colección de varios objetos para formar uno nuevo.



Composición *versus* agregación



Agregación ocurre cuando el objeto contenido en otro puede tener sentido como objeto único (rombo vacío).

```
class ChessSet:

    def __init__(self, piezas):
        self.piezas = piezas
        self.tablero = Board()

piezas = list((Pieza("Torre"), Pieza("Alfil"), ...))
juego = ChessSet(piezas)
```

Al hacer **composición**, se define que un objeto está compuesto por otro que "existe" al mismo tiempo que él.

Al hacer **agregación** se *agregan* objetos ya existentes a otro objeto.

Herencia

- Corresponde a una especialización
- *Overriding*

Herencia

Vehículo

Ruedas
Tamaño
Color
Encender
Mover

```
class Vehículo:
    def __init__(self):
        self.ruedas = ...
        ...

    def encender(self):
        print("Encendiendo vehículo")
```

Marca
Modelo
Airbag
Abrir maleta

Nº estudiantes
Filas asientos
Lista Colegios

Cilindros
Tipo Llantas
Encender

Armamento
Radar
Mover

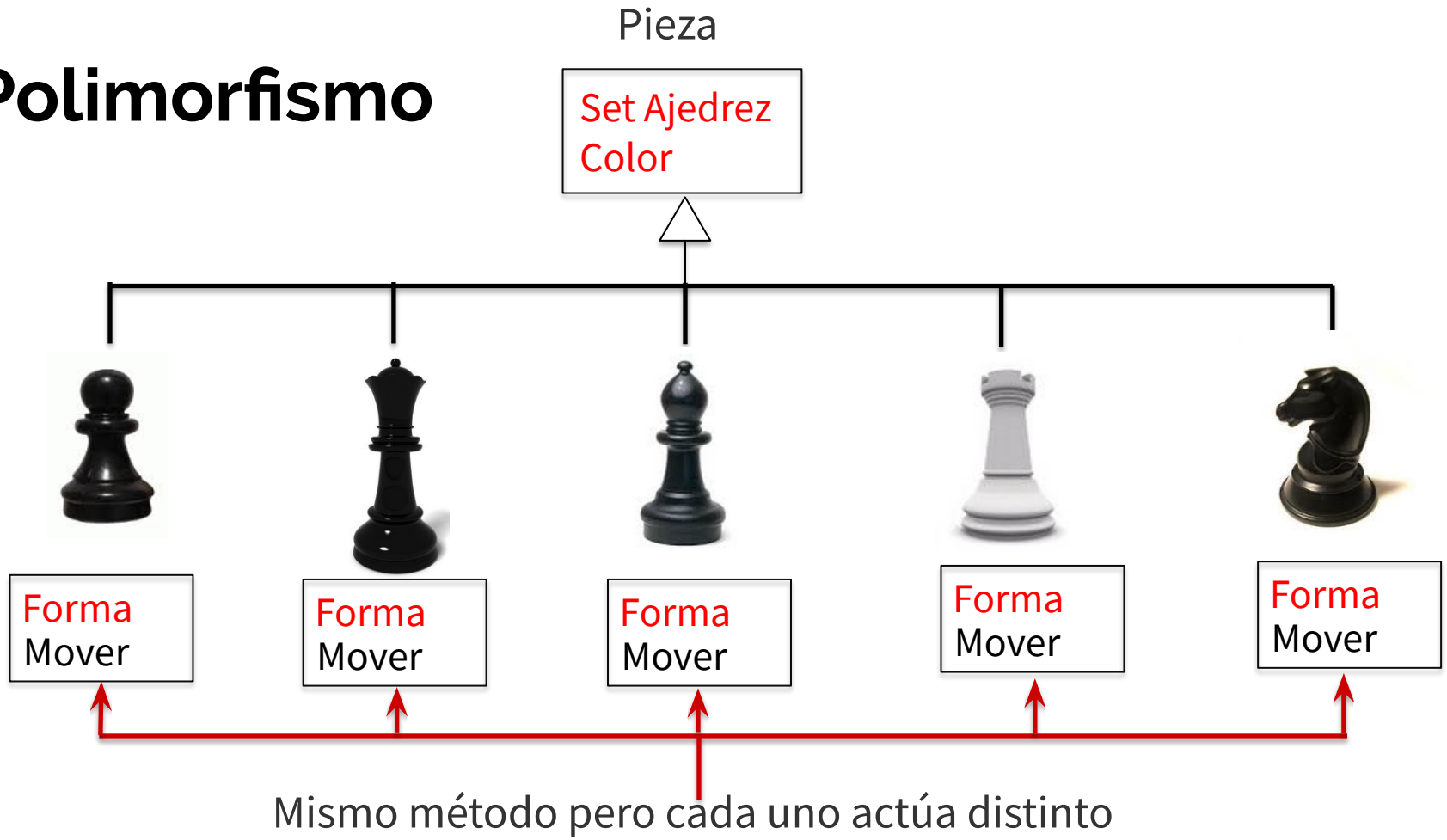
```
class Moto(Vehículo):
    def __init__(self):
        self.cilindros = ...
        ...

    def encender(self):
        print("Encendiendo moto")
```

Overriding

Polimorfismo y *duck typing*

Diferentes objetos ofrecen una misma interfaz, pero se comportan de forma distinta.

Polimorfismo




```
class Peón(Pieza):
    def __init__(self, f):
        self.forma = f
        ...

    def mover(self):
        print("Movimiento de peón")
```

```
class Torre(Pieza):
    def __init__(self, f):
        self.forma = f
        ...

    def mover(self):
        print("Movimiento de torre")
```

```
t1 = Torre("torre")
p1 = Peón("peón")
t1.mover()
p1.mover()
```

Duck typing

Forma
Mover

Forma
Mover

Forma
Mover

Forma
Mover

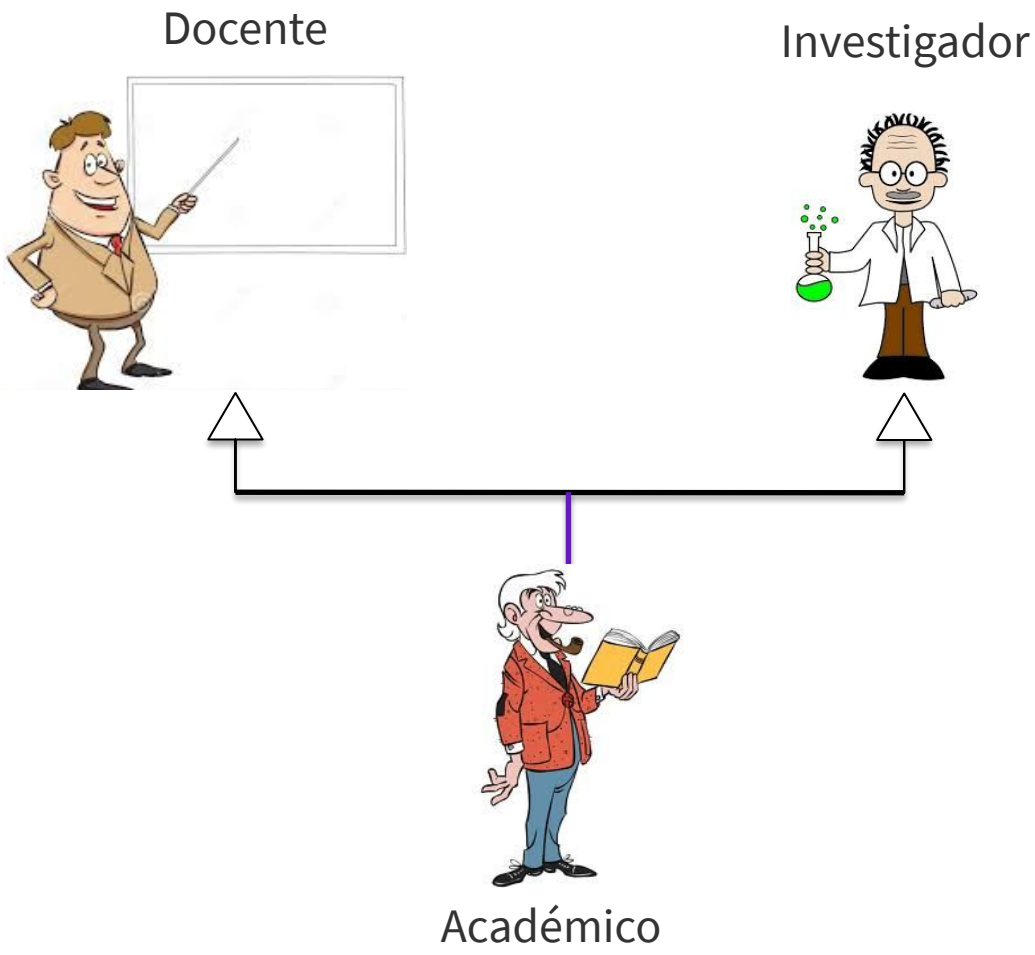
Raza
...
Mover

```
class Peón(Pieza):
    def __init__(self, f):
        self.forma = f
        ...

    def mover(self):
        print("Movimiento de peón")
```

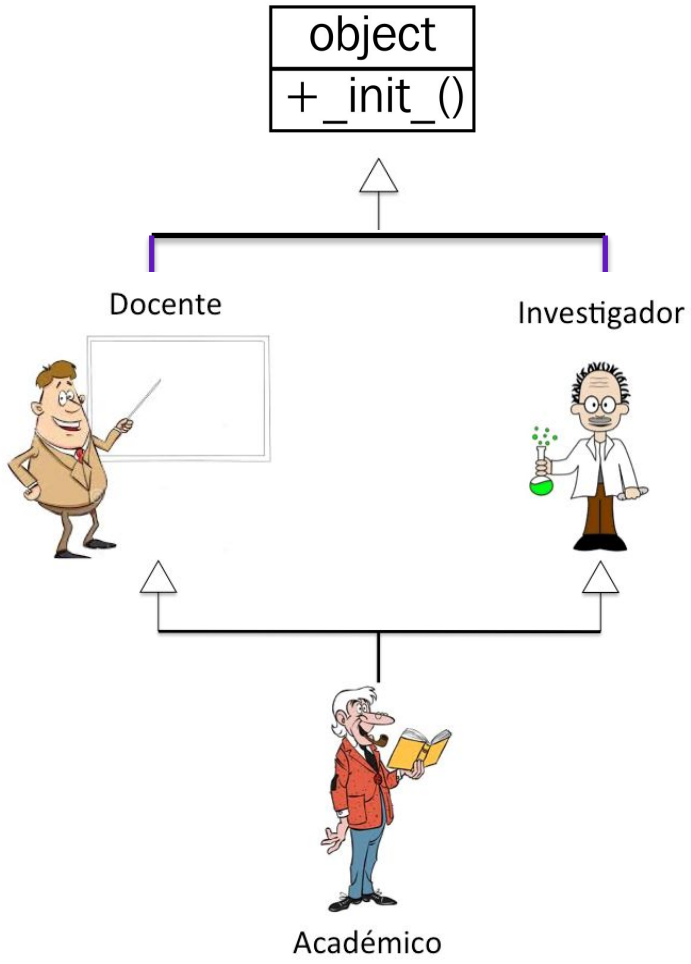
```
class Mono:
    def __init__(self, r):
        self.raza = r
        ...

    def mover(self):
        print("Movimiento de mono")
```

```
m = Mono("chimpancé")
p = Peón("peón")
m.mover()
p.mover()
```

Herencia múltiple



Multi-herencia y el problema del diamante

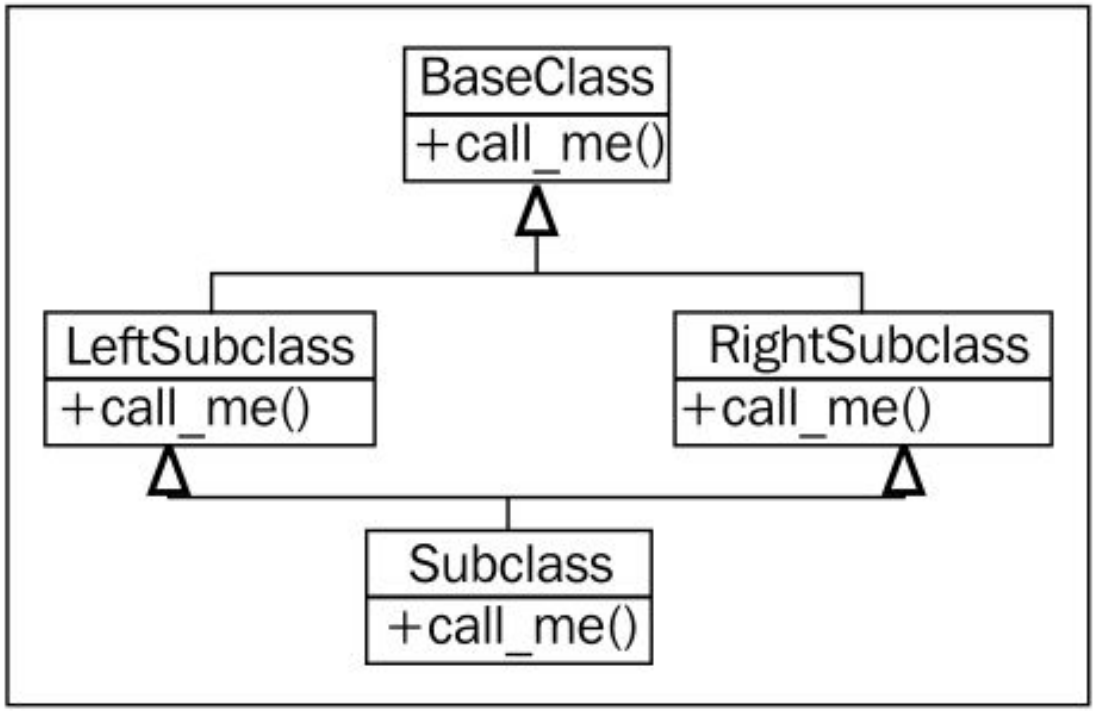


¿Qué clase es `super()` de la clase `Académico`?

¿Cuántas veces se llama a `__init__` de `object`?

```
class Académico(Docente, Investigador):
    def __init__(self):
        super().__init__()
        ...
```

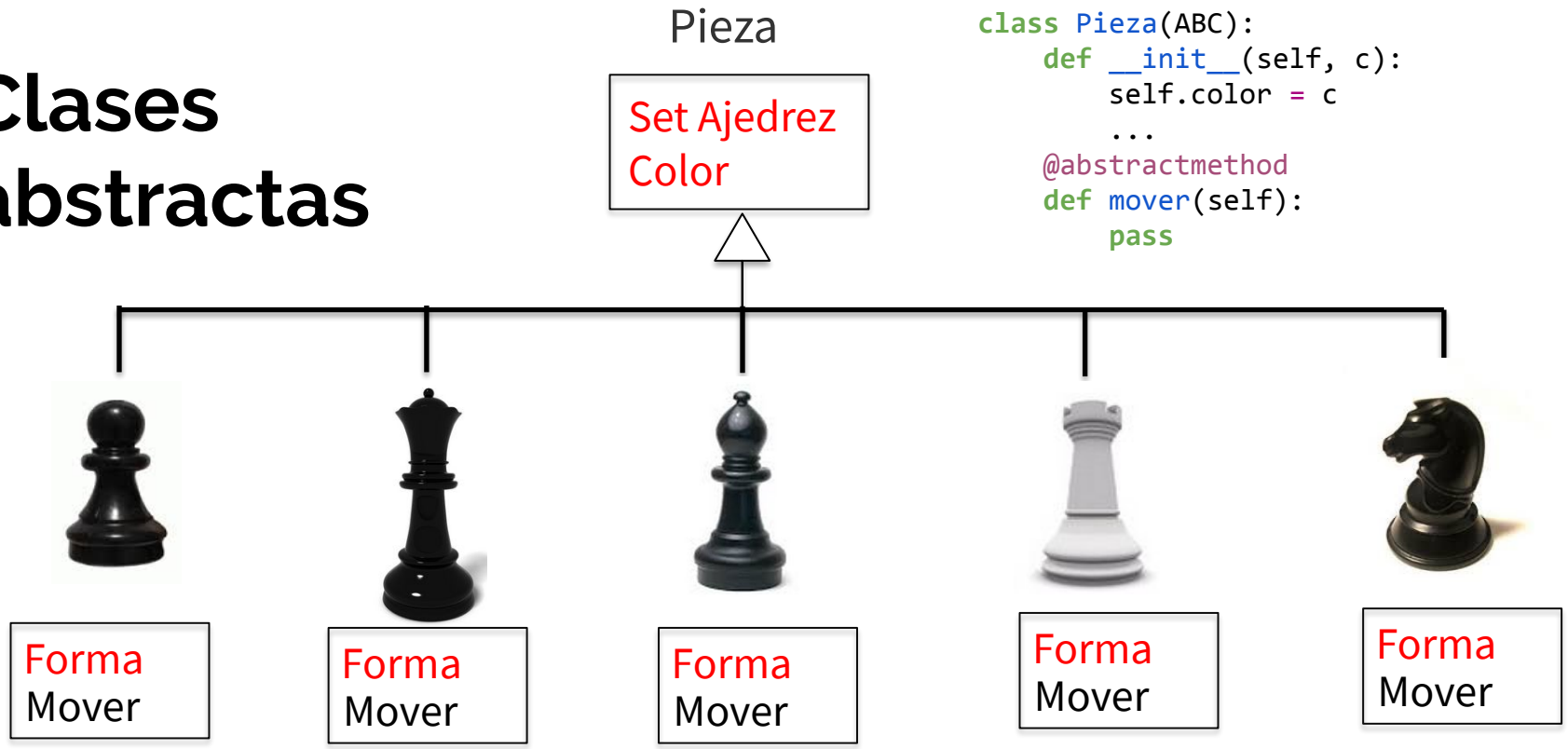
Multi-herencia y el problema del diamante



Clases abstractas

¿Para qué?

Clases abstractas




```python
class Pieza(ABC):
    def __init__(self, c):
        self.color = c
        ...
    @abstractmethod
    def mover(self):
        pass
```

Nunca se instancia una Pieza. Solo se instancia las clases que heredan de ella.

#OK. Peón implementa el método abstracto

```python
class Peón(Pieza):
    def __init__(self, f):
        self.forma = f
        ...

    def mover(self):
        print("Movimiento de peón")
```

#NotOK. Torre no implementa el método abstracto

```python
class Torre(Pieza):
    def __init__(self, f):
        self.forma = f
        ...
```

Overloading

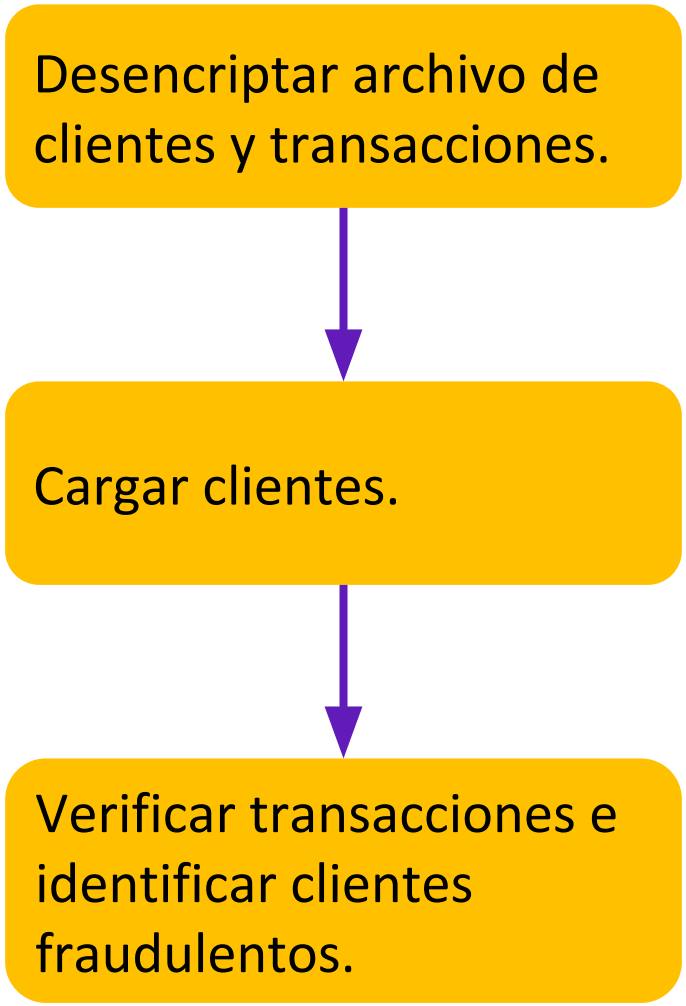
Un ejemplo

Actividad

1. En el *syllabus*, vayan a la carpeta “Actividades” y descarguen el enunciado de la actividad 1 (AC01) https://github.com/IIC2233/syllabus
2. Trabajen **individualmente** hasta las 16:30.
3. Recuerden hacer *commit* y *push* cada cierto tiempo.

Cierre

Diagrama de flujo de AC01




```
def recuperar_archivo(ruta)

class DrPintoDesencriptador
```

```
class BancoSeguroDCC(BancoDCC)

    def cargar_clientes(self, ruta)
```

```
def verificar_historial(self, historial)

def validar_montos(self, ruta)
```

Objetos

¿Se imaginan haber hecho esto sin clases que nos ordenan el código?

No solo permite modelar objetos, permite ordenar y estructurar el código en paquetes de funcionalidades, similar al uso de **módulos**.

Objetos

```
# main.py

banco_dcc_seguro = BancoDCCSeguro()
banco_dcc_seguro.cargar_clientes("banco_seguro/clientes.txt")


# recuperar_bd.py

def recuperar_archivo(ruta):

    desencriptador = DrPintoDesencriptador()

    desencriptador.ruta = ruta

    return desencriptador.texto_desencriptado
```

Properties

Podemos ocupar *properties* como herramienta para validar un atributo...

```
class ClienteSeguro(Cliente):

    # Asumiendo que los demás métodos están implementados...

    @saldo_actual.setter
    def saldo_actual(self, nuevo_saldo):
        if nuevo_saldo < 0:
            # ¡Es fraudulento!
            self.tiene_fraude = True
            self.saldo = None
        else:
            # Todo OK
            self.saldo = nuevo_saldo
```

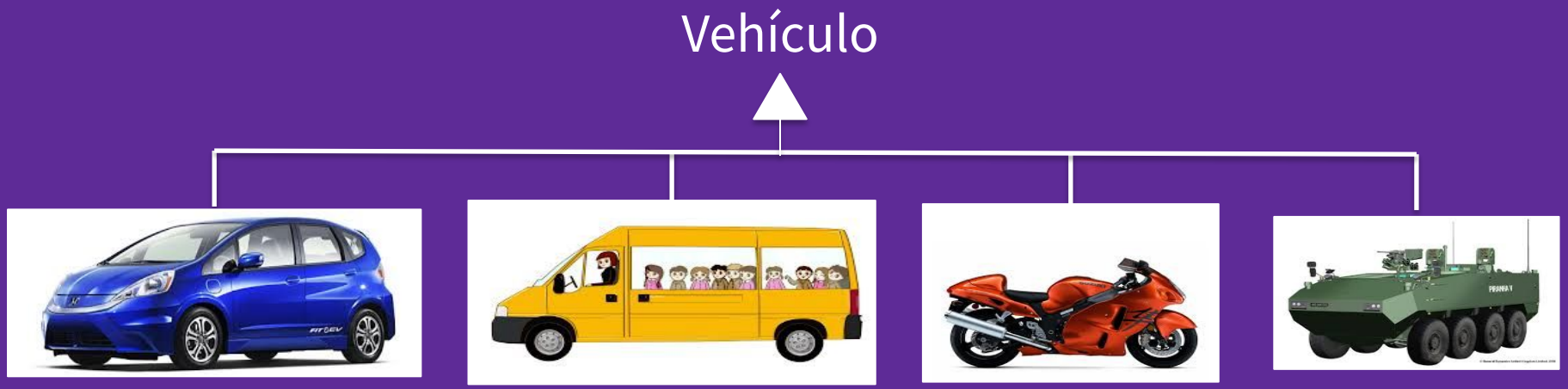
Properties

… pero también podemos ocuparlas para calcular atributos dinámicos o variables

```
class DrPintoDesencriptador(Cliente):

    # Asumiendo que los demás métodos están implementados...

    @texto_desencriptado.getter
    def texto_desencriptado(self):
        # Simplemente llamamos al método ya existente
        return self.desencriptar()
```

Herencia

Podemos utilizar herencia para hacer una mejor modelación...



Herencia

… y también como herramienta para “especializar” a nuestras clases

```
class ClienteSeguro(Cliente):

    # Asumiendo que los demás métodos están implementados...

    def deposito_seguro(self, dinero):
        # Usamos el depositar original, de la clase Cliente
        super().depositar(dinero)
        # Usamos nuestra property especializada
        self.saldo_actual = self.saldo
        with open(...) as archivo:
            transaccion = f“{self.id_cliente},depositar,{dinero}”
            archivo.write(transaccion)
```

Herencia

… y también como herramienta para “especializar” a nuestras clases

```
class BancoSeguroDCC(BancoDCC):

    # Asumiendo que los demás métodos están implementados...

    def validar_monto_clientes(self, ruta):
        with open(...) as archivo:
            archivo.readline() # Saltamos la primera línea
            for linea in archivo:
                info_cliente = linea.strip().split(“,”)
                # Usamos el método original del BancoDCC...
                cliente = self.buscar_cliente(info_cliente[0])
                # ...pero nuestro Cliente (ClienteSeguro)
                if cliente.saldo_actual != int(info_cliente[2]):
                    cliente.tiene_fraude = True
```

Programación Orientada a Objetos

Semana 02 - Jueves 21 de marzo 2019