Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos de Programación Orientada a Objetos (OOP)

Profesor: Hans Löbel

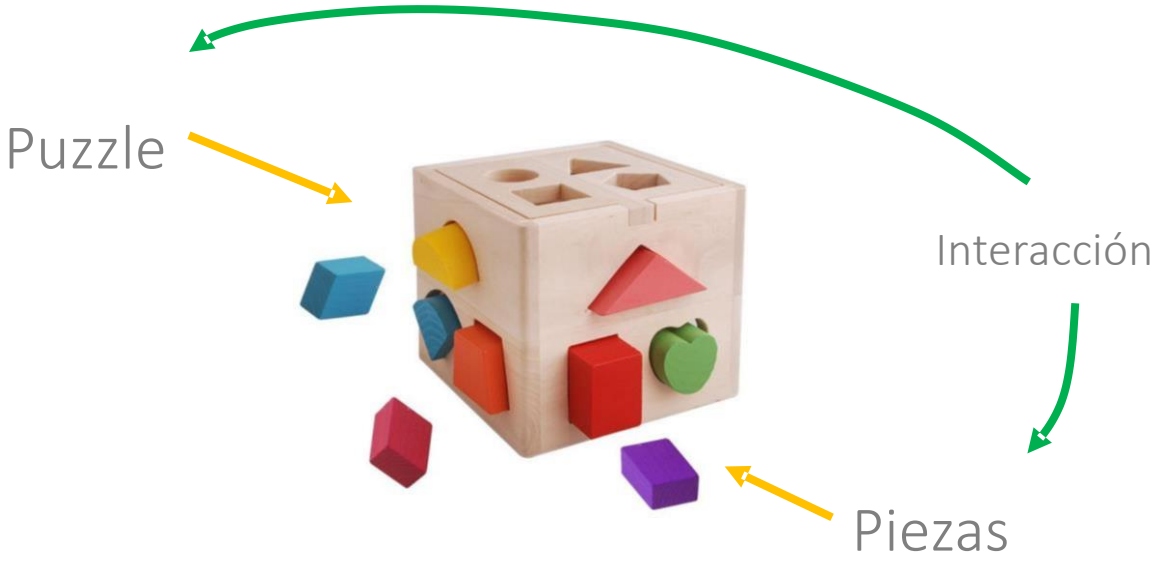
Partamos con un ejemplo conceptual

Cuando hablamos de un curso, ¿en qué estamos pensando?

Todas estas maneras de “modelarlo” representan distintas abstracciones del concepto curso, cada una más o mensos adecuadas para distintas tareas.

Esto cambia un poco cuando tenemos elementos físicos

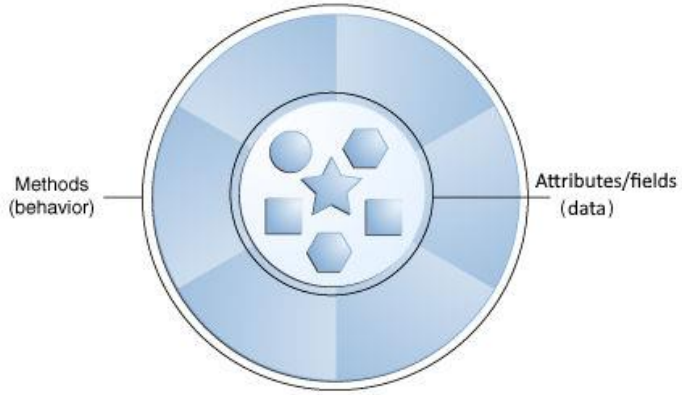
Los objetos físicos ya están modelados, por lo que traen definido su todo y sus partes constituyentes, pero además, una manera de interactuar con ellos



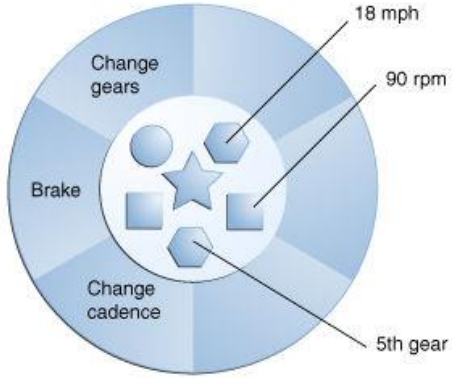
Objetos de software combinan ambas ideas

En el área de desarrollo de software, un objeto es una colección de datos que además tiene asociado comportamientos.

- Datos: describen el estado y/o composición de los objetos. Se les conoce como atributos o campos del objeto.
- Comportamientos: representan acciones que realiza el objeto, o realizan sobre él, que pueden generar cambios en su estado. Se les conoce como métodos del objeto.



(a) A software object



(b) Bicycle modelled as a software object

Ejemplo: datos y comportamiento

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

¿Qué es entonces OOP?

La programación orientada a objetos (OOP) implica que los programas estarán orientados a modelar las funcionalidades a través de la interacción entre objetos por medio de sus datos y comportamiento.

Para definir un objeto, creamos una plantilla llamada clase

Cada objeto es una instancia de la clase Auto

Objeto 1

Objeto 2

Objeto 3

Clase Auto

```python
class Departamento:
    def __init__(self, _id, mts2, valor, num_dorms, num_banos):
        self._id = _id
        self.mts2 = mts2
        self.valor = valor
        self.num_dorms = num_dorms
        self.num_banos = num_banos
        self.vendido = False

    def vender(self):
        if not self.vendido:
            self.vendido = True
        else:
            print("Departamento {} ya se vendió".format(self._id))
```

```python
d1 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
print(d1.vendido)
d1.vender()
print(d1.vendido)
d1.vender()
```

```
False
True
Departamento 1 ya se vendió
```

```
d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
d3.vender()
d4 = d1

print(d1 == d2)
print(d1 == d3)
print(d1 == d4)

d4.vendido = False
print(d1.vendido == d4.vendido)
```

```python
d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
d3.vender()
d4 = d1

print(d1 == d2)
print(d1 == d3)
print(d1 == d4)

d4.vendido = False
print(d1.vendido == d4.vendido)
```

```
False
False
True
True
```

Un concepto fundamental es el de interfaz de un objeto

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

Un concepto fundamental es el de interfaz de un objeto

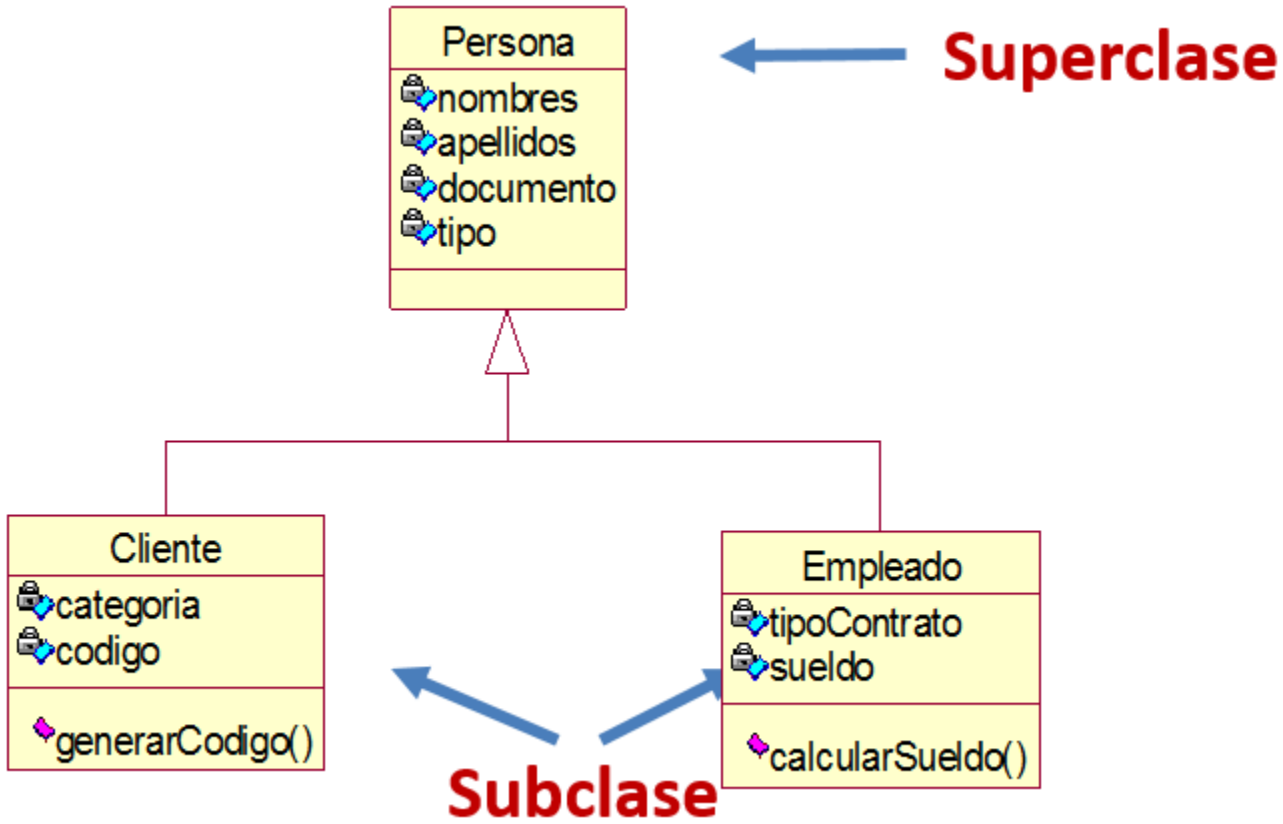
Interface
Turn on Turn off Volume up Volume down Switch to next channel Switch to previous channel
Current channel Volume level

```python
class Televisor:
    ''' Clase que modela un televisor.
    '''

    def __init__(self, pulgadas, marca):
        self.pulgadas = pulgadas
        self.marca = marca
        self.encendido = False
        self.canal_actual = 0

    def encender(self):
        self.encendido = True

    def apagar(self):
        self.encendido = False

    def cambiar_canal(self, nuevo_canal):
        self._codificar_imagen()
        self.canal_actual = nuevo_canal

    def __codificar_imagen(self):
        print("Estoy convirtiendo una señal eléctrica en la imagen que estás viendo.")
```

El nivel de detalle de la interfaz define nuestra abstracción

Vendedor
Nombre
Nº autos vendidos
Comisión asignada

Herencia nos permite modelar clases similares sin reescribir todo de nuevo



```python
import numpy as np

class Variable:
    def __init__(self, data):
        self.data = np.array(data)

    def representante(self):
        pass

class Ingresos(Variable):
    def representante(self):
        return np.mean(self.data)

class Comuna(Variable):
    def representante(self):
        ind = np.argmax([np.sum(self.data == c) for c in self.data])  # el que mas se repite
        return self.data[ind]

class Puesto(Variable):
    categorias = {'Gerente': 1, 'SubGerente': 2, 'Analista': 3,
                  'Alumno en Practica': 4} # class (or static) variable

    def representante(self):
        return self.data[np.argmin([Puesto.categorias[c] for c in self.data])]#la categoria mas alta acorde con el diccionario
```

Multiherencia es poco común, pero puede ser útil

```python
class Investigador:
    def __init__(self, area):
        self.area = area

class Docente:
    def __init__(self, departamento):
        self.departamento = departamento

class Academico(Docente, Investigador):
    def __init__(self, nombre, area_investigacion, departamento):
        #esto no es del todo correcto, coming soon...
        Investigador.__init__(self, area_investigacion)
        Docente.__init__(self, departamento)
        self.nombre = nombre

p1 = Academico("Juan Perez", "Inteligencia de Máquina", "Ciencia De La Computación")
print(p1.nombre)
print(p1.area)
print(p1.departamento)
```

```
Juan Perez
Inteligencia de Máquina
Ciencia De La Computación
```

Clases abstractas permiten formalizar interfaz

```python
from abc import ABCMeta, abstractmethod

class Base(metaclass=ABCMeta):
    @abstractmethod
    def func_1(self):
        pass

    @abstractmethod
    def func_2(self):
        pass

class SubClase(Base):
    def func_1(self):
        pass

    # Nuevamente olvidamos declarar el método func_2

print('Es subclase: {}'.format(issubclass(SubClase, Base)))
print('Es instancia: {}'.format(isinstance(SubClase(), Base)))
```

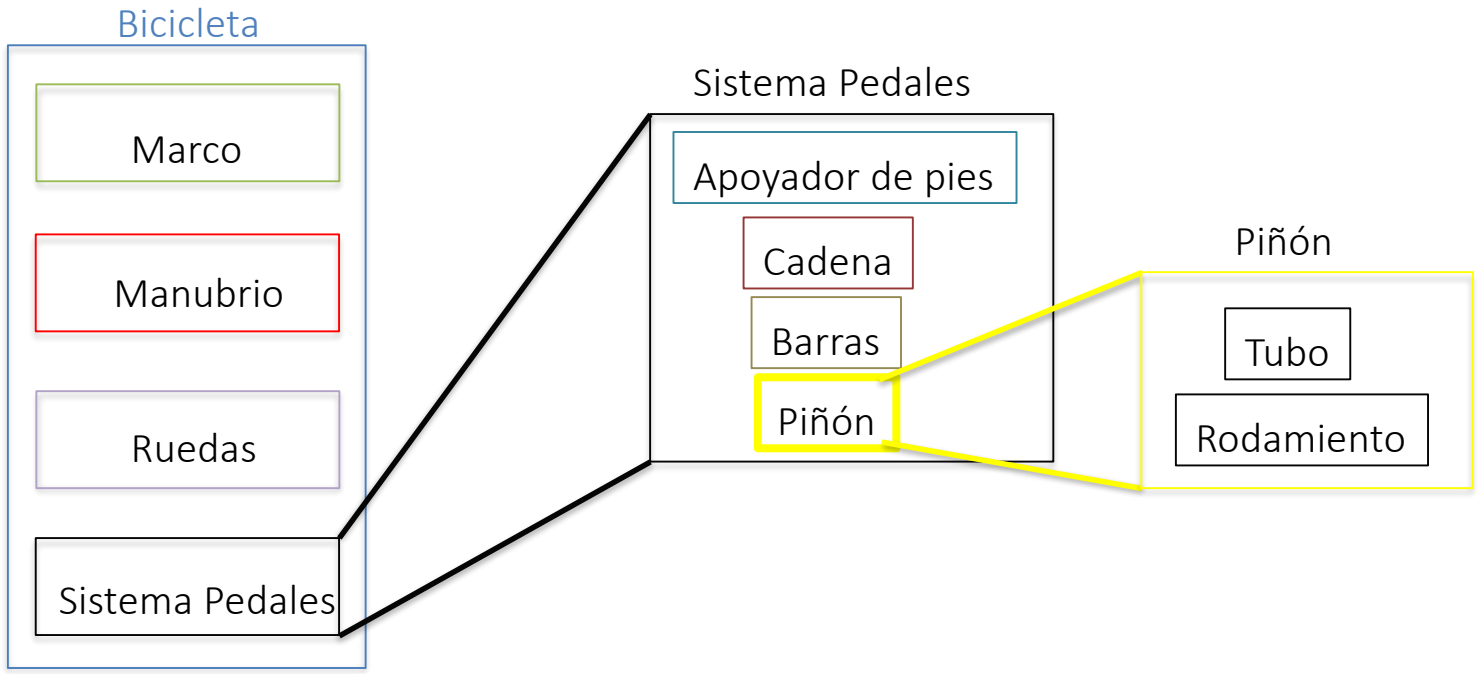
```
Es subclase: True
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-657925633d3a> in <module>()
     17
     18 print('Es subclase: {}'.format(issubclass(SubClase, Base)))
---> 19 print('Es instancia: {}'.format(isinstance(SubClase(), Base)))

TypeError: Can't instantiate abstract class SubClase with abstract methods func_2
```

Clases abstractas permiten formalizar interfaz

```python
from abc import ABCMeta, abstractmethod

class Base(metaclass=ABCMeta):
    @abstractmethod
    def func_1(self):
        pass

    @abstractmethod
    def func_2(self):
        pass

class SubClase(Base):

    def func_1(self):
        pass

    def func_2(self):
        pass

    # Nuevamente olvidamod declarar el método func_2

c = SubClase()
print('Subclass: {}'.format(issubclass(SubClase, Base)))
print('Instance: {}'.format(isinstance(SubClase(), Base)))
```

```
Subclass: True
Instance: True
```

También es posible modelar objetos como atributos de otros objetos, mediante agregación y composición



```python
#Ejemplo: Programa para manejar postits en un panel mural

import datetime

class PostIt:
    ''' Representa un post it, contiene un mensaje, guarda un conjunto de tags
        y responde si hay match con ciertos tags
        Contiene ademas un id
    '''
    last_id = 0 #variable estática para manejar el ultimo id generado

    def __init__(self, mensaje, tags = ''):
        self.mensaje = mensaje
        self.tags = tags
        self.creation_date = datetime.date.today()
        self._id = PostIt.last_id #variable de la clase para manejar el ultimo id generado
        PostIt.last_id += 1

    def match(self, keyword):
        ''' determina si el mensaje de la nota contiene el keyword o no'''
        return keyword in self.mensaje or keyword in self.tags
```

```
class Panel:
    ''' Representa un panel con un conjunto de postits (con memos) pegados
        cada hoja ademas de un memo tiene tags, asi podemos buscar hojas
        tambien podemos modificarlas
    '''

    def __init__(self):
        self.postit1 = PostIt('')
        self.postit2 = PostIt('')
        self.postit3 = PostIt('')
```

Agregación/composición vs herencia

- NO TIENEN NADA QUE VER!
- Si bien ambos son mecanismo para modelar, estructuralmente difieren de manera fundamental.
- Herencia busca facilitar la especialización de las clases, sin requerir repetir código.
- Agregación y composición buscan aumentar el nivel de abstracción de las clases, al permitir tipos de dato complejos (otras clases) como atributos.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos de Programación Orientada a Objetos (OOP)

Profesor: Hans Löbel