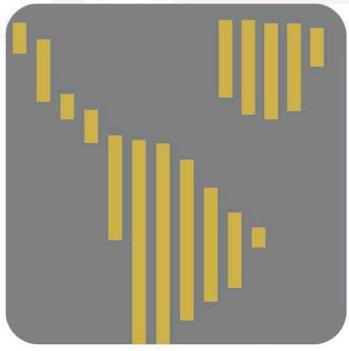


PROMiDAT
IBEROAMERICANO

Programa Iberoamericano de
Formación en Minería de Datos

www.promidat.com / info@promidat.com / (506) 4030-1205 / 4030-1114



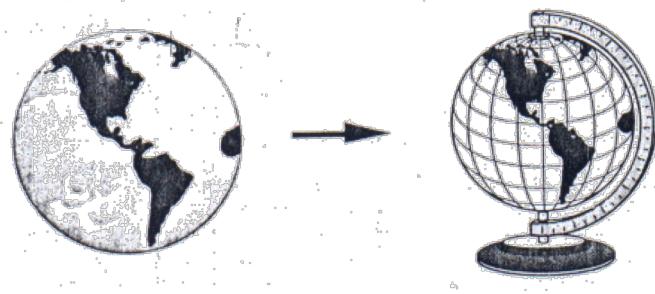
Ciencia de Datos con Python



Definiciones en Orientación a Objetos 2º Parte

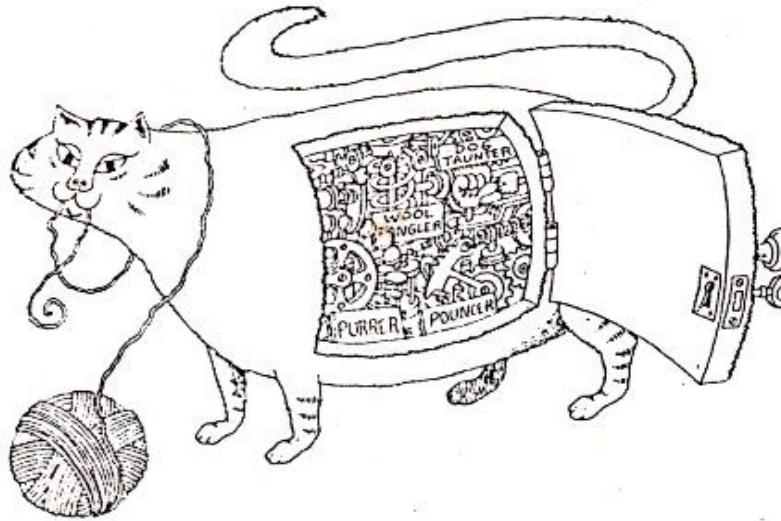
Paradigma Orientación a Objetos

- ❖ Se basan en la idea de que el Dominio de una Aplicación y los Requerimientos se pueden ***modelar, programar e implementar*** basado:
 - OO = Objetos + Métodos + Asociaciones + Herencia + Polimorfismo



Encapsulación

Concepts

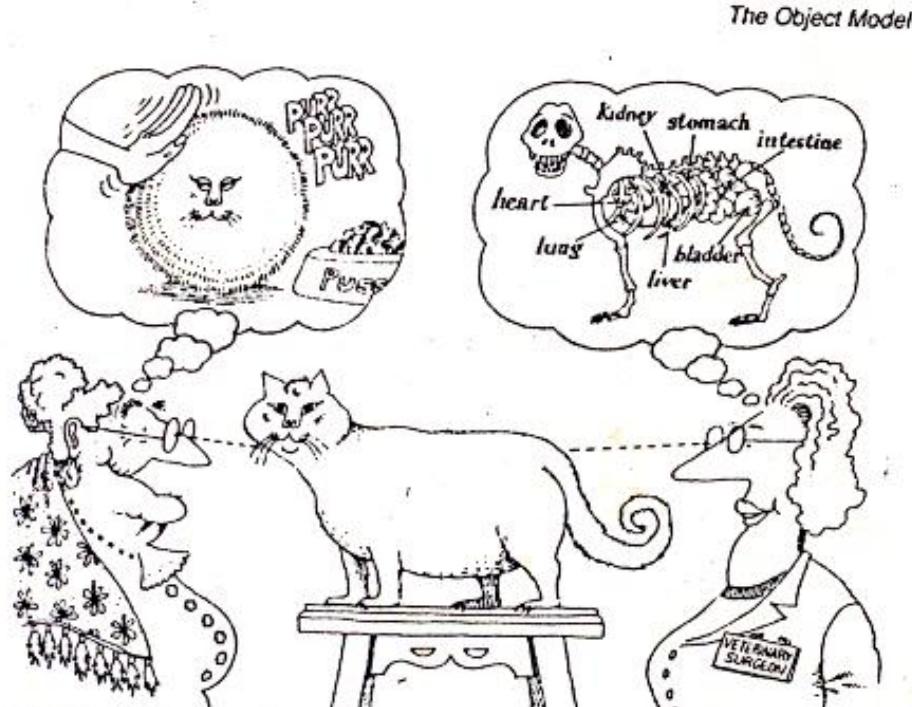


Encapsulation hides the details of the implementation of an object.

Abstracción [Booch]

- *La abstracción denota las características esenciales de un objeto que lo distinguen de todos los otros tipos de objetos y provee una clara definición de las fronteras conceptuales, relativas a las perspectivas del usuario.*

Abstracción



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Clases Abstractas en Python (Abstract base classes, o ABCs clases)

- *Las clases base abstractas, o clases tipo ABC, definen un conjunto de métodos y atributos que una subclase (clase derivada) debe implementar para ser considerada una instancia de ese tipo de clase.*
- *El módulo abc de Python proporciona las herramientas que necesita para hacer esto. “from abc import ...”*

Clases Abstractas en Python

(Abstract base classes, o ABCs clases)

```
from abc import ABCMeta, abstractmethod

# Clase Abstracta, ABC Class
class Base(metaclass = ABCMeta):
    @abstractmethod
    def __str__(self):
        pass
    @abstractmethod
    def Captura(self):
        pass|
```

Clases Abstractas en Python (Abstract base classes, o ABCs clases)

```
from Base import Base

class Libro(Base):
    def __init__(self, nombre = "", anio = 0):
        self.__nombre = nombre
        self.__anio = anio
    @property
    def nombre(self):
        return self.__nombre
    @property
    def anio(self):
        return self.__anio
    @nombre.setter
    def nombre(self, nuevo_nombre):
        self.__nombre = nuevo_nombre
    @anio.setter
    def anio(self, nuevo_anio):
        self.__anio = nuevo_anio
    # Los siguientes métodos usan los set y get NO los atributos
    def __str__(self):
        return "Nombre del Libro: %s\nAño: %i" % (self.nombre, self.anio)
    def Captura(self):
        self.nombre = input("Nombre del Libro:")
        self.anio = int(input("Año del Libro:"))
```



Ligamiento Tardío y Temprano

los lenguajes de programación estructurados parten del hecho incuestionable del ligamiento (*binding*) temprano, es decir, que el tipo de los parámetros se determina *a priori*, en términos de lenguajes de programación, en tiempo de compilación. Este enfoque, aún cuando simplifica los lenguajes, limita mucho el poder de modelación de la realidad, es por esto que la Orientación a Objetos incluye la posibilidad del ligamiento tardío, y lenguajes como Python, SmallTalk y C++ ya lo han implementado.

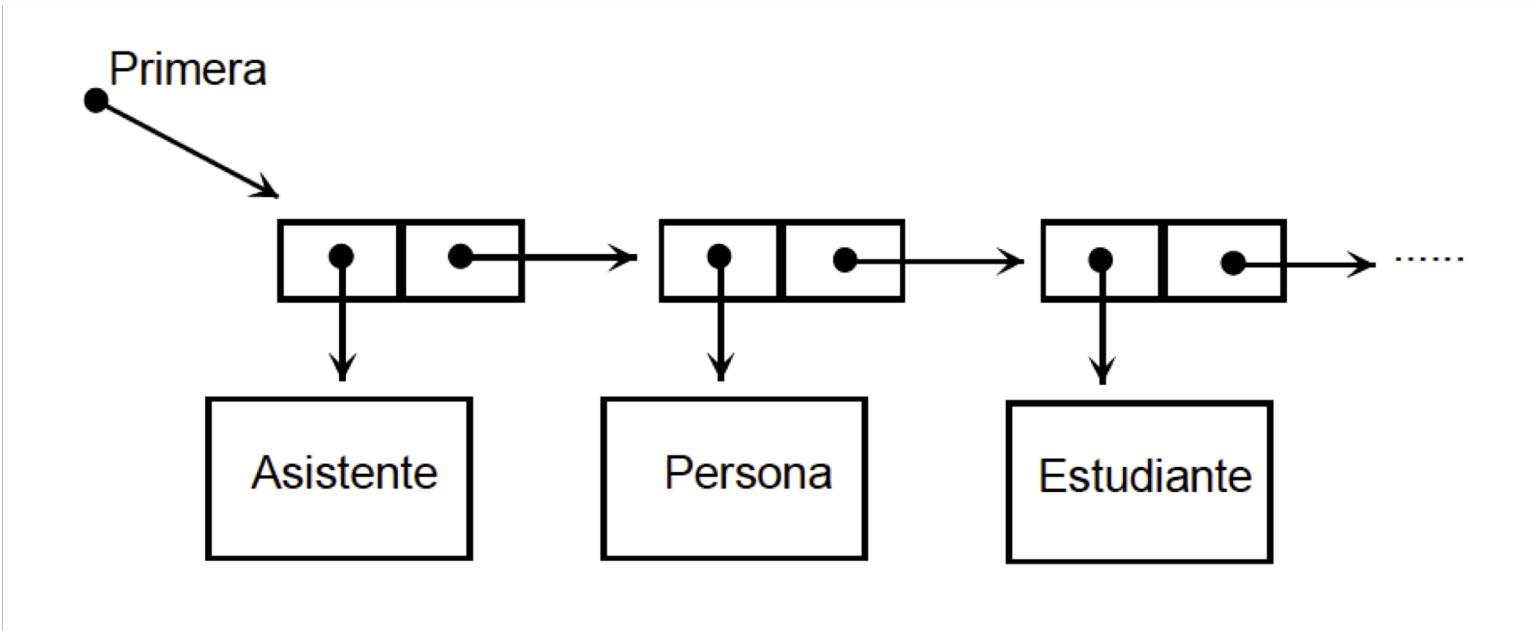
Polimorfismo

- El concepto de polimorfismo está íntimamente ligado al concepto de ligamiento tardío. En orientación a objetos tiene que ver con el hecho de que existen métodos (métodos virtuales) que pueden tener diferente implementación en clases distintas, lo que permite que en tiempo de ejecución el sistema determine cuál utilizar, de acuerdo con el tipo de parámetro con que es invocada.

Polimorfismo

```
class SuenaAudio:  
    def __init__(self, nombre_archivo):  
        if not nombre_archivo.endswith(self.ext):  
            raise Exception("formato invalido")  
        self.nombre_archivo = nombre_archivo  
  
class SuenaMP3(SuenaAudio):  
    ext = "mp3"  
    def suena(self):  
        print("Sonando {} como un mp3".format(self.nombre_archivo))  
  
class SuenaWav(SuenaAudio):  
    ext = "wav"  
    def suena(self):  
        print("Sonando {} como un wav".format(self.nombre_archivo))  
  
class SuenaOgg(SuenaAudio):  
    ext = "ogg"  
    def suena(self):  
        print("Sonando {} como un ogg".format(self.nombre_archivo))
```

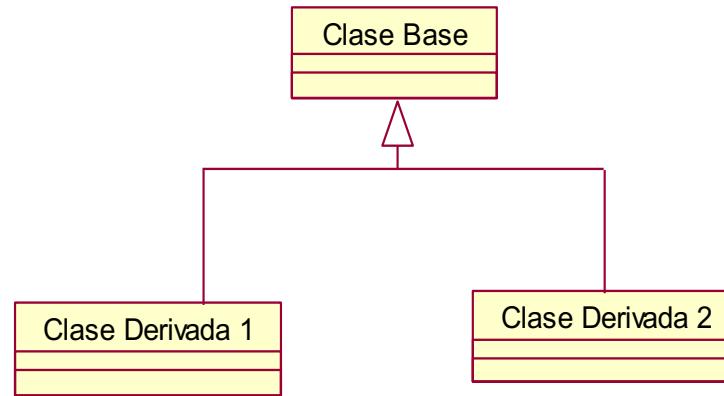
Listas Polimórficas



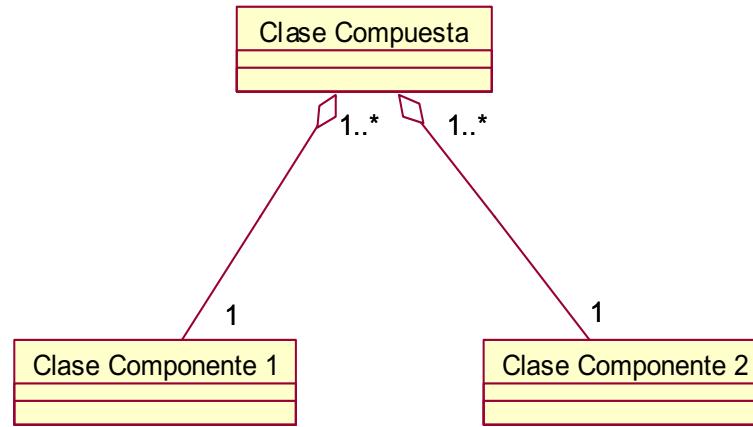
Polimorfismo

```
def mostrarLista(lista):
    print("\n"*50)
    for i in range(len(lista)):
        print(lista[i])
    print(15 * "*" + "\n")
```

Notación gráfica de la Herencia en UML

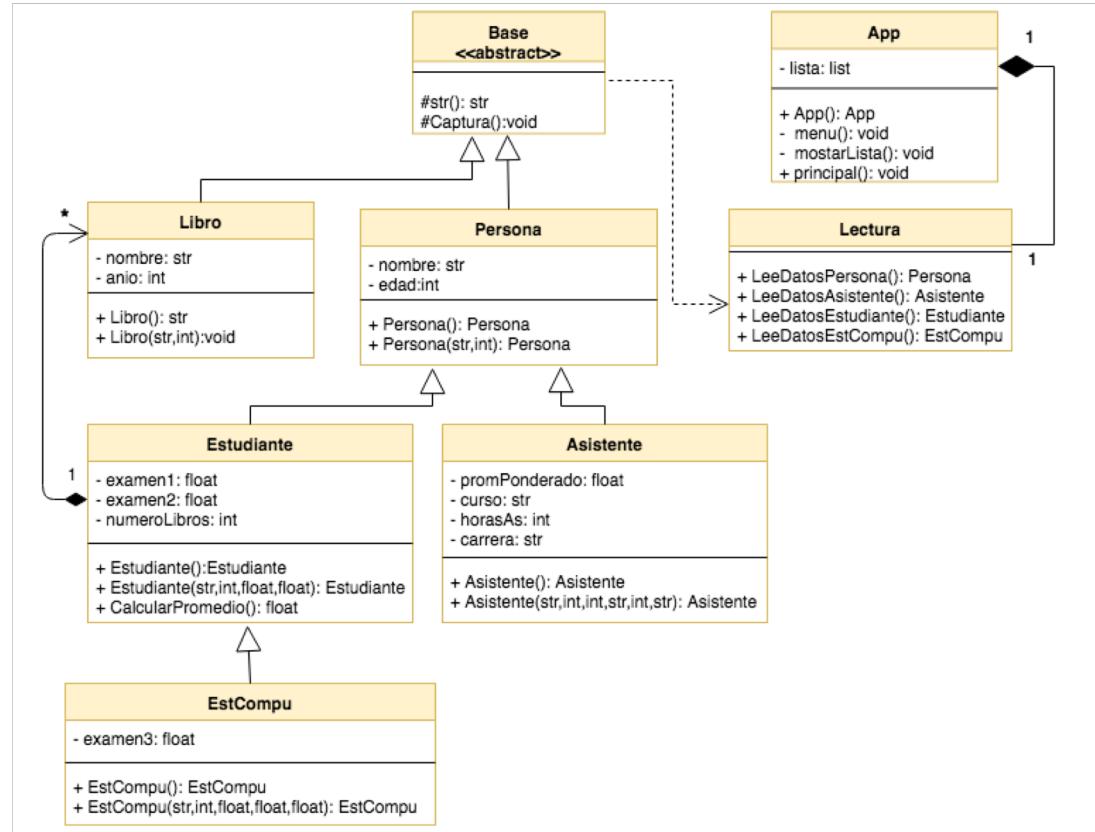


Relaciones de *Composición* (Com-Com) en UML

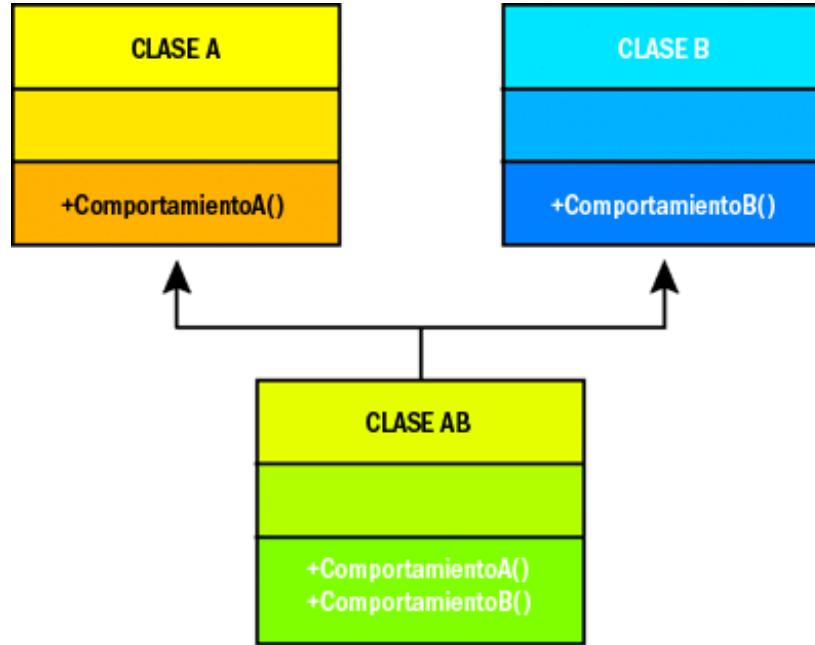


Jerarquía de Clases

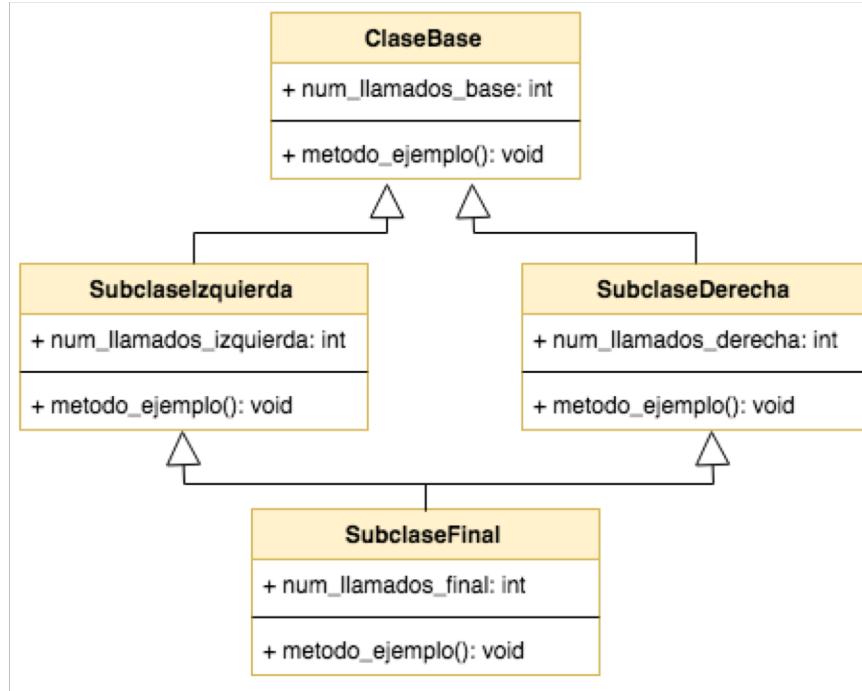
(Ver Código Python)



Herencia Múltiple



Herencia Múltiple en Python

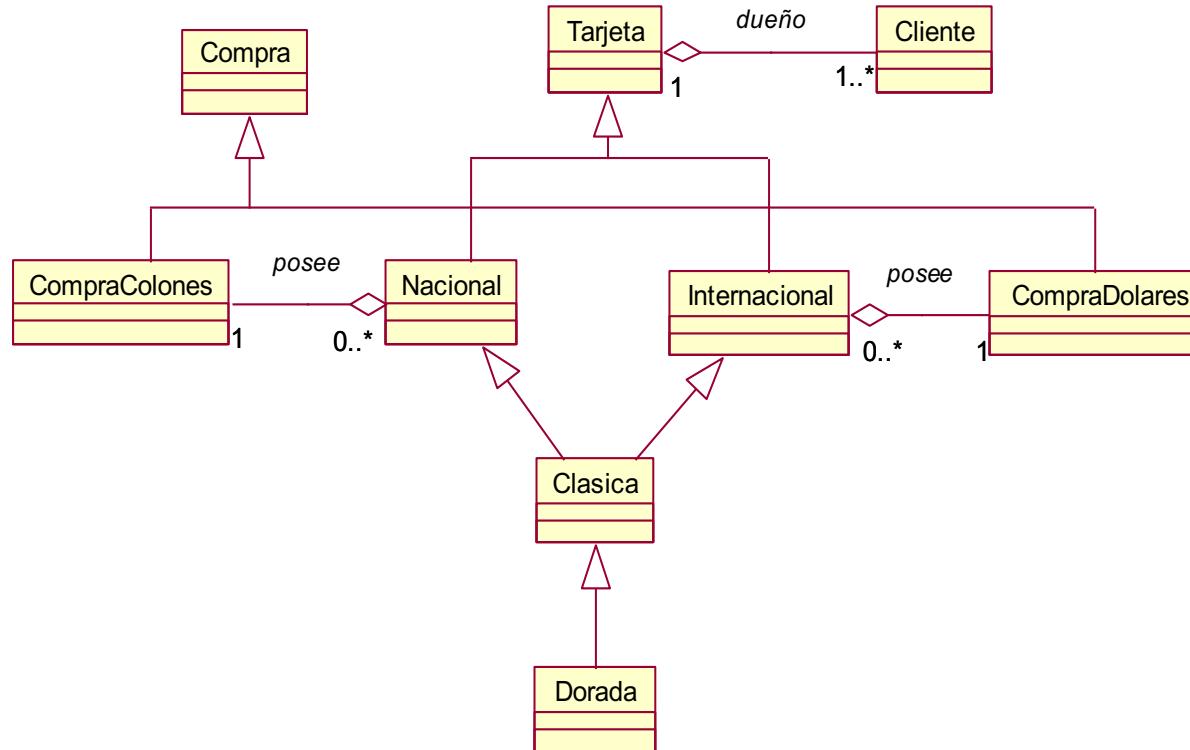


Herencia Múltiple en Python

```
class ClaseBase:  
    num_llamados_base = 0  
    def metodo_ejemplo(self):  
        print("Llamando metodo_ejemplo en la clase Base")  
        self.num_llamados_base += 1  
  
class SubclaseIzquierda(ClaseBase):  
    num_llamados_izquierda = 0  
    def metodo_ejemplo(self):  
        super().metodo_ejemplo()  
        print("Llamando metodo_ejemplo en Izquierda SubclaseFinal")  
        self.num_llamados_izquierda += 1  
  
class SubclaseDerecha(ClaseBase):  
    num_llamados_derecha = 0  
    def metodo_ejemplo(self):  
        super().metodo_ejemplo()  
        print("Llamando metodo_ejemplo en Derecha SubclaseFinal")  
        self.num_llamados_derecha += 1  
  
class SubclaseFinal(SubclaseIzquierda, SubclaseDerecha):  
    num_llamados_final = 0  
    def metodo_ejemplo(self):  
        super().metodo_ejemplo()  
        print("Llamando metodo_ejemplo en SubclaseFinal")  
        self.num_llamados_final += 1
```



Jerarquía de Clases



Modularidad

La modularidad es un concepto que tampoco es nuevo en la Orientación a Objetos y se utiliza ampliamente en lenguajes como Modula-2. Un módulo agrupa un conjunto de procedimientos relacionados, así como los datos que ellos manipulan. Este es un principio importante que permite en buena medida la reutilización del código, además, la modularidad pretende dividir un programa o sistema en subsistemas o subprogramas lo cual contribuye ampliamente a disminuir la complejidad del software.

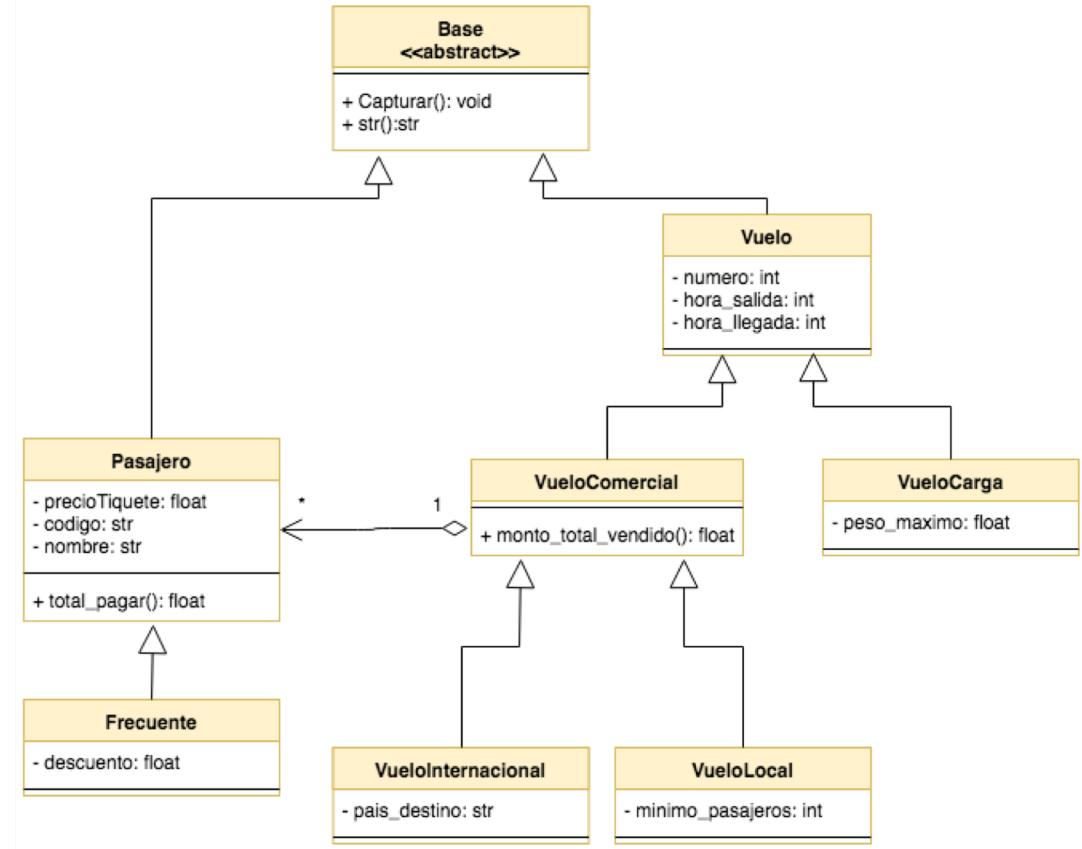
Modularidad

Concepts



Modularity packages abstractions into discrete units.

Jerarquía de Clases



¿Qué es Orientado a Objetos y qué no lo es?

- Dave Thomas en su artículo "What is an Object?" define cuatro conceptos fundamentales en la Orientación a Objetos; estos son:
 - Encapsulación.
 - Paso de mensajes (solicitud de operaciones genéricas).
 - Herencia de clases, y
 - Ligamiento tardío y temprano.

¿Qué es Orientado a Objetos y qué no lo es?

- Mientras que para Peter Wegner es suficiente tener:
 - Objetos.
 - Clases, y
 - Herencia de clases.

¿Qué es Orientado a Objetos y qué no lo es?

- Pero para Booch, según lo expresa en existen cuatro conceptos fundamentales en la Orientación a Objetos, los cuales son:
 - Abstracción.
 - Encapsulación.
 - Modularidad y
 - Herencia.



Programa Iberoamericano de
Formación en Minería de Datos



www.promidat.com
info@promidat.com
(506) 4030-1205 / 4030-1114