INSTITUTO SUPERIOR POLITÉCNICO CÓRDOBA

Fundamentos del Enfoque Orientada a Objetos

Contenido

Fundamentos del enfoque orientado a objetos (EOO)

El Enfoque Orientado a Objeto se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son:

- Jerarquías (herencia, agregación y composición)
- Abstracción
- Encapsulamiento
- Modularidad

Otros elementos a destacar (aunque no fundamentales) son:

- Polimorfismo
- Tipificación
- Concurrencia
- Persistencia.

Los cuales se describirán a continuación:

Jerarquías

Las clases no se diseñan para que trabajen de manera aislada, el objeto es que se puedan relacionar entre sí de manera que puedan compartir atributos y métodos y así resolver un problema.

La capacidad de establecer jerarquías entre las clases es una característica que hace que la programación orientada a objetos sea diferente a la programación tradicional (estructurada). Esto se debe principalmente a que el código existente se puede escalar y reutilizar sin tener que volver a escribirlo cada vez.

Herencia

En Programación Orientada a Objetos la herencia es un proceso mediante el cual se puede crear una clase hija que hereda de una clase padre, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobreescribir los métodos o atributos, o incluso definir unos nuevos. Para crear una clase hija se debe pasar como parámetro la clase de la que se quiere heredar.

Por ejemplo en python, se puede definir una clase padre llamada Auto y crear una clase hija llamada Convertible que hereda de Auto de la siguiente manera:

```
# Definimos una clase padre
class Auto:
    pass
# Creamos una clase hija que hereda de la padre
class Convertible(Auto):
    pass
```

En resumen, la *herencia* es un mecanismo por el cual podemos definir una nueva clase *B* en términos de otra clase *A* ya definida, pero de forma que la clase *B* obtiene todos los miembros definidos en la clase *A* sin necesidad de hacer una redeclaración explícita. El sólo hecho de indicar que la clase *B* hereda (o deriva) desde la clase *A*, hace que la clase *B* incluya todos los miembros de *A* como propios (a los cuales podrá acceder en mayor o menor medida de acuerdo al modificador de acceso [*public*, *private*] que esos miembros tengan en *A*).

Es decir que la herencia permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial) y evitando así la repetición de código y permitiendo la reusabilidad.

Cuando la clase *B* hereda de la clase *A*, se dice que hay una *relación de herencia* entre ellas, y se modela en UML con una flecha continua terminada en punta cerrada. La flecha parte de la nueva clase (o clase derivada) que sería *B* en nuestro ejemplo, y termina en la clase desde la cual se hereda (que es *A* en nuestro caso):

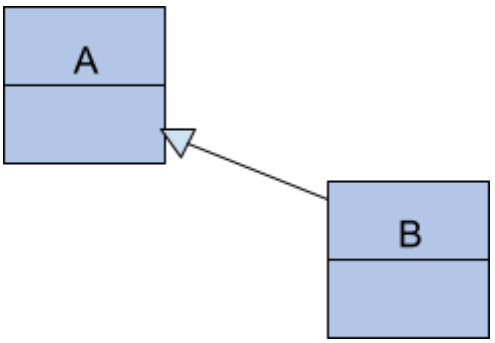

Figura 8: (UML) Diagrama de clases: Representación de la Herencia

La clase desde la cual se hereda, se llama ***super clase***, y las clases que heredan desde otra se llaman *subclases* o *clases derivadas*: de hecho, la herencia también se conoce como *derivación de clases*.

Una ***jerarquía*** *de clases* es un conjunto de clases relacionadas por herencia. La clase en la cual nace la jerarquía que se está analizando se designa en general como *clase base* de la jerarquía. La idea es que la *clase base* reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. El siguiente gráfico muestra una jerarquía de clases:

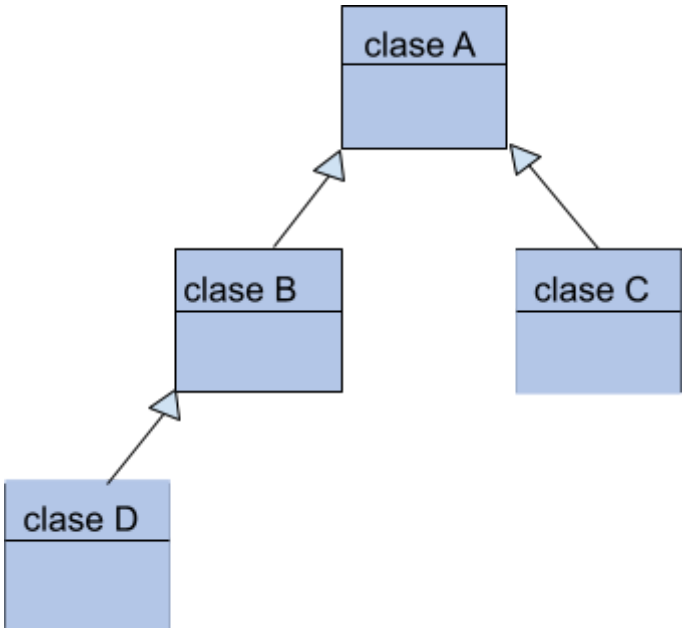

Figura 9: (UML) Diagrama de clases: Jerarquía de Clases

En esta jerarquía, la *clase base* es *"Clase A"*. Las clases *"Clase B"* y *"Clase C"* son *derivadas directas* de *"Clase A"*. Note que *"Clase D"* deriva en forma directa desde *"Clase B"*, pero en *forma indirecta* también deriva desde *"Clase A"*, por lo tanto todos los elementos definidos en *"Clase A"* también estarán contenidos en *"Clase D"*. Siguiendo con el ejemplo, *"Clase B"* es super clase de *"Clase D"*, y *"Clase A"* es super clase de *"Clase B"* y *"Clase C"*.

En general, se podrían definir esquemas de jerarquías de clases en base a dos categorías o formas de herencia:

Herencia simple: Si se siguen reglas de *herencia simple*, entonces *una clase puede tener una y sólo una superclase directa*. El gráfico anterior es un ejemplo de una jerarquía de clases que siguen *herencia simple*. La *Clase D* tiene una sola superclase directa que es *Clase B*. No hay problema en que a su vez esta última derive a su vez desde otra clase, como *Clase A* en este caso. El hecho es que en *herencia simple*, a nivel

de gráfico UML, sólo puede existir *una* flecha que parta desde la clase derivada hacia alguna superclase.

Herencia múltiple: Si se siguen reglas de *herencia múltiple*, entonces *una clase puede tener tantas superclases directas como se desee*. En la gráfica UML, puede haber varias flechas partiendo desde la clase derivada hacia sus superclases. El siguiente esquema muestra una jerarquía en la que hay *herencia múltiple*: note que *Clase D* deriva en forma directa desde las clases *Clase B* y *Clase C*, y esa situación es un caso de herencia múltiple. Sin embargo, note también que la relación que existe entre las *Clase B* y *Clase C* contra *Clase A* es de *herencia simple*; tanto *Clase B* como *Clase C* tienen una y sólo una superclase directa: *Clase A*.

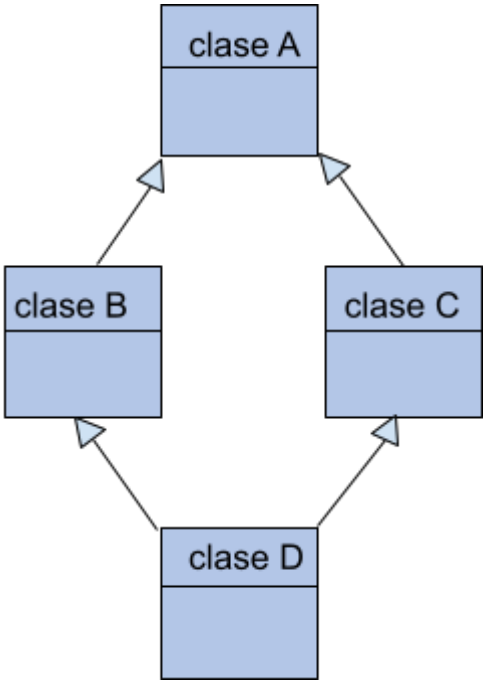

Figura 10: (UML) Diagrama de Clases: Herencia Múltiple

No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la implementación *de interfaces.*

Veamos un ejemplo en Python:

Necesitamos crear dos clases que llamaremos Suma y Resta que derivan de una superclase llamada Operación como sigue:

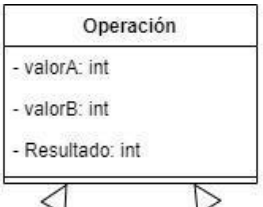

Figura 11: (UML) Diagrama de Clases de las operaciones operaciones Suma y Resta.

En python, seguir los siguientes pasos:

1- Definir la superclase operación como sigue:

```
class Operacion:

    def __init__(self, valorA, valorB):
        self.__valorA = valorA
        self.__valorB = valorB

    @property
    def valorA(self):
        return self.__valorA

    @valorA.setter
    def valorA(self, valor):
        self.__valorA = valor

    @property
    def valorB(self):
        return self.__valorB

    @valorB.setter
    def valorB(self, valor):
        self.__valorB = valor
```

2- Luego, extender las subclases suma y resta como sigue:

```
class Suma(Operacion):

    def __init__(self, valorA, valorB):
        super().__init__(valorA, valorB)

    def sumar(self):
        return self.valorA + self.valorB
```

```
class Resta(Operacion):

    def __init__(self, valorA, valorB):
        super().__init__(valorA, valorB)

    def restar(self):
        return self.valorA - self.valorB
```

3- Crear instancias de la clase suma y resta:

```
# Crear una instancia de la clase Suma
mi_suma = Suma(10, 5)

# Llamar al método sumar
resultado = mi_suma.sumar()

# Imprimir el resultado en consola
print("El resultado es:", resultado)
```

```
# Crear una instancia de la clase Resta
mi_resta = Resta(10, 5)

# Llamar al método restar
resultado = mi_resta.restar()
# Imprimir el resultado en consola
print("El resultado es:", resultado)
```

Agregación y Composición

Las jerarquías de agregación y composición son asociaciones entre clases del tipo “es parte de”.

Para comprenderlo mejor, pensemos en un auto y sus partes. Aquellas partes del auto que son elementales para su existencia y funcionamiento, como por ej. el motor, corresponden a las jerarquías de composición mientras que, aquellas partes que no lo son, ej. radio, corresponden a la jerarquía de agregación.

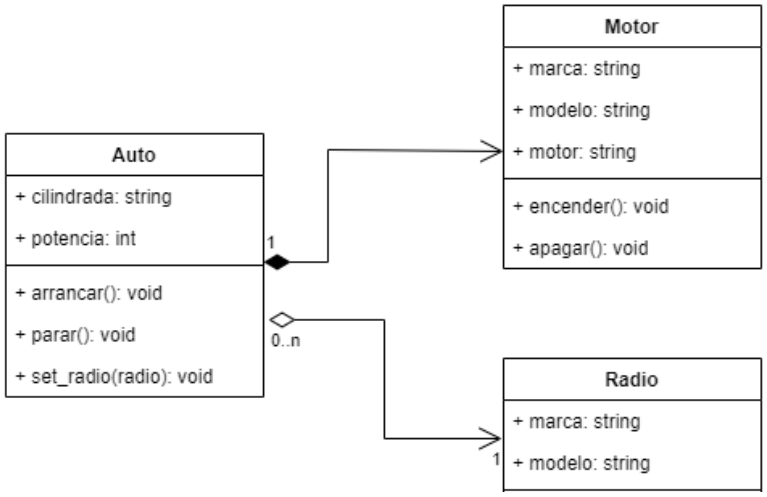

Figura 12: (UML) Diagrama de clases: Agregación y Composición

Ejemplo en Python:

```
class Motor:
    def __init__(self, cilindrada, potencia):
        self.cilindrada = cilindrada
        self.potencia = potencia

    def encender(self):
        print("El motor se ha encendido.")

    def apagar(self):
        print("El motor se ha apagado.")

class Auto:
    def __init__(self, marca, modelo, motor):
       self.marca = marca
```

```
        self.modelo = modelo
        self.motor = motor

    def set_radio(self, radio):
        self.radio = radio

    def arrancar(self):
        print("El auto ha arrancado.")
        self.motor.encender()

    def parar(self):
        print("El auto se ha detenido.")
        self.motor.apagar()

class Radio:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
```

Como puedes observar en el ejemplo, la clase Auto está compuesta por un motor y puede o no tener una radio (observa que no se requiere la radio en el constructor) por ende, el motor es necesario para apagar y encender el vehículo siendo una parte elemental del auto (composición) pero no siendo así, con la radio (agregación). Es decir que, podríamos crear una instancia de Auto sin necesidad de agregar la radio. El mismo, sería capaz de seguir funcionando.

Abstracción

Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

“Una abstracción se centra en la visión externa de un objeto por lo tanto sirve para separar el comportamiento esencial de un objeto de su implementación. La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos. Se puede caracterizar el comportamiento de un objeto de acuerdo a los servicios que presta a otros objetos, así como las operaciones que puede realizar sobre otros objetos”
(http://ceaer.edu.ar/wp-content/uploads/2018/04/Apunte-Teorico-de-Programacion-OO.pdf)

Los mecanismos de abstracción usados en el EOO para extraer y definir las abstracciones son:

“1- La **GENERALIZACIÓN**. Mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las

semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas.

En consecuencia, a través de la generalización:

- La superclase almacena datos generales de las subclases
- Las subclases almacenan sólo datos particulares.

2- La **ESPECIALIZACIÓN** es lo contrario de la generalización. La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.

3- La **AGREGACIÓN**. Mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la descomposición.

4- La **CLASIFICACIÓN**. Consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La ejemplificación es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular"
(http://www.udla.edu.co/documentos/docs/Programas%20Academicos/Tecnologia%20en%20Informatica%20y%20sistemas/Compilados/Compilado%20Programacion%20II.pdf)

La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones.

En resumen, las clases y objetos deberían estar al nivel de abstracción adecuado. Ni demasiado alto ni demasiado bajo.

Encapsulamiento

El encapsulamiento se refiere a la técnica de denegar el acceso directo a los atributos y métodos internos de una clase desde el exterior. A diferencia de otros lenguajes de programación orientados a objetos, Python no tiene una sintaxis específica para definir miembros privados. Sin embargo, se puede simular el encapsulamiento precediendo los atributos y métodos con dos barras bajas __ como indicando que son "especiales". Esto hace que los miembros sean más difíciles de acceder desde el exterior, pero aún pueden ser accedidos si se sabe cómo hacerlo.

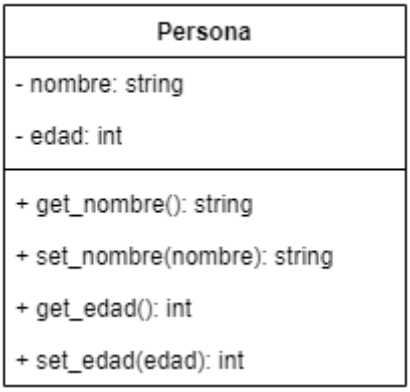

Figura 13: (UML) Diagrama de clases: Encapsulamiento.

A continuación, podemos ver el siguiente ejemplo:

```python
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre
        self.__edad = edad

    def get_nombre(self):
        return self.__nombre

    def set_nombre(self, nombre):
        self.__nombre = nombre

    def get_edad(self):
        return self.__edad

    def set_edad(self, edad):
        self.__edad = edad
```

En este ejemplo, la clase Persona tiene dos atributos privados, __nombre y __edad, que solo pueden ser accedidos desde dentro de la clase. También tiene cuatro métodos, dos para obtener (get) y dos para establecer (set) el valor de cada uno de los atributos. Los métodos get y set son necesarios porque los atributos son privados y no pueden ser accedidos directamente desde fuera de la clase. De esta manera, el encapsulamiento protege los atributos de una clase de ser modificados incorrectamente desde el exterior.

Modularidad

La modularidad se refiere a la técnica de dividir un programa en módulos o componentes más pequeños y manejables, que pueden ser reutilizados en diferentes partes del programa o en diferentes programas. Cada módulo contiene una parte del programa completo y se puede importar a otro módulo para su uso. La modularidad en Python ayuda a mantener el código organizado, facilita la reutilización de código y hace que el proceso de desarrollo sea más eficiente.

A continuación, podemos ver el siguiente ejemplo:

Supongamos que tenemos un programa que necesita realizar algunas operaciones matemáticas como la suma, la resta, la multiplicación y la división. En lugar de escribir todas las funciones en un solo archivo, podemos dividir el programa en módulos separados para cada operación matemática. Podemos crear un archivo **suma.py** que contenga una función sumar que suma dos números, un archivo **resta.py** que contenga una función restar que resta dos números, un archivo **multiplicacion.py** que contenga una función multiplicar que multiplica dos números, y un archivo **division.py** que contenga una función dividir que divide dos números. Luego, podemos importar cada módulo en nuestro programa principal y usar las funciones según sea necesario.

```
# suma.py
def sumar(a, b):
    return a + b

# resta.py
def restar(a, b):
    return a - b

# multiplicacion.py
def multiplicar(a, b):
    return a * b

# division.py
def dividir(a, b):
    if b == 0:
        return "No se puede dividir por cero"
    else:
        return a / b

# programa principal
import suma
import resta
import multiplicacion
import division

a = 10
b = 5

print("La suma de", a, "y", b, "es", suma.sumar(a, b))
print("La resta de", a, "y", b, "es", resta.restar(a, b))
print("La multiplicación de", a, "y", b, "es", multiplicacion.multiplicar(a, b))
print("La división de", a, "y", b, "es", division.dividir(a, b))
```

En este ejemplo, se han creado cuatro módulos separados, cada uno con una función que realiza una operación matemática diferente. Luego, se han importado los módulos en el programa principal y se han utilizado las funciones según sea necesario. Este enfoque de modularidad hace que el código sea más fácil de mantener, ya que cada módulo se puede modificar o actualizar por separado sin afectar al resto del programa. Además, si necesitamos realizar las mismas operaciones matemáticas en otro programa, podemos simplemente importar los módulos existentes en lugar de tener que volver a escribir todo el código.

Polimorfismo

El polimorfismo se refiere a la capacidad de un objeto de tomar diferentes formas o comportarse de diferentes maneras en función del contexto en el que se utiliza. En otras palabras, el polimorfismo permite que un objeto de una clase se comporte como si fuera de otra clase. Esto se puede lograr (entre otras formas) mediante el uso de herencia y sobrescritura de métodos..

Para comprenderlo mejor, muchas veces tenemos una subclase que hereda de una clase base por lo que obtiene todos los métodos, campos, propiedades y eventos de la clase base pero, en ocasiones vamos a necesitar un comportamiento diferente en las clases derivadas (o subclase).

Ejemplo:

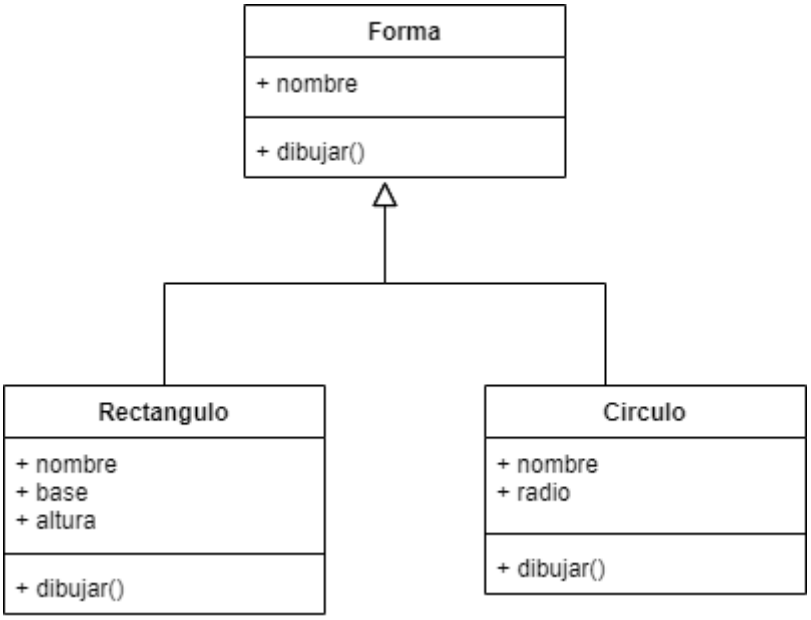

Figura 13. (UML) Diagrama de Clases: Polimorfismo.

Del diagrama de clases anterior podemos deducir que no será lo mismo dibujar un rectángulo que un círculo por lo que el comportamiento deberá ser distinto (polimorfismo).

Ejemplo de polimorfismo en base a herencia en python:

```python
class Forma:
    def __init__(self, nombre):
        self.nombre = nombre

    def dibujar(self):
        pass

class Rectangulo(Forma):
    def dibujar(self):
```

```
        return "dibuja un rectángulo!"

class Circulo(Form):
    def dibujar(self):
        return "dibuja un círculo!"



forma1 = Rectangulo("R1")
forma2 = Circulo("C1")

dibujar(forma1) # Imprime "dibuja un rectángulo!"
dibujar(forma2) # Imprime "dibuja un círculo!"
```

Tipificación

"Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas."
(http://ceaer.edu.ar/wp-content/uploads/2018/04/Apunte-Teorico-de-Programacion-OO.pdf)

Concurrencia

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.
La concurrencia permite a dos objetos actuar al mismo tiempo.

Persistencia

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).
Conserva el estado de un objeto en el tiempo y en el espacio.

Referencias

Apuntes Programa Clip - Felipe Steffolani

https://www.freecodecamp.org/

Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales,
Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela.
Programación Orientada a Objetos.
https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teoria/concepts.html

https://www.typescriptlang.org/

https://barcelonageeks.com/composicion-de-funciones-en-python/

111Mil. Módulo Programación Orientada a Objetos.
https://github.com/111milprogramadores/apuntes/blob/master/Programacion%20Orientada%20a%20Objetos/Apunte%20Teorico%20de%20Programacion%20OO.pdf