ChatGPT

Programación Orientada a Objetos en Python

Colaboración y **composición** son conceptos clave en POO que definen cómo se relacionan los objetos. La **colaboración** ocurre cuando varios objetos se comunican entre sí mediante llamadas a métodos (envío de mensajes) para resolver un problema. Por ejemplo, un objeto `Carrito` puede llamar a métodos de objetos `Producto` para sumar precios, mostrando una colaboración fluida [1] [2]. En contraste, la **composición** es un tipo de relación "todo-parte" en la que un objeto complejo **contiene** internamente otros objetos más simples. Se ilustra con la frase "tiene un" (has-a); por ejemplo, un objeto `Casa` puede estar compuesto por objetos `Habitación` y `Baño` [3] [4].

- **Colaboración entre objetos:** implica que dos o más objetos **cooperan** para resolver una tarea, comunicándose mediante métodos públicos [1]. Cada objeto se enfoca en su responsabilidad y "colabora" llamando métodos del otro. No hay relación de propiedad fuerte; simplemente usan las funcionalidades de los demás.
- **Composición de objetos:** define que un objeto ("todo") **incluye** instancias de otras clases como sus partes (componentes). Cada componente mantiene cierta independencia, pero existe una relación dependiente: si el objeto contenedor se destruye, sus partes suelen destruirse también. Se expresa con "tiene un" [4] [3].
- **Diferencias clave:** en la colaboración, los objetos se relacionan mediante **asociación funcional** (unos usan a otros) sin que uno pertenezca al otro, mientras que en la composición hay **propiedad** y fuerte dependencia "todo-parte". En otras palabras, la colaboración es sobre **interacción** (mensajes entre objetos), y la composición es un **enlace estructural** ("has-a") donde un objeto es parte integral de otro [1] [3].

Por ejemplo, en Python podemos ilustrar ambos conceptos:

```
# Ejemplo de composición y colaboración

class Motor:
    def __init__(self, potencia):
        self.potencia = potencia
    def arrancar(self):
        print(f"Motor de {self.potencia}CV arrancado")

class Coche:
    def __init__(self, potencia_motor):
        # Composición: el Coche tiene (contiene) un Motor
        self.motor = Motor(potencia_motor)
    def conducir(self):
        # Colaboración: el Coche usa al Motor para realizar su función
        self.motor.arrancar()
        print("El coche está en marcha")

class Conductor:
    def conducir_auto(self, coche: Coche):
        # Colaboración: el Conductor llama al método conducir del Coche
        coche.conducir()
```

```
        print("El conductor está conduciendo")

# Uso de las clases
mi_coche = Coche(150)
mi_conductor = Conductor()
mi_conductor.conducir_auto(mi_coche)
```

En este código, `Coche` **composicionalmente** incluye un `Motor` (tiene un motor). Luego `Conductor` **colabora** llamando al método `conducir()` del coche, que a su vez colabora con el motor. Esto ejemplifica cómo la **composición** (parte-todo) y la **colaboración** (interacción funcional) pueden combinarse para resolver un problema sencillo.

Clases en Python: Constructores, Accesores y Mutadores

En Python, las **clases** se definen con la palabra clave `class`. El **constructor** de la clase es el método especial `__init__`, que inicializa los atributos de cada instancia. Por ejemplo, en `def __init__(self, nombre, edad): self.nombre = nombre` se crean los atributos `nombre` y `edad`. Los **métodos accesores** (getters) son funciones que **devuelven** el valor de un atributo, mientras que los **mutadores** (setters) **modifican** ese valor. Aunque Python permite acceder a atributos directamente, es buena práctica usar getters y setters para mantener la encapsulación [5].

- **Constructor (`__init__`)**: método especial que se ejecuta al crear una instancia. Suele recibir parámetros para establecer los atributos iniciales.
- **Métodos accesores (getters)**: devuelven el valor de un atributo de la instancia (por ejemplo, `get_nombre(self)` retorna `self.nombre`). Permiten leer datos internos.
- **Métodos mutadores (setters)**: modifican o actualizan un atributo (por ejemplo, `set_edad(self, edad)` asigna `self.edad = edad`). Facilitan cambiar el estado respetando reglas internas.

```
# Ejemplo de clase con constructor, accesores y mutadores

class Estudiante:
    def __init__(self, nombre, edad):
        self._nombre = nombre   # Atributos con guión bajo indican “privado”
por convención
        self._edad = edad

    # Accesores (getters)
    def get_nombre(self):
        return self._nombre

    def get_edad(self):
        return self._edad

    # Mutadores (setters)
    def set_nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

    def set_edad(self, nueva_edad):
        if nueva_edad >= 0:
```

```
            self._edad = nueva_edad
        else:
            print("Edad inválida")

# Uso de la clase Estudiante
est = Estudiante("Ana", 21)
print(est.get_nombre())   # Imprime "Ana"
est.set_edad(22)
print(est.get_edad())     # Imprime 22
```

Este ejemplo muestra una clase `Estudiante` con constructor y métodos para acceder y modificar sus atributos de manera controlada. La encapsulación se fortalece mediante getters y setters, tal como indican las buenas prácticas [5].

Sobrecarga de Métodos en Python

La **sobrecarga de métodos** es un tipo de polimorfismo que permite a una clase definir varios métodos con el mismo nombre pero **distintas firmas** (diferentes parámetros) [6]. Su utilidad radica en permitir al mismo método funcionar de varias formas según los datos que recibe (por ejemplo, sumar dos números o tres con el mismo nombre de método). Sin embargo, Python **no soporta sobrecarga de métodos de forma nativa** [7]. En lugar de definir múltiples métodos iguales, se suelen usar parámetros opcionales, `*args` o comprobaciones internas para manejar distintos casos en un único método.

- **Polimorfismo por sobrecarga:** conceptualmente permite múltiples comportamientos con el mismo nombre. Es común en lenguajes estáticos como Java o C++ [6], donde el compilador elige el método adecuado según los argumentos.
- **Python y sobrecarga:** en Python la última definición de un método con el mismo nombre reemplaza a las anteriores. Para simular sobrecarga se usan trucos como argumentos por defecto, paquetes de argumentos (`*args`, `**kwargs`), o bibliotecas especiales [7]. Esto mantiene una interfaz simple sin duplicar nombres de método.

```
# Ejemplo de simulación de sobrecarga en Python

class Calculadora:
    def sumar(self, a, b=None):
        # Si se pasan dos números, suma ambos; si sólo uno, lo retorna
directamente
        if b is not None:
            return a + b
        return a

calc = Calculadora()
print(calc.sumar(5, 3))  # Imprime 8 usando dos parámetros
print(calc.sumar(10))    # Imprime 10 usando un solo parámetro (por defecto
b=None)
```

En este código, el método `sumar` admite dos usos: con un argumento o con dos. Internamente evalúa cuántos parámetros recibió y actúa en consecuencia. Así logramos el efecto de **sobrecarga** en Python,

adaptando la lógica según los argumentos. Esto ejemplifica que, aunque Python no tenga sobrecarga implícita [7], se puede ofrecer una interfaz flexible según las necesidades.

Referencias: Se han usado fuentes actualizadas y ejemplos prácticos para ilustrar cada concepto en Python [1] [3] [4] [5] [7]. Cada definición y ejemplo está fundamentado en documentación de POO y guías de Python recientes.

---

[1] [3] [6] ¿Qué es la Programación Orientada a Objetos (POO) y cuáles son sus principios fundamentales? - CodersLink
https://coderslink.com/talento/blog/que-es-la-programacion-orientada-a-objetos-poo-y-cuales-son-sus-principios-fundamentales/

[2] Programación Orientada a Objetos
https://mrebi.com/es/python/python-oop/

[4] Introducción a la programación orientada objetos | PDF
https://es.slideshare.net/slideshow/introduccion-a-la-poo-21547304/21547304

[5] Guía para Principiantes de la Programación Orientada a Objetos (POO) en Python
https://kinsta.com/es/blog/programacion-orientada-objetos-python/

[7] Overloading: Sobrecarga de funciones en Python
https://blog.damavis.com/overloading-sobrecarga-de-funciones-en-python/