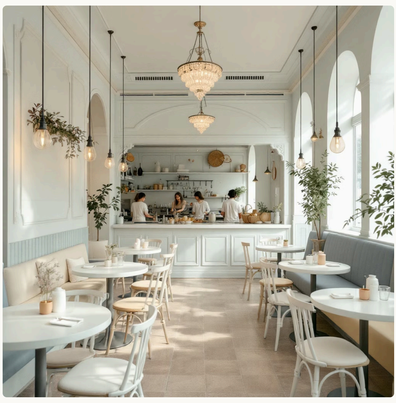
Modelado de un Sistema de Restaurante con Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) ofrece un enfoque robusto para diseñar sistemas complejos, como el de un restaurante. Permite descomponer el problema en entidades manejables (objetos) que interactúan entre sí, facilitando la creación de software modular y fácil de mantener.

¿Qué es la POO y por qué es importante?

La POO organiza el software alrededor de "objetos" que representan elementos del mundo real. Esto mejora la claridad, la reutilización del código y la capacidad de mantenimiento, crucial para sistemas dinámicos.

- **Ejemplo:** Objetos como **Mesa**, **Cliente**, **Pedido** y **Plato**, cada uno con sus datos y comportamientos específicos.



Conceptos clave de la POO aplicados a un restaurante

Aplicar la POO en un sistema de restaurante implica definir y utilizar clases, objetos, atributos y métodos para simular el funcionamiento del negocio.

- **Clases:** Plantillas para crear objetos (ej: **Mesa**, **Pedido**, **Plato**).
- **Objetos:** Instancias de clases (ej: **mesa1** de la clase **Mesa**).
- **Atributos:** Características de una clase (ej: **numero**, **capacidad** en **Mesa**).
- **Métodos:** Acciones que realizan los objetos (ej: **ocupar()** en **Mesa**).

```
class Mesa:
    def __init__(self, numero, capacidad):
        self.numero = numero
        self.capacidad = capacidad
        self.ocupada = False

    def ocupar(self):
        self.ocupada = True

    def liberar(self):
        self.ocupada = False
```

Definición de Clases: Cliente y Pedido

Las clases **Cliente** y **Pedido** demuestran cómo se estructuran las entidades en el sistema.

Clase Cliente

```
class Cliente:
    def __init__(self, nombre, telefono):
        self.nombre = nombre
        self.telefono = telefono

    def hacer_reserva(self, mesa):
        # Lógica para reservar una mesa
        pass
```

Diferencia entre Clase, Instancia y Objeto: La clase es el "molde" (**Mesa**), mientras que la instancia o objeto es el resultado de usar ese molde (ej: **mesa1 = Mesa(1, 4)**).

Clase Pedido y sus Atributos

```
class Pedido:
    def __init__(self, numero, mesa, cliente):
        self.numero = numero
        self.mesa = mesa
        self.cliente = cliente
        self.platos = []
        self.estado = 'pendiente'
```

Los atributos relevantes para un pedido son: **numero**, **mesa**, **cliente**, **platos** (una lista) y **estado**.

Estado, Métodos y Comportamiento de los Objetos

El **estado** de un objeto define su condición actual, mientras que los **métodos** determinan su **comportamiento**.

Estado del Objeto

El estado de un objeto puede cambiar. Por ejemplo, una mesa puede estar "libre" u "ocupada".

```
mesa1 = Mesa(1, 4)
mesa1.ocupar()  # Estado cambia a ocupada
mesa1.liberar() # Estado cambia a libre
```

Diferencia entre Atributo y Estado: Un atributo es una característica fija (ej: **capacidad**), mientras que el estado es el valor actual de un atributo (ej: **ocupada = True**).

Métodos y Comportamiento

Los métodos son funciones que definen las acciones que un objeto puede realizar.

```
class Reserva:
    def realizar_reserva(self, cliente, mesa):
        # Lógica para reservar la mesa
        pass
```

El comportamiento es el efecto de ejecutar uno o varios métodos. Un **Pedido** puede agregar platos y calcular su total.

```
pedido1 = Pedido(1, mesa1, cliente1)
pedido1.agregar_plato(Plato("Hamburguesa", 8.5))
pedido1.calcular_total()
```

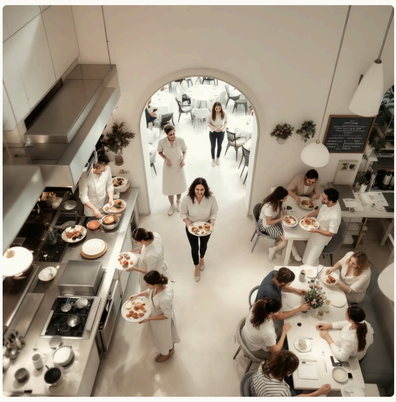
Diferencia entre Método y Comportamiento: El método es la función (**agregar_plato**), y el comportamiento es el resultado (el pedido suma platos y actualiza su total).

Principios Básicos de la POO

Dos pilares fundamentales de la POO son la **Abstracción** y el **Encapsulamiento**, esenciales para la robustez del sistema.

Abstracción

Simplifica el sistema mostrando solo los detalles relevantes. La clase **Pedido**, por ejemplo, abstrae la lógica de gestión de platos y cálculo del total, sin exponer los detalles internos al usuario.



Encapsulamiento

Protege los datos internos de los objetos, permitiendo que sean modificados solo a través de métodos específicos. Esto asegura la integridad de los datos y el control sobre cómo se interactúa con el objeto.

```
class Pedido:
    def __init__(self, numero):
        self.__numero = numero  # atributo privado

    def get_numero(self):
        return self.__numero
```