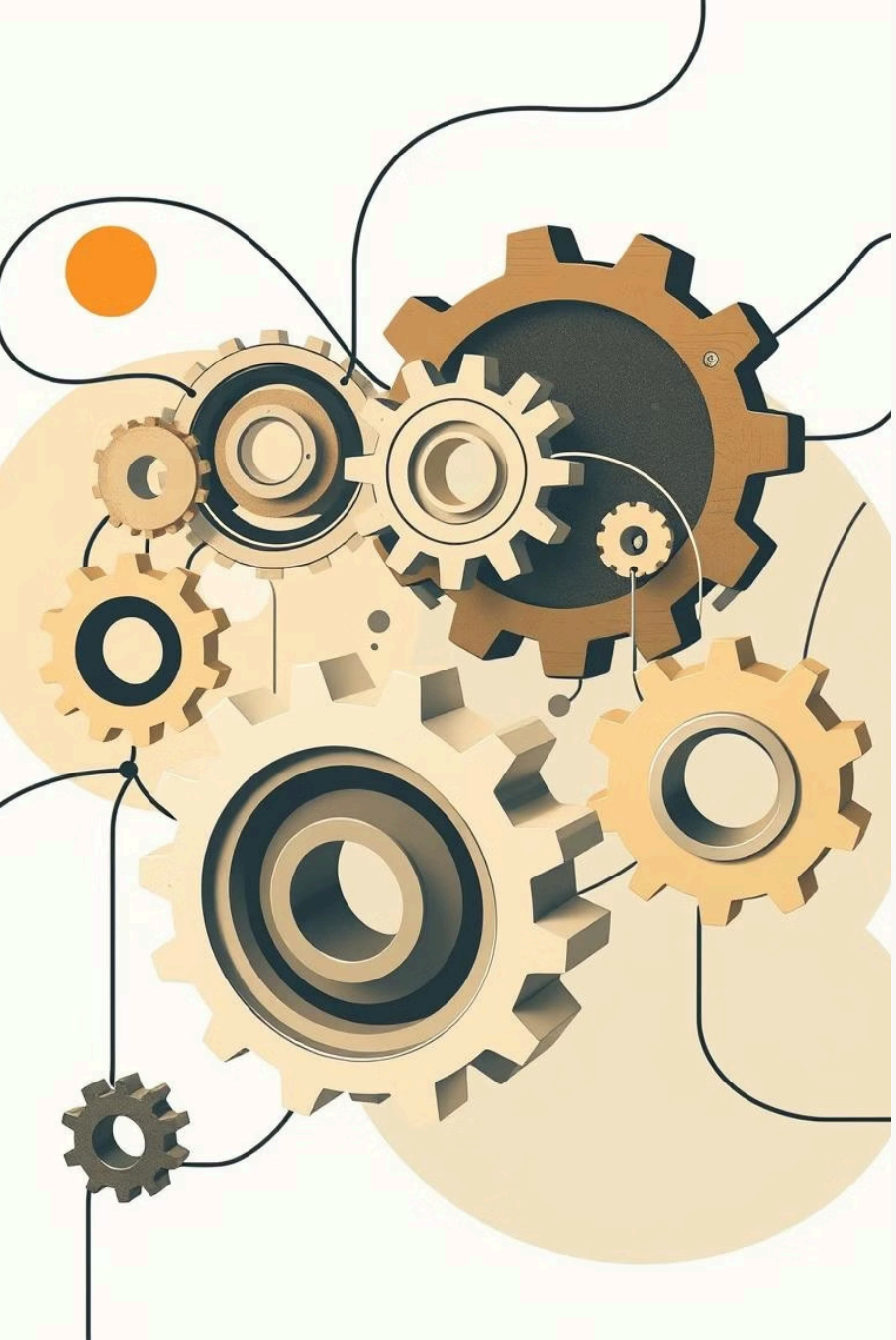

Patrón de Diseño Strategy: Elegancia y Flexibilidad en la Selección de Algoritmos

En el desarrollo de software moderno, la capacidad de adaptar el comportamiento de una aplicación en tiempo real es crucial. El Patrón de Diseño Strategy ofrece una solución elegante para este desafío, permitiendo la definición y el uso intercambiable de algoritmos.

- Permite definir una familia de algoritmos intercambiables.
- El contexto delega la ejecución a un objeto estrategia.
- Cambia el comportamiento en tiempo de ejecución sin modificar el contexto.
- Ideal para evitar condicionales complejos y mejorar mantenimiento.

¿Cómo Funciona el Patrón Strategy? Componentes Clave

El Patrón Strategy se compone de tres elementos principales que trabajan conjuntamente para lograr la flexibilidad deseada: el Contexto, la interfaz de Estrategia y las Estrategias Concretas. Esta estructura facilita la encapsulación de algoritmos y su selección dinámica.

1	**Contexto** Clase principal que mantiene una referencia a un objeto estrategia.
2	**Estrategia (Interface)** Define la interfaz común para todas las estrategias concretas.
3	**Estrategias Concretas** Implementan las diferentes variantes del algoritmo definido por la interfaz Estrategia.

Ejemplo: Ordenación con Estrategias Intercambiables

Imaginemos que necesitamos ordenar una lista de números. Podemos tener diferentes algoritmos de ordenación (burbuja, rápida, etc.). Con el patrón Strategy, el Contexto (la clase que maneja la lista) no necesita saber cómo se ordena, solo delega la tarea a la estrategia que se le asigne.

Esto nos permite cambiar el método de ordenación en tiempo de ejecución, simplemente asignando una nueva estrategia al Contexto. Esto reduce la complejidad del código y lo hace más fácil de mantener y extender.

```
from abc import ABC, abstractmethod

class OrdenacionStrategy(ABC):
 @abstractmethod
 def ordenar(self, data):
 pass

class OrdenacionBurbuja(OrdenacionStrategy):
 def ordenar(self, data):
 n = len(data)
 for i in range(n-1):
 for j in range(0, n-i-1):
 if data[j] > data[j+1]:
 data[j], data[j+1] = data[j+1], data[j]
 return data

class OrdenacionRapida(OrdenacionStrategy):
 def ordenar(self, data):
 if len(data) <= 1:
 return data
 pivot = data[len(data) // 2]
 left = [x for x in data if x < pivot]
 middle = [x for x in data if x == pivot]
 right = [x for x in data if x > pivot]
 return self.ordenar(left) + middle + self.ordenar(right)

class ContextoOrdenacion:
 def __init__(self, strategy: OrdenacionStrategy):
 self._strategy = strategy

 def set_strategy(self, strategy: OrdenacionStrategy):
 self._strategy = strategy

 def ejecutar_ordenacion(self, data):
 print(f"Ordenando con: {type(self._strategy).__name__}")
 return self._strategy.ordenar(data)

if __name__ == "__main__":
 data = [3, 1, 4, 1, 5, 9, 2, 6]

 context = ContextoOrdenacion(OrdenacionBurbuja())
 print(context.ejecutar_ordenacion(data.copy()))

 context.set_strategy(OrdenacionRapida())
 print(context.ejecutar_ordenacion(data.copy()))
```