



Diseño de Sistemas

-Patrones de diseño-

AÑO 2005



- Dolar: \$3,50
- Copas del mundo: 2
- Cantidad de equipos en 1era: 20 (Apertura y Clausura)
- Años de edad de Messi: 18
- Presidente: Nestor Kirchner



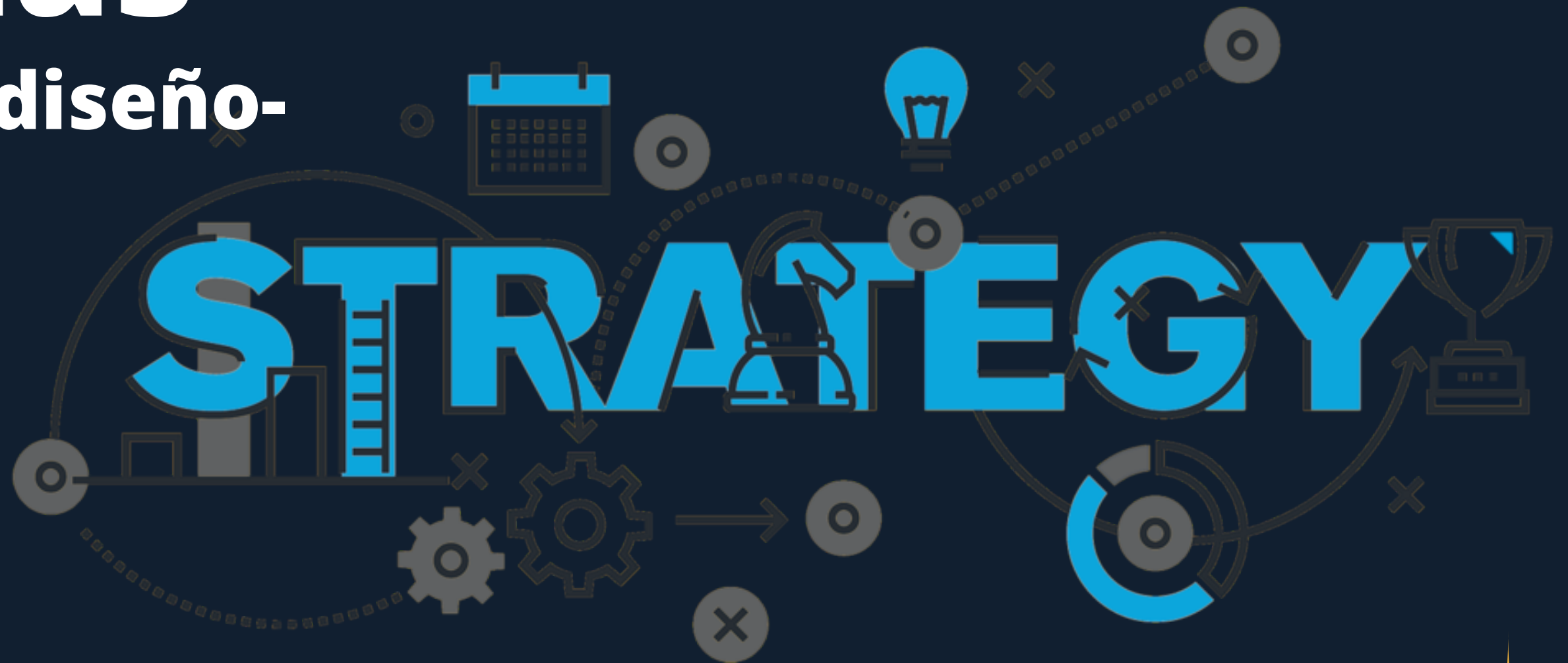


Felicidades, son los creadores de:



Diseño de Sistemas

-Patrones de diseño-





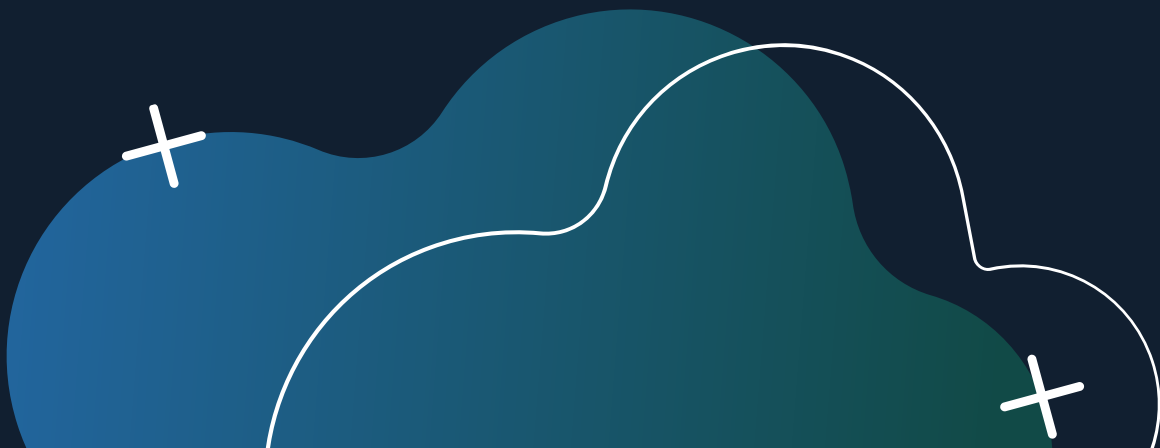
¿De qué vamos a hablar?

- Qué es?
- Cuándo se usa?
- Beneficios
- Estructura
- Ejemplo práctico/codificado
- Comparación con otros patrones
- Conclusión

¿Qué es?

“Strategy” es un *patrón de diseño de comportamiento* que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

**Patrón de diseño de comportamiento: Un patrón de diseño de comportamiento define interacciones claras y eficientes entre objetos para resolver problemas específicos del comportamiento.*





¿Cuándo se usa?

- Cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- Cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
- Para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
- Cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.





Beneficios




-Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.

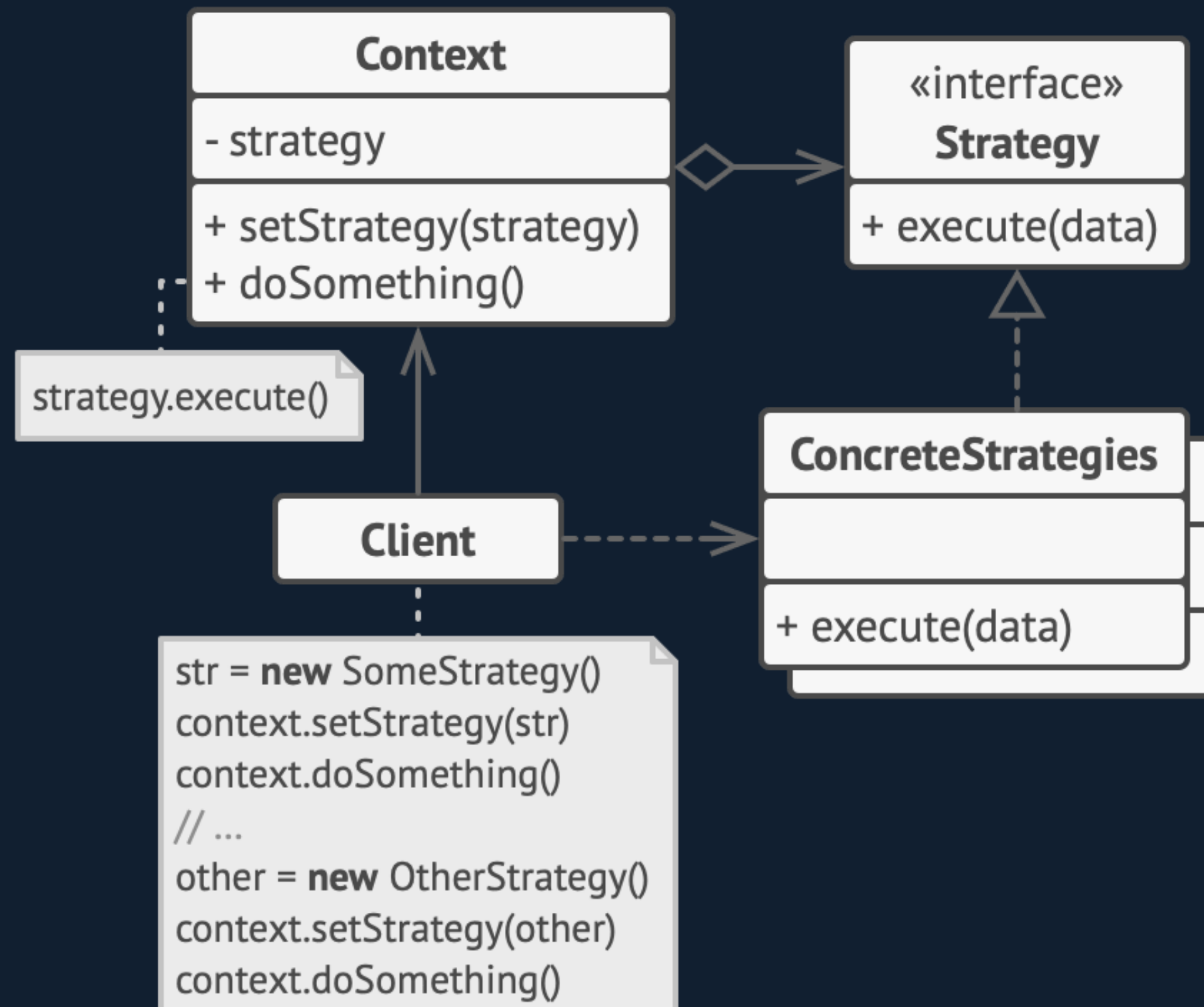
-Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.

-Puedes sustituir la herencia por composición.

-Principio de abierto/cerrado. Puedes introducir nuevas estrategias sin tener que cambiar el contexto.



Estructura





Ejemplo práctico/codificado



Primero, crearemos las clases para las estrategias:

```
ejemplo.py > ...  
1  # Estrategia para conductores  
2  class ConductorStrategy:  
3      def calcular_ruta(self, origen, destino):  
4          print(f"Calculando ruta para conductor desde {origen} hasta {destino} basada en el tiempo.")  
5  
6  # Estrategia para peatones  
7  class PeatonStrategy:  
8      def calcular_ruta(self, origen, destino):  
9          print(f"Calculando ruta para peatón desde {origen} hasta {destino} basada en la distancia.")  
10
```






Ejemplo práctico/codificado

A continuación, crearemos la clase Contexto que utilizará una de las estrategias según el tipo de usuario:

```
11 class NavegacionContexto:
12     def __init__(self, estrategia):
13         self.estrategia = estrategia
14
15     def calcular_ruta(self, origen, destino):
16         self.estrategia.calcular_ruta(origen, destino)
17
```



Ejemplo práctico/codificado

```
18 # Código principal
19 if __name__ == "__main__":
20     tipo_usuario = input("¿Eres conductor o peatón? ").lower()
21
22     if tipo_usuario == "conductor":
23         estrategia = ConductorStrategy()
24     elif tipo_usuario == "peatón":
25         estrategia = PeatonStrategy()
26     else:
27         print("Tipo de usuario no válido.")
28         exit()
29
30     origen = input("Ingrese el punto de origen: ")
31     destino = input("Ingrese el punto de destino: ")
32
33     # Creamos el contexto con la estrategia seleccionada y calculamos la ruta
34     contexto = NavegacionContexto(estrategia)
35     contexto.calcular_ruta(origen, destino)
36
```



Comparación contra otros patrones

- Bridge, State, Strategy (y, hasta cierto punto, Adapter) tienen estructuras muy similares.
- Command y Strategy pueden resultar similares porque puedes usar ambos para parametrizar un objeto con cierta acción.
- Decorator te permite cambiar la piel de un objeto, mientras que Strategy te permite cambiar sus entrañas.





Conclusión

El patrón Strategy permite que las aplicaciones sean flexibles y puedan adaptarse a diferentes escenarios





Gracias