STRATEGY PATTERN

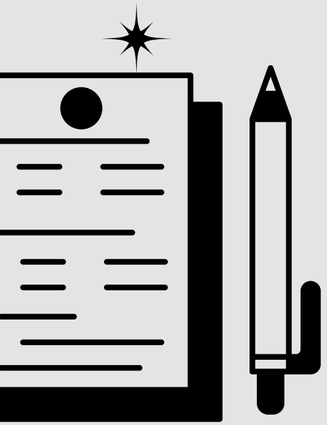
Santiago Cardona López
Juan Camilo Arias Ospina

Sandra Haro

STRATEGY PATTERN

¿En qué consiste?

Es un patrón de diseño de comportamiento que permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Este patrón permite que el algoritmo varíe independientemente de los clientes que lo utilizan, promoviendo la flexibilidad y la reutilización del código



CASO DE ESTUDIO

Supongamos que estamos desarrollando una aplicación que calcula el precio de los boletos de transporte. Dependiendo del tipo de usuario (por ejemplo, adulto, niño, estudiante), se aplicará una estrategia diferente para calcular el precio final del boleto. Utilizaremos el patrón Strategy para encapsular estas estrategias de cálculo de precios

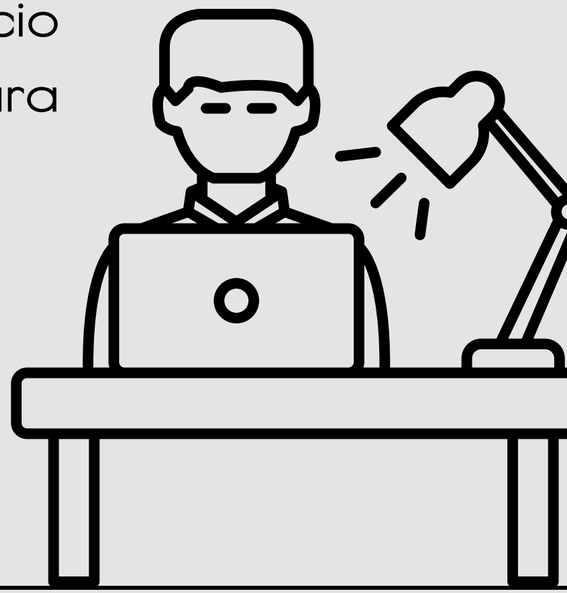


DIAGRAMA DE CLASES

Implementación brindada en el ejemplo

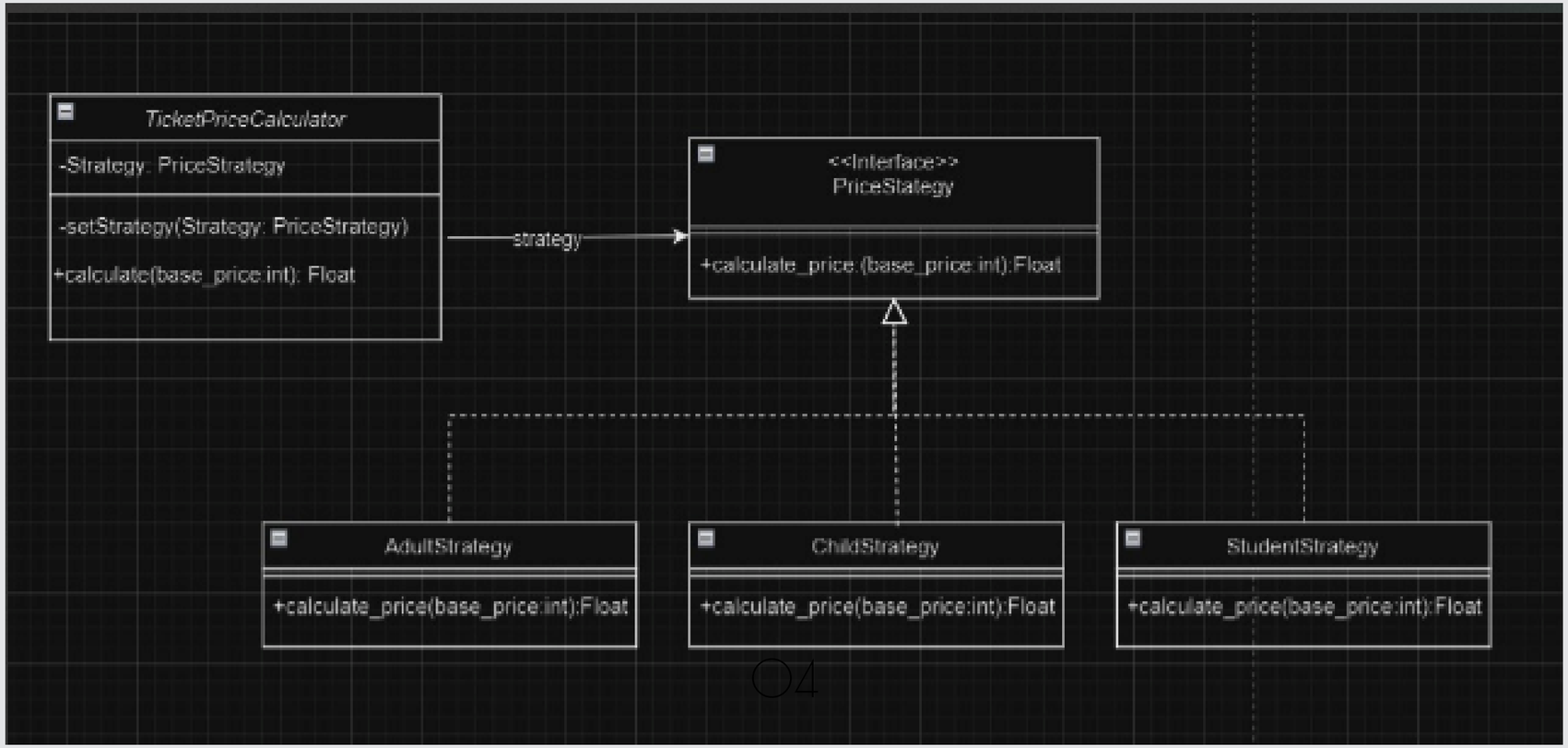
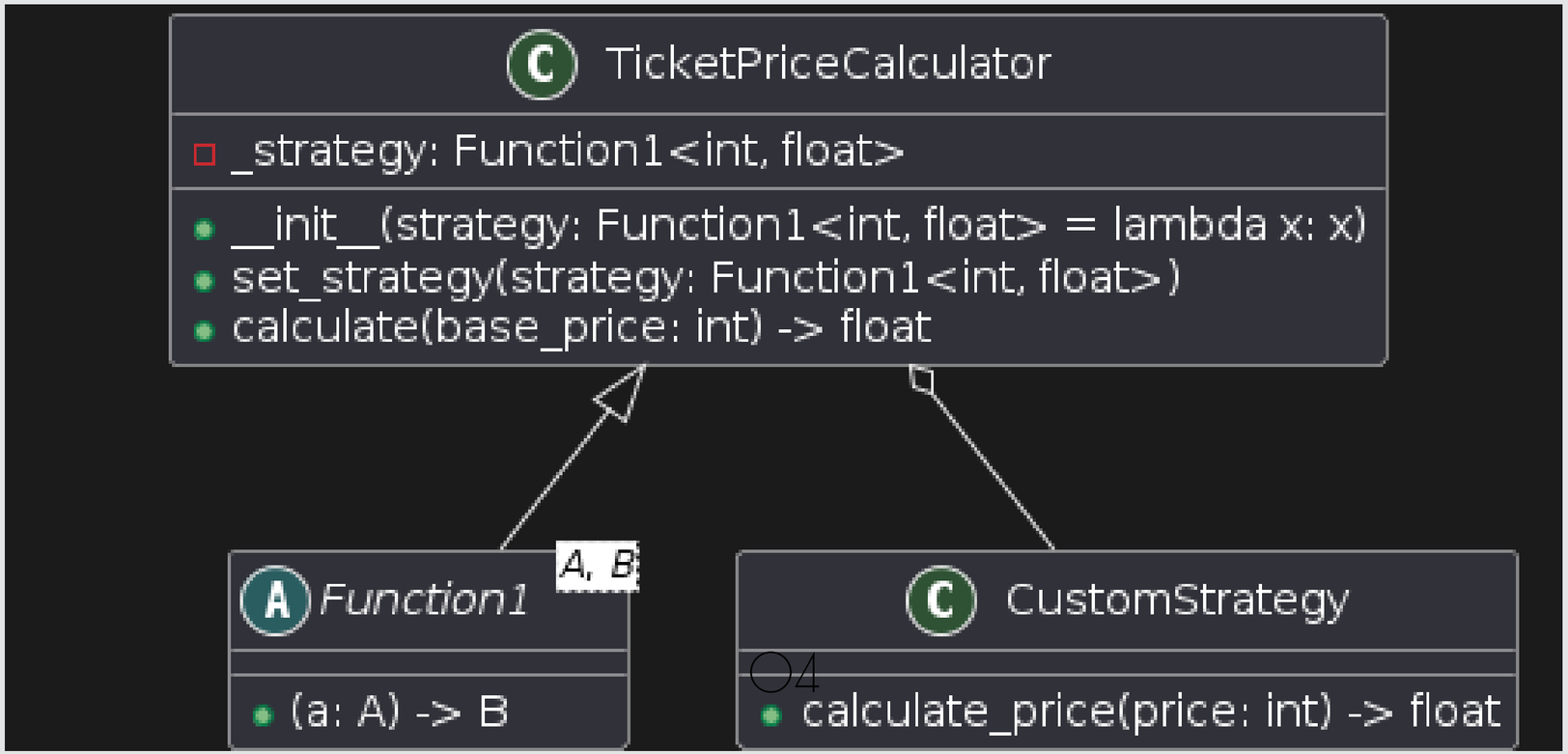


DIAGRAMA DE CLASES

Implementación planteada



COMPARACIÓN DE IMPLEMENTACIONES

Lambdas	Clases
• Más concisa y fácil de escribir. • Ideal para estrategias simples y rápidas	Separa claramente cada estrategia en su propia clase, lo que puede hacer que el código sea más legible y mantenible a largo plazo.
• Menos flexible para estrategias complejas. • Adecuado para funciones pequeñas e independientes.	• Más flexible para implementar estrategias complejas. • Facilita la incorporación de métodos adicionales y atributos específicos de cada estrategia

COMPARACIÓN DE IMPLEMENTACIONES

Lambdas	Clases
• Puede violar el principio de responsabilidad única si las estrategias comienzan a ser más complejas. • No es tan obvio qué hace cada estrategia sin un buen nombre o documentación.	• Sigue mejor el principio de responsabilidad única y el principio abierto/cerrado. • Facilita la adición de nuevas estrategias sin modificar el código existente.
• Generalmente más rápido de escribir y ejecutar para estrategias simples.	• Puede tener una ligera sobrecarga debido a la creación de instancias de clases, pero en la mayoría de los casos, esta diferencia es insignificante..

CÓDIGO PYTHON

Implementación brindada en el ejemplo :

```python
from abc import ABC, abstractmethod
# Strategy interface
class PriceStrategy(ABC):
    @abstractmethod
    def calculate_price(self, base_price: int) -> float:pass

# Concrete strategies
class AdultStrategy(PriceStrategy):
    def calculate_price(self, base_price: int) -> float:return base_price
class ChildStrategy(PriceStrategy):
    def calculate_price(self, base_price: int) -> float:return base_price * 0.5
class StudentStrategy(PriceStrategy):
    def calculate_price(self, base_price: int) -> float:return base_price * 0.8

# Context
class TicketPriceCalculator:
    def __init__(self, strategy: PriceStrategy):self._strategy = strategy
    def set_strategy(self, strategy: PriceStrategy):self._strategy = strategy
    def calculate(self, base_price: int) -> float:return self._strategy.calculate_price(base_price)

# Client
if __name__ == "__main__":
    calculator = TicketPriceCalculator(AdultStrategy())
    print("Adult price:", calculator.calculate(100))
    calculator.set_strategy(ChildStrategy())
    print("Child price:", calculator.calculate(100))
    calculator.set_strategy(StudentStrategy())
    print("Student price:", calculator.calculate(100))
```

CÓDIGO PYTHON

Implementación planteada con Lambdas:

```python
# Context
class TicketPriceCalculator:
    def __init__(self, strategy):
        self._strategy = strategy

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

    def calculate(self, base_price: int) -> float:
        return self._strategy(base_price)

# Client
if __name__ == "__main__":
    calculator = TicketPriceCalculator(lambda x: x)
    print("Adult price:", calculator.calculate(100))

    calculator.set_strategy(lambda x: x * 0.5)
    print("Child price:", calculator.calculate(100))

    calculator.set_strategy(lambda x: x * 0.8)
    print("Student price:", calculator.calculate(100))

```

Casos de uso en la vida cotidiana

CASO # 1

En el mundo de las finanzas, el patrón Estrategia se puede utilizar para implementar diferentes estrategias de inversión. Cada estrategia de inversión se puede implementar como una clase separada que implementa la interfaz Estrategia. Un inversor puede entonces cambiar su estrategia de inversión simplemente cambiando la referencia al objeto Estrategia que está utilizando.

CASO # 2

Elección de atuendo: Una persona puede tener diferentes estrategias de elección de atuendo dependiendo del evento al que asista. Por ejemplo:

Una estrategia de elección de atuendo formal para eventos de negocios o reuniones importantes.

Una estrategia de elección de atuendo casual para eventos sociales o reuniones informales.

Una estrategia de elección de atuendo deportivo para eventos deportivos o actividades físicas.

En este caso, la persona puede cambiar su estrategia de elección de atuendo en tiempo de ejecución, dependiendo del evento al que asista.

Gracias