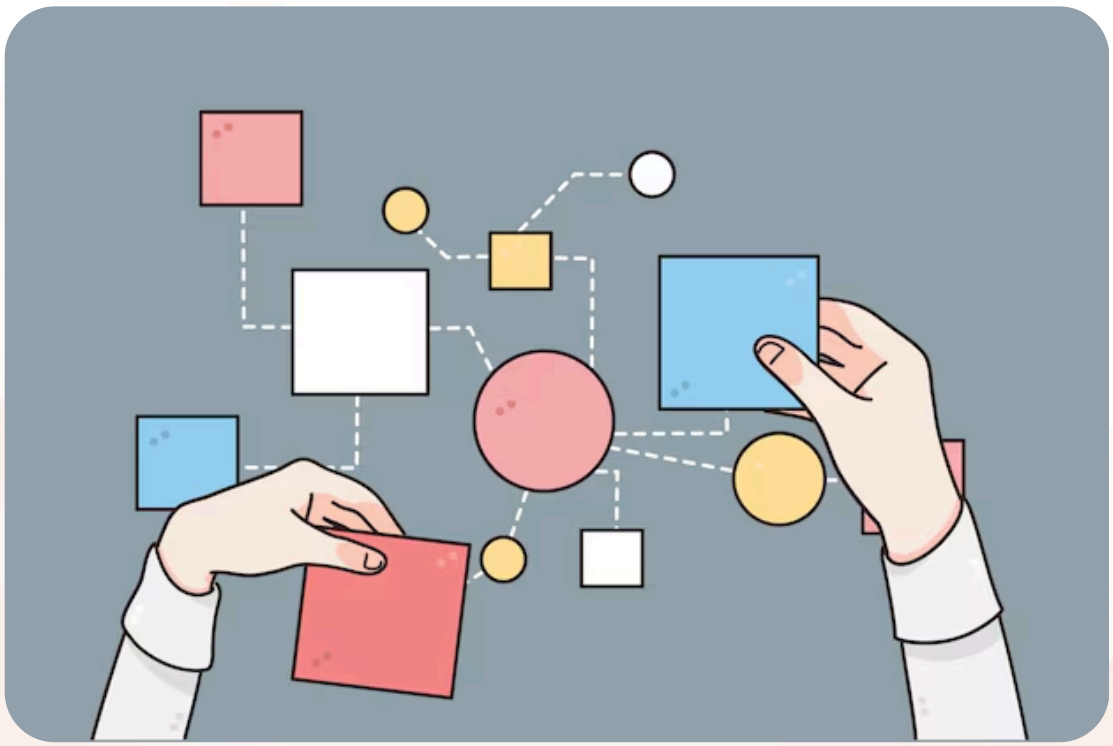
The Strategy Design Pattern



@Ankitz007

Definition

*The **Strategy Pattern** is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from clients that use it, promoting flexibility and maintainability.*

Imagine You're Ordering Pizza 🍕
You want to pay for your pizza, but you have different options:

- *Cash* 💵
- *Credit Card* 💳
- *PayPal* 🏦

Each method does the same thing (pays for the pizza) but in a different way.

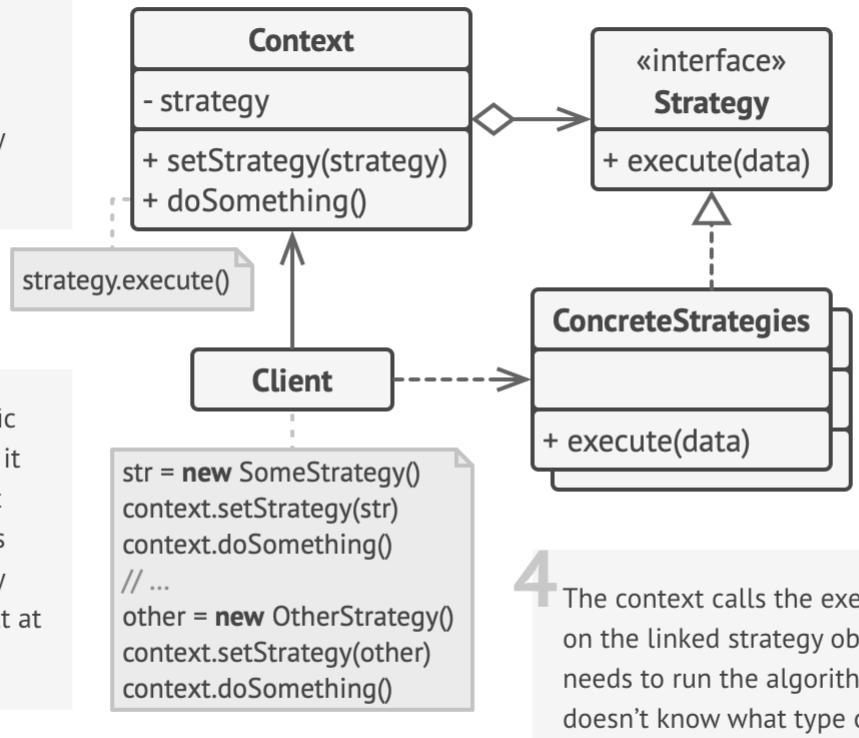
Now, Let's Relate This to Code 💻
The Strategy Pattern is like having different payment methods ready.

You can choose which method to use at runtime, without changing your main program.

@Ankitz007

Structure

1 The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.



2 The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

3 **Concrete Strategies** implement different variations of an algorithm the context uses.

4 The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.

5 The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

Source: *Refactoring.Guru*

@Ankitz007

Key Characteristics

Encapsulation of Behavior *– Instead of hardcoding different behaviors (algorithms) inside a single class, we separate them into different strategy classes.*

- ***Benefit****: Each algorithm is independent, making it easy to modify without affecting others.*

Interchangeability *– All strategies follow the same interface, allowing them to be easily swapped at runtime.*

- ***Benefit****: The client code does not need to change when switching strategies.*

Decoupling *– The main class (Context) doesn't directly implement algorithms; it only references a strategy interface.*

- ***Benefit****: The client is not tightly coupled to any specific algorithm, reducing dependencies.*

Open/Closed Principle *– The system is open for extension but closed for modification–we can add new strategies without modifying existing code.*

- ***Benefit****: No need to alter existing classes when adding new strategies.*

@Ankitz007

When to use?

✅ ***Need Dynamic Algorithm Switching*** *– Enables selecting and switching algorithms dynamically at runtime.*
Use Case*: A navigation app (Google Maps) that switches between Shortest Route, Fastest Route, or Scenic Route based on user preference.*

✅ ***Need Separation of Business Logic*** *– Keeps core logic independent of algorithm implementation details for better maintainability.*
Use Case: *A data compression tool (WinRAR, 7-Zip) that supports ZIP, RAR, and GZIP compression without modifying the core application.*

✅ ***Need Reduced Class Duplication*** *– Merges multiple similar classes by extracting behavior into separate strategy classes.*
Use Case: *A payment gateway (Amazon) with multiple payment methods (Cards, Wallets) handled by separate strategies instead of multiple classes.*

✅ ***Need to Remove Complex Conditional Statements*** *– Replaces large conditional statements with a flexible strategy-based delegation model.*
Use Case: *A loan interest calculation in Banks where they have different interest rates based on loan type (Home, Car, Personal).*

@Ankitz007

When NOT to use?

❌ **When Behavior Variations Are Rare**

- *If you only have two or three strategies, using a Strategy pattern might be overkill.*

❌ **When Performance Overhead Is a Concern**

- *The Strategy pattern introduces additional objects and dynamic method calls.*

❌ **When Clients Should Not Be Aware of Strategy Implementation**

- *If you don't want the client to deal with selecting and managing strategy objects.*

❌ **When Behavior Is Tightly Coupled with Context**

- *If the behavior depends on the internal state of the context, encapsulating it as a separate strategy might break encapsulation.*

❌ **When Concrete Strategies Share Too Much Code**

- *If all strategies share a lot of code, extracting them into separate classes may lead to code duplication.*

@Ankitz007

Code Example

```python
from abc import ABC, abstractmethod

# Strategy Interface
class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

# Concrete Strategies
class CreditCardPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using Credit Card.")

class PayPalPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using PayPal.")

# Context
class ShoppingCart:
    def __init__(self, payment_strategy: PaymentStrategy):
        self.payment_strategy = payment_strategy
    def checkout(self, amount):
        self.payment_strategy.pay(amount)

# Usage
cart1 = ShoppingCart(CreditCardPayment())
cart1.checkout(100)
cart2 = ShoppingCart(PayPalPayment())
cart2.checkout(200)
```


@Ankitz007

Real World Examples

Payment Methods *in Online Shopping (Amazon, eBay, Flipkart, etc.)*

- *Interface: PaymentStrategy*
- *Concrete Strategies: CreditCardPayment, PayPalPayment, etc.*
- *Context: ShoppingCart*

Route Calculation *in Google Maps*

- *Interface: RouteStrategy*
- *Concrete Strategies: CarRoute, BikeRoute, WalkingRoute*
- *Context: NavigationSystem*

Sorting Emails *in Gmail*

- *Interface: EmailSortingStrategy*
- *Concrete Strategies: PrimaryInbox, PromotionsInbox, SpamFilter*
- *Context: EmailClient*

Image Filters *in Instagram / Photoshop*

- *Interface: ImageFilter*
- *Concrete Strategies: BlackAndWhiteFilter, SepiaFilter, BlurFilter*
- *Context: PhotoEditor*

@Ankitz007

Real World Examples

Video Playback Quality *in YouTube / Netflix*

- *Interface: VideoQualityStrategy*
- *Concrete Strategies: AutoQuality, HDQuality, UltraHDQuality, etc.*
- *Context: VideoPlayer*

Difficulty Modes *in games*

- *Interface: DifficultyStrategy*
- *Concrete Strategies: EasyMode, NormalMode, HardMode, ExpertMode*
- *Context: GameEngine*

Text-to-Speech *strategies*

- *Interface: VoiceStrategy*
- *Concrete Strategies: MaleVoice, FemaleVoice, AIEnhancedVoice, etc.*
- *Context: TextToSpeechEngine*

Auto Text Formatting *in Word Processors (MS Word, Google Docs, etc.)*

- *Interface: TextFormatStrategy*
- *Concrete Strategies: BoldFormat, ItalicFormat, UnderlineFormat, etc.*
- *Context: WordProcessor*

@Ankitz007