**📌 التعريف**

**Strategy Pattern** هو من **أنماط السلوك (Behavioral Patterns)** في الـ Design Patterns.  
الفكرة الأساسية:

أن تفصل **الخوارزمية (Algorithm)** عن الكائن الذي يستخدمها، بحيث يمكن تبديل الخوارزميات (أو الإستراتيجيات) أثناء التنفيذ **دون تعديل الكود الأساسي**.

**📌 لماذا نحتاجه؟**

* عندما يكون لدينا عدة طرق (خوارزميات) لتحقيق نفس المهمة.
* بدلاً من كتابة **if/else أو switch** في كل مكان، نضع هذه الخوارزميات في **كائنات مستقلة**، ونمررها ديناميكياً للـ Context (الكائن الرئيسي).

**📌 المكونات الأساسية**

1. **Strategy (الواجهة/Interface)**:  
   تحدد العقد (contract) أو الدوال التي يجب أن تنفذها جميع الاستراتيجيات.
2. **Concrete Strategies (الاستراتيجيات الفعلية)**:  
   كل كلاس يمثل خوارزمية مختلفة ويطبق الواجهة.
3. **Context (السياق)**:  
   الكائن الذي يستخدم الاستراتيجية. يحتفظ بمؤشر/مرجع إلى Strategy، ويمكنه تبديله في أي وقت.

**📌 مثال توضيحي (بايثون)**

نفترض أننا نريد تنفيذ طرق مختلفة لفرز البيانات (**Sorting**):

from typing import List

# 1. Strategy Interface

class SortStrategy:

def sort(self, data: List[int]) -> List[int]:

pass

# 2. Concrete Strategies

class BubbleSort(SortStrategy):

def sort(self, data: List[int]) -> List[int]:

print("Using Bubble Sort")

return sorted(data) # للتبسيط

class QuickSort(SortStrategy):

def sort(self, data: List[int]) -> List[int]:

print("Using Quick Sort")

return sorted(data) # للتبسيط

# 3. Context

class SortContext:

def \_\_init\_\_(self, strategy: SortStrategy):

self.\_strategy = strategy

def set\_strategy(self, strategy: SortStrategy):

self.\_strategy = strategy

def sort(self, data: List[int]) -> List[int]:

return self.\_strategy.sort(data)

# الاستخدام

data = [5, 3, 8, 1, 2]

context = SortContext(BubbleSort())

print(context.sort(data)) # Using Bubble Sort

context.set\_strategy(QuickSort())

print(context.sort(data)) # Using Quick Sort

**📌 الفوائد**

✅ مرونة: تغيير الخوارزمية أثناء التشغيل دون تعديل الكود.  
✅ تقليل التكرار: لا نحتاج if/else ضخمة.  
✅ سهولة إضافة خوارزميات جديدة دون لمس الـ Context.

**📌 أمثلة واقعية**

* في لعبة، يمكن تغيير **سلوك الهجوم (AttackStrategy)** للشخصية: سيف / قوس / سحر.
* في أنظمة الدفع: **PayPalStrategy** ، **CreditCardStrategy** ، **CryptoStrategy**.
* في تطبيق ضغط الملفات: **ZipStrategy** ، **RarStrategy** ، **7zStrategy**.

**📌 التعريف**

**Decorator Pattern** هو من **أنماط البنية (Structural Patterns)** في الـ Design Patterns.  
الفكرة الأساسية:

يسمح بإضافة **وظائف جديدة (Responsibilities)** إلى الكائنات **بشكل ديناميكي** (أثناء التنفيذ) دون الحاجة لتغيير الكود الأصلي أو إنشاء فروع كثيرة من الوراثة.

**📌 لماذا نحتاجه؟**

* عندما نريد **توسيع سلوك الكائنات** بطريقة مرنة وقابلة للتغيير.
* بدلاً من كتابة عشرات الأصناف (subclasses) لكل حالة، يمكننا بناء وظائف إضافية على الكائنات الأصلية **عن طريق تغليفها (Wrapping)**.
* يعطينا **مرونة أكبر** مقارنة بالوراثة (Inheritance).

**📌 المكونات الأساسية**

1. **Component (المكوّن الأساسي)**:  
   الواجهة أو الكلاس الأساسي الذي يحدد السلوك الأصلي.
2. **Concrete Component (المكوّن الفعلي)**:  
   الكائن الأساسي الذي نريد إضافة وظائف جديدة له.
3. **Decorator (المُزخرف الأساسي)**:  
   كلاس يحتوي على مرجع إلى Component، ويطبق نفس الواجهة.
4. **Concrete Decorators (المُزخرفات الفعلية)**:  
   تضيف وظائف جديدة للكائنات عبر إعادة تعريف أو توسيع السلوك.

**📌 مثال توضيحي (بايثون)**

نفترض أن عندنا كائن **قهوة Coffee**، ونريد أن نضيف له إضافات (مثل الحليب والسكر) بدون إنشاء عشرات الأصناف المختلفة:

# 1. Component

class Coffee:

def cost(self):

return 5

# 2. Decorator base

class CoffeeDecorator(Coffee):

def \_\_init\_\_(self, coffee):

self.\_coffee = coffee

def cost(self):

return self.\_coffee.cost()

# 3. Concrete Decorators

class MilkDecorator(CoffeeDecorator):

def cost(self):

return self.\_coffee.cost() + 2

class SugarDecorator(CoffeeDecorator):

def cost(self):

return self.\_coffee.cost() + 1

# الاستخدام

coffee = Coffee()

print("Basic Coffee:", coffee.cost()) # 5

coffee\_with\_milk = MilkDecorator(coffee)

print("Coffee + Milk:", coffee\_with\_milk.cost()) # 7

coffee\_with\_milk\_and\_sugar = SugarDecorator(coffee\_with\_milk)

print("Coffee + Milk + Sugar:", coffee\_with\_milk\_and\_sugar.cost()) # 8

**📌 الفوائد**

✅ مرونة في إضافة وظائف جديدة دون تعديل الكود الأساسي.  
✅ تقليل الحاجة إلى إنشاء أصناف فرعية كثيرة (subclasses).  
✅ يمكن دمج عدة مزخرفات بسهولة (مثل قهوة + حليب + سكر).

**📌 أمثلة واقعية**

* **Java I/O Streams**:  
  BufferedReader يزخرف FileReader ليضيف إمكانية القراءة بكفاءة.
* **واجهة المستخدم (UI Components)**:  
  يمكن زخرفة زر Button ليضيف ظل أو لون أو حركة إضافية.
* **نظام إشعارات**:  
  يمكن زخرفة إشعار عادي ليتم إرساله عبر البريد أو SMS أو Push Notifications.

**📌 التعريف**

**Builder Pattern** هو من **أنماط الإنشاء (Creational Patterns)** في الـ Design Patterns.  
الفكرة الأساسية:

يسمح بإنشاء كائنات معقدة **خطوة بخطوة (Step by Step)**، مع إمكانية التحكم في عملية البناء دون الحاجة إلى Constructor ضخم مليء بالمعاملات.

**📌 لماذا نحتاجه؟**

* عندما يكون لدينا كائن معقد يحتوي على **خيارات متعددة** (Optional Parameters).
* لتجنب **التعقيد** الناتج عن Constructors بعدد كبير من البارامترات.
* للفصل بين **إنشاء الكائن** (Construction) و **تمثيله النهائي** (Representation).

**📌 المكونات الأساسية**

1. **Product (الناتج/الكائن النهائي):**  
   الكائن المعقد الذي نريد إنشاءه (مثل: بيت، سيارة، كمبيوتر).
2. **Builder (الباني):**  
   واجهة أو كلاس يحدد خطوات البناء (Build Part A, Build Part B…).
3. **Concrete Builder (الباني الفعلي):**  
   يطبق خطوات البناء لبناء نسخة محددة من الكائن.
4. **Director (المُوجّه):**  
   مسؤول عن **ترتيب** عملية البناء (يبني الكائن خطوة بخطوة).

**📌 مثال توضيحي (بايثون)**

نفترض أننا نريد بناء **بيت (House)** يحتوي على:

* الجدران (walls)
* السقف (roof)
* المسبح (pool) (اختياري)

# 1. Product

class House:

def \_\_init\_\_(self):

self.walls = None

self.roof = None

self.pool = None

def \_\_str\_\_(self):

return f"House with {self.walls}, {self.roof}, pool: {self.pool}"

# 2. Builder

class HouseBuilder:

def \_\_init\_\_(self):

self.house = House()

def build\_walls(self, walls):

self.house.walls = walls

return self

def build\_roof(self, roof):

self.house.roof = roof

return self

def build\_pool(self, pool=True):

self.house.pool = pool

return self

def get\_result(self):

return self.house

# 3. Director (اختياري)

class Director:

def \_\_init\_\_(self, builder):

self.builder = builder

def build\_simple\_house(self):

return self.builder.build\_walls("Brick Walls").build\_roof("Concrete Roof").get\_result()

def build\_luxury\_house(self):

return self.builder.build\_walls("Marble Walls").build\_roof("Glass Roof").build\_pool(True).get\_result()

# الاستخدام

builder = HouseBuilder()

director = Director(builder)

simple\_house = director.build\_simple\_house()

print(simple\_house) # House with Brick Walls, Concrete Roof, pool: None

luxury\_house = director.build\_luxury\_house()

print(luxury\_house) # House with Marble Walls, Glass Roof, pool: True

**📌 الفوائد**

✅ يفصل **عملية الإنشاء** عن الكائن نفسه.  
✅ يجعل الكود أكثر وضوحًا عند إنشاء كائنات معقدة.  
✅ يدعم **المرونة** (يمكن بناء نسخ مختلفة من الكائن نفسه).  
✅ يقلل من الاعتماد على **Constructors ضخمة**.

**📌 أمثلة واقعية**

* إنشاء كائن **StringBuilder** في Java.
* بناء كائن **Query SQL** بشكل ديناميكي.
* في واجهات المستخدم (UI)، يمكن بناء نافذة مع عناصر مختلفة (أزرار، قوائم، نصوص) حسب الحاجة.

**📌 التعريف**

**🔹 Factory Pattern**

**Factory Pattern** هو من **أنماط الإنشاء (Creational Patterns)**.  
الفكرة الأساسية:

إخفاء تفاصيل **إنشاء الكائنات** عن العميل (Client)، بحيث يطلب العميل كائنًا من المصنع (Factory) ويحصل على الكائن المناسب دون أن يعرف كيف تم إنشاؤه.

**🔹 Factory Method Pattern**

**Factory Method** هو **تطوير لفكرة Factory**:

بدلاً من أن يكون لدينا كلاس Factory واحد فقط، يتم تعريف **واجهة/دالة مصنع** (Factory Method) وتُترك مهمة تحديد الكائن المناسب للأصناف الفرعية (Subclasses).

**📌 لماذا نحتاجهما؟**

* لتقليل **الاعتماد المباشر** على الأصناف (Loose Coupling).
* تسهيل **إضافة أنواع جديدة** من الكائنات دون تغيير الكود الأساسي.
* تحسين **قابلية الصيانة والتوسع**.

**📌 المكونات الأساسية**

1. **Product (المنتج):**  
   الواجهة أو الكلاس الأساسي الذي يمثل الكائنات المراد إنشاؤها.
2. **Concrete Products (المنتجات الفعلية):**  
   الأصناف التي تنفذ المنتج (مثل: Dog, Cat).
3. **Factory / Creator (المصنع):**  
   يحتوي على منطق إنشاء الكائنات.
4. **Concrete Factory / Subclasses:**  
   تقوم بإرجاع كائن من النوع المطلوب.

**📌 مثال توضيحي (بايثون)**

**🔹 Factory Pattern**

# 1. Product

class Animal:

def speak(self):

pass

# 2. Concrete Products

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

# 3. Factory

class AnimalFactory:

@staticmethod

def create\_animal(animal\_type: str) -> Animal:

if animal\_type == "dog":

return Dog()

elif animal\_type == "cat":

return Cat()

else:

raise ValueError("Unknown animal type")

# الاستخدام

factory = AnimalFactory()

dog = factory.create\_animal("dog")

cat = factory.create\_animal("cat")

print(dog.speak()) # Woof!

print(cat.speak()) # Meow!

**🔹 Factory Method Pattern**

# 1. Product

class Document:

def open(self):

pass

# 2. Concrete Products

class WordDocument(Document):

def open(self):

return "Opening Word Document"

class PDFDocument(Document):

def open(self):

return "Opening PDF Document"

# 3. Creator (Factory Method)

class Application:

def create\_document(self):

pass

# 4. Concrete Creators

class WordApp(Application):

def create\_document(self):

return WordDocument()

class PDFApp(Application):

def create\_document(self):

return PDFDocument()

# الاستخدام

word\_app = WordApp()

doc1 = word\_app.create\_document()

print(doc1.open()) # Opening Word Document

pdf\_app = PDFApp()

doc2 = pdf\_app.create\_document()

print(doc2.open()) # Opening PDF Document

**📌 الفوائد**

✅ تقليل الاعتماد المباشر على الأصناف (Loose Coupling).  
✅ تسهيل إضافة أنواع جديدة من الكائنات دون تعديل الكود الأساسي.  
✅ فصل منطق الإنشاء عن الاستخدام.  
✅ أكثر تنظيمًا عند وجود أنواع متعددة من الكائنات.

**📌 أمثلة واقعية**

* **Java:**  
  Calendar.getInstance() → يعيد النوع المناسب حسب المنطقة الزمنية.
* **GUI Frameworks:**  
  زر (Button) في Windows يختلف عن زر في Mac، لكن كليهما ينشأ عبر Factory.
* **Databases:**  
  يمكن اختيار نوع الاتصال (MySQL, PostgreSQL, SQLite) عبر Factory.

🔹 **Adapter Pattern**

**تعريف:**  
Adapter Pattern هو من أنماط الهيكلية (Structural Patterns).  
الفكرة الأساسية:

* السماح لكائن (Client) باستخدام واجهة (Interface) لكائن آخر غير متوافق مع واجهته مباشرة، عن طريق **تغليف الكائن الأصلي في Adapter** يقوم بتحويل الواجهة.
* بمعنى آخر، **Adapter يعمل كجسر** بين واجهة العميل والواجهة التي يقدمها الكائن الحالي.

📌 **لماذا نحتاجه؟**

* لتقليل التعارض بين الأنظمة أو الكائنات المختلفة.
* لتمكين استخدام كائنات قديمة (Legacy) أو مكتبات خارجية دون تعديل الكود الأساسي.
* لتحسين قابلية التوسع وإعادة الاستخدام.

📌 **المكونات الأساسية**

1. **Target (الواجهة المستهدفة):**  
   الواجهة التي يتوقعها العميل.
2. **Client:**  
   يستخدم Target دون أن يعرف عن التفاصيل الداخلية للكائن الذي سيتعامل معه.
3. **Adaptee (الكائن المراد تكييفه):**  
   الكائن الذي يمتلك واجهة مختلفة عن Target.
4. **Adapter:**  
   الكائن الذي يحول استدعاءات العميل إلى استدعاءات يفهمها Adaptee.

📌 **مثال توضيحي (بايثون)**

# 1. Target

class MediaPlayer:

def play(self, filename: str):

pass

# 2. Adaptee

class VLCPlayer:

def play\_vlc(self, filename: str):

print(f"Playing VLC file: {filename}")

class MP4Player:

def play\_mp4(self, filename: str):

print(f"Playing MP4 file: {filename}")

# 3. Adapter

class MediaAdapter(MediaPlayer):

def \_\_init\_\_(self, player\_type: str):

if player\_type == "vlc":

self.player = VLCPlayer()

elif player\_type == "mp4":

self.player = MP4Player()

else:

raise ValueError("Unknown player type")

def play(self, filename: str):

if isinstance(self.player, VLCPlayer):

self.player.play\_vlc(filename)

elif isinstance(self.player, MP4Player):

self.player.play\_mp4(filename)

# 4. Client

class AudioPlayer(MediaPlayer):

def play(self, filename: str, filetype: str):

if filetype == "mp3":

print(f"Playing MP3 file: {filename}")

else:

adapter = MediaAdapter(filetype)

adapter.play(filename)

# الاستخدام

player = AudioPlayer()

player.play("song.mp3", "mp3") # Playing MP3 file: song.mp3

player.play("movie.vlc", "vlc") # Playing VLC file: movie.vlc

player.play("video.mp4", "mp4") # Playing MP4 file: video.mp4

📌 **الفوائد**

✅ السماح بتكامل أنظمة غير متوافقة بسهولة.  
✅ تقليل الحاجة لتعديل كائنات موجودة (Open/Closed Principle).  
✅ تسهيل استخدام مكتبات أو واجهات خارجية.  
✅ تحسين قابلية الصيانة وإعادة الاستخدام.

📌 **أمثلة واقعية**

* **Power adapters** في الواقع: تحويل المقابس لتتناسب مع الأجهزة.
* **Wrappers لمكتبات خارجية** في البرمجة، مثل استخدام مكتبات صوت وفيديو مختلفة عبر واجهة موحدة.
* **واجهات قواعد البيانات**: تحويل استدعاءات قاعدة بيانات قديمة لتتوافق مع ORM حديث.

🔹 **Observer / Listener Pattern**

**تعريف:**  
Observer Pattern هو من أنماط السلوك (Behavioral Patterns).  
الفكرة الأساسية:

* السماح لكائن (Subject) بإعلام عدة كائنات أخرى (Observers/Listeners) عندما تتغير حالته.
* بمعنى آخر، **الموضوع (Subject) يُبقي قائمة بالمشتركين ويحدثهم تلقائيًا عند أي تغيير**.

📌 **لماذا نحتاجه؟**

* لفصل الكائن الذي يرسل التغييرات عن الكائنات التي تستجيب لها (Loose Coupling).
* لتسهيل إضافة أو إزالة المستمعين (Observers) دون تعديل الكود الأساسي.
* لتنفيذ تحديثات تلقائية (Auto-update) في واجهات المستخدم أو الأنظمة التفاعلية.

📌 **المكونات الأساسية**

1. **Subject (الموضوع):**  
   الكائن الذي يحتفظ بالحالة ويبلغ الـObservers عند تغييراتها.
2. **Observer / Listener (المستمع):**  
   الكائن الذي يتلقى إشعارات عند تغييرات الموضوع.
3. **Concrete Subject:**  
   نسخة فعلية من Subject تحتوي على البيانات الفعلية وتحدث Observers عند تغييراتها.
4. **Concrete Observer:**  
   نسخة فعلية من Observer تقوم بتنفيذ رد فعل عند تلقي إشعار من Subject.

📌 **مثال توضيحي (بايثون)**

# 1. Observer (الواجهة)

class Observer:

def update(self, message: str):

pass

# 2. Concrete Observers

class EmailSubscriber(Observer):

def update(self, message: str):

print(f"Email Subscriber received: {message}")

class SMSSubscriber(Observer):

def update(self, message: str):

print(f"SMS Subscriber received: {message}")

# 3. Subject

class Newsletter:

def \_\_init\_\_(self):

self.subscribers = []

def subscribe(self, observer: Observer):

self.subscribers.append(observer)

def unsubscribe(self, observer: Observer):

self.subscribers.remove(observer)

def notify(self, message: str):

for observer in self.subscribers:

observer.update(message)

# الاستخدام

newsletter = Newsletter()

email\_user = EmailSubscriber()

sms\_user = SMSSubscriber()

newsletter.subscribe(email\_user)

newsletter.subscribe(sms\_user)

newsletter.notify("New edition released!")

# Email Subscriber received: New edition released!

# SMS Subscriber received: New edition released!

newsletter.unsubscribe(sms\_user)

newsletter.notify("Another update!")

# Email Subscriber received: Another update!

📌 **الفوائد**

✅ تقليل الاعتماد المباشر بين الكائنات (Loose Coupling).  
✅ تسهيل إضافة أو إزالة Observers في أي وقت.  
✅ تحديث تلقائي عند تغييرات الموضوع (Auto-update).  
✅ مناسب جدًا لتطبيقات واجهة المستخدم، الأنظمة التفاعلية، وإشعارات الأحداث.

📌 **أمثلة واقعية**

* **واجهات المستخدم (GUI):**  
  الضغط على زر Button يرسل إشعار لجميع المستمعين (Listeners) المرتبطين بالزر.
* **إشعارات التطبيقات:**  
  عندما تصل رسالة جديدة أو تغيير في حالة الشبكة، يتم إبلاغ جميع المشتركين.
* **الأسواق المالية:**  
  تحديث الأسعار في الوقت الحقيقي لجميع المشتركين عند تغير الأسعار.

تمام، بنفس الأسلوب، هنا شرح **Singleton Pattern**:

🔹 **Singleton Pattern**

**تعريف:**  
Singleton Pattern هو من أنماط الإنشاء (Creational Patterns).  
الفكرة الأساسية:

* التأكد من أن **كائن واحد فقط** من صنف معين موجود طوال دورة حياة البرنامج.
* توفير **نقطة وصول عالمية (Global Access Point)** لهذا الكائن.

📌 **لماذا نحتاجه؟**

* عندما نريد إدارة موارد مشتركة (مثل الاتصال بقاعدة بيانات أو سجل إعدادات التطبيق).
* لتجنب إنشاء نسخ متعددة من كائن يمثل حالة واحدة مشتركة (Shared State).
* لتسهيل التحكم وإدارة الكائن بشكل مركزي.

📌 **المكونات الأساسية**

1. **Private Constructor (المُنشئ الخاص):**  
   لمنع إنشاء كائنات جديدة من الخارج.
2. **Static / Class Instance (نسخة ثابتة للكائن):**  
   لتخزين الكائن الوحيد وإعادته عند الحاجة.
3. **Public Access Method (طريقة الوصول العامة):**  
   طريقة للحصول على النسخة الوحيدة من الكائن.

📌 **مثال توضيحي (بايثون)**

class Singleton:

\_instance = None # النسخة الوحيدة محفوظة هنا

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

# يمكن تهيئة البيانات هنا

return cls.\_instance

# الاستخدام

obj1 = Singleton()

obj2 = Singleton()

print(obj1 is obj2) # True, كلاهما نفس النسخة

📌 **الفوائد**

✅ ضمان وجود نسخة واحدة فقط من الكائن.  
✅ توفير نقطة وصول عالمية للكائن.  
✅ إدارة الموارد المشتركة بكفاءة.  
✅ منع التعارض أو التكرار عند التعامل مع كائن يمثل حالة مركزية.

📌 **أمثلة واقعية**

* **الاتصال بقاعدة البيانات (Database Connection Pool)**:  
  لا نحتاج أكثر من اتصال مركزي واحد لإدارة كل الطلبات.
* **سجل الإعدادات (Configuration Manager):**  
  لتخزين إعدادات التطبيق في كائن واحد مشترك.
* **طباعه السجلات (Logging):**  
  كائن واحد لإدارة الكتابة إلى ملفات السجل أو وحدة الطباعة.

تمام، بنفس الأسلوب، هنا شرح **State Pattern**:

🔹 **State Pattern**

**تعريف:**  
State Pattern هو من أنماط السلوك (Behavioral Patterns).  
الفكرة الأساسية:

* السماح لكائن (Context) بتغيير سلوكه (Behavior) عند تغير **حالته الداخلية (Internal State)**.
* بمعنى آخر، **الكائن يتصرف بشكل مختلف اعتمادًا على حالته الحالية** دون الحاجة لاستخدام جمل شرطية متعددة (if/else).

📌 **لماذا نحتاجه؟**

* لتقليل التعقيد الناتج عن جمل شرطية كبيرة (if/else أو switch).
* لتسهيل إضافة حالات جديدة دون تعديل الكود الأساسي.
* لفصل منطق كل حالة في كلاس مستقل (Single Responsibility).

📌 **المكونات الأساسية**

1. **Context (السياق):**  
   الكائن الذي يحتوي على الحالة الحالية ويعتمد عليها في سلوكه.
2. **State (الواجهة/الواجهة المجردة):**  
   تحدد واجهة لكل حالة، عادة تشمل دوال يمثل السلوك المتوقع.
3. **Concrete States (الحالات الفعلية):**  
   كائنات تمثل الحالات المختلفة، كل حالة تنفذ السلوك الخاص بها.

📌 **مثال توضيحي (بايثون)**

# 1. State (الواجهة)

class State:

def handle(self, context):

pass

# 2. Concrete States

class LockedState(State):

def handle(self, context):

print("Door is locked. Unlock to open.")

context.state = UnlockedState()

class UnlockedState(State):

def handle(self, context):

print("Door is unlocked. You can open it now.")

context.state = LockedState()

# 3. Context

class Door:

def \_\_init\_\_(self):

self.state = LockedState() # الحالة الافتراضية

def press\_button(self):

self.state.handle(self)

# الاستخدام

door = Door()

door.press\_button() # Door is locked. Unlock to open.

door.press\_button() # Door is unlocked. You can open it now.

door.press\_button() # Door is locked. Unlock to open.

📌 **الفوائد**

✅ تقليل التعقيد الناتج عن جمل شرطية متعددة.  
✅ فصل منطق كل حالة في كلاس مستقل.  
✅ تسهيل إضافة حالات جديدة دون تعديل الكود الأساسي.  
✅ تعزيز مبدأ **Single Responsibility** و **Open/Closed Principle**.

📌 **أمثلة واقعية**

* **الأبواب الذكية:** تتغير حالة الباب بين Locked وUnlocked وOpen اعتمادًا على الإدخال.
* **أجهزة الفيديو/مشغلات الموسيقى:** حالات التشغيل مثل Playing, Paused, Stopped.
* **آلات البيع (Vending Machine):** حالة الانتظار، حالة قبول النقود، حالة إصدار المنتج.

تمام، بنفس الأسلوب، هنا شرح **Model-View-Controller (MVC)**:

🔹 **Model-View-Controller (MVC)**

**تعريف:**  
MVC هو من أنماط التصميم المعمارية (Architectural Patterns).  
الفكرة الأساسية:

* فصل **البيانات والمنطق (Model)** عن **واجهة المستخدم (View)** وعن **إدارة التفاعل (Controller)**.
* كل جزء مسؤول عن مهمة محددة، مما يسهل الصيانة والتوسع وإعادة الاستخدام.

📌 **لماذا نحتاجه؟**

* لفصل منطق الأعمال عن واجهة المستخدم (Separation of Concerns).
* لتسهيل التعديل أو استبدال واجهة المستخدم دون التأثير على منطق البيانات.
* لتحسين تنظيم الكود وإعادة استخدام المكونات المختلفة.
* لدعم أنظمة تفاعلية ومعقدة بطريقة منظمة.

📌 **المكونات الأساسية**

1. **Model (النموذج):**
   * يمثل البيانات والمنطق الأساسي للتطبيق.
   * يقوم بتخزين الحالة وإجراء العمليات على البيانات.
   * لا يهتم بكيفية عرض البيانات للمستخدم.
2. **View (العرض):**
   * مسؤول عن عرض البيانات للمستخدم.
   * يستقبل البيانات من Model ويعرضها بشكل مناسب.
   * لا يحتوي على منطق أعمال.
3. **Controller (المتحكم):**
   * مسؤول عن استقبال مدخلات المستخدم (Input).
   * يحدّد ما يجب فعله عند تفاعل المستخدم مع الواجهة.
   * يتواصل مع Model لتحديث البيانات ويدير View لعرض النتائج.

📌 **مثال توضيحي (بايثون)**

# 1. Model

class Student:

def \_\_init\_\_(self, name, grade):

self.name = name

self.grade = grade

def update\_grade(self, grade):

self.grade = grade

# 2. View

class StudentView:

def display(self, student):

print(f"Student: {student.name}, Grade: {student.grade}")

# 3. Controller

class StudentController:

def \_\_init\_\_(self, model, view):

self.model = model

self.view = view

def set\_grade(self, grade):

self.model.update\_grade(grade)

def show\_student(self):

self.view.display(self.model)

# الاستخدام

student = Student("Ahmed", 90)

view = StudentView()

controller = StudentController(student, view)

controller.show\_student() # Student: Ahmed, Grade: 90

controller.set\_grade(95)

controller.show\_student() # Student: Ahmed, Grade: 95

📌 **الفوائد**

✅ فصل واجهة المستخدم عن منطق البيانات.  
✅ تسهيل الصيانة والتوسع في التطبيقات الكبيرة.  
✅ إعادة استخدام المكونات المختلفة (Model, View, Controller).  
✅ يدعم تطوير الفرق الكبيرة بطريقة منظمة.

📌 **أمثلة واقعية**

* **تطبيقات الويب:** Django, Ruby on Rails تستخدم MVC.
* **تطبيقات سطح المكتب:** Java Swing، PyQt تدعم فصل View وController.
* **الألعاب:** فصل منطق اللعبة (Model) عن الرسم والواجهة (View) وعن التفاعل (Controller).

إذا أحببت، أستطيع أن أرسم لك **مخطط UML مبسط للـ MVC** يوضح علاقة Model وView وController وكيفية التفاعل بينهم، بنفس أسلوب Factory وAdapter وObserver وSingleton وState.

هل تريد أن أرسمه لك؟