IE University BCSAI 2025 CSAI.3.M.B_C3_493621

Software Design Patterns

Sources

- [**Head First Design Patterns**. Kathy Sierra. 2004. O`Reilly](%5BO%60Reilly.%20Head%20First%5D%20-%20Head%20First%20Design%20Patterns%20...%20-%20GitHub%20%20GitHub%20https://raw.githubusercontent.com%20%E2%80%BA%20master%20%E2%80%BA%20%5BO%60...)
- [**Clean Architecture**. Robert C. Martin. O`Reilly](https://agorism.dev/book/software-architecture/%28Robert%20C.%20Martin%20Series%29%20Robert%20C.%20Martin%20-%20Clean%20Architecture_%20A%20Craftsman%E2%80%99s%20Guide%20to%20Software%20Structure%20and%20Design-Prentice%20Hall%20%282017%29.pdf)
- [**Design Patterns: Elements of Reusable Object-Oriented Software**. Erich Gamma. 1994 O`Reilly](https://github.com/deepakkum21/Books/blob/master/Design%20Patterns%20-%20Elements%20of%20Reusable%20Object%20Oriented%20Software%20-%20GOF.pdf)

Introduction

Software design patterns are **reusable solutions** to **recurring challenges** in software development. They provide structured approaches that make systems more maintainable, scalable, and easier to understand.

Patterns are commonly grouped into three categories: **creational**, which focus on flexible and efficient *object creation*; **structural**, which deal with *organizing relationships* between classes and components; and **behavioral**, which *define communication and responsibilities* among objects.

By applying these patterns, developers gain a shared language and proven strategies to build **clearer, more reliable software.**

Creational patterns

> *Focus on flexible and efficient object creation*

Summary

- *Singleton*: one instance only.
- *Factory Method*: delegates object creation to subclasses.
- *Builder*: step-by-step construction of complex objects.

Factory Method Pattern

Key Idea

> *Think of a **restaurant kitchen**. Customers (clients) don't know how dishes are prepared. They just order through the **menu (factory method interface)**. The kitchen (subclasses) decides which cook and which recipe to use.*

The **Factory Method** defines an **interface for creating objects**, but lets **subclasses decide which concrete class to instantiate**.

- Instead of calling a constructor (`new`) directly, you call a **factory method**.
- This **decouples** object creation from object usage.

Why use it?

- To make code **more flexible**: you can introduce new product types without changing client code.
- To follow the **Open/Closed Principle** (open for extension, closed for modification).
- To centralize and control **object creation logic**.

Pros & Cons

Pros

- Encapsulates creation logic in one place.
- Makes it easy to add new product types without modifying client code.
- Promotes loose coupling.

Cons

- Introduces more classes/abstraction, which can make code more complex.
- Sometimes overkill if object creation is simple.

Example

`Transport` is the **product interface**; `Truck` and `Ship` are concrete products.

```
from __future__ import annotations
from abc import ABC, abstractmethod

# ----- Product interface -----
class Transport(ABC):
    @abstractmethod
    def deliver(self, cargo: str) -> None:
        ...

class Truck(Transport):
    def deliver(self, cargo: str) -> None:
        print(f"Delivering '{cargo}' by road in a truck.")

class Ship(Transport):
    def deliver(self, cargo: str) -> None:
        print(f"Delivering '{cargo}' by sea in a ship.")
```

`Logistics` is the **creator** that defines `create_transport()` (the factory method).

`RoadLogistics` and `SeaLogistics` override the factory method to instantiate different products.

```
# ----- Creator (uses factory method) -----
class Logistics(ABC):
    def plan_delivery(self, cargo: str) -> None:
        vehicle = self.create_transport()     # <-- factory method
        print("Planning route...")
        vehicle.deliver(cargo)

    @abstractmethod
    def create_transport(self) -> Transport:
        ...

class RoadLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Truck()

class SeaLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Ship()
```

The client picks which factory to use at runtime (`factory_map`), and the rest of the code stays closed to modification but open to extension.

```
# ----- Client code -----
def main(kind: str, cargo: str) -> None:
    factory_map = {
        "road": RoadLogistics,
        "sea":  SeaLogistics,
    }
    # choose factory at runtime
    logistics: Logistics = factory_map[kind]()
    logistics.plan_delivery(cargo)

if __name__ == "__main__":
    main("road", "Apples")
    main("sea", "Cars")
```

Singleton Pattern

Key Ideas

> *Think of it like a **single CEO of a company** — no matter how many times you ask "who's the CEO?", you always get the same person in charge.*

1. **One and only one object** → No matter how many times you "create" it, you always get the same instance.
2. **Global access** → The instance can be accessed from anywhere in the code (like a shared resource).
3. **Controlled instantiation** → The class itself controls how and when the single instance is created.

Why use it?

- To manage **shared resources** (like a configuration manager, logging system, database connection, or thread pool).
- To enforce **a single point of truth** in the program.
- To avoid conflicts and inconsistencies caused by multiple instances.

Pros & Cons

Pros

- Centralized control of a resource.
- Saves memory (only one object created).
- Easy to access globally.

Cons

- Can make testing harder (since it's a global state).
- Can hide dependencies (code might rely on the singleton without making it obvious).
- Sometimes overused when a simpler solution (like passing an object explicitly) would be better.

Concept

- **Interface** that declares **behaviour** to be implemented.
- **Subclasses** decide **concrete behaviour**.

Example

```
def singleton(cls):
    instances = {}
    def wrapper(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]
    return wrapper

@singleton
class Singleton:
    def __init__(self, value):
        self.value = value

# Example usage
s1 = Singleton(42)
s2 = Singleton(99)

print(s1 is s2)       # True
print(s1.value)       # 42
print(s2.value)       # 42 (same instance as s1)
```

Builder Pattern

Key Idea

> *Think of building a **custom burger** at a restaurant: The **builder** (chef) adds ingredients step by step. The **director** (waiter/menu) controls the order of steps.*

The **Builder pattern** separates the **construction of a complex object** from its **representation**.

- Instead of a huge constructor with many optional parameters, you build the object **step by step**.
- The same building process can create **different representations**.

Why use it?

- To construct objects that need **many configuration options**.
- To avoid **telescoping constructors** (constructors with many parameters that are hard to read/maintain).

- To make object creation **more readable and flexible**.

⚖️ Pros & Cons

✅ Pros

- Clean and readable object creation.
- Can reuse the same construction process for different outputs.
- Encapsulates construction logic in one place.

❌ Cons

- Adds extra classes (builder, director, etc.).
- Can be overkill if objects are simple.

Example

`Computer` class defines what are the **attributes** of the entity to be build.

```
# Product
class Computer:
    def __init__(self):
        self.cpu = None
        self.gpu = None
        self.memory = None
        self.storage = None

    def __str__(self):
        return f"CPU: {self.cpu}, GPU: {self.gpu}, RAM: {self.memory}, Storage: {self.storage}"
```

`ComputerBuilder` class defines which steps or **behaviours** should be defined to create `Computer`.

```

# Builder
class ComputerBuilder:
    def __init__(self):
        self.computer = Computer()

    def set_cpu(self, cpu):
        self.computer.cpu = cpu
        return self  # allows chaining

    def set_gpu(self, gpu):
        self.computer.gpu = gpu
        return self

    def set_memory(self, memory):
        self.computer.memory = memory
        return self

    def set_storage(self, storage):
        self.computer.storage = storage
        return self

    def build(self):
        return self.computer
```

`Director` is helper class that impose execution order for the builder.

```
# Director (optional: encapsulates construction order)
class Director:
    def __init__(self, builder: ComputerBuilder):
        self.builder = builder

    def build_gaming_pc(self):
        return (self.builder
                .set_cpu("Intel i9")
                .set_gpu("RTX 4090")
                .set_memory("32GB")
```

```
                .set_storage("2TB SSD")
                .build())

    def build_office_pc(self):
        return (self.builder
                .set_cpu("Intel i5")
                .set_gpu("Integrated")
                .set_memory("16GB")
                .set_storage("512GB SSD")
                .build())
```

Finally we can call the `Director` instance to create different kind of `Computer` objects.

```
# Client code
builder = ComputerBuilder()
director = Director(builder)

gaming_pc = director.build_gaming_pc()
office_pc = director.build_office_pc()

print("Gaming PC:", gaming_pc)
print("Office PC:", office_pc)
```

Output of the above code:

```
Gaming PC: CPU: Intel i9, GPU: RTX 4090, RAM: 32GB, Storage: 2TB SSD
Office PC: CPU: Intel i5, GPU: Integrated, RAM: 16GB, Storage: 512GB SSD
```

Creational patterns

> *deal with organizing relationships between classes and components*

Summary

- *Adapter* → makes incompatible interfaces work together.
- *Proxy* → controls access to another object (lazy load, security, caching, etc.).
- *Facade* → provides a simplified interface to a complex subsystem.

Adapter Pattern

Key Idea

> *Think of a **power adapter**: Your laptop plug (client) expects a certain socket shape (target interface). The wall outlet (adaptee) has a different shape. The adapter converts between them so the laptop can charge.*

The **Adapter pattern** allows **incompatible interfaces** to work together.

- It acts as a **translator** between two classes that couldn’t normally interact.
- The client uses a **target interface**, and the adapter makes an **adaptee** fit into it.

Why use it?

- To integrate **legacy code** or **third-party libraries** with new systems.
- To avoid rewriting existing code when interfaces don’t match.
- To promote **reusability** by making different APIs work together.

Pros & Cons

Pros

- Increases flexibility: lets you use existing classes without modification.
- Promotes code reuse and compatibility.
- Makes systems easier to extend.

Cons

- Adds extra layers → can introduce complexity.
- If overused, may hide the need for a better overall design.

Example

Suppose we have a modern app that expects a `PaymentProcessor` interface, but we need to integrate with a **legacy PayPal system**.

Target = `PaymentProcessor` (what our system expects).

```
class  PaymentProcessor:
    def  pay(self, amount):
        raise NotImplementedError
```

Adaptee = `PayPal` (existing class with a different method).

```
# Adaptee (legacy class with incompatible interface)
class  PayPal:
    def  send_payment(self, amount):
        print(f"Processing ${amount} payment through PayPal")
```

Adapter = `PayPalAdapter` (wraps PayPal to make it compatible).

```
class  PayPalAdapter(PaymentProcessor):
    def  __init__(self, paypal: PayPal):
        self.paypal = paypal

    def  pay(self, amount):
        # Translate the call
        self.paypal.send_payment(amount)
```

Client = `complete_purchase()` (works only with the target interface, not with legacy code directly).

```
def complete_purchase(processor: PaymentProcessor, amount: int):
    processor.pay(amount)

# Usage paypal = PayPal()
processor = PayPalAdapter(paypal)
complete_purchase(processor, 100)
```

Output:

```
`Processing  $100 payment through PayPal`
```

Proxy Pattern

Key Idea

> *Think of a **personal assistant (proxy)**: Instead of calling the CEO directly (real subject), you ask the assistant. The assistant decides if your request is important enough to forward, schedules it, or blocks it.*

The **Proxy pattern** provides a **surrogate or placeholder** for another object to **control access** to it.

- A proxy object has the **same interface** as the real object.
- The proxy decides whether to forward the request to the real object, add extra behavior, or restrict access.

Why use it?

- To **control access** to sensitive or resource-heavy objects.
- To implement **lazy initialization** (create the real object only when needed).
- To add **logging, caching, or security checks** transparently.
- To work with **remote objects** as if they were local.

Pros & Cons

Pros

- Adds an extra layer of control without changing client code.
- Can optimize performance (e.g., lazy loading, caching).
- Can enhance security (access control, permissions).

 Cons

- Adds complexity (extra classes).
- Can introduce latency if misused (extra indirection).
- If overused, may make debugging harder.

Example

Suppose we have a heavy `Video` class for playing videos. We use a proxy to **delay loading** until needed.

Subject interface = `Video`.

```
from abc import ABC, abstractmethod

# Subject interface
class Video(ABC):
    @abstractmethod
    def play(self):
        pass
```

RealSubject = `RealVideo` (expensive to create).

```
# Real Subject
class RealVideo(Video):
    def __init__(self, filename):
        self.filename = filename
        self.load_video()  # Expensive operation

    def load_video(self):
        print(f"Loading video file: {self.filename}")

    def play(self):
        print(f"Playing video: {self.filename}")
```

Proxy = `ProxyVideo` (controls access, loads lazily).

```
# Proxy
class ProxyVideo(Video):
    def __init__(self, filename):
        self.filename = filename
        self._real_video = None

    def play(self):
        if self._real_video is None:
            # Lazy initialization
            self._real_video = RealVideo(self.filename)
        self._real_video.play()
```

Client = interacts with the proxy as if it were the real object.

```
# Client code
video1 = ProxyVideo("movie.mp4")
video2 = ProxyVideo("trailer.mp4")

# Only loads when actually played
video1.play()
video2.play()
video1.play()  # Already loaded, just plays
```

Output:

```
Loading  video  file: movie.mp4
Playing  video: movie.mp4
Loading  video  file: trailer.mp4
Playing  video: trailer.mp4
Playing  video: movie.mp4`
```

Facade Pattern

🔑 Key Idea

> *Think of a **universal remote control (facade)**: The TV, DVD player, and sound system each have their own complex controls (subsystems). Instead of using all three remotes, you use **one remote** that provides a **simple interface**: Power, Play, Volume.*

The **Facade pattern** provides a **simplified interface** to a **complex subsystem**.

- Instead of exposing many detailed classes and methods, the facade exposes a **unified, easy-to-use API**.
- Clients interact with the facade, not directly with the subsystem.

🎯 Why use it?

- To **simplify usage** of a complicated library/framework.
- To reduce **coupling** between clients and subsystems.
- To provide a **single entry point** to a system.
- To make the code more **readable and maintainable**.

⚖️ Pros & Cons

✅ Pros

- Shields clients from subsystem complexity.
- Makes code easier to use and understand.
- Reduces dependencies (clients only depend on the facade).
- Encourages loose coupling.

❌ Cons

- Can become a “god object” if it grows too large.
- May limit access to advanced features of the subsystem.
- Adds another abstraction layer.

Example

Suppose we have a **home theater system** with multiple components: **Subsystems** = `DVDPlayer`, `Projector`, `SoundSystem`.

```
class DVDPlayer:
    def on(self): print("DVD Player ON")
    def play(self, movie): print(f"Playing movie: {movie}")

class Projector:
    def on(self): print("Projector ON")
    def set_input(self, source): print(f"Projector input set to {source}")

class SoundSystem:
    def on(self): print("Sound System ON")
    def set_volume(self, level): print(f"Volume set to {level}")
```

Facade = `HomeTheaterFacade`, which provides a single, simple method `watch_movie()`.

```
class HomeTheaterFacade:
    def __init__(
        self,
        dvd: DVDPlayer,
        projector: Projector,
        sound: SoundSystem
    ):
        self.dvd = dvd
        self.projector = projector
        self.sound = sound

    def watch_movie(self, movie):
        print("Get ready to watch a movie...")
        self.dvd.on()
        self.projector.on()
        self.projector.set_input("DVD")
```

```
        self.sound.on()
        self.sound.set_volume(20)
        self.dvd.play(movie)
```

Client = interacts only with the facade, not with the complex subsystems.

```
dvd = DVDPlayer()
projector = Projector()
sound = SoundSystem()

home_theater = HomeTheaterFacade(dvd, projector, sound)
home_theater.watch_movie("Inception")
```

Behavioral patterns

> *define communication and responsibilities among objects.*

Summary

- *Observer* → defines one-to-many dependency for automatic notifications.
- *Strategy* → interchangeable algorithms chosen at runtime.
- *Command* → encapsulates requests as objects (supports undo, queue, log).

Observer Pattern

Key Idea

> *Think of a **YouTube channel**: The channel is the **subject** (publisher). Then subscribers are the **observers**. When the channel uploads a new video, all subscribers **get notified automatically**.*

The **Observer pattern** defines a **one-to-many dependency** between objects.

- When the **subject** (or publisher) changes state, all its **observers** (or subscribers) are **notified automatically**.
- Promotes a **publish–subscribe** mechanism without tight coupling.

Why use it?

- To implement **event-driven systems** (GUI, messaging, notifications).
- To keep objects **in sync** without hard dependencies.
- To allow **dynamic subscription** and **loose coupling**.

Pros & Cons

Pros

- Decouples subject from observers.
- Easy to add/remove observers at runtime.
- Useful for broadcast-style communication.

Cons

- Can lead to **unexpected updates** if too many observers.
- If not managed carefully, may cause **memory leaks** (dangling observers).
- Debugging event chains can be tricky.

Example

Let’s build a simple stock price notifier system:

Observers = `EmailNotifier`, `SMSNotifier`.

```
from abc import ABC, abstractmethod

# Observer interface
class Observer(ABC):
    @abstractmethod
    def update(self, price):
        pass

# Concrete Observers
```

```
class EmailNotifier(Observer):
    def update(self, price):
        print(f"[Email] Stock price updated to {price}")

class SMSNotifier(Observer):
    def update(self, price):
        print(f"[SMS] Stock price updated to {price}")
```

Subject = `Stock` (maintains state and observers).

```
class Stock:
    def __init__(self):
        self._observers = []
        self._price = None

    def attach(self, observer: Observer):
        self._observers.append(observer)

    def detach(self, observer: Observer):
        self._observers.remove(observer)

    def set_price(self, price):
        print(f"\nStock price set to {price}")
        self._price = price
        self._notify()

    def _notify(self):
        for observer in self._observers:
            observer.update(self._price)
```

Client = attaches/detaches observers dynamically.

```
stock = Stock()
email = EmailNotifier()
sms = SMSNotifier()

stock.attach(email)
stock.attach(sms)

stock.set_price(100)
stock.set_price(105)

stock.detach(sms)
stock.set_price(110)
```

Output

```
Stock price set to 100
[Email] Stock price updated to 100
[SMS] Stock price updated to 100

Stock price set to 105
[Email] Stock price updated to 105
[SMS] Stock price updated to 105

Stock price set to 110
[Email] Stock price updated to 110
```

Strategy Pattern

Key Idea

> *Think of a **navigation app**: The app = **context**. Different routing methods (car, bike, walk) = **strategies**. You can switch from "shortest route" to "scenic route" without rewriting the app — just change the strategy.*

The **Strategy pattern** defines a **family of algorithms**, encapsulates each one, and makes them **interchangeable at runtime**.

- The client chooses which **strategy (algorithm)** to use without changing its own code.
- Promotes **composition over inheritance**: behavior is delegated to the chosen strategy object.

Why use it?

- To **switch between algorithms dynamically** (e.g., different sorting methods).
- To avoid long conditional logic (`if/elif` chains for algorithm selection).
- To follow the **Open/Closed Principle** (new strategies can be added without modifying existing code).

Pros & Cons

Pros

- Makes algorithms interchangeable.
- Keeps client code clean and focused.
- Encourages reusable, testable, and maintainable algorithms.

Cons

- Increases the number of classes/objects.
- The client must be aware of different strategies and when to use them.
- Can be overkill for simple problems.

Example

Suppose we want different payment strategies in an e-commerce app:

Strategy interface = `PaymentStrategy`.

```
from abc import ABC, abstractmethod

class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount):
        pass
```

Concrete strategies = `CreditCardPayment`, `PayPalPayment`, `BitcoinPayment`.

```
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")

class BitcoinPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using Bitcoin")
```

Context = `ShoppingCart`, which delegates payment to the chosen strategy.

```
class ShoppingCart:
    def __init__(self, payment_strategy: PaymentStrategy):
        self.payment_strategy = payment_strategy
        self.items = []

    def add_item(self, item, price):
        self.items.append((item, price))

    def checkout(self):
        total = sum(price for _, price in self.items)
        self.payment_strategy.pay(total)
```

Client = selects which strategy to use at runtime.

```
cart1 = ShoppingCart(CreditCardPayment())
cart1.add_item("Book", 20)
```

```
cart1.add_item("Pen", 5)
cart1.checkout()

cart2 = ShoppingCart(PayPalPayment())
cart2.add_item("Shoes", 50)
cart2.checkout()

cart3 = ShoppingCart(BitcoinPayment())
cart3.add_item("Laptop", 1000)
cart3.checkout()
```

Command Pattern

Key Idea

> *Think of a **restaurant order system**: The customer = **invoker** (wants food). The waiter writes the order = **command** (encapsulates the request). The chef = **receiver** (executes the order). The customer doesn't talk directly to the chef, and the waiter can queue, cancel, or reorder requests.*

The **Command pattern** turns a **request (an action)** into a **standalone object**.

- The object (command) encapsulates **what action to perform, and on which receiver**.
- This allows actions to be **queued, logged, undone/redone, or executed later**.

Why use it?

- To implement **undo/redo functionality**.
- To **queue or schedule** operations.
- To **log and replay** actions (e.g., macros).
- To decouple **senders** (who request an action) from **receivers** (who perform the action).

Pros & Cons

Pros

- Decouples request invoker from request executor.
- Supports undo/redo and history of actions.
- Makes it easy to add new commands without changing existing code.
- Can support macros (combine multiple commands).

Cons

- Increases the number of classes (one per command).
- Can add complexity if commands are very simple.

Example

Let's make a **text editor** with undo support: **Command interface** = `Command`.

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

    @abstractmethod
    def undo(self):
        pass
```

Receiver = `TextEditor` (knows how to perform the action).

```
class TextEditor:
    def __init__(self):
        self.text = ""

    def write(self, content):
        self.text += content

    def erase(self, count):
```

```
        self.text = self.text[:-count]

    def __str__(self):
        return self.text
```

Concrete command = `WriteCommand`.

```
class WriteCommand(Command):
    def __init__(self, editor: TextEditor, content: str):
        self.editor = editor
        self.content = content

    def execute(self):
        self.editor.write(self.content)

    def undo(self):
        self.editor.erase(len(self.content))
```

Invoker = `EditorInvoker` (executes/undoes commands).

```
# Invoker
class EditorInvoker:
    def __init__(self):
        self.history = []

    def execute_command(self, command: Command):
        command.execute()
        self.history.append(command)

    def undo(self):
        if self.history:
            command = self.history.pop()
            command.undo()
```

Client = creates commands and passes them to the invoker.

```
editor = TextEditor()
invoker = EditorInvoker()

# Execute commands
invoker.execute_command(WriteCommand(editor, "Hello "))
invoker.execute_command(WriteCommand(editor, "World!"))
print(editor)  # Hello World!

# Undo last action
invoker.undo()
print(editor)  # Hello

# Undo again
invoker.undo()
print(editor)  # (empty)
```

Output

```
Hello World!
Hello
```

Summary

1. Creational Patterns - *object creation focus*

- **Singleton** → You need **only one instance** of a class, accessible globally.

  > *Q to ask: “Do I need to enforce a single global object?”*

- **Factory Method** → You want to **delegate object creation** to subclasses, choosing which product to return.

  > *Q to ask: “Do I want to vary which object is created without changing client code?”*

- **Builder** → You need to build a **complex object step by step**, possibly with different configurations.

  > *Q to ask: “Do I need to construct objects with many optional parts or variations?”*

2. Structural Patterns - *class/object composition focus*

- **Adapter** → You want to make **two incompatible interfaces work together**.

  > *Q to ask: “Do I need to wrap an existing class to match a required interface?”*

- **Proxy** → You want to **control access** to another object (e.g., add lazy loading, caching, logging, security).

  > *Q to ask: “Do I need an object that stands in front of another object and decides when/how it is accessed?”*

- **Facade** → You want to **simplify a complex system** by providing a single unified interface.

  > *Q to ask: “Do I need to hide subsystem complexity behind a simple API?”*

3. Behavioral Patterns - *object interaction focus*

- **Observer** → You need a **one-to-many notification mechanism** (publish/subscribe).

  > *Q to ask: “Do I need multiple objects to react automatically when another object changes?”*

- **Strategy** → You need to choose between **different algorithms/behaviors at runtime**.

  > *Q to ask: “Do I want to swap algorithms without changing client code?”*

- **Command** → You need to turn an **action into an object** so it can be executed later, undone, or queued.

  > *Q to ask: “Do I want to log, undo, or queue operations?”*

⚡ Quick Differentiation

- **Global single object?** → Singleton.
- **Need to decide *which* product to create?** → Factory Method.
- **Complex object built step by step?** → Builder.
- **Incompatible interfaces?** → Adapter.
- **Control access / stand-in for real object?** → Proxy.
- **Simplify subsystem API?** → Facade.
- **Notify many listeners automatically?** → Observer.
- **Choose between interchangeable algorithms?** → Strategy.
- **Encapsulate actions (undo, queue, log)?** → Command.

💡 Think of it this way:

- **Creational** = *“How do I create objects?”*
- **Structural** = *“How do I compose objects?”*
- **Behavioral** = *“How do objects interact?”*