





Introduction to Design Patterns

- **Design Patterns** are **reusable solutions** to common software design problems.
- **Key Characteristics:**
 -  **Reusability** – Can be applied across multiple projects.
 -  **Standardization** – Provides a **shared language** for developers.
 -  **Efficiency** – Saves time by avoiding re-solving known problems.
 -  **Flexibility** – Can be **adapted** to fit different requirements.

Pros and Cons

✓ Pros:

- Improves **code maintainability** and **scalability**.
- Encourages **modular**, **well-structured**, and **flexible** programs.
- Reduces **development time** by using **tested solutions**.

✗ Cons:

- **Not a replacement** for good design thinking.
- Adds **complexity** if misused.
- **Language-dependent** – Some patterns may not fit all programming paradigms.

Types of Design Patterns

Design patterns are categorized into **three main types**:

Type	Purpose	Examples
Creational	Focuses on how objects are created	Singleton, Factory, Builder, Prototype
Structural	Deals with object relationships & composition	Adapter, Decorator, Proxy, Composite
Behavioral	Defines object interactions & responsibilities	Strategy, Observer, Command, State

Creational Design Patterns (Object Creation)

Singleton

Ensures **only one instance** of a class exists and provides a global access point.

✓ Use When:

- Only **one instance makes sense** (e.g., **database connection**, **logger**).

✗ Avoid When:

- May introduce **global state issues**.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

# Usage
singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2) # ✓ True (same instance)
```

Factory Method

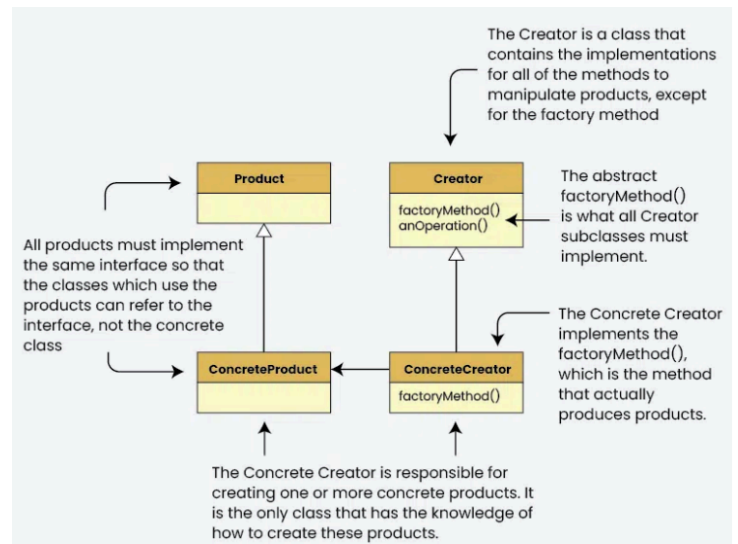
Provides an **interface for creating objects** without specifying their concrete classes.

✓ Use When:

- Object creation is **complex or varies by context**.

✗ Avoid When:

- **Overkill** for simple object creation.



```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def get_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            return None

# Usage
animal = AnimalFactory.get_animal("dog")
print(animal.speak()) # ✓ Woof!
```

Abstract Factory Pattern

The **Abstract Factory Pattern** is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

✓ Use When:

- You need to create related objects that work together within a family of products.
- When your code needs to handle multiple variations of objects, but you want to ensure the objects from the same family work together.

✗ Avoid When:

- The object creation is not related to any families or variations, and you only need a simple object creation process.

```
# Abstract Factory
class ComputerFactory:
    def create_computer(self):
        raise NotImplementedError("create_computer() must be implemented")

# Concrete Factories
class PCFactory(ComputerFactory):
    def create_computer(self):
        return PC()

class LaptopFactory(ComputerFactory):
    def create_computer(self):
        return Laptop()

# Abstract Product
class Computer:
```

```

def get_type(self):
    raise NotImplementedError("get_type() must be implemented")

# Concrete Products
class PC(Computer):
    def get_type(self):
        return "PC"

class Laptop(Computer):
    def get_type(self):
        return "Laptop"

# Usage
def get_computer(factory: ComputerFactory):
    return factory.create_computer()

pc_factory = PCFactory()
laptop_factory = LaptopFactory()

pc = get_computer(pc_factory)
laptop = get_computer(laptop_factory)

print(f"Created a {pc.get_type()}") # Output: Created a PC
print(f"Created a {laptop.get_type()}") # Output: Created a Laptop

```

Builder

Separates **object construction from representation**.

✔ Use When:

- Creating **complex objects with many parameters**.

✗ Avoid When:

- **Too much complexity** for simple objects.

```

class Burger:
    def __init__(self):
        self.ingredients = []

    def add_ingredient(self, ingredient):
        self.ingredients.append(ingredient)
        return self

    def show(self):
        return f"Burger with {', '.join(self.ingredients)}"

# Usage
burger =
Burger().add_ingredient("Lettuce").add_ingredient("Tomato").add_ingredient("Cheese")
print(burger.show()) # ✔ Burger with Lettuce, Tomato, Cheese

```

Prototype

Creates new objects **by copying an existing one**.

✔ Use When:

- Object **creation is expensive**.

✗ Avoid When:

- Objects have **deep dependencies or circular references**.

```

import copy

class Prototype:
    def clone(self):
        return copy.deepcopy(self)

class Car(Prototype):
    def __init__(self, model):
        self.model = model

# Usage
car1 = Car("Tesla Model S")
car2 = car1.clone()
print(car1.model == car2.model) # ✔ True (cloned object)

```

Object Creation Methods Trade-offs

Simplicity vs. Flexibility:

Factory Method & Abstract Factory: More flexible but adds complexity.

Builder: Handles complex objects but sacrifices simplicity.

Performance:

Prototype: Best for expensive object initialization, cloning is faster than creating new objects.

Other Patterns: Can incur overhead, especially with dynamic type resolution.

Object Complexity:

Factory Method/Prototype: Good for simple objects.

Builder: Ideal for complex objects with many parameters.

Scalability:

Abstract Factory & Builder: Scale well for larger, extensible systems.

Singleton: Can cause issues in large systems due to global state.

Testing and Maintenance:

Factory Method & Abstract Factory: Easier testing due to Dependency Inversion.

Singleton: Harder to test due to global state.

Object Creation Methods Trade-offs

Simplicity vs. Flexibility:

Factory Method & Abstract Factory: More flexible but adds complexity.

Builder: Handles complex objects but sacrifices simplicity.

Performance:

Prototype: Best for expensive object initialization, cloning is faster than creating new objects.

Other Patterns: Can incur overhead, especially with dynamic type resolution.

Object Complexity:

Factory Method/Prototype: Good for simple objects.

Builder: Ideal for complex objects with many parameters.

Scalability:

Abstract Factory & Builder: Scale well for larger, extensible systems.

Singleton: Can cause issues in large systems due to global state.

Testing and Maintenance:

Factory Method & Abstract Factory: Easier testing due to Dependency Inversion.

Singleton: Harder to test due to global state.

Structural Design Patterns (Object Composition & Relationships)

Adapter

🔗 **Acts as a bridge** between incompatible interfaces.

✔ Use When:

- Need to **integrate legacy code** into a new system.

✗ Avoid When:

- Direct modification is possible.

```

class EuropeanPlug:
    def plug_in(self):
        return "Using European plug"

class USAdapter:
    def __init__(self, european_plug):
        self.european_plug = european_plug

    def plug_in(self):
        return
self.european_plug.plug_in().replace("European", "US")

# Usage
plug = EuropeanPlug()
adapter = USAdapter(plug)
print(adapter.plug_in()) # ✔ Using US plug

```

Decorator

🔗 **Dynamically adds behavior** to objects **without modifying the original class**.

✔ Use When:

- Adding **flexibility** without modifying existing code.
- ❌ **Avoid When:**
- Too many decorators can **complicate debugging**.

```
def make_bold(func):
    def wrapper():
        return f"<b>{func()}</b>"
    return wrapper

@make_bold
def say_hello():
    return "Hello"

print(say_hello()) # ✅ <b>Hello</b>
```

Composite

- 📁 Allows **objects to be treated the same way**, whether **individual or grouped**.
- ✅ **Use When:**
- Need to **manage hierarchies** (e.g., **file systems, UI components**).
- ❌ **Avoid When:**
- If **object relationships are simple**.

```
class FileSystem:
    def show(self):
        pass

class File(FileSystem):
    def __init__(self, name):
        self.name = name

    def show(self):
        return f"File: {self.name}"

class Folder(FileSystem):
    def __init__(self, name):
        self.name = name
        self.children = []

    def add(self, child):
        self.children.append(child)

    def show(self):
        return f"Folder: {self.name}, contains: {[child.show() for child in self.children]}"

# Usage
folder = Folder("Documents")
folder.add(File("Resume.pdf"))
folder.add(File("Project.docx"))
print(folder.show())
```

Proxy

- 🛡️ **Controls access** to another object.
- ✅ **Use When:**
- **Lazy initialization, security, logging**.
- ❌ **Avoid When:**
- **Extra indirection may hurt performance**.

```
class RealImage:
    def __init__(self, filename):
        self.filename = filename

    def display(self):
        return f"Displaying {self.filename}"

class ImageProxy:
```

```
    def __init__(self, filename):
        self.filename = filename
        self.image = None

    def display(self):
        if not self.image:
            self.image = RealImage(self.filename)
        return self.image.display()

# Usage
proxy = ImageProxy("photo.jpg")
print(proxy.display()) # ✅ Displays photo only when accessed
```

Facade Pattern

The **Facade Pattern** provides a simplified interface to a complex subsystem. It hides the complexities of interacting with multiple subsystem classes and their interactions, making it easier for clients to use the system.

When to Use:

- Provide a simple interface to a complex or tightly coupled subsystem.
- Decouple clients from the internal structure of a subsystem.
- Work with legacy systems that have poorly organized or inconsistent interfaces.

Pros:

- **Simplifies interaction:** Clients don't need to deal with the complexities of subsystem classes.
- **Loose coupling:** Clients are decoupled from the subsystems, making the system easier to maintain and modify.
- **Cohesive interface:** The facade encapsulates subsystem logic and provides a single cohesive interface.

Cons:

- **Limited access to advanced features:** The simplified interface might hide some of the advanced features of the subsystems.
- **Additional complexity:** The facade layer could add unnecessary complexity if the subsystems are already simple.

```
# Subsystem Class 1
class Subsystem1:
    def operation1(self):
        print("Subsystem1: Performing operation1.")

    def operation2(self):
        print("Subsystem1: Performing operation2.")

# Subsystem Class 2
class Subsystem2:
    def operationA(self):
        print("Subsystem2: Performing operationA.")

    def operationB(self):
        print("Subsystem2: Performing operationB.")

# Facade Class
class Facade:
    def __init__(self):
        self.subsystem1 = Subsystem1()
        self.subsystem2 = Subsystem2()

    def simplified_operation(self):
        print("Facade: Simplifying subsystem interactions.")
        self.subsystem1.operation1()
        self.subsystem2.operationA()
        self.subsystem1.operation2()
        self.subsystem2.operationB()

# Client
if __name__ == "__main__":
    facade = Facade()
    print("Client: Using the facade to interact with subsystems.")
    facade.simplified_operation()
```

Bridge Pattern

The **Bridge Pattern** is a structural design pattern that separates an object's interface (abstraction) from its implementation. This separation allows both the abstraction and implementation to vary independently, making the system more flexible and extensible.

When to Use the Bridge Pattern:

- **Decouple abstraction and implementation** to allow independent variations. It allows you to add new features without changing the existing code.
- **Support multiple dimensions of variation**, such as platforms (Windows/Mac), devices (Mobile/Desktop), or output formats (PDF/HTML).
- **Avoid class explosion** that happens when using inheritance for multiple variations of abstractions and implementations.

Without Bridge:

In a traditional approach, multiple subclasses are created for every combination of abstraction and implementation. This can lead to a **class explosion**, where the number of classes grows exponentially as variations of the abstraction and implementation increase.

With Bridge:

The **Bridge Pattern** decouples the abstraction (like a `Dialog` or `Button`) from the implementation (like a `WindowsButton` or `MacButton`). This makes it easier to add new button types or dialogs without creating many new subclasses.

```
# Without Bridge: Multiple subclasses for each variation

class WindowsButton:
    def render(self):
        print("Rendering Windows Button.")

    def on_click(self):
        print("Windows Button Clicked.")

class MacButton:
    def render(self):
        print("Rendering Mac Button.")

    def on_click(self):
        print("Mac Button Clicked.")

class Dialog:
    def __init__(self, button):
        self.button = button

    def render(self):
        self.button.render()
        self.button.on_click()

# Usage
windows_button = WindowsButton()
mac_button = MacButton()
dialog = Dialog(windows_button) # or Dialog(mac_button)
dialog.render()

Example with Bridge:
python
Copy
# With Bridge: Separating abstraction from implementation

# Implementor: Button interface
class Button:
    def render(self):
        pass

    def on_click(self):
        pass

# Concrete Implementors: Windows and Mac buttons
class WindowsButton(Button):
    def render(self):
        print("Rendering Windows Button.")

    def on_click(self):
        print("Windows Button Clicked.")

class MacButton(Button):
    def render(self):
        print("Rendering Mac Button.")

    def on_click(self):
```

```
print("Mac Button Clicked.")

# Abstraction: Dialog with a Button
class Dialog:
    def __init__(self, button: Button):
        self.button = button

    def render(self):
        self.button.render()
        self.button.on_click()

# Usage
windows_button = WindowsButton()
mac_button = MacButton()
dialog1 = Dialog(windows_button) # Dialog using WindowsButton
dialog2 = Dialog(mac_button)     # Dialog using MacButton

dialog1.render()
dialog2.render()
```

Flyweight Pattern

The **Flyweight Pattern** is a structural design pattern that minimizes memory usage by sharing as much data as possible between similar objects. It reduces the number of objects created by sharing common intrinsic data and separating it from extrinsic data that varies.

Real-World Use Cases:

- **Font Rendering:** Text editors and word processors, like MS Word, use flyweights to represent characters, as characters are often repeated.
- **Game Graphics:** Objects like trees, tiles, or bullets in games where their intrinsic properties (such as model, texture) can be shared.
- **Network Applications:** Managing shared connections or session objects.
- **Document Editors:** Formatting elements such as bold or italicized text, which can be shared across multiple uses.

When to Use Flyweight:

- When your application creates a large number of similar objects, and **memory usage is a concern**.
- When objects can share most of their state, while only a small part varies (intrinsic vs. extrinsic state).

Pros:

- **Minimizes memory consumption:** Especially when dealing with many similar objects, as shared data is reused.
- **Reduces object instantiation overhead:** By reusing existing objects, it avoids the cost of creating many similar objects.
- **Easy management and updating:** The shared objects are centralized and managed via the factory, making it easier to maintain and update them.

Cons:

- **Complexity in managing states:** Handling both intrinsic and extrinsic states can complicate the design.
- **Concurrency concerns:** If shared objects are modified in a multi-threaded environment, synchronization might be required.
- **Limited use case:** Only applicable when a large number of similar objects with shared data are needed.

```
class Character:
    def __init__(self, symbol):
        self.symbol = symbol # Intrinsic state (shared)

    def display(self, font_size, font_color):
        print(f"Character: '{self.symbol}', Size: {font_size}, Color: {font_color}")

class CharacterFactory:
    _characters = {}

    def get_character(self, symbol):
        if symbol not in self.symbol_pool:
            self.symbol_pool[symbol] = Character(symbol)
            print(f"Creating new Character instance for '{symbol}'")
        return self.symbol_pool[symbol]

# Client code
if __name__ == "__main__":
    factory = CharacterFactory()

    char_a1 = factory.get_character('A')
    char_a1.display(12, 'red')
```

```

char_a2 = factory.get_character('A')
char_a2.display(14, 'blue')

char_b = factory.get_character('B')
char_b.display(16, 'green')

print(f"\nTotal unique Character instances:
{factory.total_characters()}")

```

Behavioral Design Patterns (Object Interaction & Responsibilities)

Strategy

✚ **Encapsulates different algorithms** and makes them interchangeable.

✔ **Use When:**

- You want to **replace conditional logic** with **polymorphism**.

✗ **Avoid When:**

- Too many strategy classes** can become hard to manage.

```

class PayPalPayment:
    def pay(self, amount):
        return f"Paid {amount} via PayPal"

class CreditCardPayment:
    def pay(self, amount):
        return f"Paid {amount} via Credit Card"

class PaymentProcessor:
    def __init__(self, strategy):
        self.strategy = strategy

    def process_payment(self, amount):
        return self.strategy.pay(amount)

# Usage
payment = PaymentProcessor(PayPalPayment())
print(payment.process_payment(100)) # ✔ Paid 100 via PayPal

```

Observer

👂 **One-to-many relationship** where multiple objects are **notified of changes**.

✔ **Use When:**

- You need **real-time updates** (e.g., **stock prices**, **event listeners**).

✗ **Avoid When:**

- Too many observers** can **cause performance overhead**.

```

class Subject:
    def __init__(self):
        self.observers = []

    def attach(self, observer):
        self.observers.append(observer)

    def notify(self, message):
        for observer in self.observers:
            observer.update(message)

class Observer:
    def update(self, message):
        pass

class EmailSubscriber(Observer):
    def update(self, message):
        print(f"Email received: {message}")

# Usage
subject = Subject()
subscriber = EmailSubscriber()
subject.attach(subscriber)
subject.notify("New Update Available!") # ✔ Email received:
New Update Available!

```

Command

👤 **Encapsulates a request** as an object.

✔ **Use When:**

- Undo/redo** or **queueing operations** (e.g., **remote control system**).

✗ **Avoid When:**

- Adds **complexity** when not needed.

```

# Command Interface
class Command:
    def execute(self):
        pass

# Receiver class
class Light:
    def on(self):
        print("Light is ON")

    def off(self):
        print("Light is OFF")

# Concrete commands
class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.off()

# Invoker class
class RemoteControl:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def press_button(self):
        self.command.execute()

# Client
light = Light()
remote = RemoteControl()

remote.set_command(LightOnCommand(light))
remote.press_button() # 🗨️ Light is ON

remote.set_command(LightOffCommand(light))
remote.press_button() # 🗨️ Light is OFF

```

State

🔄 **Changes an object's behavior** based on its **state**.

✔ **Use When:**

- Behavior depends on state changes** (e.g., **traffic lights**, **vending machines**).

✗ **Avoid When:**

- Too many states** can lead to **state explosion**.

```

class Command:
    def execute(self):
        pass

class LightOnCommand(Command):
    def execute(self):
        return "Light turned ON"

class LightOffCommand(Command):
    def execute(self):
        return "Light turned OFF"

class RemoteControl:
    def __init__(self, command):
        self.command = command

    def press(self):
        return self.command.execute()

# Usage
remote = RemoteControl(LightOnCommand())
print(remote.press()) # ✔ Light turned ON

```

Template Method

 Defines an algorithm's structure but lets subclasses override steps.

✓ Use When:

- A workflow remains the same but **steps vary** (e.g., **data parsers**).

✗ Avoid When:

- Too much dependency on inheritance.

```
from abc import ABC, abstractmethod

class DataParser(ABC):
    # Template method
    def parse(self, data):
        self.read(data)
        self.process()
        self.save()

    def read(self, data):
        print(f"Reading data: {data}")

    def save(self):
        print("Saving results")

    @abstractmethod
    def parse(self):
        pass

class JSONParser(DataParser):
    def parse(self):
        print("Parsing data as JSON")

class XMLParser(DataParser):
    def parse(self):
        print("Parsing data as XML")

# Usage
json_parser = JSONParser()
json_parser.parse()

xml_parser = XMLParser()
xml_parser.parse()
```

• Iterator

- sequentially access elements of a collection without exposing its underlying representation.
- Example
 - Traversing elements of a linked list, tree, or graph.
 - Iterating over lines, words, or characters in a file.
 - Providing access to live data streams (e.g., sensor data or API responses).
- When to use
 - traverse a collection without exposing its implementation details.
 - want multiple traversal methods for a collection.
- Components
 - Iterator: The interface or abstract class for iterating.
 - Concrete Iterator: Implements iteration for a specific collection.
 - Aggregate: Interface for the collection.
 - Concrete Aggregate: Implements the collection and provides an iterator.

- Encapsulation: Hides the internal structure of the collection.
- Consistency: Provides a uniform interface for iterating over different collections.
- Reusability: Iterator logic can be reused with minimal changes.

◦ Cons

- Increased Complexity: Requires additional classes for iterator implementation.
- Performance Overhead: Can introduce slight overhead, especially for simple collections.

```
#include <iostream>
#include <vector>
#include <memory>
#include <string>
```

```
// Iterator Interface
```

```
class Iterator {
public:
    virtual bool hasNext() const = 0;
    virtual std::string next() = 0;
    virtual ~Iterator() = default;
};
```

```
// Concrete Iterator
```

```
class BookIterator : public Iterator {
private:
    const std::vector<std::string>& books;
    size_t index;

public:
    explicit BookIterator(const std::vector<std::string>& books)
        : books(books), index(0) {}

    bool hasNext() const override {
        return index < books.size();
    }
```

```
};

std::string next() override {
    return books[index++];
}
```

```
// Aggregate Interface
```

```
class Collection {
public:
    virtual std::shared_ptr<Iterator> createIterator() const = 0;
    virtual ~Collection() = default;
};
```

```
// Concrete Aggregate
```

```
class BookCollection : public Collection {
private:
    std::vector<std::string> books;

public:
    void addBook(const std::string& book) {
        books.push_back(book);
    }

    std::shared_ptr<Iterator> createIterator() const override {
        return std::make_shared<BookIterator>(books);
    }
};
```

```
// Client Code
```

```
int main() {
    BookCollection collection;
    collection.addBook("Design Patterns");
    collection.addBook("Effective C++");
    collection.addBook("Clean Code");

    auto iterator = collection.createIterator();

    std::cout << "Iterating over book collection:\n";
    while (iterator->hasNext()) {
        std::cout << iterator->next() << "\n";
    }

    return 0;
}
```


● Chain of Responsibility

- allows multiple objects to process a request without tightly coupling the sender and receiver.
- Example
 - Handler Interface: Defines the contract for processing requests.
 - Concrete Handlers: Implement specific handling logic.
 - Chain Setup: Connect handlers to form the chain.
- When to use
 - multiple objects might handle a request, and the handler doesn't need to be specified explicitly.
 - dynamically define or change the chain of handling.
- Components
 - Handler:
 - Defines an interface for processing a request and a reference to the next handler in the chain.
 - Concrete Handlers:
 - Implement the handler interface to process specific requests or pass them along.
 - Client:
 - Initiates the request and sends it into the chain.

○ Pros

- Decoupled Senders and Receivers:
 - Senders don't need to know which handler will process the request.
- Flexible Chain Construction:
 - Handlers can be easily added, removed, or reordered.
- Open/Closed Principle:
 - New handlers can be added without modifying existing code.

○ Cons

- Potential for Unhandled Requests:
 - If no handler processes a request, it might go unhandled.
- Debugging Complexity:
 - Tracing the flow of a request through the chain can be challenging.
- Performance Overhead:
 - Passing requests through multiple handlers may introduce latency.

● Mediator

- What is it
 - centralize communication between multiple objects (many-to-many)
 - reduce direct dependencies and simplifying collaboration
- Example
 - Centralized event handling in GUI systems.
 - Messaging hubs in distributed systems.
- When to use
 - multiple objects need to communicate but shouldn't directly depend on each other.
 - communication logic between objects becomes complex.
- Components
 - Mediator Interface: Defines how colleagues communicate.
 - Concrete Mediator: Manages interactions between colleagues.
 - Colleague Interface: Represents participants in communication.
 - Concrete Colleagues: Objects that depend on the mediator.

● Visitor

- add new operations to existing object structures without modifying their classes.
- Example
 - Syntax tree processing in compilers (See impl of LLVM).
 - Reporting or analytics systems for hierarchical data.
- When to use
 - perform unrelated operations on objects in a structure.
 - object structure is stable but operations frequently change.
- Components
 - Visitor Interface: Declares operations for each type of element.
 - Concrete Visitors: Implements specific behavior for each type of element.
 - Element Interface: Declares an accept() method.
 - Concrete Elements: Implements the accept() method to interact with visitors.
 - Object Structure: Manages the collection of elements.

● Memento

- capture and externalize an object's internal state without violating encapsulation.
- Example
 - Undo features in text editors, drawing applications, and IDEs.
 - Saving checkpoints in games or simulations.
- When to use
 - implement undo or rollback functionality.
 - the internal state of an object needs to be saved and restored without breaking encapsulation.
- Components
 - Memento:
 - Represents the saved state of an object.
 - Stores state but prevents modification.
 - Originator:
 - The object whose state is saved and restored.
 - It creates a memento and uses it to restore its state.
 - Caretaker:
 - The object that holds the memento but doesn't modify it.
 - It can request a memento to store the originator's state and pass it back when needed.

✓ Trade-offs in Behavioral Patterns

1. Simplicity vs. Complexity

- **Low Complexity:** Easy to implement; less flexible.
 - Examples: Command, Iterator, State, Template Method, Memento.
- **High Complexity:** More flexible but harder to implement.
 - Examples: Mediator, Observer, Strategy, Visitor.

2. Coupling

- **Low Coupling:** Easier changes, but more abstraction. (Mediator, Observer, Strategy, Visitor)
- **High Coupling:** Simpler design, less flexibility. (Template Method, State, Memento)

3. Flexibility & Extensibility

- **High Flexibility:** Easy to extend, but higher complexity. (Strategy, Observer, Visitor)
- **Low Flexibility:** Simpler but rigid designs. (State, Template Method)

4. Reusability

- **High Reuse:** Command, Strategy, Visitor (more complex, reusable)
- **Low Reusability:** Easier initial design, less adaptable. (State, Memento)

4. Maintainability

- **High:** Easier to update due to modularity (Observer, Strategy, Visitor)
- **Low:** Simpler but harder to adapt later (Template Method, State, Memento)

5. Performance

- **High Overhead:** Increased complexity and memory use. (Visitor, Mediator, Observer)
- **Low Overhead:** Efficient but less adaptable (State, Iterator, Template Method)

Conclusion

- 1 **Creational Patterns** help with **object creation** (e.g., **Singleton**, **Factory**).
- 2 **Structural Patterns** define **object composition** (e.g., **Adapter**, **Proxy**).
- 3 **Behavioral Patterns** manage **object interactions** (e.g., **Observer**, **Strategy**).
- 4 Each pattern has **trade-offs**, so **choose wisely based on the problem**.