



~~Concepts~~ Design Patterns in OOP Development

Reusable Solutions in Object-Oriented Design
Object-Oriented Programming 2/2567

@wichit2s

<https://wichit2s.github.io/courses/oop/>



Objectives



- Understand what design patterns are and their purpose.



- Identify the three main categories of design patterns.



- Explore examples of commonly used patterns.



- Discuss why and when to use design patterns.



What?



“Design patterns are reusable solutions to common software design problems.”

Key Features: Not code, but templates for solving design problems.

Based on proven, time-tested solutions.

Why are they important?

Improve code readability and maintainability.

Promote best practices in object-oriented design.

Real-Life Analogy



A government
issuing only one
passport per citizen.



A bakery creating
various types of
pastries.



Social media
notifications when
someone likes your
post.



When and Why to Use Design Patterns



When to Use?

- Facing recurring design challenges in your code.
- Need for scalable and maintainable solutions.

Why Use?

- Save time by not reinventing the wheel.
- Make code more understandable for other developers.



Best Practices for Learning Design Patterns



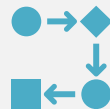
Learn one pattern at a time.



Understand the problem the pattern solves.



Look for patterns in existing codebases.



Practice implementing patterns in small projects.



Categories



1. Creational Patterns

object creation mechanisms.

Example: Singleton, Factory Method.

2. Structural Patterns

object composition and relationships.

Example: Adapter, Composite, Decorator.

3. Behavioral Patterns

communication between objects.

Example: Observer, Strategy, Command.

Singleton Pattern

- What?
- Why?
- Code?
- Applications?

What is the Singleton Pattern?

“A design pattern that ensures a class has only one instance and provides a global point of access to it.”

Key Features:

- Single Instance

Only one object of the class exists.

- Controlled Access

Centralized control of the instance.

- Lazy Initialization

Instance is created only when needed.

Why Use the Singleton Pattern?

- Avoid redundant object creation for shared resources.
- *Centralize control for managing configurations or logging.*
- Ensures consistency when working with global objects.

- **Common Use Cases**

- Database connections.
- Logging systems.
- Configuration settings.



Example of Singleton Pattern

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

```
# Usage:
obj1 = Singleton()
obj2 = Singleton()
print(obj1 is obj2) # Output: True
```



Applications of Singleton Pattern

Logging Systems

- Ensures that all components log to the same file or console.

Configuration Managers

- Provides a centralized way to access configuration settings.

Database Connection Pools

- Maintains a single shared connection object for performance.

Caching Systems

- Centralizes the storage and retrieval of frequently used data.



Benefits and Limitations

Benefits

- Ensures single instance.
- Reduces resource usage.
- Promotes consistency.

Limitations

- Can introduce global state, making testing difficult.
- Overuse may lead to tightly coupled code.

Conclusion

Summary

- The Singleton pattern ensures a single instance of a class.
- Useful for managing shared resources, logging, and configurations.

Questions?

Factory Method Pattern

- What?
- Why?
- Code?
- Applications?

What is the Factory Method Pattern?

“A creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.”

Key Features:

- Promotes loose coupling between client code and the object creation logic.
- Delegates object creation to a method, allowing subclasses to specify the type of object to instantiate.

Why Use the Factory Method Pattern?

- Avoids direct instantiation of objects, making the code more modular and testable.
- Allows for easy addition of new object types without modifying existing code (Open/Closed Principle).
- Useful when the exact type of object to create is determined at runtime.

- **Common Use Cases**

- Creating objects with shared initialization logic.
- Managing families of related objects.

Example of Factory Method Pattern

```
from abc import ABC, abstractmethod

# Abstract Creator
class DocumentCreator(ABC):
    @abstractmethod
    def create_document(self):
        pass

# Concrete Creators
class PDFCreator(DocumentCreator):
    def create_document(self):
        return PDFDocument()

class WordCreator(DocumentCreator):
    def create_document(self):
        return WordDocument()

# Products
class Document(ABC):
    @abstractmethod
    def display(self):
        pass

class PDFDocument(Document):
    def display(self):
        return "Displaying a PDF Document"

class WordDocument(Document):
    def display(self):
        return "Displaying a Word Document"
```

Client Code

```
def client_code(creator: DocumentCreator):
    doc = creator.create_document()
    print(doc.display())
```

Usage

```
client_code(PDFCreator()) # Output: Displaying a PDF Document
client_code(WordCreator()) # Output: Displaying a Word Document
```



Applications of Factory Method Pattern

File Parsers:

Creating different parsers for XML, JSON, or CSV files.

UI Components:

Generating platform-specific UI elements (e.g., Windows buttons vs. MacOS buttons).

Database Drivers:

Managing connections to different databases (e.g., MySQL, PostgreSQL).

Game Development:

Creating different types of characters or enemies based on the game level.



Benefits and Limitations

Benefits

- Reduces code duplication by centralizing object creation logic.
- Promotes scalability and adherence to SOLID principles.

Limitations

- Adds complexity due to the introduction of additional classes and methods.
- Can be overkill for simple object creation tasks.

Conclusion

Summary

- The Factory Method pattern is used to centralize and simplify object creation.
- It is especially useful when dealing with dynamically determined or complex object hierarchies.

Questions?

Adapter Pattern

- What?
- Why?
- Code?
- Applications?

What is the Factory Method Pattern?


“A structural design pattern that allows incompatible interfaces to work together by acting as a bridge.”

Key Features:

- Converts the interface of a class into another interface that a client expects.
- Used to make existing classes work together without modifying their source code.

Analogy:

Like a power adapter that converts an international plug to fit into a local socket.



Why Use Adapter Pattern?

- Enables reusability of existing code that is incompatible with the new system.
- Promotes flexibility by decoupling client code from specific implementations.
- **Common Use Cases**
 - integrating a legacy system into a new system.
 - use a third-party library with an incompatible interface.

Example of Adapter Pattern

Existing class with incompatible interface

class OldPrinter:

```
def print_text(self, text):  
    return f"Printing text: {text}"
```

Target interface expected by the client

class NewPrinterInterface:

```
def print(self, message):  
    pass
```

Adapter class

class PrinterAdapter(NewPrinterInterface):

```
def __init__(self, old_printer):  
    self.old_printer = old_printer
```

```
def print(self, message):  
    return self.old_printer.print_text(message)
```

Client code

```
def client_code(printer: NewPrinterInterface):  
    print(printer.print("Hello, Adapter Pattern!"))
```

Usage

```
old_printer = OldPrinter()  
adapter = PrinterAdapter(old_printer)  
client_code(adapter)
```



Applications of Adapter Pattern

- **Legacy System Integration:**

Making old systems compatible with modern APIs.

- **Third-Party Libraries:**

Adapting a library's interface to meet the requirements of your system.

- **Cross-Platform Development:**

Adapting code to work with different operating systems or environments.

- **Game Development:**

Allowing different input devices (e.g., joystick, keyboard) to interact with the same game engine interface.



Benefits and Limitations

Benefits

- Allows reuse of existing functionality without modifying its code.
- Decouples the client from the implementation details of the adapted class.

Limitations

- Can increase the complexity of the codebase.
- Overuse can lead to excessive layers of abstraction..

Conclusion

Summary

- The Adapter Pattern helps bridge the gap between incompatible interfaces.
- It's a versatile pattern for integrating legacy code or third-party libraries into modern applications.

Questions?

Composite Pattern

- What?
- Why?
- Code?
- Applications?

What is the Composite Pattern?

“A structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies.”

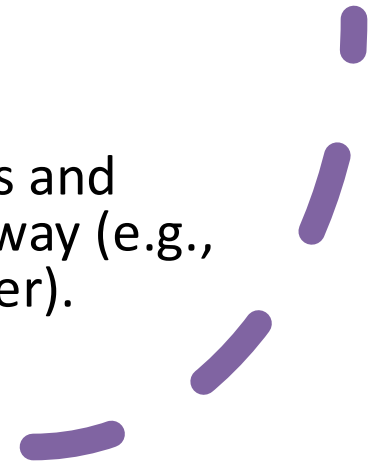
“It lets clients treat individual objects and compositions of objects uniformly.”

Key Features:

- Works with tree-like structures (e.g., folders, files, menus).
- Defines a common interface for all objects in the composition.

Analogy:

Think of a file system where files and folders are treated in the same way (e.g., opening a file vs. opening a folder).



Why Use Composite Pattern?

- Simplifies code by treating individual and composite objects uniformly.
 - Makes it easier to work with complex hierarchical structures.
 - Promotes flexibility by enabling dynamic tree structures.
-
- **Common Use Cases**
 - When dealing with tree-like structures such as file systems, organizational hierarchies, or GUIs.
 - When you want to perform operations on individual and composite objects in the same way.

```
from abc import ABC, abstractmethod
```

```
# Component interface
class Graphic(ABC):
    @abstractmethod
    def render(self): pass
```

```
# Leaf class
class Circle(Graphic):
    def render(self):
        return "Circle"
```

```
class Rectangle(Graphic):
    def render(self):
        return "Rectangle"
```

```
# Composite class
class Drawing(Graphic):
    def __init__(self):
        self.children = []

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

    def remove(self, graphic):
        self.children.remove(graphic)

    def render(self):
        results = [child.render() for child in self.children]
        return " + ".join(results)
```

Example of Composite Pattern

```
# Client code
circle = Circle()
rectangle = Rectangle()
```

```
drawing = Drawing()
drawing.add(circle)
drawing.add(rectangle)
```

```
print(drawing.render()) # Output: Circle +
Rectangle
```




Applications of Composite Pattern

File Systems:

- Treating files and folders as components of a single hierarchy.

GUI Frameworks:

- Combining widgets like buttons, text fields, and panels into a unified interface.

Organizational Hierarchies:

- Representing departments and employees in a company structure.

Graphics Editors:

- Grouping shapes like lines, circles, and rectangles for rendering or transformations.



Benefits and Limitations

Benefits

- Makes the client code simpler by working with individual and composite objects in the same way.
- Supports recursive structures like trees.
- Promotes flexibility in adding new component types.

Limitations

- Can make the design overly general and complex.
- May be difficult to restrict which components can be part of a composite.

Conclusion

Summary

- The Composite Pattern is ideal for working with tree structures and part-whole hierarchies.
- It simplifies client interactions with complex structures by treating objects uniformly.

Questions?

Decorator Pattern

- What?
- Why?
- Code?
- Applications?

What is the Decorator Pattern?

“A structural design pattern that allows behavior to be added to an individual object, dynamically, without modifying the code of the object or its class.”

Key Features:

- Uses a "wrapper" object to enhance the behavior of the original object.
- Multiple decorators can be combined to create flexible behavior.

Analogy:

Think of decorating a cake by adding multiple layers of icing or toppings, one at a time.

Why Use Decorator Pattern?

- Allows for dynamic addition of responsibilities to objects.
 - Avoids creating a large number of subclasses for every possible combination of behaviors.
 - Promotes the Open/Closed Principle (open for extension, closed for modification).
-
- **Common Use Cases**
 - When you need to add responsibilities dynamically without affecting other instances.
 - When subclassing is impractical due to a large number of potential combinations.

```

# Base component
class Coffee:
    def cost(self):
        return 5 # Base cost

    def description(self):
        return "Basic Coffee"

# Decorator base class
class CoffeeDecorator:
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost()

    def description(self):
        return self._coffee.description()

# Concrete decorators
class Milk(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 2

    def description(self):
        return self._coffee.description() + ", Milk"

class Sugar(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 1

    def description(self):
        return self._coffee.description() + ", Sugar"

```

Example of Decorator Pattern

```

# Client code
basic_coffee = Coffee()
print(basic_coffee.description(), "-> $", basic_coffee.cost())

milk_coffee = Milk(basic_coffee)
print(milk_coffee.description(), "-> $", milk_coffee.cost())

sugar_milk_coffee = Sugar(milk_coffee)
print(sugar_milk_coffee.description(), "-> $",
sugar_milk_coffee.cost())

```

```

Basic Coffee -> $ 5
Basic Coffee, Milk -> $ 7
Basic Coffee, Milk, Sugar -> $ 8

```



Applications of Decorator Pattern

User Interfaces

- Adding scrollbars, borders, or shadows to UI components.

Logging Systems

- Adding logging functionality to existing methods dynamically.

Data Processing Pipelines

- Wrapping data transformations like encryption, compression, or validation.

Web Development

- Middleware in web frameworks like Flask or Django, where decorators are used to add authentication or caching.



Benefits and Limitations

Benefits

- Provides a flexible alternative to subclassing for extending functionality.
- Enables dynamic and reusable extensions to object behavior.

Limitations

- Can result in complex and hard-to-read code when multiple decorators are combined.
- Debugging can be challenging as behavior is distributed across multiple layers.

Conclusion

Summary

- The Decorator Pattern allows dynamic addition of behavior to objects without modifying their structure.
- It is widely used in user interfaces, logging systems, and data processing pipelines.

Questions?

Observer Pattern

- What?
- Why?
- Code?
- Applications?

What is the Observer Pattern?

“A behavioral design pattern where an object (subject) maintains a list of its dependents (observers) and notifies them automatically of any state changes.”

Key Features:

- Establishes a one-to-many relationship between objects.
- Observers can dynamically subscribe or unsubscribe to the subject.

Analogy:

Think of a newsletter subscription: subscribers get notified whenever there's a new issue.

Why Use Observer Pattern?

- Promotes loose coupling between the subject and its observers.
 - Ensures that all dependent objects are updated automatically when the subject's state changes.
 - Allows dynamic addition and removal of observers.
-
- **Common Use Cases**
 - When changes to one object require updates to other objects.
 - When you need to maintain consistency between related objects without tightly coupling them.

Example of Decorator Pattern

```
class Subject:
    def __init__(self):
        self._observers = []
    def attach(self, observer):
        self._observers.append(observer)
    def detach(self, observer):
        self._observers.remove(observer)
    def notify(self):
        for observer in self._observers:
            observer.update(self)
```

```
class WeatherStation(Subject):
    def __init__(self):
        super().__init__()
        self._temperature = 0
    @property
    def temperature(self):
        return self._temperature
    @temperature.setter
    def temperature(self, value):
        self._temperature = value
        self.notify()
```

```
class Observer:
    def update(self, subject): pass
```

```
class PhoneDisplay(Observer):
    def update(self, subject):
        print(f"Phone Display: Temperature is {subject.temperature}°C")
```

```
class WindowDisplay(Observer):
    def update(self, subject):
        print(f"Window Display: Temperature is {subject.temperature}°C")
```

```
# Client code
weather_station = WeatherStation()
phone_display = PhoneDisplay()
window_display = WindowDisplay()

weather_station.attach(phone_display)
weather_station.attach(window_display)

weather_station.temperature = 25
# Output:
# Phone Display: Temperature is 25°C
# Window Display: Temperature is 25°C
```

Phone Display: Temperature is 25°C
Window Display: Temperature is 25°C



Applications of Observer Pattern

Real-Time Systems

- Weather apps updating displays when data changes.

Event-Driven Programming

- GUI components reacting to user actions like button clicks.

Notification Systems

- Stock market applications notifying users of price changes.

Data Streams

- Social media apps updating feeds when new posts are available.



Benefits and Limitations

Benefits

- Ensures a consistent state between objects without direct dependencies.
- Promotes flexibility with dynamic subscription management.

Limitations

- Can lead to performance issues if there are too many observers or frequent notifications.
- Difficult to debug due to indirect communication between subjects and observers.

Conclusion

Summary

- The Observer Pattern is ideal for implementing event-driven systems and maintaining consistency between objects.
- It decouples the subject and its observers, making it easier to extend or modify behavior.

Questions?

Strategy Pattern

- What?
- Why?
- Code?
- Applications?

What is Strategy Pattern?

“A behavioral design pattern that enables selecting an algorithm's behavior at runtime by encapsulating it within a class.”

Key Features:

- Defines a family of algorithms.
- Encapsulates each algorithm into a separate class.
- Makes algorithms interchangeable without modifying the client code.

Analogy:

Think of choosing different routes on a GPS: shortest route, fastest route, or avoiding tolls.

Why Use Strategy Pattern?

- Promotes the Open/Closed Principle by allowing new algorithms to be added without changing existing code.
 - Simplifies maintenance by separating algorithm logic from the client code.
 - Reduces the need for conditionals (if/elif) to choose between algorithms.
-
- **Common Use Cases**
 - When multiple algorithms can be applied to a problem, and the choice can change at runtime.
 - When you want to eliminate hardcoded logic in the client code.

```
from abc import ABC, abstractmethod
```

```
# Strategy interface
```

```
class PaymentStrategy(ABC):
```

```
    @abstractmethod
```

```
    def pay(self, amount):
```

```
        pass
```

```
# Concrete strategies
```

```
class CreditCardPayment(PaymentStrategy):
```

```
    def __init__(self, card_number):
```

```
        self.card_number = card_number
```

```
    def pay(self, amount):
```

```
        print(f"Paid ${amount} using Credit Card {self.card_number}")
```

```
class PayPalPayment(PaymentStrategy):
```

```
    def __init__(self, email):
```

```
        self.email = email
```

```
    def pay(self, amount):
```

```
        print(f"Paid ${amount} using PayPal ({self.email})")
```

```
# Context class
```

```
class PaymentContext:
```

```
    def __init__(self, strategy: PaymentStrategy):
```

```
        self._strategy = strategy
```

```
    def set_strategy(self, strategy: PaymentStrategy):
```

```
        self._strategy = strategy
```

```
    def execute_payment(self, amount):
```

```
        self._strategy.pay(amount)
```

Example of Strategy Pattern

```
# Client code
```

```
if __name__ == "__main__":
```

```
    credit_card = CreditCardPayment("1234-5678-9876-5432")
```

```
    paypal = PayPalPayment("user@example.com")
```

```
    context = PaymentContext(credit_card)
```

```
    context.execute_payment(100) # Output: Paid $100 using Credit  
Card 1234-5678-9876-5432
```

```
    context.set_strategy(paypal)
```

```
    context.execute_payment(200) # Output: Paid $200 using PayPal  
(user@example.com)
```

Paid \$100 using Credit Card 1234-5678-9876-5432

Paid \$200 using PayPal (user@example.com)



Applications of Strategy Pattern

Payment Systems

- Allow users to choose between multiple payment methods (e.g., credit card, PayPal, cryptocurrency).

Sorting Algorithms

- Choose between different sorting strategies (e.g., quicksort, mergesort) at runtime.

Data Compression

- Dynamically select compression algorithms (e.g., ZIP, RAR, GZIP).

Game Development

- Switch between different AI behaviors or strategies dynamically.



Benefits and Limitations

Benefits

- Simplifies client code by encapsulating algorithms.
- Makes the code flexible and easily extendable by adding new strategies.
- Reduces tight coupling between algorithms and client code.

Limitations

- Increases the number of classes in the system due to encapsulated strategies.
- The client must be aware of all available strategies to choose the appropriate one.

Conclusion

Summary

- The Strategy Pattern allows dynamic selection of algorithms at runtime, promoting flexibility and reusability.
- It is widely used in systems requiring interchangeable behaviors.

Questions?

Command Pattern

- What?
- Why?
- Code?
- Applications?

What is Command Pattern?

“A behavioral design pattern that encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations.”

Key Features:

- Decouples the sender (who initiates a request) from the receiver (who executes the request).
- Encapsulates requests as objects, enabling undo/redo functionality or queuing requests.

Analogy:

Think of a restaurant: the waiter takes an order (command) and passes it to the kitchen (receiver) to execute.

Why Use Command Pattern?

- Promotes loose coupling between sender and receiver.
- Enables easy implementation of undo, redo, or logging functionalities.
- Supports queuing and scheduling of requests.

Common Use Cases

- When you need to encapsulate requests or operations as objects.
- When implementing undo/redo functionality in applications.
- When managing a queue of operations or deferred execution.

Example of Command Pattern

```
from abc import ABC, abstractmethod
```

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

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

```
class LightOnCommand(Command):  
    def __init__(self, light):  
        self.light = light  
    def execute(self):  
        self.light.turn_on()  
    def undo(self):  
        self.light.turn_off()
```

```
class LightOffCommand(Command):  
    def __init__(self, light):  
        self.light = light  
    def execute(self):  
        self.light.turn_off()  
    def undo(self):  
        self.light.turn_on()
```

Receiver

```
class Light:  
    def turn_on(self):  
        print("The light is ON")  
    def turn_off(self):  
        print("The light is OFF")
```

Invoker

```
class RemoteControl:  
    def __init__(self):  
        self.history = []  
  
    def press(self, command):  
        command.execute()  
        self.history.append(command)
```

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

Client code

```
if __name__ == "__main__":  
    light = Light()  
    light_on = LightOnCommand(light)  
    light_off = LightOffCommand(light)  
    remote = RemoteControl()
```

```
remote.press(light_on) # Output: The light is ON  
remote.press(light_off) # Output: The light is OFF  
remote.undo_last()    # Output: The light is ON
```



Applications of Command Pattern

Undo/Redo Functionality:

- Text editors, drawing apps, and version control systems.

GUI Applications:

- Button click actions in graphical interfaces (e.g., Save, Open, Close).

Job Queues:

- Scheduling tasks or jobs in systems.

Smart Home Systems:

- Controlling devices (e.g., lights, thermostats) with commands issued via remote control or apps.



Benefits and Limitations

Benefits

- Decouples the sender and receiver, making the system more flexible.
- Easy to extend by adding new commands without modifying existing code.
- Supports logging, queuing, and undo functionality.

Limitations

- Can increase complexity by adding many command classes.
- Overhead for simple tasks that don't require such flexibility.

Conclusion

Summary

- The Command Pattern encapsulates requests as objects, enabling flexibility and extensibility.
- Commonly used in applications requiring undo/redo or deferred execution.

Questions?