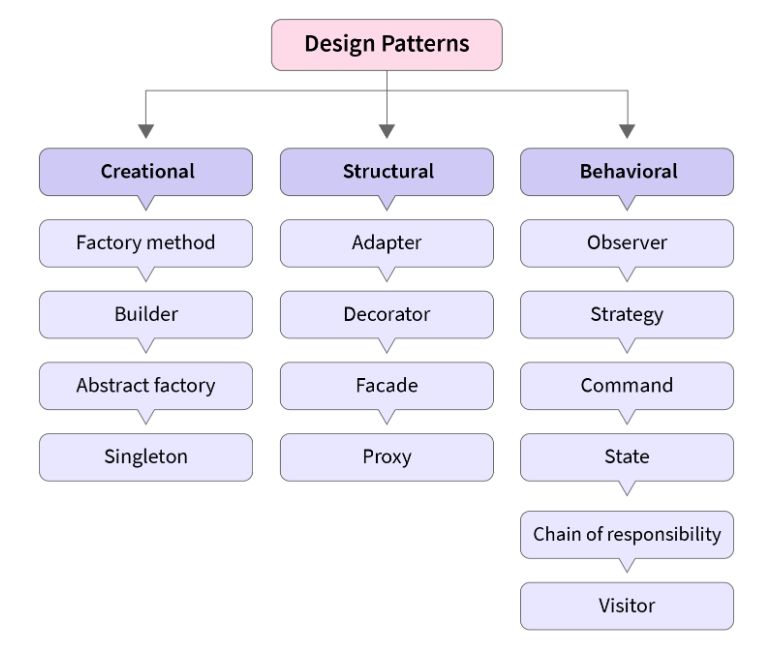
Design Patterns



A design pattern is a general, reusable solution to a common problem within a given context in software design. It is not a finished design that can be directly transformed into code, but rather a template that describes how to solve a problem that can be used in many different situations. Design patterns provide a standardized approach to solving common design issues, enhancing code readability, maintainability, and communication among developers. They capture best practices and knowledge gained by experienced developers and can be categorized into three main types: creational, structural, and behavioral patterns.

Design patterns can be categorized into three main types:

1. **Creational Patterns**: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or add unnecessary complexity to the design. Creational design patterns solve this problem by controlling this object creation. Examples include:
   * **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
   * **Factory Method**: Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.
   * **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
   * **Builder**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
   * **Prototype**: Specifies the kind of objects to create using a prototypical instance, and creates new objects by copying this prototype.
2. **Structural Patterns**: These patterns deal with object composition or the way objects are structured to form larger structures. They help ensure that if one part of a system changes, the entire system does not need to be refactored. Examples include:
   * **Adapter**: Allows incompatible interfaces to work together by acting as a bridge between them.
   * **Decorator**: Adds additional responsibilities to an object dynamically.
   * **Facade**: Provides a simplified interface to a complex subsystem.
   * **Composite**: Composes objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions uniformly.
   * **Proxy**: Provides a surrogate or placeholder for another object to control access to it.
3. **Behavioral Patterns**: These patterns are concerned with algorithms and the assignment of responsibilities between objects. They help in defining how objects interact and communicate with each other. Examples include:
   * **Observer**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   * **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
   * **Command**: Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
   * **Chain of Responsibility**: Passes a request along a chain of handlers. Each handler decides either to process the request or to pass it to the next handler in the chain.
   * **State**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Factory Pattern

The Factory Pattern, specifically the Factory Method Pattern, is 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. This pattern is particularly useful when the exact type of object to be created isn't known until runtime, or when you want to delegate the responsibility of instantiation to subclasses.



1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2. **Concrete Products** are different implementations of the product interface.
3. The **Creator** class declares the factory method that returns new product objects. It’s important that the return type of this method matches the product interface. You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.
4. **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn’t have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Here's a detailed look at the Factory Method Pattern in Python :

**Intent**

The Factory Method Pattern defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.

**Problem**

Imagine you're creating a logistics management application. In the initial version, the transportation type (e.g., trucks) is hard-coded into the logistics class. As the application evolves, you need to add support for different types of transportation (e.g., ships). Hard-coding each type of transportation into the logistics class is not scalable or maintainable.

**Solution**

The Factory Method Pattern suggests moving the object creation logic to subclasses. This allows the logistics class to delegate the creation of transportation objects to subclasses without knowing the specific types of objects to be created.

**Structure**

1. **Product**: Defines the interface for objects that the factory method creates.
2. **Concrete Product**: Implements the Product interface.
3. **Creator**: Declares the factory method, which returns an object of type Product.
4. **Concrete Creator**: Implements the factory method to create an instance of a Concrete Product.

**Example in Python**

Here's a Python example demonstrating the Factory Method Pattern using a logistics context:

from abc import ABC, abstractmethod

# Product Interface

class Transport(ABC):

@abstractmethod

def deliver(self) -> str:

pass

# Concrete Products

class Truck(Transport):

def deliver(self) -> str:

return "Deliver by land in a box."

class Ship(Transport):

def deliver(self) -> str:

return "Deliver by sea in a container."

# Creator Class

class Logistics(ABC):

@abstractmethod

def create\_transport(self) -> Transport:

pass

def plan\_delivery(self) -> str:

transport = self.create\_transport()

return transport.deliver()

# Concrete Creators

class RoadLogistics(Logistics):

def create\_transport(self) -> Transport:

return Truck()

class SeaLogistics(Logistics):

def create\_transport(self) -> Transport:

return Ship()

# Client Code

def client\_code(logistics: Logistics) -> None:

print(f"Client: {logistics.plan\_delivery()}")

if \_\_name\_\_ == "\_\_main\_\_":

print("App: Launched with RoadLogistics.")

client\_code(RoadLogistics())

print("\nApp: Launched with SeaLogistics.")

client\_code(SeaLogistics())

**Explanation**

1. **Transport** is the product interface, defining the deliver method.
2. **Truck** and **Ship** are concrete products implementing the Transport interface.
3. **Logistics** is the creator class, declaring the factory method create\_transport.
4. **RoadLogistics** and **SeaLogistics** are concrete creators that override the factory method to create instances of Truck and Ship, respectively.

**Benefits**

* **Single Responsibility Principle**: The Factory Method Pattern keeps the product creation code separate from the product usage code.
* **Open/Closed Principle**: You can introduce new types of products without changing existing code.

In the context of the Factory Method Pattern, here are the definitions for Product, Concrete Product, Creator, and Concrete Creator:

1. **Product**:
   * **Definition**: The Product is an interface or abstract class that defines the operations that all concrete products must implement.
   * **Purpose**: It serves as a common type for all objects that the factory method creates.
   * **Example in Python**:

python

Copy code

from abc import ABC, abstractmethod

class Transport(ABC):

@abstractmethod

def deliver(self) -> str:

pass

1. **Concrete Product**:
   * **Definition**: The Concrete Product is a class that implements the Product interface or inherits from the abstract class. Each concrete product corresponds to a specific type of product created by the factory method.
   * **Purpose**: It provides the actual implementation of the Product interface.
   * **Example in Python**:

python

Copy code

class Truck(Transport):

def deliver(self) -> str:

return "Deliver by land in a box."

class Ship(Transport):

def deliver(self) -> str:

return "Deliver by sea in a container."

1. **Creator**:
   * **Definition**: The Creator is an abstract class or interface that declares the factory method, which returns an object of type Product. The Creator may also contain some core business logic that relies on Product objects, which are returned by the factory method.
   * **Purpose**: It provides the method signature for creating products but defers the actual creation to its subclasses.
   * **Example in Python**:

class Logistics(ABC):

@abstractmethod

def create\_transport(self) -> Transport:

pass

def plan\_delivery(self) -> str:

transport = self.create\_transport()

return transport.deliver()

1. **Concrete Creator**:
   * **Definition**: The Concrete Creator is a class that inherits from the Creator and implements the factory method. It overrides the factory method to create and return an instance of a Concrete Product.
   * **Purpose**: It specifies which concrete product to instantiate.
   * **Example in Python**:

python

Copy code

class RoadLogistics(Logistics):

def create\_transport(self) -> Transport:

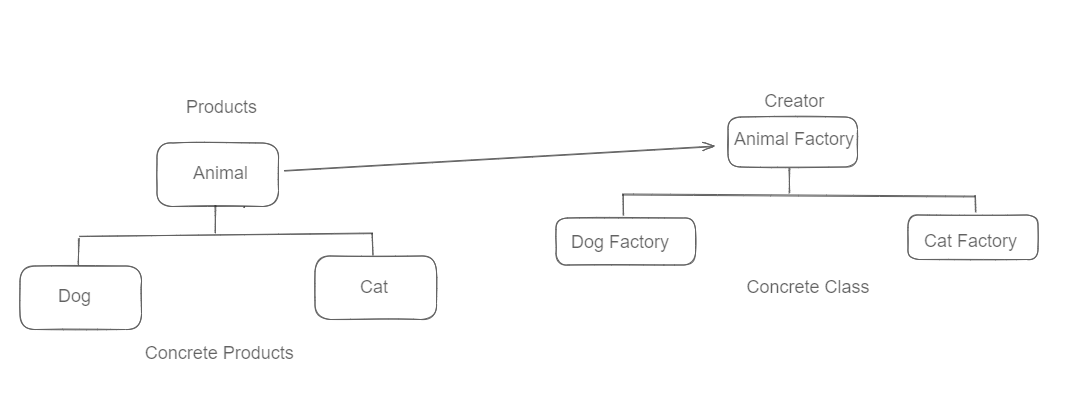
return Truck()

class SeaLogistics(Logistics):

def create\_transport(self) -> Transport:

return Ship()

Another Example



from abc import ABC, abstractmethod

#Products

class Animal(ABC):

@abstractmethod

def speak(self):

pass

#Concrete Products

class Dog(Animal):

def speak(self):

return "Woof!"

#Concrete Products

class Cat(Animal):

def speak(self):

return "Meow!"

#Creator Class

class AnimalFactory(ABC):

@abstractmethod

def create\_animal(self):

pass

#Concrete Creator

class DogFactory(AnimalFactory):

def create\_animal(self):

return Dog()

#Concrete Creator

class CatFactory(AnimalFactory):

def create\_animal(self):

return Cat()

# Example usage:

dog\_factory = DogFactory()

dog = dog\_factory.create\_animal()

print(dog.speak()) # Output: Woof!

cat\_factory = CatFactory()

cat = cat\_factory.create\_animal()

print(cat.speak()) # Output: Meow!

Another Example

class PaymentProcessor:

def process\_payment(self, amount):

raise NotImplementedError

class CreditCardProcessor(PaymentProcessor):

def process\_payment(self, amount):

# Process credit card payment using credit card processing API

return "Process through Credit Card"

class PayPalProcessor(PaymentProcessor):

def process\_payment(self, amount):

# Process PayPal payment using PayPal API

return "Process through PayPal Card"

class ApplePayProcessor(PaymentProcessor):

def process\_payment(self, amount):

# Process Apple Pay payment using Apple Pay API

return "Process through PayPal Card"

class PaymentProcessorFactory:

def get\_payment\_processor(self, payment\_method):

if payment\_method == "credit\_card":

return CreditCardProcessor()

elif payment\_method == "paypal":

return PayPalProcessor()

elif payment\_method == "apple\_pay":

return ApplePayProcessor()

else:

raise ValueError("Invalid payment method")

factory = PaymentProcessorFactory()

credit\_card\_processor = factory.get\_payment\_processor("credit\_card")

print(credit\_card\_processor.process\_payment(100.0))

paypal\_processor = factory.get\_payment\_processor("paypal")

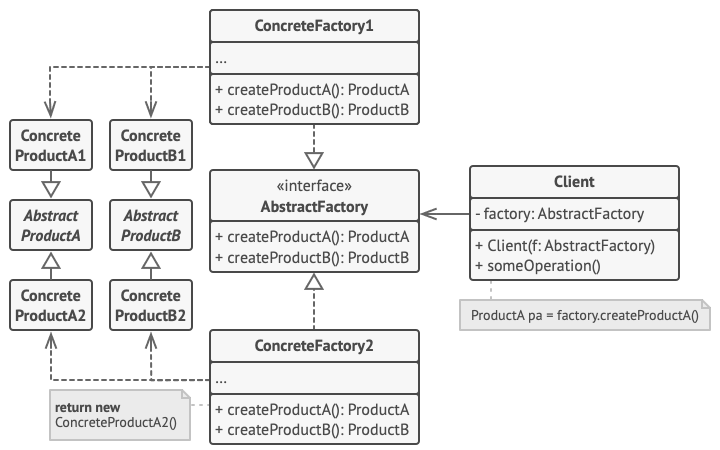
print(paypal\_processor.process\_payment(50.0))

apple\_pay\_processor = factory.get\_payment\_processor("apple\_pay")

print(apple\_pay\_processor.process\_payment(25.0))

Abstract Factory Pattern

**Abstract Factory** is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.



Here's a simple example in Python to illustrate the Abstract Factory Pattern:

### Abstract Factory Pattern in Python

1. **Abstract Factory Interface**: Defines methods for creating abstract products.
2. **Concrete Factory Classes**: Implement the creation methods for concrete products.
3. **Abstract Product Interfaces**: Define interfaces for a set of related products.
4. **Concrete Product Classes**: Implement the interfaces defined by the abstract product interfaces.

#### Step-by-Step Implementation

1. **Abstract Product Interfaces**

from abc import ABC, abstractmethod

class AbstractProductA(ABC):

@abstractmethod

def operation\_a(self) -> str:

pass

class AbstractProductB(ABC):

@abstractmethod

def operation\_b(self) -> str:

pass

1. **Concrete Product Classes**

class ConcreteProductA1(AbstractProductA):

def operation\_a(self) -> str:

return "Result of ConcreteProductA1"

class ConcreteProductA2(AbstractProductA):

def operation\_a(self) -> str:

return "Result of ConcreteProductA2"

class ConcreteProductB1(AbstractProductB):

def operation\_b(self) -> str:

return "Result of ConcreteProductB1"

class ConcreteProductB2(AbstractProductB):

def operation\_b(self) -> str:

return "Result of ConcreteProductB2"

1. **Abstract Factory Interface**

class AbstractFactory(ABC):

@abstractmethod

def create\_product\_a(self) -> AbstractProductA:

pass

@abstractmethod

def create\_product\_b(self) -> AbstractProductB:

pass

1. **Concrete Factory Classes**

class ConcreteFactory1(AbstractFactory):

def create\_product\_a(self) -> AbstractProductA:

return ConcreteProductA1()

def create\_product\_b(self) -> AbstractProductB:

return ConcreteProductB1()

class ConcreteFactory2(AbstractFactory):

def create\_product\_a(self) -> AbstractProductA:

return ConcreteProductA2()

def create\_product\_b(self) -> AbstractProductB:

return ConcreteProductB2()

1. **Client Code**

def client\_code(factory: AbstractFactory) -> None:

product\_a = factory.create\_product\_a()

product\_b = factory.create\_product\_b()

print(product\_a.operation\_a())

print(product\_b.operation\_b())

if \_\_name\_\_ == "\_\_main\_\_":

factory1 = ConcreteFactory1()

client\_code(factory1)

factory2 = ConcreteFactory2()

client\_code(factory2)

### Explanation

1. **Abstract Product Interfaces**: AbstractProductA and AbstractProductB define the interfaces for products.
2. **Concrete Product Classes**: ConcreteProductA1, ConcreteProductA2, ConcreteProductB1, and ConcreteProductB2 are concrete implementations of the abstract products.
3. **Abstract Factory Interface**: AbstractFactory defines methods for creating abstract products.
4. **Concrete Factory Classes**: ConcreteFactory1 and ConcreteFactory2 implement the creation methods for concrete products.
5. **Client Code**: The client code uses the abstract factory to create products. It doesn't need to know the concrete classes of the products it works with, as long as it works with products through their abstract interfaces.

This pattern provides flexibility and decouples the creation of products from their usage, allowing you to change the product families without modifying the client code.

**Singleton Patern**

**Singleton** is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

The singleton pattern is a design pattern that restricts the instantiation of a class to a single instance. This is useful when exactly one object is needed to coordinate actions across the system. The singleton pattern ensures that a class has only one instance and provides a global point of access to it.



Singleton Pattern Implementation Approach 1

class Singleton:

\_instance = None

@classmethod

def get\_instance(cls):

if cls.\_instance is None:

cls.\_instance = cls()

return cls.\_instance

if \_\_name\_\_ == "\_\_main\_\_":

s1 = Singleton().get\_instance()

s2 = Singleton().get\_instance()

if s1 is s2:

print("Same instance")

else:

print("Different instances")

Singleton Pattern Implementation Approach 2

class SingletonMeta(type):

\_instances = {}

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

if cls not in cls.\_instances:

cls.\_instances[cls] = super().\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instances[cls]

class Singleton(metaclass=SingletonMeta):

pass

if \_\_name\_\_ == "\_\_main\_\_":

s1 = Singleton()

s2 = Singleton()

if s1 is s2:

print("Same instance")

else:

print("Different instances")

**Why simple singlemaeta fails with threading**?

The simple SingletonMeta implementation may fail in a multi-threaded environment because it is not thread-safe. If two or more threads attempt to create an instance of the singleton at the same time, it is possible that multiple instances could be created. This happens because the check to see if an instance already exists and the creation of the instance are not atomic (i.e., not a single, indivisible operation).

**Example of the Problem**

Consider this scenario:

1. Thread A checks if the instance exists and sees that it does not.
2. Thread B also checks if the instance exists and sees that it does not.
3. Both Thread A and Thread B proceed to create a new instance, resulting in two instances being created.

**Adding Thread Safety with a Lock**

To prevent this, you can use a lock to ensure that only one thread can execute the instance creation code at a time. Here’s how you can modify the SingletonMeta to be thread-safe:

import threading

class SingletonMeta(type):

\_instances = {}

\_lock = threading.Lock() # Add a lock object to ensure thread safety

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

with cls.\_lock: # Ensure that only one thread can create an instance at a time

if cls not in cls.\_instances:

cls.\_instances[cls] = super().\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instances[cls]

class Singleton(metaclass=SingletonMeta):

pass

# Usage in a multithreaded context

def create\_singleton():

singleton = Singleton()

print(f"Singleton instance ID: {id(singleton)}")

threads = [threading.Thread(target=create\_singleton) for \_ in range(10)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

**Explanation of the Thread-Safe Implementation**

1. **Lock Initialization**: A lock object \_lock is created as a class attribute of SingletonMeta.
2. **Lock Acquisition**: The with cls.\_lock: statement ensures that the following block of code is executed by only one thread at a time.
3. **Instance Check and Creation**: Within the locked block, it checks if the instance already exists. If it does not, it creates a new instance. This ensures that no two threads can create the instance simultaneously.
4. **Thread Execution**: In the usage example, multiple threads are created and started. Each thread attempts to create an instance of Singleton, but due to the lock, only one instance is created.

**Why the Simple Implementation Fails**

Without the lock, the simple implementation could look like this:

class SingletonMeta(type):

\_instances = {}

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

if cls not in cls.\_instances:

cls.\_instances[cls] = super().\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instances[cls]

class Singleton(metaclass=SingletonMeta):

pass

In this case, if multiple threads execute the \_\_call\_\_ method at the same time, they could all pass the if cls not in cls.\_instances check before any of them actually create the instance, leading to multiple instances being created.

**Builder Pattern**

**Builder** is a creational design pattern, which allows constructing complex objects step by step.

The Builder pattern is a creational design pattern that separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It aims to handle the construction of complex objects by breaking it down into multiple steps, encapsulating the logic for each step within separate builder objects.

### Key Components of the Builder Pattern:

1. **Product**: Represents the complex object to be created. It typically contains multiple parts or components that need to be assembled.
2. **Builder**: Defines an abstract interface for creating parts of a Product. It declares methods for assembling the different parts of the product.
3. **Concrete Builder**: Implements the Builder interface to build and assemble parts of the Product. Each Concrete Builder provides a specific implementation of how to build the parts.
4. **Director (optional)**: Controls the construction process by invoking the Builder's methods to build the parts sequentially. It abstracts the client code from knowing the specifics of how the parts are assembled.

from abc import ABC,abstractmethod

#Define Product

class Pizza:

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

self.ingredients = []

def add\_ingredients(self, ingredient):

self.ingredients.append(ingredient)

def list\_ingredients(self) -> None:

print(f"Pizza Ingridients : {','.join(self.ingredients)}")

#Define Builder Interface

class PizzaBuilder(ABC):

@property

@abstractmethod

def pizza(self) -> None:

pass

@abstractmethod

def produce\_dough(self):

pass

@abstractmethod

def produce\_sauce(self):

pass

@abstractmethod

def produce\_topping(self):

pass

@abstractmethod

def produce\_cheese(self):

pass

class VegPizzaBuilder(PizzaBuilder):

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

self.reset()

def reset(self) -> None:

self.\_pizza = Pizza()

@property

def pizza(self) -> Pizza:

return self.\_pizza

def produce\_dough(self) -> None:

self.pizza.add\_ingredients("Whole White Dough")

def produce\_sauce(self) -> None:

self.pizza.add\_ingredients("Tomato Sauce")

def produce\_topping(self) -> None:

self.pizza.add\_ingredients("Onions, Mushrooms, Olives")

def produce\_cheese(self) -> None:

self.pizza.add\_ingredients("Mozzarella Cheese")

class MeatPizzaBuilder(PizzaBuilder):

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

self.reset()

def reset(self)-> None:

self.\_pizza = Pizza()

@property

def pizza(self) -> Pizza:

return self.\_pizza

def produce\_dough(self) -> None:

self.pizza.add\_ingredients("Regular Dough")

def produce\_sauce(self) -> None:

self.pizza.add\_ingredients("Barbecue Sauce")

def produce\_topping(self) -> None:

self.pizza.add\_ingredients("Pepperoni, Sausage, Bacon")

def produce\_cheese(self) -> None:

self.pizza.add\_ingredients("Cheddar Cheese")

#Create the Dictator

class PizzaDirector:

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

self.\_builder = None

@property

def builder(self) -> PizzaBuilder:

return self.\_builder

@builder.setter

def builder(self, builder: PizzaBuilder) -> None:

self.\_builder = builder

def build\_vegetarian\_pizza(self) -> None:

self.builder.produce\_dough()

self.builder.produce\_sauce()

self.builder.produce\_topping()

self.builder.produce\_cheese()

def build\_meat\_pizza(self) -> None:

self.builder.produce\_dough()

self.builder.produce\_sauce()

self.builder.produce\_topping()

self.builder.produce\_cheese()

if \_\_name\_\_ == "\_\_main\_\_":

director = PizzaDirector()

print("Vegetarian Pizza: ")

veg\_builder = VegPizzaBuilder()

director.builder = veg\_builder

director.build\_vegetarian\_pizza()

veg\_builder.pizza.list\_ingredients()

print("\n")

print("Meat Lovers Pizza: ")

meat\_builder = MeatPizzaBuilder()

director.builder = meat\_builder

director.build\_meat\_pizza()

meat\_builder.pizza.list\_ingredients()

print("\n")

print("Custom Pizza: ")

custom\_builder = VegPizzaBuilder()

custom\_builder.produce\_dough()

custom\_builder.produce\_sauce()

custom\_builder.produce\_cheese()

custom\_builder.pizza.list\_ingredients()

Adapter Pattern

The Adapter Design Pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a bridge between two incompatible interfaces, enabling them to work together without changing their existing code.

**Key Concepts:**

1. **Target Interface**: The interface that the client expects.
2. **Adaptee**: The existing class with a different interface that needs to be adapted.
3. **Adapter**: A class that implements the target interface and translates the requests from the client to the adaptee in a form the adaptee can understand.

**Definition:**

The Adapter Design Pattern converts the interface of a class into another interface that the client expects. It allows classes to work together that couldn't otherwise because of incompatible interfaces by providing a wrapper that translates the client’s calls to the adaptee's methods.

**Real-World Analogy:**

Imagine you have a plug that fits into European electrical sockets, but you are in the USA where the sockets are different. An adapter would be a device that you plug your European plug into, and on the other end, it fits into the USA socket. The adapter does not change the actual plug or the socket; it just makes them compatible with each other.

**Detailed Example: Payment Gateway Integration**

Imagine you have an e-commerce application that needs to support multiple payment gateways like Stripe and PayPal. Each gateway has its own API and methods for processing payments. By using the Adapter Pattern, you can create a common interface for payment processing and then implement adapters for each gateway.

# Target Interface

class PaymentProcessor:

def process\_payment(self, amount):

pass

# Adaptee 1

class Stripe:

def charge(self, amount):

return f"Charging {amount} using Stripe"

# Adaptee 2

class PayPal:

def make\_payment(self, amount):

return f"Paying {amount} using PayPal"

# Adapter for Stripe

class StripeAdapter(PaymentProcessor):

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

self.stripe = stripe

def process\_payment(self, amount):

return self.stripe.charge(amount)

# Adapter for PayPal

class PayPalAdapter(PaymentProcessor):

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

self.paypal = paypal

def process\_payment(self, amount):

return self.paypal.make\_payment(amount)

# Client code

def main():

stripe = Stripe()

paypal = PayPal()

stripe\_adapter = StripeAdapter(stripe)

paypal\_adapter = PayPalAdapter(paypal)

print(stripe\_adapter.process\_payment(100)) # Output: Charging 100 using Stripe

print(paypal\_adapter.process\_payment(200)) # Output: Paying 200 using PayPal

if \_\_name\_\_ == "\_\_main\_\_":

main()

In this example:

* PaymentProcessor is the target interface expected by the client.
* Stripe and PayPal are the existing classes with their specific methods.
* StripeAdapter and PayPalAdapter are adapters that implement the PaymentProcessor interface, translating calls to the respective methods of Stripe and PayPal.

This allows the client code to process payments without worrying about the specific details of each payment gateway.

# Real-world Use Cases: Adapter in Action

The Adapter pattern is a powerful design pattern with various practical applications. Here are some common use cases:

1. **Legacy System Integration:**When you need to integrate a legacy system or library with modern code, the Adapter pattern can make the transition smoother. It allows you to wrap the legacy code with an adapter, ensuring it conforms to the expected interface of the new system.
2. **Third-Party Libraries:**When working with third-party libraries or APIs that do not align with your system’s requirements, adapters can serve as intermediaries. They translate the third-party interface into one that your codebase understands.
3. **Interface Evolution:**As your software evolves, you may encounter situations where the interfaces of existing classes need to change. Adapters can help maintain backward compatibility by presenting the old interface while internally implementing the new one.

# Advantages of the Adapter Pattern

The Adapter pattern offers several advantages when applied judiciously:

1. **Code Reusability:**Adapters facilitate code reuse by integrating existing classes without altering their source, reducing redundancy and enabling well-tested code in new contexts.
2. **Principle of Single Responsibility:** The Adapter pattern aligns with the principle of single responsibility, ensuring that each class has a clear and specific role, thus improving code organization and maintainability.
3. **Open/Closed Principle:** By adapting existing interfaces rather than modifying them directly, the Adapter pattern adheres to the open/closed principle, allowing for extension without altering existing code, enhancing system stability.
4. **Integration of Third-Party Code:**When using external libraries or APIs, the Adapter pattern eases their integration, shielding your codebase from external changes and minimizing disruptions during updates.

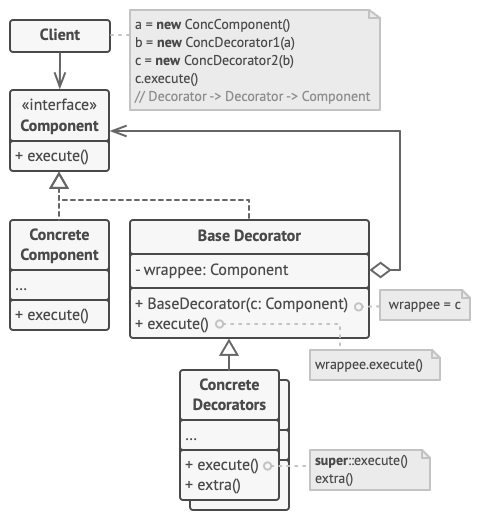
# Considerations and Potential Drawbacks

1. **Complexity:** Adapters can add complexity, especially with many to manage, impacting code maintainability.
2. **Performance Overhead:** The added layer may introduce slight performance overhead, a concern for high-performance applications.
3. **Design Implications:** Adapter use may signal original design issues; consider comprehensive refactoring for core interface incompatibilities.

Decorator Pattern

The decorator pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. This is useful for adhering to the Open/Closed Principle, which states that objects should be open for extension but closed for modification.

**Key Concepts**

* **Component**: An interface or abstract class that defines the behavior.
* **Concrete Component**: The class that implements the Component interface.
* **Decorator**: An abstract class that implements the Component interface and contains a reference to a Component object.
* **Concrete Decorators**: Classes that extend the Decorator class and add new behavior.
* 

### Class Definitions

#### 1. Component Interface

python

Copy code

class Component():

"""

The base Component interface defines operations that can be altered by

decorators.

"""

def operation(self) -> str:

pass

* **Purpose**: This is an interface for components. It declares a method operation that can be implemented or altered by decorators.
* **Method**: operation is defined but not implemented, serving as a placeholder for subclasses.

#### 2. ConcreteComponent

python

Copy code

class ConcreteComponent(Component):

"""

Concrete Components provide default implementations of the operations. There

might be several variations of these classes.

"""

def operation(self) -> str:

return "ConcreteComponent"

* **Purpose**: This class provides a concrete implementation of the Component interface.
* **Method**: operation returns the string "ConcreteComponent".

#### 3. Decorator Base Class

class Decorator(Component):

"""

The base Decorator class follows the same interface as the other components.

The primary purpose of this class is to define the wrapping interface for

all concrete decorators. The default implementation of the wrapping code

might include a field for storing a wrapped component and the means to

initialize it.

"""

\_component: Component = None

def \_\_init\_\_(self, component: Component) -> None:

self.\_component = component

@property

def component(self) -> Component:

"""

The Decorator delegates all work to the wrapped component.

"""

return self.\_component

def operation(self) -> str:

return self.\_component.operation()

* **Purpose**: This is the base class for all decorators. It implements the Component interface and stores a reference to a Component object.
* **Constructor**: \_\_init\_\_ initializes the decorator with a Component object.
* **Property**: component returns the wrapped Component.
* **Method**: operation delegates the call to the wrapped Component's operation method.

#### 4. ConcreteDecoratorA

class ConcreteDecoratorA(Decorator):

"""

Concrete Decorators call the wrapped object and alter its result in some

way.

"""

def operation(self) -> str:

"""

Decorators may call parent implementation of the operation, instead of

calling the wrapped object directly. This approach simplifies extension

of decorator classes.

"""

return f"ConcreteDecoratorA({self.component.operation()})"

* **Purpose**: This class extends Decorator and modifies the behavior of the operation method.
* **Method**: operation calls the operation of the wrapped component and adds its own behavior, returning a modified string.

#### 5. ConcreteDecoratorB

class ConcreteDecoratorB(Decorator):

"""

Decorators can execute their behavior either before or after the call to a

wrapped object.

"""

def operation(self) -> str:

return f"ConcreteDecoratorB({self.component.operation()})"

* **Purpose**: Similar to ConcreteDecoratorA, this class extends Decorator and modifies the behavior of the operation method.
* **Method**: operation calls the operation of the wrapped component and adds its own behavior, returning a modified string.

### Client Code

def client\_code(component: Component) -> None:

"""

The client code works with all objects using the Component interface. This

way it can stay independent of the concrete classes of components it works

with.

"""

print(f"RESULT: {component.operation()}", end="")

* **Purpose**: This function demonstrates the use of the Component interface. It can work with both simple and decorated components.
* **Method**: operation is called on the provided component and its result is printed.

### Main Execution

if \_\_name\_\_ == "\_\_main\_\_":

# This way the client code can support both simple components...

simple = ConcreteComponent()

print("Client: I've got a simple component:")

client\_code(simple)

print("\n")

# ...as well as decorated ones.

#

# Note how decorators can wrap not only simple components but the other

# decorators as well.

decorator1 = ConcreteDecoratorA(simple)

decorator2 = ConcreteDecoratorB(decorator1)

print("Client: Now I've got a decorated component:")

client\_code(decorator2)

* **Purpose**: Demonstrates how the client code can handle both plain and decorated components.
* **Steps**:
  1. Create a ConcreteComponent instance and pass it to client\_code.
  2. Wrap the ConcreteComponent with ConcreteDecoratorA and then with ConcreteDecoratorB.
  3. Pass the decorated component to client\_code.

### Execution Result

1. When the simple component is used:

Client: I've got a simple component:

RESULT: ConcreteComponent

1. When the decorated component is used:

Client: Now I've got a decorated component:

RESULT: ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))

The decorators modify the output of the operation method by wrapping additional behavior around the original component's behavior.Top of Form

Façade Pattern

The **Facade Design Pattern** is a structural pattern that provides a simplified interface to a complex subsystem or set of interfaces. It involves creating a single class, called the facade, which hides the complexities of the subsystem and provides a more straightforward, unified interface to the client.

**Key Points:**

1. **Simplification:** The facade simplifies the interface for the client, making it easier to use the subsystem without needing to understand its intricacies.
2. **Decoupling:** It decouples the client code from the subsystem, promoting better modularity and flexibility in the codebase.
3. **Unified Interface:** It presents a single, unified interface to the client for interacting with multiple classes or components of the subsystem.
4. **Complexity Hiding:** The facade hides the complexity of the underlying subsystem by delegating client requests to appropriate subsystem objects.
5. **Easier Maintenance:** By providing a simpler interface, the facade can make the system easier to maintain and extend.

**Intent**

The Facade pattern is intended to:

* Provide a unified interface to a set of interfaces in a subsystem.
* Define a higher-level interface that makes the subsystem easier to use.

**Problem**

When you have a complex system with many interacting components, it can be difficult for clients to use the system correctly and efficiently. Clients need to understand the intricacies of the system to perform operations.

**Solution**

The Facade pattern provides a simplified interface to these complex subsystems. This interface delegates the client requests to the appropriate components within the subsystem. By doing so, it hides the complexity and makes the subsystem easier to use.

**from** \_\_future\_\_ **import** annotations

**class** **Facade**:

"""

The Facade class provides a simple interface to the complex logic of one or

several subsystems. The Facade delegates the client requests to the

appropriate objects within the subsystem. The Facade is also responsible for

managing their lifecycle. All of this shields the client from the undesired

complexity of the subsystem.

"""

**def** **\_\_init\_\_**(self, subsystem1: Subsystem1, subsystem2: Subsystem2) -> **None**:

"""

Depending on your application's needs, you can provide the Facade with

existing subsystem objects or force the Facade to create them on its

own.

"""

self.\_subsystem1 = subsystem1 **or** Subsystem1()

self.\_subsystem2 = subsystem2 **or** Subsystem2()

**def** **operation**(self) -> str:

"""

The Facade's methods are convenient shortcuts to the sophisticated

functionality of the subsystems. However, clients get only to a fraction

of a subsystem's capabilities.

"""

results = []

results.append("Facade initializes subsystems:")

results.append(self.\_subsystem1.operation1())

results.append(self.\_subsystem2.operation1())

results.append("Facade orders subsystems to perform the action:")

results.append(self.\_subsystem1.operation\_n())

results.append(self.\_subsystem2.operation\_z())

**return** "\n".join(results)

**class** **Subsystem1**:

"""

The Subsystem can accept requests either from the facade or client directly.

In any case, to the Subsystem, the Facade is yet another client, and it's

not a part of the Subsystem.

"""

**def** **operation1**(self) -> str:

**return** "Subsystem1: Ready!"

# ...

**def** **operation\_n**(self) -> str:

**return** "Subsystem1: Go!"

**class** **Subsystem2**:

"""

Some facades can work with multiple subsystems at the same time.

"""

**def** **operation1**(self) -> str:

**return** "Subsystem2: Get ready!"

# ...

**def** **operation\_z**(self) -> str:

**return** "Subsystem2: Fire!"

**def** **client\_code**(facade: Facade) -> **None**:

"""

The client code works with complex subsystems through a simple interface

provided by the Facade. When a facade manages the lifecycle of the

subsystem, the client might not even know about the existence of the

subsystem. This approach lets you keep the complexity under control.

"""

print(facade.operation(), end="")

**if** \_\_name\_\_ == "\_\_main\_\_":

# The client code may have some of the subsystem's objects already created.

# In this case, it might be worthwhile to initialize the Facade with these

# objects instead of letting the Facade create new instances.

subsystem1 = Subsystem1()

subsystem2 = Subsystem2()

facade = Facade(subsystem1, subsystem2)

client\_code(facade)

### Example in Python

Suppose we have a home theater system with several components like a DVD player, a projector, an amplifier, and lights. Without a Facade, you would need to control each component individually. The Facade pattern can simplify this by providing a single interface for controlling the entire home theater.

#### Step-by-Step Implementation:

1. **Subsystem Classes**: These are the complex parts of the system.

class DVDPlayer:

def on(self):

print("DVD Player on")

def play(self, movie):

print(f"Playing {movie}")

def off(self):

print("DVD Player off")

class Projector:

def on(self):

print("Projector on")

def wide\_screen\_mode(self):

print("Projector in widescreen mode")

def off(self):

print("Projector off")

class Amplifier:

def on(self):

print("Amplifier on")

def set\_volume(self, level):

print(f"Amplifier volume set to {level}")

def off(self):

print("Amplifier off")

class TheaterLights:

def dim(self, level):

print(f"Theater lights dimmed to {level}%")

def on(self):

print("Theater lights on")

1. **Facade Class**: This class provides a simplified interface to control the home theater system.

class HomeTheaterFacade:

def \_\_init\_\_(self, dvd, projector, amp, lights):

self.dvd = dvd

self.projector = projector

self.amp = amp

self.lights = lights

def watch\_movie(self, movie):

print("Get ready to watch a movie...")

self.lights.dim(10)

self.projector.on()

self.projector.wide\_screen\_mode()

self.amp.on()

self.amp.set\_volume(5)

self.dvd.on()

self.dvd.play(movie)

def end\_movie(self):

print("Shutting movie theater down...")

self.lights.on()

self.projector.off()

self.amp.off()

self.dvd.off()

1. **Client Code**: The client interacts with the Facade to control the home theater system.

if \_\_name\_\_ == "\_\_main\_\_":

dvd = DVDPlayer()

projector = Projector()

amp = Amplifier()

lights = TheaterLights()

home\_theater = HomeTheaterFacade(dvd, projector, amp, lights)

home\_theater.watch\_movie("Inception")

print("\nAfter the movie...\n")

home\_theater.end\_movie()

### Explanation:

* **Subsystem Classes**: Each class represents a component of the home theater system with methods to control it.
* **Facade Class**: HomeTheaterFacade provides methods watch\_movie and end\_movie to simplify the interaction with the subsystem classes.
* **Client Code**: The client uses the Facade to watch and end the movie, without needing to directly interact with each subsystem class.

### Benefits of the Facade Pattern:

* **Simplifies Usage**: It hides the complexities of the subsystem from the client.
* **Loose Coupling**: Reduces the dependency of client code on the subsystem classes.
* **Improves Readability**: Makes the client code easier to read and maintain.

### Drawbacks:

* **Less Flexibility**: The Facade might limit the functionality accessible to the client since it only exposes certain parts of the subsystem's functionality.

The Facade pattern is particularly useful when you have a complex system and you want to provide a simple interface to the clients. This way, clients can perform high-level operations without getting bogged down by the details of the subsystem.

Proxy Pattern

The Proxy Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. This pattern involves creating a proxy class that acts as an interface to something else, such as a network connection, a large object in memory, a file, or some other resource that is expensive or impractical to duplicate.

**Key Concepts of Proxy Pattern:**

1. **Proxy Class**: Acts as an intermediary between the client and the real object. It controls access to the real object, allowing certain actions to be performed before or after the request reaches the real object.
2. **Real Subject**: The actual object that the proxy represents. The proxy controls access to this object.
3. **Client**: The client interacts with the proxy instead of the real subject. The proxy can then decide whether to forward the request to the real subject or handle it itself.

**Types of Proxies:**

1. **Virtual Proxy**: Manages the creation and initialization of the expensive object. It can delay the creation of the object until it is needed.
2. **Remote Proxy**: Represents an object that is located in a different address space. This is often used in distributed systems.
3. **Protection Proxy**: Controls access to the original object based on access rights.
4. **Smart Proxy**: Performs additional actions when an object is accessed, such as reference counting or lazy initialization.

**Example:**

Imagine a scenario where you have a Database class that connects to a database and executes queries. Establishing a connection to the database might be resource-intensive, so you don't want to create a connection until it's actually needed.

class Database:

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

self.connect()

def connect(self):

print("Connecting to the database...")

def execute\_query(self, query):

print(f"Executing query: {query}")

class DatabaseProxy:

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

self.\_database = None

def execute\_query(self, query):

if self.\_database is None:

self.\_database = Database()

self.\_database.execute\_query(query)

# Client code

proxy = DatabaseProxy()

proxy.execute\_query("SELECT \* FROM users")

**Benefits:**

1. **Controlled Access**: The proxy controls access to the real object, providing a layer of security or access control.
2. **Lazy Initialization**: Delays the creation and initialization of the object until it is necessary.
3. **Performance Optimization**: Can help optimize performance by controlling expensive operations and managing resources efficiently.

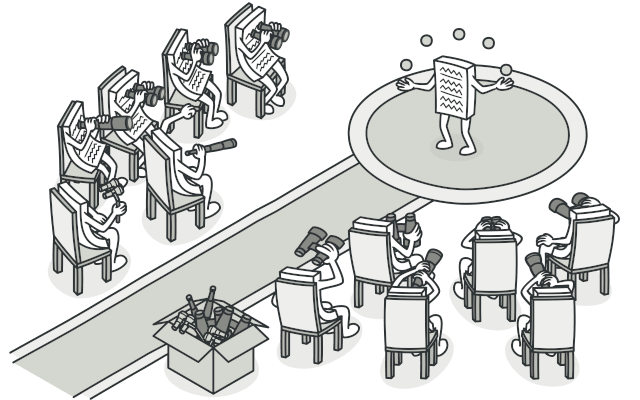
**Drawbacks:**

1. **Complexity**: Introduces additional complexity into the system by adding another layer of abstraction.
2. **Overhead**: May add some overhead due to the extra level of indirection.

The Proxy Pattern is useful in scenarios where object creation, access, or usage needs to be controlled, delayed, or managed in some way. It provides a flexible and efficient way to manage resources and control access to objects.

Observer Pattern

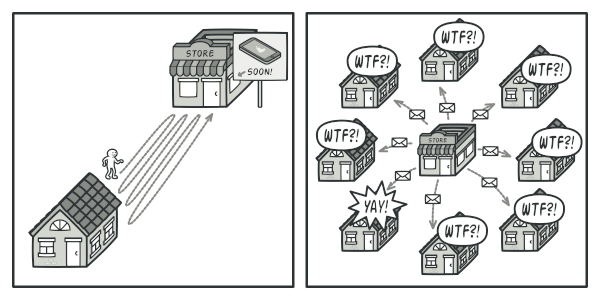
**Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.



## Problem

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it’s a new model of the iPhone) which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.



Visiting the store vs. sending spam

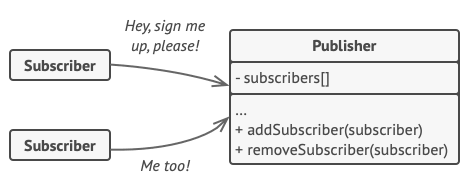
On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it’d upset other customers who aren’t interested in new products.

It looks like we’ve got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

## Solution

The object that has some interesting state is often called subject, but since it’s also going to notify other objects about the changes to its state, we’ll call it publisher. All other objects that want to track changes to the publisher’s state are called subscribers.

The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. Fear not! Everything isn’t as complicated as it sounds. In reality, this mechanism consists of 1) an array field for storing a list of references to subscriber objects and 2) several public methods which allow adding subscribers to and removing them from that list.

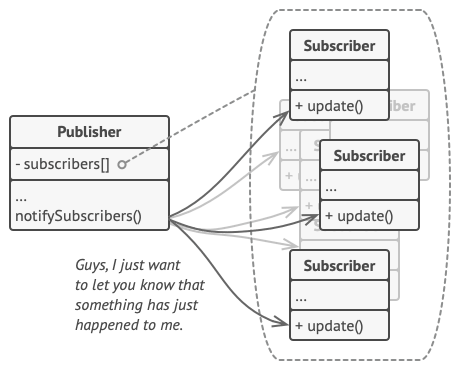


A subscription mechanism lets individual objects subscribe to event notifications.

Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

Real apps might have dozens of different subscriber classes that are interested in tracking events of the same publisher class. You wouldn’t want to couple the publisher to all of those classes. Besides, you might not even know about some of them beforehand if your publisher class is supposed to be used by other people.

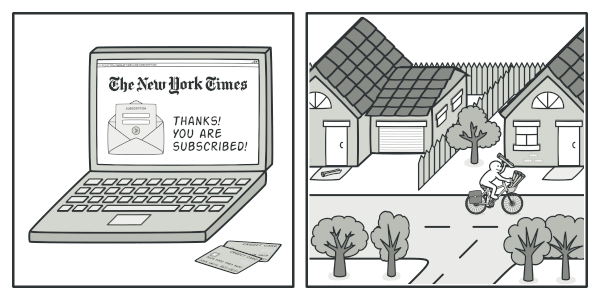
That’s why it’s crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface. This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.



Publisher notifies subscribers by calling the specific notification method on their objects.

If your app has several different types of publishers and you want to make your subscribers compatible with all of them, you can go even further and make all publishers follow the same interface. This interface would only need to describe a few subscription methods. The interface would allow subscribers to observe publishers’ states without coupling to their concrete classes.

## Real-World Analogy



Magazine and newspaper subscriptions.

If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.

The publisher maintains a list of subscribers and knows which magazines they’re interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

**Strategy Pattern**

**Strategy** is a behavioral design pattern that turns a set of behaviors into objects and makes them interchangeable inside original context object.

The original object, called context, holds a reference to a strategy object. The context delegates executing the behavior to the linked strategy object. In order to change the way the context performs its work, other objects may replace the currently linked strategy object with another one.

Here's a breakdown of the strategy pattern:

1. **Context**: This is the class that uses a Strategy. The Context doesn't implement the behavior directly but instead delegates it to one of the Strategy objects.
2. **Strategy Interface**: This is an interface common to all supported strategies. The Context uses this interface to call the algorithm defined by a Concrete Strategy.
3. **Concrete Strategies**: These are classes that implement the Strategy interface, each providing a different implementation of the algorithm.

By using the strategy pattern, you can easily switch between different algorithms or policies at runtime without modifying the Context. This promotes the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. It also allows for better organization and maintainability of code by separating the concerns of algorithm implementation from the context in which they are used.

**Real World Analogy**

Imagine that you have to get to the airport. You can catch a bus, order a cab, or get on your bicycle. These are your transportation strategies. You can pick one of the strategies depending on factors such as budget or time constraints.

### Example: Strategy Pattern in a Context of a Travel Planner Application

Let's consider a travel planner application that can recommend different modes of transportation (e.g., car, bus, bicycle). Depending on the user's preference or current traffic conditions, the application should be able to switch between different transportation strategies.

#### Strategy Interface

from abc import ABC, abstractmethod

class TransportationStrategy(ABC):

@abstractmethod

def travel(self, start: str, destination: str):

pass

#### Concrete Strategies

class CarStrategy(TransportationStrategy):

def travel(self, start: str, destination: str):

print(f"Traveling from {start} to {destination} by car.")

class BusStrategy(TransportationStrategy):

def travel(self, start: str, destination: str):

print(f"Traveling from {start} to {destination} by bus.")

class BicycleStrategy(TransportationStrategy):

def travel(self, start: str, destination: str):

print(f"Traveling from {start} to {destination} by bicycle.")

class TravelPlanner:

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

self.\_strategy = strategy

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

self.\_strategy = strategy

def plan\_trip(self, start: str, destination: str):

self.\_strategy.travel(start, destination)

#### Client Code

if \_\_name\_\_ == "\_\_main\_\_":

car\_strategy = CarStrategy()

bus\_strategy = BusStrategy()

bicycle\_strategy = BicycleStrategy()

planner = TravelPlanner(car\_strategy)

planner.plan\_trip("Home", "Office")

planner.set\_strategy(bus\_strategy)

planner.plan\_trip("Home", "Office")

planner.set\_strategy(bicycle\_strategy)

planner.plan\_trip("Home", "Park")

In this example:

* The TransportationStrategy interface defines the common method travel for all transportation strategies.
* CarStrategy, BusStrategy, and BicycleStrategy are concrete implementations of the transportation strategies.
* TravelPlanner is the context that uses a transportation strategy to plan a trip.
* The client code demonstrates how to use different transportation strategies at runtime by setting the desired strategy on the TravelPlanner and planning a trip.

Command Pattern

The Command Pattern is a behavioral design pattern that turns a request into a stand-alone object containing all the information about the request. This transformation allows for parameterizing methods with different requests, delaying or queuing a request's execution, and supporting undoable operations.

**Key Components of the Command Pattern**

1. **Command Interface**: Declares an interface for executing an operation. It typically defines an execute() method that performs the command.
2. **Concrete Command**: Implements the Command interface and defines the binding between a receiver and an action. It often includes a reference to the receiver object that performs the actual work and implements the execute() method to call the receiver's methods.
3. **Receiver**: The object that knows how to perform the operations associated with carrying out a request. The receiver executes the actual command's action.
4. **Invoker**: Asks the command to carry out the request. The invoker does not know about the receiver's operations but only knows about the command interface. It stores a command object and executes it when requested.
5. **Client**: Creates a command object and sets its receiver. The client decides which commands to execute and when.

**Example**

Consider a simple example of a text editor with operations like copying, pasting, and undoing.

* **Command Interface (Command)**:

class Command:

def execute(self):

pass

* **Concrete Command (CopyCommand)**:

class CopyCommand(Command):

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

self.editor = editor

def execute(self):

self.editor.copy()

* **Receiver (Editor)**:

class Editor:

def copy(self):

print("Text copied")

* **Invoker**:

class Button:

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

self.command = command

def click(self):

self.command.execute()

* **Client**:

editor = Editor()

copyCommand = CopyCommand(editor)

button = Button(copyCommand)

button.click() # Output: Text copied

In this example:

* The Command interface declares an execute() method.
* The CopyCommand is a concrete command that implements execute() by calling the copy() method on the Editor.
* The Editor is the receiver that knows how to perform the actual copy operation.
* The Button acts as the invoker, which stores the command and triggers its execution.
* The client sets up the command and associates it with the receiver and invoker.

**Use Cases**

* **Parameterizing objects with operations**: The Command Pattern allows you to store commands as objects, enabling different invocations.
* **Queueing operations**: You can queue commands for later execution.
* **Undoable operations**: By storing the history of commands, you can implement undo functionality.
* **Logging changes**: Commands can be used to log changes for auditing purposes.

The Command Pattern decouples the sender of a request from the receiver, providing flexibility in how requests are processed and managed.

State Pattern

The State pattern is a behavioral design pattern used to manage the state-specific behavior of an object. It allows an object to change its behavior when its internal state changes, making the object appear as if it has changed its class. This pattern is especially useful when an object's behavior depends on its state, and it must change its behavior at runtime depending on that state.

**Key Concepts**

1. **Context**: This is the class that contains the state and delegates behavior to the current state object. It maintains an instance of a State subclass that represents its current state.
2. **State Interface**: This defines the interface for encapsulating the behavior associated with a particular state. All concrete state classes implement this interface.
3. **Concrete States**: These are the classes that implement the State interface and define the behavior for each specific state.

**How It Works**

1. **State Interface**: Defines methods that concrete states should implement. These methods typically correspond to actions that can be performed in the context.
2. **Concrete State Classes**: Implement the State interface and provide the specific behavior for each state. They often include logic to transition to other states.
3. **Context Class**: Maintains a reference to a State instance and delegates state-specific behavior to the current state object. It also manages state transitions.

**Example**

Imagine a simple context with a traffic light. The traffic light can be in one of three states: Green, Yellow, or Red. Each state has its behavior for when the light changes.

#State Inteface

class TrafficLightState:

    def handle(self, context):

        raise NotImplementedError

#Concrete State Class

class GreenLightState(TrafficLightState):

    def handle(self, context):

        print("Green light! Go!")

        context.set\_state(YellowLightState())

class YellowLightState(TrafficLightState):

    def handle(self, context):

        print("Yellow light! Slow down!")

        context.set\_state(RedLightState())

class RedLightState(TrafficLightState):

    def handle(self, context):

        print("Red light! Stop!")

        context.set\_state(GreenLightState())

#Context Class

class TrafficLight:

    def \_\_init\_\_(self):

        self.\_state = GreenLightState()

    def set\_state(self, state):

        self.\_state = state

    def change(self):

        self.\_state.handle(self)

traffic\_light = TrafficLight()

for \_ in range(6):

    traffic\_light.change()

In this example, the TrafficLight object starts in the GreenLightState. When change() is called, it delegates the behavior to the current state object, which then changes the state to the next one.

The State pattern helps in making the state-specific behavior more organized and easier to manage, especially when there are multiple states and state transitions in a system.