OBJECT ORIENTED DESIGN: CONCEPTS, PRINCIPLES, AND PATTERNS

Cédric Bohnert

MISE A JOUR Septembre 2024

TABLE DES MATIERES

1. OOD CONCEPTS

ENCAPSULATION

Encapsulation is the practice of bundling data (attributes) and methods that operate on the data within a single unit or class. This mechanism hides the internal state of the object and only exposes a controlled interface for interaction, protecting the integrity of the data.

```
class BankAccount:
  def __init__(self, owner, balance=0):
    self.owner = owner
    self.__balance = balance  # Private attribute

  def deposit(self, amount):
    if amount > 0:
      self.__balance += amount
    else:
      print("Deposit amount must be positive")

  def withdraw(self, amount):
    if 0 < amount <= self.__balance:
      self.__balance -= amount
    else:
      print("Insufficient funds or invalid amount")

  def get_balance(self):
    return self.__balance
```

Private Attributes: Attributes prefixed with double underscores (__) are private and cannot be accessed directly outside the class.

Public Methods: Methods like **deposit**, **withdraw**, and **get_balance** provide controlled access to the private data.

ABSTRACTION

Abstraction involves hiding the complex implementation details and showing only the essential features of an object. This simplifies interaction and makes it easier to understand how to use the object without needing to know its internal workings.

```
from abc import ABC, abstractmethod

class Shape(ABC):
  @abstractmethod
  def area(self):
    pass

  @abstractmethod
  def perimeter(self):
    pass

class Circle(Shape):
  def __init__(self, radius):
    self.radius = radius

  def area(self):
    return 3.14 * self.radius ** 2

  def perimeter(self):
    return 2 * 3.14 * self.radius
```

Abstract Base Class (ABC): **Shape** is an abstract base class with abstract methods **area** and **perimeter**.

Concrete Implementation: **Circle** is a concrete class that implements the abstract methods.

INHERITANCE

Inheritance allows a new class to inherit attributes and methods from an existing class. It promotes code reuse and establishes a natural hierarchy between classes.

```
class Animal:
  def speak(self):
    return "Animal sound"

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

Base Class: **Animal** is the base class with a method **speak**.

Derived Class: **Dog** inherits from **Animal** and overrides the **speak** method.

POLYMORPHISM

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the same method name to invoke different behaviors based on the object's class.

```
def make_animal_speak(animal):
  print(animal.speak())

dog = Dog()
make_animal_speak(dog)  # Outputs: Woof!

animal = Animal()
make_animal_speak(animal)  # Outputs: Animal sound
```

Method Overriding: The **speak** method is overridden in the **Dog** class.

Dynamic Dispatch: The method called is determined at runtime based on the actual object type.

2. OOD SOLID PRINCIPLES

Object-Oriented Design (OOD) principles are guidelines that help create software that is modular, flexible, and maintainable. The most well-known principles are often summarized using the acronym SOLID, which represents five key principles.

SINGLE RESPONSABILITY PRINCIPLE (SRP)

A class should have only one reason to change, meaning it should have only one job or responsibility. This promotes maintainability and flexibility.

```
class Invoice:
  def __init__(self, number, amount):
    self.number = number
    self.amount = amount

  def generate_invoice(self):
    return f"Invoice {self.number}: ${self.amount}"

class EmailSender:
  def send_email(self, message):
    print(f"Sending email: {message}")

# Example usage
invoice = Invoice(123, 456)
email_sender = EmailSender()
email_sender.send_email(invoice.generate_invoice())
```

Invoice Class: Responsible only for managing invoice details.

EmailSender Class: Responsible only for sending emails.

OPEN/CLOSED PRINCIPLE (OCP)

Software entities should be open for extension but closed for modification. This means you should be able to extend the functionality of a class without changing its existing code.

```
class Shape(ABC):
  @abstractmethod
  def area(self):
    pass

class Rectangle(Shape):
  def __init__(self, width, height):
    self.width = width
    self.height = height

  def area(self):
    return self.width * self.height

class AreaCalculator:
  def calculate_area(self, shape):
    return shape.area()

# Example usage
rectangle = Rectangle(10, 5)
calculator = AreaCalculator()
print(calculator.calculate_area(rectangle))  # Outputs: 50
```

Shape Class: Defines an interface for shapes.

Rectangle Class: Implements the **Shape** interface and calculates area.

AreaCalculator Class: Can calculate area for any shape that conforms to the **Shape** interface.

LISKOV SUBSTITUTION PRINCIPLE (LSP)

Subtypes must be substitutable for their base types without altering the correctness of the program. This ensures that derived classes extend the base class without changing its behavior.

```
class Bird:
  def fly(self):
    return "Flying"

class Penguin(Bird):
  def fly(self):
    raise Exception("Penguins can't fly")

def make_bird_fly(bird):
  print(bird.fly())

# Example usage
bird = Bird()
make_bird_fly(bird)  # Outputs: Flying

penguin = Penguin()
make_bird_fly(penguin)  # Raises an exception
```

Bird Class: Provides a **fly** method.

Penguin Class: Breaks LSP by overriding **fly** with behavior that doesn't fit the base class contract.

INTERFACE SEGREGATION PRINCIPLE (ISP)

Clients should not be forced to depend on interfaces they do not use. It's better to have small, specific interfaces rather than a large, general-purpose one.

```python
class Workable(ABC):
  @abstractmethod
  def work(self):
    pass

class Eatable(ABC):
  @abstractmethod
  def eat(self):
    pass

class Employee(Workable, Eatable):
  def work(self):
    return "Working"

  def eat(self):
    return "Eating"

# Example usage
employee = Employee()
print(employee.work())  # Outputs: Working
print(employee.eat())  # Outputs: Eating
```

Workable and Eatable Interfaces: Separate interfaces for working and eating.

Employee Class: Implements both interfaces, adhering to ISP by not forcing unrelated methods.

DEPENDENCY INVERSION PRINCIPLE (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions. This reduces coupling between components and increases flexibility.


```python
class DataFetcher(ABC):
  @abstractmethod
  def fetch(self):
    pass

class APIDataFetcher(DataFetcher):
  def fetch(self):
    return "Data from API"

class DataProcessor:
  def __init__(self, fetcher: DataFetcher):
    self.fetcher = fetcher

  def process_data(self):
    data = self.fetcher.fetch()
    print(f"Processing {data}")

# Example usage
fetcher = APIDataFetcher()
processor = DataProcessor(fetcher)
processor.process_data()  # Outputs: Processing Data from API
```

DataFetcher Interface: Defines the abstraction for data fetching.

APIDataFetcher Class: Provides implementation of the data fetching.

DataProcessor Class: Depends on the abstraction (**DataFetcher**), allowing for flexible implementations.

OTHER PRINCIPLES

In addition to SOLID, there are other principles in object-oriented design such as:

DRY (Don't Repeat Yourself): Avoid code duplication by ensuring that every piece of knowledge has a single, unambiguous, authoritative representation within a system.

KISS (Keep It Simple, Stupid): Design should be as simple as possible, avoiding unnecessary complexity.

YAGNI (You Aren't Gonna Need It): Don't add functionality until it's necessary.

3. OOD DESIGN PATTERNS

Object-Oriented Design (OOD) patterns are commonly categorized into three main types as outlined by the “Gang of Four” (GoF) in their influential book “Design Patterns: Elements of Reusable Object-Oriented Software”. They identified 23 design patterns, and these patterns are divided into three categories.

CREATIONAL PATTERNS

Focus on the process of object creation, aiming to make a system independent of how its objects are created, composed, and represented.

ABSTRACT FACTORY

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is useful when a system should be independent of how its products are created.

```python
# Abstract factory and product interfaces
class AbstractFactory:
    def create_product_a(self):
        pass

    def create_product_b(self):
        pass

class AbstractProductA:
    def do_something(self):
        pass
```



```python
class AbstractProductB:
    def do_something_else(self):
        pass

# Concrete products
class ConcreteProductA1(AbstractProductA):
    def do_something(self):
        return "Product A1"

class ConcreteProductB1(AbstractProductB):
    def do_something_else(self):
        return "Product B1"

class ConcreteProductA2(AbstractProductA):
    def do_something(self):
        return "Product A2"

class ConcreteProductB2(AbstractProductB):
    def do_something_else(self):
        return "Product B2"

# Concrete factories
class ConcreteFactory1(AbstractFactory):
    def create_product_a(self):
        return ConcreteProductA1()

    def create_product_b(self):
        return ConcreteProductB1()

class ConcreteFactory2(AbstractFactory):
    def create_product_a(self):
        return ConcreteProductA2()

    def create_product_b(self):
        return ConcreteProductB2()

# Client code
def client(factory: AbstractFactory):
    product_a = factory.create_product_a()
```

```
    product_b = factory.create_product_b()
    print(product_a.do_something())
    print(product_b.do_something_else())

# Usage
factory1 = ConcreteFactory1()
client(factory1)

factory2 = ConcreteFactory2()
client(factory2)
```

BUILDER

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

```
# Product to be built
class Product:
    def __init__(self):
        self.parts = []

    def add_part(self, part):
        self.parts.append(part)

    def show(self):
        return ', '.join(self.parts)

# Builder interface
class Builder:
    def build_part(self):
        pass

# Concrete builders
class ConcreteBuilderA(Builder):
    def __init__(self):
        self.product = Product()

    def build_part(self):
        self.product.add_part("Part A1")

    def get_result(self):
        return self.product

class ConcreteBuilderB(Builder):
    def __init__(self):
        self.product = Product()

    def build_part(self):
        self.product.add_part("Part B1")

    def get_result(self):
        return self.product

# Director
class Director:
    def construct(self, builder: Builder):
        builder.build_part()

# Usage
director = Director()

builder_a = ConcreteBuilderA()
director.construct(builder_a)
product_a = builder_a.get_result()
print(product_a.show())

builder_b = ConcreteBuilderB()
director.construct(builder_b)
product_b = builder_b.get_result()
print(product_b.show())
```

FACTORY METHOD

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

```
# Product interface
class Product:
    def operation(self):
        pass

# Concrete products
class ConcreteProductA(Product):
    def operation(self):
        return "ConcreteProductA"

class ConcreteProductB(Product):
    def operation(self):
        return "ConcreteProductB"

# Creator class
class Creator:
    def factory_method(self):
        pass

    def some_operation(self):
        product = self.factory_method()
        return product.operation()

# Concrete creators
class ConcreteCreatorA(Creator):
    def factory_method(self):
        return ConcreteProductA()

class ConcreteCreatorB(Creator):
    def factory_method(self):
        return ConcreteProductB()

# Usage
creator_a = ConcreteCreatorA()
print(creator_a.some_operation())

creator_b = ConcreteCreatorB()
print(creator_b.some_operation())
```


PROTOTYPE

The Prototype pattern is used to create new objects by copying an existing object, known as the prototype. This is useful when the cost of creating a new instance is more expensive than copying an existing one.

```
import copy

# Prototype interface
class Prototype:
    def clone(self):
        pass

# Concrete prototype
class ConcretePrototype(Prototype):
    def __init__(self, value):
        self.value = value

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

# Usage
prototype = ConcretePrototype("Prototype Value")
clone = prototype.clone()
print(clone.value)
```

SINGLETON

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

```python
class SingletonMeta(type):
    _instance = None

    def __call__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__call__(*args, **kwargs)
        return cls._instance

class Singleton(metaclass=SingletonMeta):
    def some_business_logic(self):
        return "Singleton instance"

# Usage
singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2)  # True
print(singleton1.some_business_logic())
```

STRUCTURAL PATTERNS

Concerned with how classes and objects are composed to form larger structures, helping ensure that systems are easier to design, understand, and maintain.

ADAPTER

The Adapter pattern allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface expected by the clients.

```python
# Existing interface
class Adaptee:
    def specific_request(self):
        return "Adaptee's specific behavior"

# Target interface
class Target:
    def request(self):
        pass

# Adapter
class Adapter(Target):
    def __init__(self, adaptee: Adaptee):
        self.adaptee = adaptee

    def request(self):
        return self.adaptee.specific_request()

# Usage
adaptee = Adaptee()
adapter = Adapter(adaptee)
print(adapter.request())  # Output: Adaptee's specific behavior
```

BRIDGE

The Bridge pattern decouples an abstraction from its implementation so that the two can vary independently. It allows you to change the implementation independently of the abstraction and vice versa.

```python
# Implementation interface
class Implementation:
    def operation_implementation(self):
        pass

# Concrete implementations
class ConcreteImplementationA(Implementation):
    def operation_implementation(self):
        return "ConcreteImplementationA"

class ConcreteImplementationB(Implementation):
    def operation_implementation(self):
        return "ConcreteImplementationB"

# Abstraction
class Abstraction:
    def __init__(self, implementation: Implementation):
        self.implementation = implementation

    def operation(self):
        return self.implementation.operation_implementation()

# Usage
implementation_a = ConcreteImplementationA()
abstraction = Abstraction(implementation_a)
print(abstraction.operation())  # Output: ConcreteImplementationA

implementation_b = ConcreteImplementationB()
abstraction = Abstraction(implementation_b)
print(abstraction.operation())  # Output: ConcreteImplementationB
```

COMPOSITE

The Composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.


```python
# Component interface
class Component:
    def operation(self):
        pass

# Leaf
class Leaf(Component):
    def __init__(self, name):
        self.name = name

    def operation(self):
        return self.name

# Composite
class Composite(Component):
    def __init__(self):
        self.children = []

    def add(self, component: Component):
        self.children.append(component)

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

# Usage
leaf1 = Leaf("Leaf 1")
leaf2 = Leaf("Leaf 2")

composite = Composite()
composite.add(leaf1)
composite.add(leaf2)

print(composite.operation())  # Output: Leaf 1 + Leaf 2
```

DECORATOR

The Decorator pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

```python
# Component interface
class Component:
    def operation(self):
        pass

# Concrete component
class ConcreteComponent(Component):
    def operation(self):
        return "ConcreteComponent"

# Decorator
class Decorator(Component):
    def __init__(self, component: Component):
        self.component = component

    def operation(self):
        return self.component.operation()

# Concrete decorator
class ConcreteDecorator(Decorator):
    def operation(self):
        return f"Decorated ({self.component.operation()})"

# Usage
component = ConcreteComponent()
decorated_component = ConcreteDecorator(component)
print(decorated_component.operation())  # Output: Decorated (ConcreteComponent)
```

FAÇADE

The Facade pattern provides a simplified interface to a complex subsystem. It shields clients from complex details and interactions of a system by providing a simpler interface.


```python
# Subsystem classes
class SubsystemA:
    def operation_a(self):
        return "Subsystem A Operation"

class SubsystemB:
    def operation_b(self):
        return "Subsystem B Operation"

# Facade
class Facade:
    def __init__(self):
        self.subsystem_a = SubsystemA()
        self.subsystem_b = SubsystemB()

    def operation(self):
        result_a = self.subsystem_a.operation_a()
        result_b = self.subsystem_b.operation_b()
        return f"Facade simplifies: {result_a} + {result_b}"

# Usage
facade = Facade()
print(facade.operation())  # Output: Facade simplifies: Subsystem A Operation + Subsystem B Operation
```

FLYWEIGHT

The Flyweight pattern is used to minimize memory usage or computational expenses by sharing as much as possible with similar objects. It is mainly used to support a large number of fine-grained objects efficiently.

```python
# Flyweight
class Flyweight:
    def __init__(self, intrinsic_state):
```

```
    self.intrinsic_state = intrinsic_state

  def operation(self, extrinsic_state):
    return f"Intrinsic: {self.intrinsic_state}, Extrinsic: {extrinsic_state}"

# Flyweight Factory
class FlyweightFactory:
  def __init__(self):
    self.flyweights = {}

  def get_flyweight(self, intrinsic_state):
    if intrinsic_state not in self.flyweights:
      self.flyweights[intrinsic_state] = Flyweight(intrinsic_state)
    return self.flyweights[intrinsic_state]

# Usage
factory = FlyweightFactory()
flyweight1 = factory.get_flyweight("State A")
flyweight2 = factory.get_flyweight("State B")
flyweight3 = factory.get_flyweight("State A")

print(flyweight1.operation("Extrinsic 1"))  # Output: Intrinsic: State A, Extrinsic: Extrinsic 1
print(flyweight2.operation("Extrinsic 2"))  # Output: Intrinsic: State B, Extrinsic: Extrinsic 2
print(flyweight1 is flyweight3)  # Output: True (same instance)
```

PROXY

The Proxy pattern provides a surrogate or placeholder for another object to control access to it. This can be useful for lazy initialization, access control, logging, and more.

```
# Subject interface
class Subject:
  def request(self):
    pass

# Real subject
class RealSubject(Subject):
  def request(self):
    return "RealSubject: Handling request"

# Proxy
class Proxy(Subject):
  def __init__(self, real_subject: RealSubject):
    self.real_subject = real_subject

  def request(self):
    # You can add additional behavior here
    print("Proxy: Checking access before forwarding request")
    return self.real_subject.request()

# Usage
real_subject = RealSubject()
proxy = Proxy(real_subject)
print(proxy.request())  # Output: Proxy: Checking access before forwarding request \n RealSubject: Handling request
```

BEHAVIORAL PATTERNS

Deal with object interaction and responsibility, focusing on how objects communicate and cooperate.

CHAIN OF RESPONSIBILITY

The Chain of Responsibility pattern 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.

```
# Handler interface
class Handler:
  def set_next(self, handler):
    pass
```

```
    def handle(self, request):
        pass

# Base handler
class BaseHandler(Handler):
    def __init__(self):
        self._next_handler = None

    def set_next(self, handler):
        self._next_handler = handler
        return handler

    def handle(self, request):
        if self._next_handler:
            return self._next_handler.handle(request)
        return None

# Concrete handlers
class ConcreteHandlerA(BaseHandler):
    def handle(self, request):
        if request == "A":
            return "Handled by ConcreteHandlerA"
        return super().handle(request)

class ConcreteHandlerB(BaseHandler):
    def handle(self, request):
        if request == "B":
            return "Handled by ConcreteHandlerB"
        return super().handle(request)

# Usage
handler_a = ConcreteHandlerA()
handler_b = ConcreteHandlerB()
handler_a.set_next(handler_b)

print(handler_a.handle("A"))  # Output: Handled by ConcreteHandlerA
print(handler_a.handle("B"))  # Output: Handled by ConcreteHandlerB
print(handler_a.handle("C"))  # Output: None
```


The Command pattern turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

```
# Command interface
class Command:
    def execute(self):
        pass

# Concrete commands
class ConcreteCommandA(Command):
    def __init__(self, receiver):
        self.receiver = receiver

    def execute(self):
        self.receiver.action_a()

class ConcreteCommandB(Command):
    def __init__(self, receiver):
        self.receiver = receiver

    def execute(self):
        self.receiver.action_b()

# Receiver
class Receiver:
    def action_a(self):
        print("Receiver: Action A")

    def action_b(self):
        print("Receiver: Action B")

# Invoker
class Invoker:
```

```
  def __init__(self):
    self._commands = []

  def set_command(self, command: Command):
    self._commands.append(command)

  def execute_commands(self):
    for command in self._commands:
      command.execute()

# Usage
receiver = Receiver()
command_a = ConcreteCommandA(receiver)
command_b = ConcreteCommandB(receiver)

invoker = Invoker()
invoker.set_command(command_a)
invoker.set_command(command_b)
invoker.execute_commands()
```

INTERPRETER

The Interpreter pattern provides a way to evaluate language grammar or expressions. It defines a representation for a grammar and an interpreter to interpret the sentences in the language.

```
# Expression interface
class Expression:
  def interpret(self, context):
    pass

# Terminal expression
class TerminalExpression(Expression):
  def __init__(self, data):
    self.data = data

  def interpret(self, context):
    return self.data in context

# Or expression
class OrExpression(Expression):
  def __init__(self, expr1, expr2):
    self.expr1 = expr1
    self.expr2 = expr2

  def interpret(self, context):
    return self.expr1.interpret(context) or self.expr2.interpret(context)

# And expression
class AndExpression(Expression):
  def __init__(self, expr1, expr2):
    self.expr1 = expr1
    self.expr2 = expr2

  def interpret(self, context):
    return self.expr1.interpret(context) and self.expr2.interpret(context)

# Usage
expr1 = TerminalExpression("Hello")
expr2 = TerminalExpression("World")
or_expr = OrExpression(expr1, expr2)
and_expr = AndExpression(expr1, expr2)

context = "Hello everyone"
print(or_expr.interpret(context))  # Output: True
print(and_expr.interpret(context))  # Output: False
```

ITERATOR

The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

```python
# Iterator interface
class Iterator:
    def __next__(self):
        pass

# Concrete iterator
class ConcreteIterator(Iterator):
    def __init__(self, collection):
        self._collection = collection
        self._index = 0

    def __next__(self):
        if self._index < len(self._collection):
            item = self._collection[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration

# Aggregate
class Aggregate:
    def __init__(self):
        self._items = []

    def add_item(self, item):
        self._items.append(item)

    def __iter__(self):
        return ConcreteIterator(self._items)

# Usage
aggregate = Aggregate()
aggregate.add_item("Item 1")
aggregate.add_item("Item 2")
aggregate.add_item("Item 3")

for item in aggregate:
    print(item)  # Output: Item 1 \n Item 2 \n Item 3
```


MEDIATOR

The Mediator pattern defines an object that encapsulates how a set of objects interact. It promotes loose coupling by preventing objects from referring to each other explicitly and letting you vary their interaction independently.

```python
# Mediator interface
class Mediator:
    def notify(self, sender, event):
        pass

# Concrete mediator
class ConcreteMediator(Mediator):
    def __init__(self, component1, component2):
        self.component1 = component1
        self.component2 = component2
        self.component1.set_mediator(self)
        self.component2.set_mediator(self)

    def notify(self, sender, event):
        if event == "A":
            print("Mediator reacts on A and triggers the following operation:")
            self.component2.do_c()
        elif event == "B":
            print("Mediator reacts on B and triggers the following operation:")
            self.component1.do_a()

# Base component
class BaseComponent:
    def __init__(self, mediator=None):
        self._mediator = mediator

    def set_mediator(self, mediator):
        self._mediator = mediator

# Concrete components
```

```
class Component1(BaseComponent):
  def do_a(self):
    print("Component 1 does A.")
    self._mediator.notify(self, "A")

class Component2(BaseComponent):
  def do_c(self):
    print("Component 2 does C.")
    self._mediator.notify(self, "B")

# Usage
component1 = Component1()
component2 = Component2()
mediator = ConcreteMediator(component1, component2)

component1.do_a()
```

MEMENTO

The Memento pattern provides the ability to restore an object to its previous state. It's useful for saving and restoring the state of an object without exposing its internal structure.

```
# Memento
class Memento:
  def __init__(self, state):
    self._state = state

  def get_state(self):
    return self._state

# Originator
class Originator:
  def __init__(self):
    self._state = None

  def set_state(self, state):
    self._state = state

  def save(self):
    return Memento(self._state)

  def restore(self, memento: Memento):
    self._state = memento.get_state()

# Caretaker
class Caretaker:
  def __init__(self, originator: Originator):
    self._originator = originator
    self._history = []

  def backup(self):
    self._history.append(self._originator.save())

  def undo(self):
    if not self._history:
      return
    memento = self._history.pop()
    self._originator.restore(memento)

# Usage
originator = Originator()
caretaker = Caretaker(originator)

originator.set_state("State 1")
caretaker.backup()

originator.set_state("State 2")
caretaker.backup()

originator.set_state("State 3")
print(f"Current State: {originator._state}") # Output: Current State: State 3

caretaker.undo()
print(f"Restored State: {originator._state}") # Output: Restored State: State 2
```

```
caretaker.undo()
print(f"Restored State: {originator._state}")  # Output: Restored State: State 1
```

OBSERVER

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```
# Subject interface
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)

# Concrete subject
class ConcreteSubject(Subject):
  def __init__(self):
    super().__init__()
    self._state = None

  @property
  def state(self):
    return self._state

  @state.setter
  def state(self, state):
    self._state = state
```



```
    self.notify()

# Observer interface
class Observer:
  def update(self, subject):
    pass

# Concrete observer
class ConcreteObserver(Observer):
  def update(self, subject):
    print(f"Observer: Reacted to the state change to: {subject.state}")

# Usage
subject = ConcreteSubject()
observer1 = ConcreteObserver()
observer2 = ConcreteObserver()

subject.attach(observer1)
subject.attach(observer2)

subject.state = "State 1"
subject.state = "State 2"
```

STATE

The State pattern allows an object to alter its behavior when its internal state changes. It appears as if the object changed its class.

```
# Context
class Context:
  def __init__(self, state):
    self.transition_to(state)

  def transition_to(self, state):
    self._state = state
    self._state.context = self
```

```
  def request(self):
    self._state.handle()

# State interface
class State:
  def set_context(self, context):
    self._context = context

  def handle(self):
    pass

# Concrete states
class ConcreteStateA(State):
  def handle(self):
    print("ConcreteStateA handles the request.")
    self._context.transition_to(ConcreteStateB())

class ConcreteStateB(State):
  def handle(self):
    print("ConcreteStateB handles the request.")
    self._context.transition_to(ConcreteStateA())

# Usage
context = Context(ConcreteStateA())
context.request()  # Output: ConcreteStateA handles the request.
context.request()  # Output: ConcreteStateB handles the request.
```

STRATEGY

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

```
# Strategy interface
class Strategy:
  def execute(self, data):
```



```
    pass

# Concrete strategies
class ConcreteStrategyA(Strategy):
  def execute(self, data):
    return sorted(data)

class ConcreteStrategyB(Strategy):
  def execute(self, data):
    return sorted(data, reverse=True)

# Context
class Context:
  def __init__(self, strategy: Strategy):
    self._strategy = strategy

  def set_strategy(self, strategy: Strategy):
    self._strategy = strategy

  def execute_strategy(self, data):
    return self._strategy.execute(data)

# Usage
data = [5, 2, 1, 4, 3]
context = Context(ConcreteStrategyA())
print(context.execute_strategy(data))  # Output: [1, 2, 3, 4, 5]

context.set_strategy(ConcreteStrategyB())
print(context.execute_strategy(data))  # Output: [5, 4, 3, 2, 1]
```

TEMPLATE METHOD

The Template Method pattern defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

```python
# Abstract class
class AbstractClass:
    def template_method(self):
        self.base_operation1()
        self.required_operation1()
        self.base_operation2()
        self.hook()
        self.required_operation2()

    def base_operation1(self):
        print("AbstractClass: Base Operation 1")

    def base_operation2(self):
        print("AbstractClass: Base Operation 2")

    def hook(self):
        pass

    def required_operation1(self):
        pass

    def required_operation2(self):
        pass

# Concrete class
class ConcreteClassA(AbstractClass):
    def required_operation1(self):
        print("ConcreteClassA: Implemented Operation 1")

    def required_operation2(self):
        print("ConcreteClassA: Implemented Operation 2")

# Usage
concrete_class = ConcreteClassA()
concrete_class.template_method()
```

VISITOR

The Visitor pattern lets you define a new operation without changing the classes of the elements on which it operates. It separates the algorithm from the object structure.

```python
# Visitor interface
class Visitor:
    def visit_concrete_element_a(self, element):
        pass

    def visit_concrete_element_b(self, element):
        pass

# Concrete visitor
class ConcreteVisitor(Visitor):
    def visit_concrete_element_a(self, element):
        print(f"ConcreteVisitor: Visiting {element.operation_a()}")

    def visit_concrete_element_b(self, element):
        print(f"ConcreteVisitor: Visiting {element.operation_b()}")

# Element interface
class Element:
    def accept(self, visitor: Visitor):
        pass

# Concrete elements
class ConcreteElementA(Element):
    def accept(self, visitor: Visitor):
        visitor.visit_concrete_element_a(self)

    def operation_a(self):
        return "ElementA"

class ConcreteElementB(Element):
    def accept(self, visitor: Visitor):
        visitor.visit_concrete_element_b(self)

    def operation_b(self):
```

```
    return "ElementB"

# Usage
elements = [ConcreteElementA(), ConcreteElementB()]
visitor = ConcreteVisitor()

for element in elements:
  element.accept(visitor)
```

4. OOD DESIGN PATTERNS RELATIONSHIPS

Design patterns are interrelated in that they often address similar problems but from different angles, and they can sometimes be used together to solve complex design challenges. They provide a vocabulary for developers to communicate solutions more effectively and to create flexible and reusable object-oriented designs.

COMPLEMENTARY PATTERNS

ADAPTER AND FACADE

Both provide a way to work with existing interfaces. The Adapter makes two incompatible interfaces compatible, while Facade provides a simplified interface to a complex subsystem. You might use a Facade to create a high-level interface and Adapters to integrate different systems into this interface.

DECORATOR AND PROXY

Both add behavior to objects, but they do so differently. The Decorator pattern adds additional behavior dynamically without modifying the object, while the Proxy controls access to an object, often adding some level of security or optimization.

COMPOSITE AND DECORATOR

Both involve a tree structure. Composite organizes objects into tree structures to represent part-whole hierarchies. Decorator uses a similar recursive composition to add responsibilities dynamically.

CHAIN OF RESPONSIBILITY AND COMMAND

Both deal with the handling of requests. The Command pattern encapsulates requests as objects, while the Chain of Responsibility passes requests along a chain of handlers. You could use Command objects to represent requests in a Chain of Responsibility.

STRATEGY AND TEMPLATE METHOD

Both define the skeleton of an algorithm. The Template Method defines the structure in a superclass and allows subclasses to override specific steps. Strategy, on the other hand, defines a family of algorithms and makes them interchangeable. You might start with a Template Method and refactor it to a Strategy pattern to make the algorithm more flexible.

STRUCTURAL PATTERNS RELATIONS

ADAPTER, BRIDGE, AND PROXY

These patterns deal with object composition and decoupling. Adapter changes the interface of an existing object, Bridge decouples an abstraction from its implementation, and Proxy controls access to another object. A Bridge can be used to separate abstract interfaces and implementations, while Adapters can convert those interfaces as needed.

FLYWEIGHT AND COMPOSITE

Both deal with object management. Flyweight is about sharing to save memory, while Composite allows treating individual and composite objects uniformly. You can use a Flyweight to manage the parts of a Composite if there are many objects with shared states.

BEHAVIORAL PATTERNS RELATIONS

OBSERVER AND MEDIATOR

Both are about managing communication between objects. The Observer pattern defines a one-to-many relationship, while the Mediator centralizes communication between objects, promoting loose coupling. A Mediator can manage a complex network of Observers.

STATE AND STRATEGY

Both patterns involve changing behavior at runtime. The State pattern changes the behavior of an object when its internal state changes, while the Strategy pattern changes behavior by switching between algorithms. You might use a State pattern to manage which Strategy to use.

VISITOR AND COMPOSITE

Visitor is often used with Composite objects. It allows adding new operations to a Composite structure without changing the classes of the elements on which it operates.

MEMENTO AND COMMAND

Both can be used to implement undo functionality. Memento captures the state of an object to restore it later, while Command can be used to store operations that can be reversed. You might store Mementos in Command objects to provide a more comprehensive undo mechanism.

CREATIONAL PATTERNS RELATIONS

ABSTRACT FACTORY AND FACTORY METHOD

Abstract Factory can use Factory Methods to create objects. Abstract Factory defines an interface for creating families of related objects, while Factory Method defines an interface for creating a single object. An Abstract Factory might have multiple Factory Methods.

BUILDER AND FACTORY METHOD

Builder focuses on constructing complex objects step by step, while Factory Method deals with the creation of a single object. A Builder could use Factory Methods internally to create parts of an object.

PROTOTYPE AND SINGLETON

Prototype creates new objects by copying an existing object. If you want to avoid creating multiple instances of a prototype, you can combine it with the Singleton pattern.

PATTERN COMBINATIONS AND COLLABORATION

COMPOSITE AND DECORATOR WITH VISITOR

A Composite or Decorator structure can be traversed by a Visitor to perform operations on each element without modifying the structure.

FACADE AND MEDIATOR

A Facade can be used to provide a unified interface to a set of subsystems, while a Mediator can manage communication between those subsystems.

CHAIN OF RESPONSIBILITY WITH COMMAND

Commands can be processed by a Chain of Responsibility to determine which handler should execute the command.

PATTERNS AS BUILDING BLOCKS

USING PATTERNS TOGETHER

In many systems, you will find multiple patterns used in conjunction. For example, you might use a Singleton to manage access to a Facade, which in turn uses a Chain of Responsibility internally to handle requests.

FROM SPECIFIC TO GENERAL

Some patterns are more specialized (like Singleton), while others provide more general solutions (like Strategy or Decorator). They can be layered together to create complex architectures.

PATTERN EVOLUTION AND REFINEMENT

PATTERN REFACTORING

During the software development process, you might refactor code from one pattern to another. For example, you might start with a simple Factory Method and later refactor it into an Abstract Factory when the need arises to create families of related objects.