**### Поведенческие паттерны проектирования**

**### 1. Итератор (Iterator)**

Итератор позволяет последовательно проходить через элементы коллекции, не раскрывая внутреннее представление этой коллекции.

\*\*Пример:\*\*

```

class MyCollection:

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

self.items = []

def add(self, item):

self.items.append(item)

def \_\_iter\_\_(self):

return MyIterator(self.items)

class MyIterator:

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

self.items = items

self.index = 0

def \_\_next\_\_(self):

if self.index < len(self.items):

result = self.items[self.index]

self.index += 1

return result

else:

raise StopIteration

# Использование

collection = MyCollection()

collection.add(1)

collection.add(2)

collection.add(3)

for item in collection:

print(item) # Вывод: 1 2 3

```

**### 2. Команда (Command)**

Команда инкапсулирует запрос как объект, позволяя параметризовать клиент с различными запросами, ставить запросы в очередь и поддерживать операции отмены.

\*\*Пример:\*\*

```

class Command:

def execute(self):

pass

class Light:

def on(self):

print("Свет включен")

def off(self):

print("Свет выключен")

class LightOnCommand(Command):

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

self.light = light

def execute(self):

self.light.on()

class LightOffCommand(Command):

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

self.light = light

def execute(self):

self.light.off()

# Использование

light = Light()

light\_on = LightOnCommand(light)

light\_off = LightOffCommand(light)

light\_on.execute() # Вывод: Свет включен

light\_off.execute() # Вывод: Свет выключен

```

**### 3. Наблюдатель (Observer)**

Наблюдатель определяет зависимость "один ко многим" между объектами, так что когда один объект изменяет свое состояние, все зависимые от него объекты уведомляются и автоматически обновляются.

\*\*Пример:\*\*

```

class Observer:

def update(self, message):

pass

class Subject:

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

self.observers = []

def attach(self, observer):

self.observers.append(observer)

def notify(self, message):

for observer in self.observers:

observer.update(message)

class ConcreteObserver(Observer):

def update(self, message):

print(f"Получено сообщение: {message}")

# Использование

subject = Subject()

observer = ConcreteObserver()

subject.attach(observer)

subject.notify("Состояние изменилось!") # Вывод: Получено сообщение: Состояние изменилось!

```

**### 4. Посетитель (Visitor)**

Посетитель позволяет добавить новые операции к объектам без изменения их классов, отделяя алгоритмы от объектов.

\*\*Пример:\*\*

```

class Visitor:

def visit(self, element):

pass

class Element:

def accept(self, visitor):

visitor.visit(self)

class ConcreteElementA(Element):

def operation(self):

return "Element A"

class ConcreteElementB(Element):

def operation(self):

return "Element B"

class ConcreteVisitor(Visitor):

def visit(self, element):

print(f"Визит к {element.operation()}")

# Использование

element\_a = ConcreteElementA()

element\_b = ConcreteElementB()

visitor = ConcreteVisitor()

element\_a.accept(visitor) # Вывод: Визит к Element A

element\_b.accept(visitor) # Вывод: Визит к Element B

```

**### 5. Посредник (Mediator)**

Посредник определяет объект, инкапсулирующий взаимодействие между множеством объектов, тем самым уменьшая зависимость между ними.

\*\*Пример:\*\*

```

class Mediator:

def send(self, message, sender):

pass

class ConcreteMediator(Mediator):

def \_\_init\_\_(self, component1, component2):

self.component1 = component1

self.component2 = component2

self.component1.mediator = self

self.component2.mediator = self

def send(self, message, sender):

if sender == self.component1:

self.component2.receive(message)

else:

self.component1.receive(message)

class Component:

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

self.mediator = None

class Component1(Component):

def send(self, message):

self.mediator.send(message, self)

def receive(self, message):

print(f"Component1 получил сообщение: {message}")

class Component2(Component):

def send(self, message):

self.mediator.send(message, self)

def receive(self, message):

print(f"Component2 получил сообщение: {message}")

# Использование

c1 = Component1()

c2 = Component2()

mediator = ConcreteMediator(c1, c2)

c1.send("Привет от Component1") # Вывод: Component2 получил сообщение: Привет от Component1

c2.send("Привет от Component2") # Вывод: Component1 получил сообщение: Привет от Component2

```

**### 6. Снимок (Memento)**

Снимок позволяет сохранять и восстанавливать состояние объекта без нарушения инкапсуляции.

\*\*Пример:\*\*

```

class Memento:

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

self.state = state

class Originator:

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

self.state = None

def create\_memento(self):

return Memento(self.state)

def set\_memento(self, memento):

self.state = memento.state

def \_\_str\_\_(self):

return f"State: {self.state}"

# Использование

originator = Originator()

originator.state = "Состояние 1"

print(originator) # State: Состояние 1

memento = originator.create\_memento()

originator.state = "Состояние 2"

print(originator) # State: Состояние 2

originator.set\_memento(memento)

print(originator) # State: Состояние 1

```

**### 7. Состояние (State)**

Состояние позволяет объектам изменять свое поведение в зависимости от состояния, в котором они находятся.

\*\*Пример:\*\*

```

class State:

def do\_action(self):

pass

class Context:

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

self.state = None

def set\_state(self, state):

self.state = state

def request(self):

self.state.do\_action()

class ConcreteStateA(State):

def do\_action(self):

print("Выполнение действия в состоянии A")

class ConcreteStateB(State):

def do\_action(self):

print("Выполнение действия в состоянии B")

# Использование

context = Context()

state\_a = ConcreteStateA()

context.set\_state(state\_a)

context.request() # Вывод: Выполнение действия в состоянии A

state\_b = ConcreteStateB()

context.set\_state(state\_b)

context.request() # Вывод: Выполнение действия в состоянии B

```

**### 8. Цепочка обязанностей (Chain of Responsibility)**

Цепочка обязанностей позволяет передавать запросы по цепочке обработчиков, пока кто-то не обработает запрос.

\*\*Пример:\*\*

```

class Handler:

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)

class ConcreteHandlerA(Handler):

def handle(self, request):

if request == "A":

return f"Handler A обработал запрос: {request}"

return super().handle(request)

class ConcreteHandlerB(Handler):

def handle(self, request):

if request == "B":

return f"Handler B обработал запрос: {request}"

return super().handle(request)

# Использование

handler\_a = ConcreteHandlerA()

handler\_b = ConcreteHandlerB()

handler\_a.set\_next(handler\_b)

print(handler\_a.handle("A")) # Вывод: Handler A обработал запрос: A

print(handler\_a.handle("B")) # Вывод: Handler B обработал запрос: B

print(handler\_a.handle("C")) # Вывод: None

```

**### 9. Шаблонный метод (Template Method)**

Шаблонный метод определяет основу операции, позволяя подклассам изменять некоторые шаги.

\*\*Пример:\*\*

```

class AbstractClass:

def template\_method(self):

self.operation1()

self.operation2()

def operation1(self):

print("Операция 1")

def operation2(self):

pass # должно быть переопределено

class ConcreteClassA(AbstractClass):

def operation2(self):

print("Операция 2A")

class ConcreteClassB(AbstractClass):

def operation2(self):

print("Операция 2B")

# Использование

a = ConcreteClassA()

a.template\_method()

# Вывод:

# Операция 1

# Операция 2A

b = ConcreteClassB()

b.template\_method()

# Вывод:

# Операция 1

# Операция 2B

```

**### Порождающие паттерны проектирования**

**### 10. Абстрактная фабрика (Abstract Factory)**

Абстрактная фабрика предоставляет интерфейс для создания семейств связанных объектов, не определяя их конкретные классы.

\*\*Пример:\*\*

```

class AbstractFactory:

def create\_product\_a(self):

pass

def create\_product\_b(self):

pass

class ConcreteFactory1(AbstractFactory):

def create\_product\_a(self):

return ProductA1()

def create\_product\_b(self):

return ProductB1()

class ConcreteFactory2(AbstractFactory):

def create\_product\_a(self):

return ProductA2()

def create\_product\_b(self):

return ProductB2()

class ProductA1:

def \_\_str\_\_(self):

return "Product A1"

class ProductB1:

def \_\_str\_\_(self):

return "Product B1"

class ProductA2:

def \_\_str\_\_(self):

return "Product A2"

class ProductB2:

def \_\_str\_\_(self):

return "Product B2"

# Использование

factory1 = ConcreteFactory1()

product\_a1 = factory1.create\_product\_a()

product\_b1 = factory1.create\_product\_b()

print(product\_a1) # Вывод: Product A1

print(product\_b1) # Вывод: Product B1

factory2 = ConcreteFactory2()

product\_a2 = factory2.create\_product\_a()

product\_b2 = factory2.create\_product\_b()

print(product\_a2) # Вывод: Product A2

print(product\_b2) # Вывод: Product B2

```

**### 11. Одиночка (Singleton)**

Одиночка гарантирует, что класс имеет только один экземпляр и предоставляет к нему глобальную точку доступа.

\*\*Пример:\*\*

```

class Singleton:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.\_instance

# Использование

singleton1 = Singleton()

singleton2 = Singleton()

print(singleton1 is singleton2) # Вывод: True

```

**### 12. Прототип (Prototype)**

Прототип позволяет создавать новые объекты, копируя существующий объект, известный как экземпляр-прототип.

\*\*Пример:\*\*

```

import copy

class Prototype:

def clone(self):

return copy.deepcopy(self)

class ConcretePrototype(Prototype):

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

self.value = value

# Использование

prototype = ConcretePrototype("Some Value")

clone = prototype.clone()

print(clone.value) # Вывод: Some Value

```

**### 13. Строитель (Builder)**

Строитель отделяет построение сложного объекта от его представления, позволяя тем создавать одинаковые представления.

\*\*Пример:\*\*

```

class Product:

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

self.parts = []

def add(self, part):

self.parts.append(part)

class Builder:

def create\_product(self):

return Product()

class ConcreteBuilder(Builder):

def build\_part(self, part):

product = self.create\_product()

product.add(part)

return product

# Использование

builder = ConcreteBuilder()

product = builder.build\_part("Part1")

print(product.parts) # Вывод: ['Part1']

```

**### 14. Фабричный метод (Factory Method)**

Фабричный метод определяет интерфейс для создания объекта, позволяя подклассам изменять тип создаваемых объектов.

\*\*Пример:\*\*

```

class Product:

def operation(self):

pass

class ConcreteProductA(Product):

def operation(self):

return "Результат операции продукта A"

class ConcreteProductB(Product):

def operation(self):

return "Результат операции продукта B"

class Creator:

def factory\_method(self):

pass

class ConcreteCreatorA(Creator):

def factory\_method(self):

return ConcreteProductA()

class ConcreteCreatorB(Creator):

def factory\_method(self):

return ConcreteProductB()

# Использование

creator\_a = ConcreteCreatorA()

product\_a = creator\_a.factory\_method()

print(product\_a.operation()) # Вывод: Результат операции продукта A

creator\_b = ConcreteCreatorB()

product\_b = creator\_b.factory\_method()

print(product\_b.operation()) # Вывод: Результат операции продукта B

```

**### Структурные паттерны проектирования**

**### 15. Адаптер (Adapter)**

Адаптер позволяет объектам с несовместимыми интерфейсами работать вместе.

\*\*Пример:\*\*

```

class Target:

def request(self):

return "Целевой запрос"

class Adaptee:

def specific\_request(self):

return "Специфический запрос"

class Adapter(Target):

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

self.adaptee = adaptee

def request(self):

return self.adaptee.specific\_request()

# Использование

adaptee = Adaptee()

adapter = Adapter(adaptee)

print(adapter.request()) # Вывод: Специфический запрос

```

**### 16. Декоратор (Decorator)**

Декоратор позволяет добавлять новое поведение объектам, оборачивая их в другой объект.

\*\*Пример:\*\*

```

class Coffee:

def cost(self):

return 5

class MilkDecorator:

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

self.coffee = coffee

def cost(self):

return self.coffee.cost() + 1

# Использование

coffee = Coffee()

print(coffee.cost()) # Вывод: 5

decorated\_coffee = MilkDecorator(coffee)

print(decorated\_coffee.cost()) # Вывод: 6

```

**### 17. Заместитель (Proxy)**

Заместитель контролирует доступ к объекту, предоставляя замену или контролируя выполнение операций над объектом.

\*\*Пример:\*\*

```

class Subject:

def request(self):

pass

class RealSubject(Subject):

def request(self):

return "Реальный запрос"

class Proxy(Subject):

def \_\_init\_\_(self, real\_subject):

self.real\_subject = real\_subject

def request(self):

# Контроль доступа

print("Proxy: выполняет дополнительные действия перед запросом.")

return self.real\_subject.request()

# Использование

real\_subject = RealSubject()

proxy = Proxy(real\_subject)

print(proxy.request()) # Вывод: Proxy: выполняет дополнительные действия перед запросом. \n Реальный запрос

```

**### 18. Легковес (Flyweight)**

Легковес позволяет экономить память, используя объекты, разделяемые множеством сущностей.

\*\*Пример:\*\*

```

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}'

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]

# Использование

factory = FlyweightFactory()

flyweight1 = factory.get\_flyweight("State1")

print(flyweight1.operation("ExtrinsicState1")) # Вывод: Intrinsic: State1, Extrinsic: ExtrinsicState1

```

**### 19. Мост (Bridge)**

Мост разделяет абстракцию и реализацию, позволяя изменять их независимо друг от друга.

\*\*Пример:\*\*

```

class Abstraction:

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

self.implementor = implementor

def operation(self):

return self.implementor.operation()

class Implementor:

def operation(self):

pass

class ConcreteImplementorA(Implementor):

def operation(self):

return "Concrete Implementor A"

class ConcreteImplementorB(Implementor):

def operation(self):

return "Concrete Implementor B"

# Использование

implementor\_a = ConcreteImplementorA()

abstraction = Abstraction(implementor\_a)

print(abstraction.operation()) # Вывод: Concrete Implementor A

implementor\_b = ConcreteImplementorB()

abstraction = Abstraction(implementor\_b)

print(abstraction.operation()) # Вывод: Concrete Implementor B

```

**### 20. Фасад (Facade)**

Фасад предоставляет упрощенный интерфейс к более сложной системе, делая ее легче для использования.

\*\*Пример:\*\*

```

class SubsystemA:

def operation\_a(self):

return "Операция A"

class SubsystemB:

def operation\_b(self):

return "Операция B"

class Facade:

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

self.subsystem\_a = SubsystemA()

self.subsystem\_b = SubsystemB()

def operation(self):

return f'{self.subsystem\_a.operation\_a()} + {self.subsystem\_b.operation\_b()}'

# Использование

facade = Facade()

print(facade.operation()) # Вывод: Операция A + Операция B

```