# Singleton

Singleton - это паттерн проектирования, который гарантирует, что у класса будет только один экземпляр. Он часто используется для глобальных объектов, таких как объекты настроек или логгера.

Пример реализации в Python:

class Singleton(type):

\_instances = {}

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

if cls not in cls.\_instances:

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

return cls.\_instances[cls]

class Logger(metaclass=Singleton):

pass

logger1 = Logger()

logger2 = Logger()

print(logger1 is logger2) # True

Здесь мы используем метакласс Singleton для того, чтобы при создании нового экземпляра класса возвращался уже существующий экземпляр из словаря \_instances.

# Adapter

Adapter - это паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Он как бы оборачивает один интерфейс в другой.

Пример в Python:

class Dog:

def bark(self):

return "woof!"

class Cat:

def meow(self):

return "meow!"

class Human:

def speak(self):

return ""

class Car:

def make\_noise(self, octane\_level):

return f"vroom{octane\_level\*'}"

class Adapter:

def \_\_init\_\_(self, obj, adapted\_methods):

self.obj = obj

self.\_\_dict\_\_.update(adapted\_methods)

def \_\_getattr\_\_(self, attr):

return getattr(self.obj, attr)

objects = []

dog = Dog()

objects.append(Adapter(dog, dict(make\_noise=dog.bark)))

cat = Cat()

objects.append(Adapter(cat, dict(make\_noise=cat.meow)))

human = Human()

objects.append(Adapter(human, dict(make\_noise=human.speak)))

car = Car()

objects.append(Adapter(car, dict(make\_noise=lambda: car.make\_noise(3))))

for obj in objects:

print(obj.make\_noise())

Здесь мы адаптируем объекты Dog, Cat, Human и Car под общий интерфейс make\_noise с помощью класса Adapter.

# Decorator

Decorator - это паттерн, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные "обёртки".

Пример в Python:

from functools import wraps

def my\_decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

print('Function:', func.\_\_name\_\_)

print('Arguments:', args, kwargs)

result = func(\*args, \*\*kwargs)

print('Result:', result)

return result

return wrapper

@my\_decorator

def add(x, y):

return x + y

@my\_decorator

def greet(name):

return f"Hello {name}"

print(add(2, 3))

print(greet('John'))

Здесь мы определяем декоратор my\_decorator, который принимает функцию, добавляет к ней дополнительную функциональность в виде логирования и возвращает обернутую функцию. Используя синтаксис @my\_decorator перед функцией мы применяем этот декоратор.

В результате функции add и greet сохраняют свою основную логику, но теперь имеют расширенную функциональность благодаря декоратору.