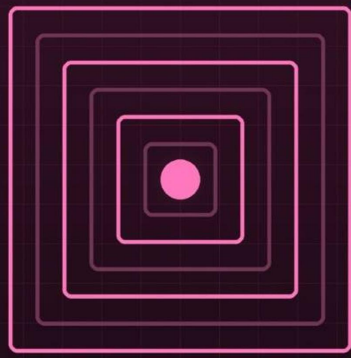
GoF · 09/23 СТРУКТУРНЫЙ

Python C++23 Rust Java



Декоратор

Decorator

Алексей Примак

Алексей Примак

Декоратор. Паттерн GoF на Python, C++23, Rust и Java

https://litres.ru/74810749
SelfPub; 2026

Аннотация

Как добавлять к объекту новые возможности без создания множества подклассов? «Декоратор» показывает, как динамически комбинировать поведение и собирать цепочки обёрток. На примере сжатия, шифрования и журналирования разобраны преимущества композиции перед наследованием. Паттерн GoF реализован на Python, C++23, Rust и Java, а также дополнен практическими вариантами и самостоятельными заданиями.

Содержание

Введение 4
Как читать 5
Глава 9. Реализация на Python 6
9.1. Зачем нужен декоратор 6
9.2. Участники и схема 7
9.3. Классический пример: сжатие и шифрование 9
Конец ознакомительного фрагмента. 14

Декоратор. Паттерн GoF на Python, C++23, Rust и Java

Введение

Эта книга посвящена одному паттерну проектирования — **Декоратор**(Decorator). Он относится к каталогу «Банды четырёх» (Gang of Four, GoF). Здесь паттерн разобран сразу в четырёх реализациях: на Python, C++23, Rust и Java. Каждая реализация — отдельная часть книги: сначала задача и общая схема, затем код, модификации, гибридные варианты и самостоятельные работы.

Как читать

Основы каждого языка здесь не повторяются: они собраны в книгах серий по соответствующему языку. Разделы каждой части нумеруются заново, поэтому ссылка «раздел 1.2» указывает на раздел той же части, где она сделана.

Глава 9. Реализация на Python

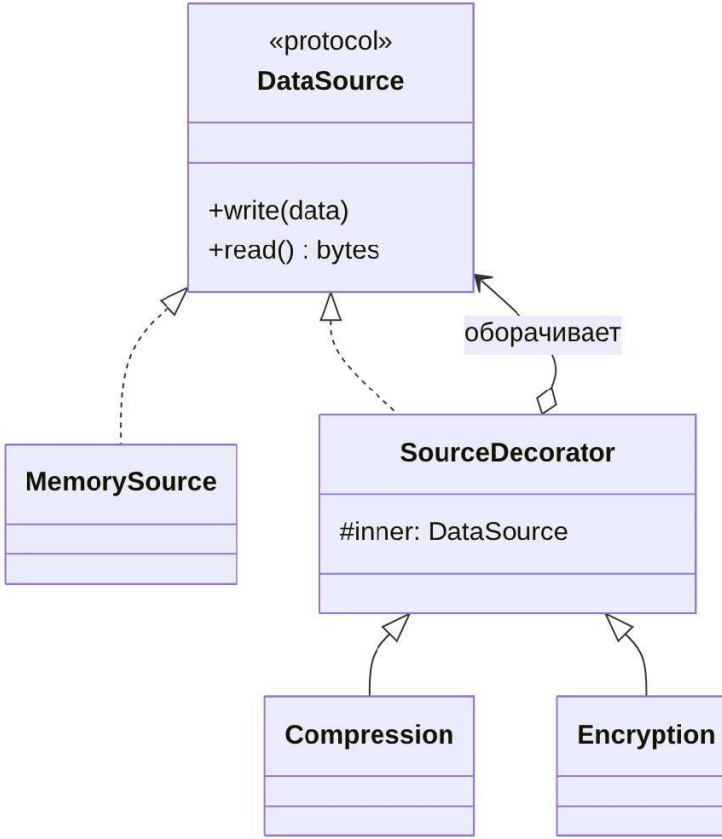
9.1. Зачем нужен декоратор

Есть класс, который читает данные из источника. Иногда данные нужно сжимать, иногда шифровать, иногда журналировать каждое обращение, а иногда — всё вместе в разном порядке. Наследование здесь не годится: CompressedEncryptedLoggedSource — тупик, а число сочетаний растёт экспоненциально.

Декоратор оборачивает объект другим объектом с тем же интерфейсом. Обёртка делает что-то до или после вызова и передаёт работу внутреннему объекту. Обёртки можно вкладывать друг в друга, как матрёшки, собирая нужную комбинацию во время работы.

В Python есть встроенный синтаксис @decorator для функций и классов. Это не совсем то же самое, что паттерн GoF: синтаксис оборачивает функцию при определении, а паттерн — объект во время работы. Но идея общая — добавить поведение, не меняя исходный код, — и в этой главе мы используем оба инструмента.

9.2. Участники и схема



Component— общий интерфейс.

Concrete Component— базовый объект.

Decorator— хранит ссылку на компонент и реализует тот же интерфейс.

Concrete Decorator— добавляет поведение.

9.3. Классический пример: сжатие и шифрование

Файл 01_data_source.py:

01_data_source.py Python

```python
import zlib
from typing import Protocol


class DataSource(Protocol):
    def write(self, data: bytes) -> None: ...
    def read(self) -> bytes: ...


class MemorySource:
    def __init__(self) -> None:
        self.stored = b""

    def write(self, data: bytes) -> None:
        self.stored = data

    def read(self) -> bytes:
        return self.stored


class SourceDecorator:
    def __init__(self, inner: DataSource) -> None:
        self.inner = inner

    def write(self, data: bytes) -> None:
        self.inner.write(data)

    def read(self) -> bytes:
        return self.inner.read()


class Compression(SourceDecorator):
    def write(self, data: bytes) -> None:
        super().write(zlib.compress(data))
```

Листинг 9.1 *— Классический пример: сжатие и шифро-*

вание

01_data_source.py (продолжение) Python

```python

    def read(self) -> bytes:
        return zlib.decompress(super().read())


class Encryption(SourceDecorator):
    def __init__(self, inner: DataSource, key: int) -> None:
        super().__init__(inner)
        self.key = key

    def _xor(self, data: bytes) -> bytes:  # учебный «шифр», не для реальной защиты
        return bytes(b ^ self.key for b in data)

    def write(self, data: bytes) -> None:
        super().write(self._xor(data))

    def read(self) -> bytes:
        return self._xor(super().read())


if __name__ == "__main__":
    text = ("Паттерн декоратор. " * 20).encode()
    raw = MemorySource()
    source = Encryption(Compression(raw), key=42)
    source.write(text)
    print("Исходный размер:", len(text))
    print("Хранится байт:  ", len(raw.stored))
    print("Начало в хранилище:", raw.stored[:8].hex())
    print("Прочитано верно:", source.read() == text)
    plain = MemorySource()
    plain.write(text)
    print("Без обёрток хранится:", len(plain.stored))
```

```
$ python 01_data_source.py
Исходный размер: 700
Хранится байт:   49
Начало в хранилище: 789cfbb5f5d7acdf
Прочитано верно: True
Без обёрток хранится: 700
```

Листинг 9.1 (продолжение) *— Классический пример: сжатие и шифрование*

Разбор. Цепочка Encryption(Compression(raw)) при записи сначала шифрует, затем сжимает, а при чтении разворачивает в обратном порядке. Клиент работает с source через обычный интерфейс write/read. Порядок обёрток важен: шифрованные данные похожи на случайные и почти не сжимаются, поэтому правильно сначала сжимать, а потом шифровать. В нашем примере XOR с одним байтом сохраняет повторы, и сжатие всё равно работает, но с настоящим шифром так не будет — это повод поменять обёртки местами.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.