GeekBrains

Архитектура и шаблоны проектирования на Python

Структурные паттерны

На этом уроке

Обзор структурных паттернов. Adapter, Bridge, Composite, Decorator, Facade, Proxy. Примеры реализации.

Оглавление

Структурные паттерны

Структурные паттерны определяют, как из классов и объектов образуются более крупные структуры.

Adapter // Адаптер

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами.

Паттерн применим на уровне класса для адаптации одного интерфейса к другому и на уровне объекта: одна из реализаций целевого интерфейса использует композицию для адаптации поведения объекта, вызывая методы объекта адаптируемого класса.

Адаптер уровня класса

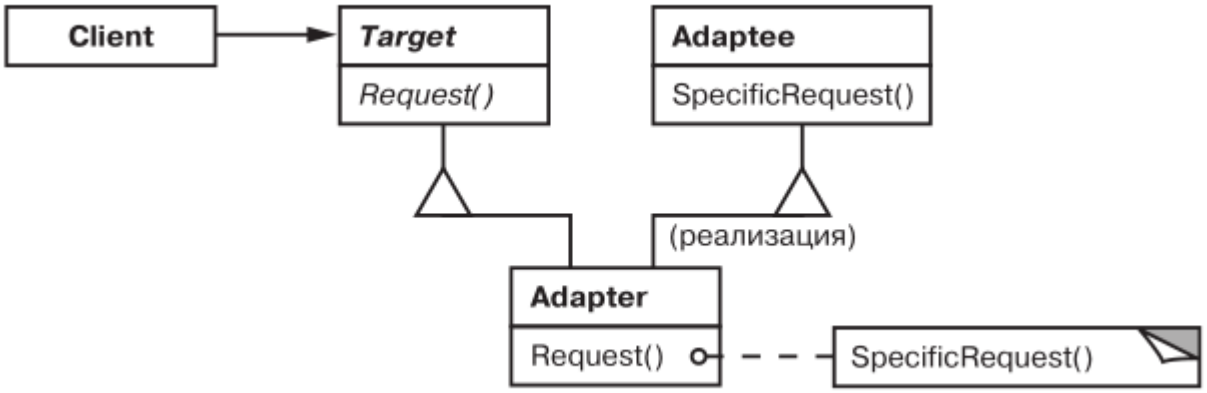


Диаграмма классов для случая адаптации интерфейсов. Здесь используется множественное наследование интерфейсов.

Target — целевой интерфейс, к этому типу следует привести интерфейс адаптируемого.

Adaptee — адаптируемый класс, чей интерфейс приводится в соответствие с целевым.

Adapter — подкласс адаптируемого класса, который имплементирует целевой интерфейс путём обёртки методов адаптируемого класса в методы целевого интерфейса. На диаграмме изображена реализация целевого метода Request() путём вызова метода SpecificRequest() адаптируемого класса.

Пример

Есть окружности, реализующие интерфейс Roundable — «имеющих радиус»:

```
// нечто круглое, имеющее радиус
class Roundable(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def get_radius(self):
        pass
```

```
// окружность имеет радиус
class Circle(Roundable):
    def __init__(self, radius):
        self._radius = radius

    def get_radius(self):
```

```
        return self._radius
```

Кроме того, есть квадрат, имеющий сторону, он совсем не круглый.

```
// квадрат со стороной side
class Square:
    def __init__(self, side):
        self._side = side

    def get_side(self):
        return self._side
```

Задача: обеспечить работу с квадратами как с круглыми предметами — например, при раскрое материала. Тут поможет паттерн *Адаптер* уровня класса.

```
// круглый квадрат
class RoundableSquare(Square, Roundable):
    def get_radius(self):
        return self.get_side() * math.sqrt(2) / 2
```

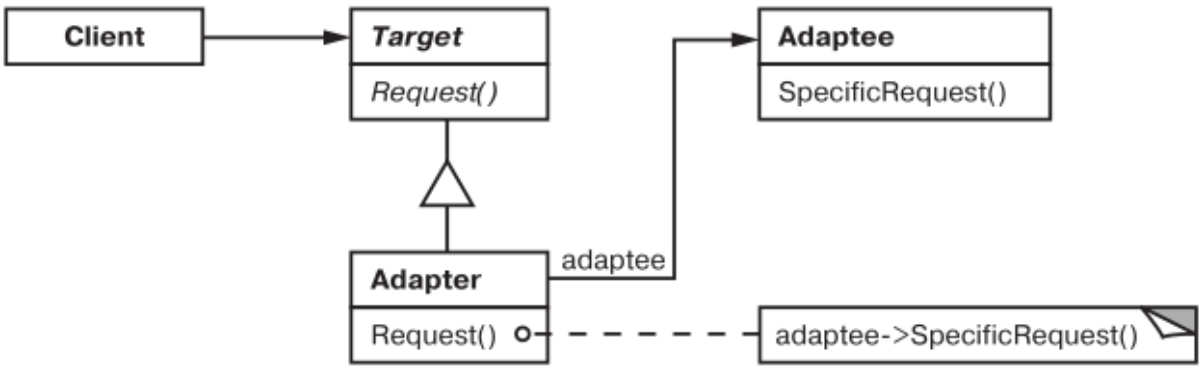
Теперь их можно использовать совместно с окружностями:

```
circle_1 = Circle(5)
roundable_square_1 = RoundableSquare(5)

print(circle_1.get_radius())
print(roundable_square_1.get_radius())

print(issubclass(circle_1.__class__, Roundable))
print(issubclass(roundable_square_1.__class__, Roundable))
```

Адаптер уровня объекта



Target — целевой интерфейс, к этому типу следует привести поведение адаптируемого объекта.

Adaptee — адаптируемый класс.

Adapter — реализация целевого интерфейса, содержит (композиция) ссылку на объект адаптируемого класса и вызывает методы адаптируемого класса. На диаграмме изображена

реализация целевого метода Request() путём вызова метода SpecificRequest() объекта адаптируемого класса.

Пример

Теперь набор окружностей и набор квадратов есть как данность. Задача: упорядочить их по радиусу, при том, что у квадратов его нет.

Пишем *Адаптер* объекта *Квадрат* к интерфейсу Roundable:

```
// адаптер квадрата к круглым фигурам
class RoundableSquareAdapter(Roundable):
    def __init__(self, adaptee):
        self._adaptee = adaptee

    // радиус квадрата - как радиус описанной окружности
    def get_radius(self):
        return self._adaptee.get_side() * math.sqrt(2) / 2
```

С его помощью сортируем совмещённый список объектов:

```
// создаем класс для сортировки объектов Roundable
class SortRoundable(RoundableSquareAdapter):
    def compare_order(self):
        if issubclass(self.__class__, Roundable):
            return self.get_radius()
        else:
            return RoundableSquareAdapter(self).get_radius()

// список окружностей и квадратов
figures_1 = [Circle(5), Square(5), Circle(2), Square(2)]

// отсортированный список окружностей и квадратов
ordered_figures = sorted(figures_1, key=SortRoundable.compare_order)
```

В результате *Адаптеры* используются только в момент сортировки, а итоговый список содержит упорядоченные разнообразные объекты.

```
// выводим значения радиуса для фигур
for item in ordered_figures:
    if issubclass(item.__class__, Roundable):
        print(item.get_radius())
    else:
        print(RoundableSquareAdapter(item).get_radius())
```

Вывод:

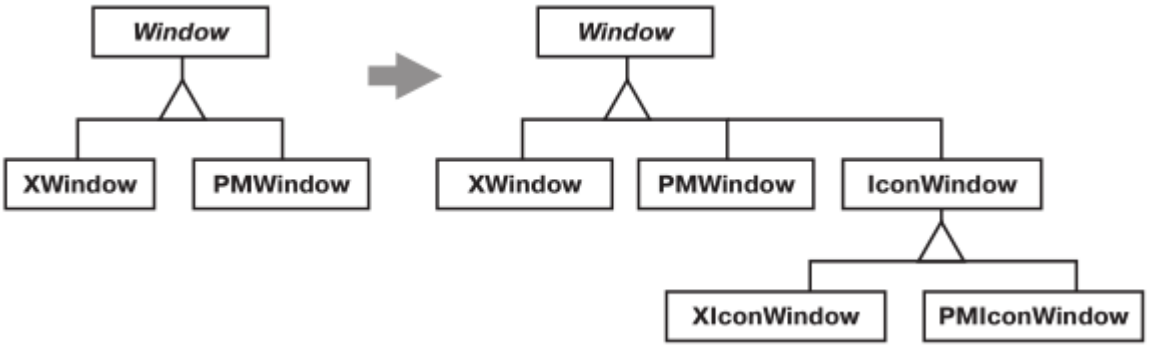
```
square: R=1.4142135623730951, side=2
circle: R=2
square: R=3.5355339059327378, side=5
circle: R=5
```

Bridge // Мост

Паттерн *Мост* призван отделить абстракцию от её реализации так, чтобы их можно было изменять независимо.

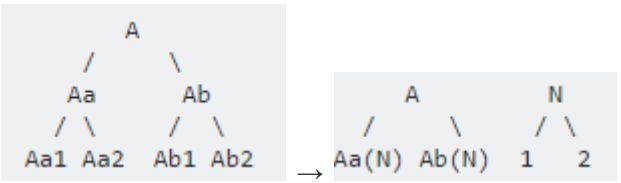
Проблема

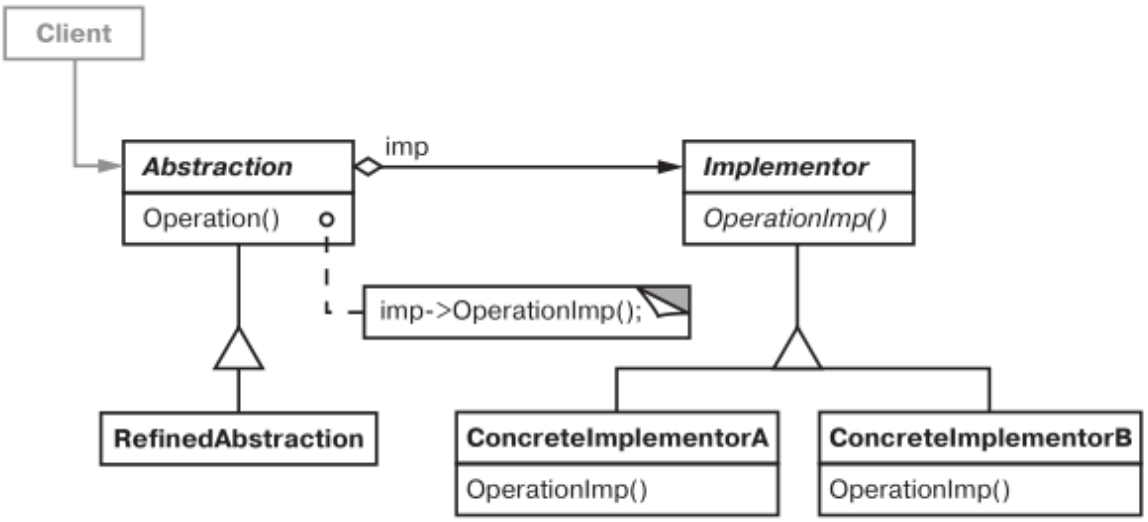
Есть иерархия абстракции окон, и нужно создать несколько разных её реализаций.



Пусть вначале иерархия абстракции состоит только из одного интерфейса Window. Мы можем легко создать две различные реализации. Затем иерархия абстракций расширяется более узким понятием IconWindow (Иконка). Для её реализации потребуется создать два новых класса, в которых либо не получится использовать существующий код XWindow и PMWindow, либо придётся его дублировать. Задача классов XIconWindow и PMIconWindow — только реализация особенностей иконок, но нам придётся еще и реализовывать в них поведение, описанное интерфейсом Window. Ответственность этих классов будет размыта, что нежелательно.

Решение предлагает структурный паттерн *Мост*. Он заменяет прямое наследование от абстракции дополнительным уровнем косвенности: абстракция, как и её наследники, хранит ссылку на базовый интерфейс иерархии различных вариантов реализаций. Его называют реализатором абстракции, а конкретные реализации — платформой.





Abstraction — интерфейс абстракции. Хранит ссылку на объект класса Implementor, для реализации функционала вызывает методы, описанные в интерфейсе ***Implementor***.

RefinedAbstraction — расширение иерархии абстракций.

Implementor — интерфейс классов реализации. Описывает методы, реализующие функционал, заданный интерфейсом *Абстракции*, но это самостоятельный корневой интерфейс, а не наследник *Абстракции*.

ConcreteImplementor — конкретные реализации интерфейса ***Implementor***.

Client — клиентский код. Использует функционал абстракции через его методы.

Пример

Датчики различных физических величин. Нас может интересовать моментальное значение, среднее за несколько измерений, за единицу времени и т. п. — это иерархия абстракций. Датчики могут измерять различные физические величины, быть разных изготовителей, разных протоколов и т. п. — это иерархия реализаций. Эти иерархии не пересекаются. Мы можем описать базовое поведение на стороне различных реализаций, связать абстракцию и реализацию, указав ссылку на интерфейс реализации, и свободны в развитии расширения абстракции и особенностей реализации базового поведения датчиков.

Сначала опишем абстракцию — абстрактный датчик, использующий реализацию физического датчика по интерфейсу, и сам интерфейс:

```
class AbstractSensor:
    def __init__(self, implementor):
        self._implementor = implementor

    def get_value(self):
        return self._implementor.get_value_impl()


class SensorImplementor(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def get_value_impl(self):
        pass
```

Описываем базовый интерфейс физического датчика. Допустим, он умеет отдавать информацию о текущем значении из некоторого ряда:

```
class BaseSensorImplementor(SensorImplementor):
   values_list = [12.3, 12.25, 12.38, 12.18, 12.41, 12.43, 12.37, 12.67]

   def __init__(self):
       self.values = iter(__class__.values_list)

   def get_value_impl(self):
       return next(self.values)
```

Реализуем функционал со стороны абстракции. Сначала реализуем базовый датчик мгновенного значения, который просто использует возможности физических датчиков:

```
class BaseSensor(AbstractSensor):
   pass


base_sensor_1 = BaseSensor(BaseSensorImplementor())
print(base_sensor_1.get_value())
```

Расширим возможности со стороны абстракции. Создадим датчик, возвращающий среднее значение по *n* последним измерениям:

```
class AverageSensor(AbstractSensor):
   def __init__(self, implementor, n):
       super().__init__(implementor)
       // очередь последних измерений
       self.queue= []
       self.n = n

   // среднее по последним n измерениям
   def get_average_value(self):
       self.queue.append(self._implementor.get_value_impl());
       if len(self.queue) > self.n:
           self.queue.pop(0)

       return sum(self.queue) / len(self.queue)


average_sensor_1 = AverageSensor(BaseSensorImplementor(), 5)
print(average_sensor_1.get_value())
[print(average_sensor_1.get_average_value()) for _ in range(7)]
```

Осталось реализовать физические датчики, например, скорости и напряжения в бортовой сети:

```
class Speedometer(SensorImplementor):
    def __init__(self, min_val, max_val):
        self.min_val = min_val
        self.max_val = max_val

    def get_value_impl(self):
        return self.min_val + random.random() * (self.max_val - self.min_val)


speedometer = Speedometer(0, 30)
average_speed = AverageSensor(speedometer, 10)

print(average_speed.get_value())
[print(average_speed.get_average_value()) for _ in range(20)]
```

```
class Voltmeter(SensorImplementor):
    def __init__(self):
        self.base_voltage = 12
        self.deviation = 0.5

    def get_value_impl(self):
        return self.base_voltage + random.random() * self.deviation * 2 -
self.deviation


voltmeter = Voltmeter()
average_voltmeter = AverageSensor(voltmeter, 5)

print(average_voltmeter.get_value())
[print(average_voltmeter.get_average_value()) for _ in range(20)]
```

Теперь можно использовать любую комбинацию абстракции и физической реализации, клиентский код:

```
class Client:
    @staticmethod
    def measure(sensor):
        print(sensor.__class__.__name__)

        // датчик мгновенного значения
        baseSensor = BaseSensor(sensor)

        // датчик усредненного значения по 5 последним измерениям
        averageSensor = AverageSensor(sensor, 5)

        // серия измерений
        for _ in range(10):
```

```
            print(f'мгновенное значение : {baseSensor.get_value():5.2f}, среднее
значение: {averageSensor.get_average_value():5.2f}')



if __name__ == '__main__':
    // создаём физические датчики
    speedometer = Speedometer(0, 25)
    voltmeter = Voltmeter()

    Client.measure(speedometer)
    Client.measure(voltmeter)
```

Важный момент — необходимо выделять общий базовый функционал на стороне реализаций.

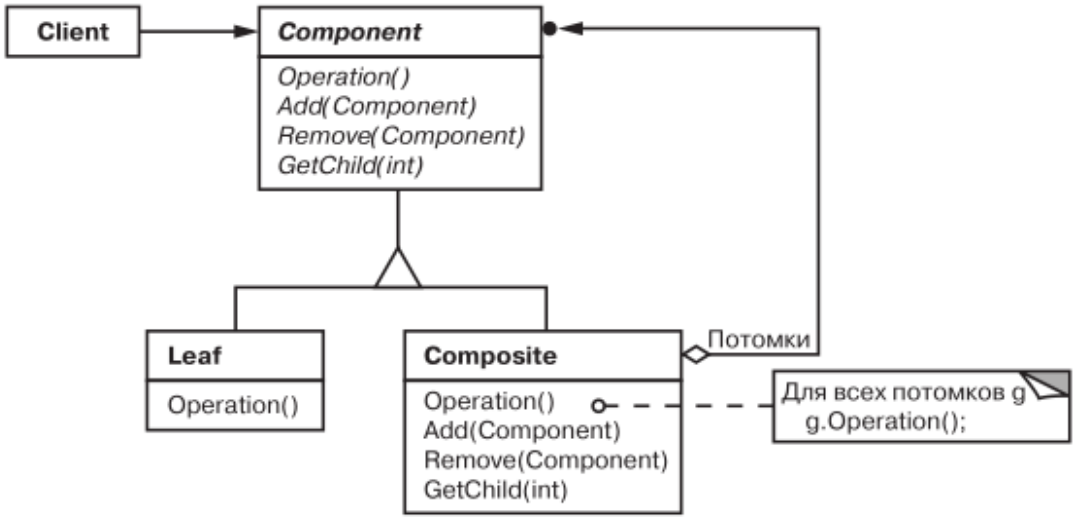
Паттерн *Мост* иногда применяют совместно с паттерном *Абстрактная фабрика*, где *Фабрика* порождает объекты платформозависимой реализации, а *Мост* структурно разделяет иерархию абстракций и иерархию реализаций.

Composite // Компоновщик

Компоновщик — паттерн, который структурирует объекты: компонует их в древовидные иерархические структуры для представления иерархий «часть — целое». Он позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Древовидные иерархические структуры часто используются в приложениях. Например, при копировании файла или папки мы даём команду (или перетаскиваем мышкой), не различая, файл это или папка. С точки зрения файловых операций они ведут себя одинаково, за исключением того, что в файл невозможно положить что-либо, а содержимое папок — это всегда набор других файлов и папок. Таким образом, мы имеем дело с минимальной единицей хранения (файлом) и контейнером (папкой), вмещающим в себя другие контейнеры или минимальные единицы. При этом необходим унифицированный интерфейс взаимодействия с этими двумя разными объектами в пределах образованной ими иерархии.

Смысл паттерна — предоставить клиентскому коду общий интерфейс для контейнера и одиночного элемента.



- ***Component*** — компонент, общий интерфейс, предоставляемый клиентскому коду.
- **Leaf** — лист дерева, узел иерархии, не имеющий потомков.
- **Composite** — составной объект, определяющий операции внутри иерархии и хранящий внутри себя компоненты-потомки.
- **Client** — клиентский код, использующий интерфейс *Component* для манипуляции объектами дерева.

Главное, чтобы реализация действия (операции) была и в классе листа, и в классе компоновщика.

Пример

Создаём абстракцию компонента, который должен выполнять определённую операцию:

```
class Component(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def operation(self):
        pass
```

Реализация компонента в виде машинной операции:

```
class MachineOperation(Component):
    def __init__(self, name):
        self.name = name

    def operation(self):
        print(self.name)
```

Композитная операция:

```
class CompositeOperation(Component):
    def __init__(self):
        self._child = set()

    def operation(self):
        for child in self._child:
            child.operation()

    def append(self, component):
        self._child.add(component)

    def remove(self, component):
        self._child.discard(component)
```

Пример использования в клиентском коде:

```
// инициализация операций
operation_1 = MachineOperation('drill 5 mm')
operation_2 = MachineOperation('drill 15 mm')
composite_1 = CompositeOperation()
composite_1.append(operation_1)
composite_1.append(operation_2)
```

```
operation_3 = MachineOperation('assemble')
operation_4 = MachineOperation('paint')
composite_2 = CompositeOperation()
composite_2.append(composite_1)
composite_2.append(operation_3)
composite_2.append(operation_4)

// использование разных по структуре операций идентично
composite_2.operation()
operation_1.operation()
```

Вывод:

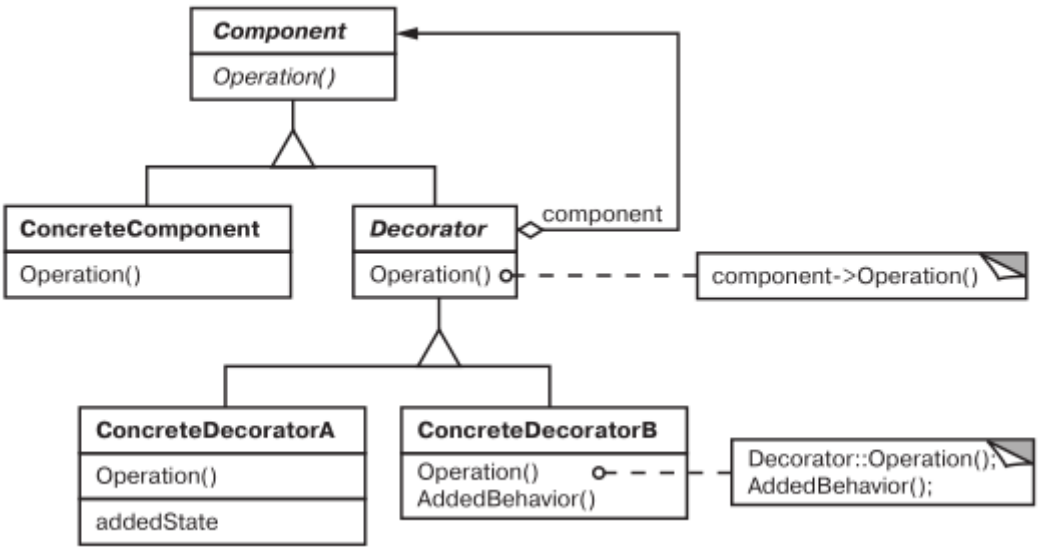
```
drill 5 mm
drill 15 mm
assemble
paint
drill 5 mm
```

Decorator // Декоратор

Динамически добавляет функционал к интерфейсу некоторой конкретной реализации. Выступает в роли динамического наследования.

Паттерн широко используется в Python: decorator, function definitions. Посмотрим на него изнутри.

Назначение: динамически добавляет объекту новые обязанности. Гибкая альтернатива порождению подклассов с целью расширения функциональности.



ConcreteComponent — конкретный класс, чья функциональность требует расширения.

Component — выделенный интерфейс конкретного класса для расширения.

Decorator — обычно абстрактный класс, задача которого — хранение ссылки на декорируемый объект. Это общая задача для всех реализаций иерархии.

ConcreteDecorator — конкретные реализации различных декораторов, расширяющих функционал конкретного класса в рамках выделенного интерфейса. Обычно инициализируются ссылкой на декорируемый объект, чем занимается абстрактный класс ***Decorator.***

Базовый объект остается неизменным (но вынужден имплементировать выделенный интерфейс), а дополнительные обязанности нанизываются, как бусины, на интерфейс базового объекта. Причём состав функционала и порядок применения варьируются динамически.

Пример

Реализуем класс абстрактного писателя Writer и его реализацию:

```
class Writer(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def write_message(self):
        pass


class ConcreteWriter(Writer):
    def write_message(self):
        print('writing message')
```

Пусть понадобились дополнительные действия перед записью: проверка длины сообщения и его сжатие. Реализуем их в виде декораторов на базе некоторого абстрактного WriterDecorator:

```
class WriterDecorator(Writer, metaclass=abc.ABCMeta):
    def __init__(self, component):
        self._component = component

    @abc.abstractmethod
    def write_message(self):
        pass


class CheckLengthDecorator(WriterDecorator):
    def write_message(self):
        print('checking message length')
        self._component.write_message()


class CompressDecorator(WriterDecorator):
    def write_message(self):
        print('compressing message')
        self._component.write_message()
        print('check compressed length')
```

Проверяем:

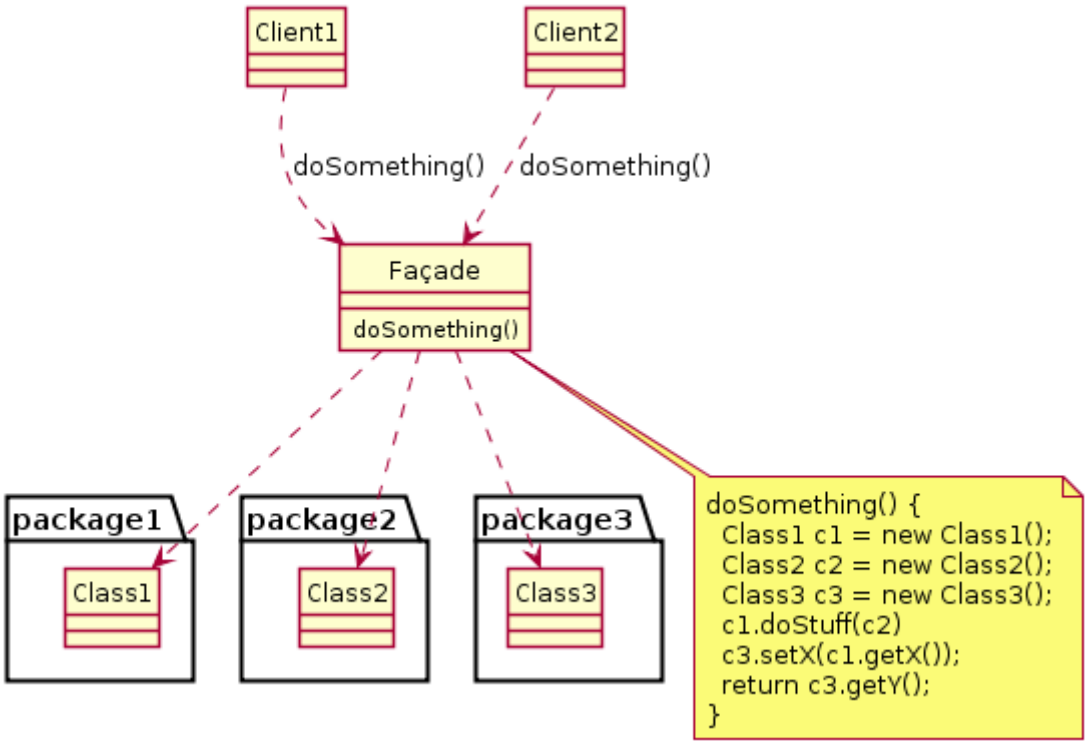
```
concrete_writer = ConcreteWriter()
check_length_decorator = CheckLengthDecorator(concrete_writer)
compress_decorator = CompressDecorator(check_length_decorator)
compress_decorator.write_message()
```

Вывод:

```
compressing message
checking message length
writing message
check compressed length
```

Facade // Фасад

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. *Фасад* определяет интерфейс более высокого уровня, упрощающий использование подсистемы.



Facade — *Фасад*, обладая информацией о строении подсистемы, делегирует вызовы клиентского кода подходящим объектам внутри неё.

Class1, Class2, Class3 — классы подсистемы: реализуют её функциональность, не знают о существовании *Фасада*.

Пример

Мы пишем приложение, отображающее цены на автомобили с разных сайтов. Мы создаём для каждого сайта свой класс, в котором есть методы для получения цен на русские и импортные авто:

```
class Site1Checker:
    def russian_auto(self):
        print('prices of russian cars on site 1')

    def foreign_auto(self):
        print('prices of foreign cars on site 1')


class Site2Checker:
    def russian_auto(self):
        print('prices of russian cars on site 2')

    def foreign_auto(self):
        print('prices of foreign cars on site 2')
```

Напишем *Фасад* вывода цен на автомобили:

```
class FacadeSiteChecker:
    def __init__(self):
        self._subsys_1 = Site1Checker()
        self._subsys_2 = Site2Checker()

    def russian_auto(self):
        self._subsys_1.russian_auto()
        self._subsys_2.russian_auto()

    def foreign_auto(self):
        self._subsys_1.foreign_auto()
        self._subsys_2.foreign_auto()
```

Здесь *Фасад* выполняет простые задачи:

1. Создаёт объекты доступа к сайтам.
2. Выполняет методы этих объектов — заменяет рутинные операции.

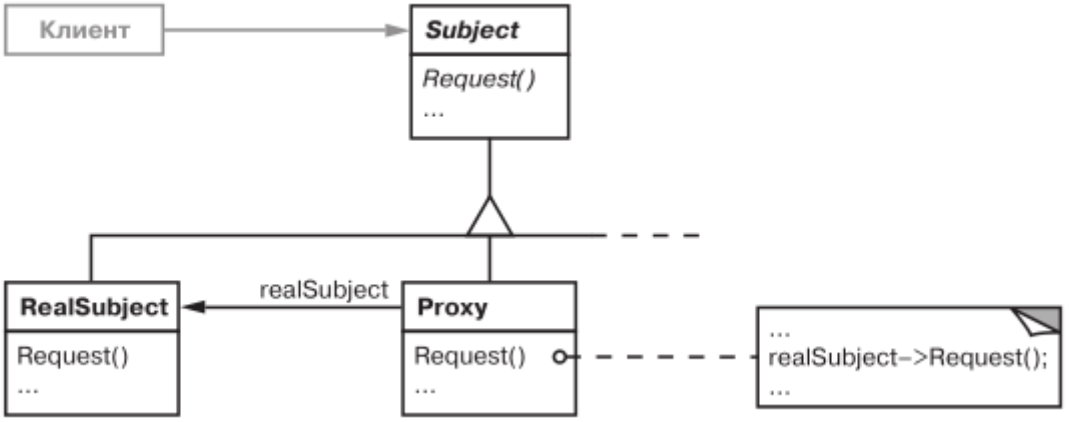
Код клиента упрощается:

```
facade_site_checker = FacadeSiteChecker()
facade_site_checker.russian_auto()
facade_site_checker.foreign_auto()
```

Используя *Фасад*, мы лишь запрашиваем его о получении объекта конкретного типа данных и не занимаемся разбором введённых строк в коде бизнес-логики.

Proxy // Заместитель

Паттерн *Заместитель* применяется, когда нужно сослаться на объект более изощрённо, чем позволяет простой указатель.



Subject — субъект, определяет общий для RealSubject и Proxy интерфейс. Поэтому класс **Proxy** можно использовать везде, где ожидается RealSubject.

Proxy — *Заместитель*: хранит ссылку, позволяющую ему обратиться к реальному субъекту.

RealSubject — реальный субъект, определяющий реальный объект, представленный *Заместителем*.

С помощью паттерна *Заместитель* при доступе к объекту вводится дополнительный уровень косвенности.

Применимость:

- Ленивая инициализация (виртуальный прокси). Есть тяжёлый объект, грузящий данные из файловой системы или базы данных. Вместо того, чтобы грузить их в начале программы, можно сэкономить ресурсы и создать объект, когда он действительно понадобится.
- Защита доступа (защищающий прокси), когда в программе есть разные типы пользователей и нужно защитить объект от неавторизованного доступа. Например, если объекты — важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).
- Кеширование объектов («умная» ссылка). *Заместитель* может подсчитывать количество ссылок на сервисный объект, которые были отданы клиенту и остаются активными. Когда все ссылки освободятся, можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных). Кроме того, *Заместитель* может отслеживать, не менял ли клиент сервисный объект. Так можно использовать объекты повторно и экономить ресурсы, особенно если речь идет о больших ресурсоёмких сервисах.
- Преобразование протоколов (удалённые прокси). Оборачиваемый сервис находится на удалённом сервере, использует другой протокол или написан на другом языке. В этом случае *Заместитель* транслирует запросы клиента в вызовы по сети (в протоколе, понятном удалённому сервису), а клиент избавляется от головной боли.
- Логирование запросов (логирующий прокси). *Заместитель* может сохранять историю обращения клиента к сервисному объекту.

Пример

В приложении требуется сервис, предоставляющий текущий курс валюты по ЦБ. Информацией о курсе валюты владеет ЦБ, но она меняется достаточно редко (обычно раз в день), а в приложении может быть запрошена весьма часто (сотни раз в день). Решение — кешировать информацию о курсе, полученную от ЦБ.

Интерфейс сервиса:

```
class CurrencyRateService(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def get_currency_rate(self, currency):
        pass
```

Реальный (медленный) сервис, запрашивающий курс валюты у истинного владельца информации:

```
class CbrCurrencyRateService(CurrencyRateService):
    def get_currency_rate(self, currency):
        // ... особенности реализации опущены
        return 0.57
```

Кеширующий прокси:

```
class ProxyCurrencyRateService(CurrencyRateService):
    def __init__(self):
        // ссылка на реальный сервис
        self.currencyRateService = CbrCurrencyRateService()

        // кеш курсов
        self.rates = dict()

    def get_currency_rate(self, currency):
        if (currency in self.rates.keys()):
            // если курс уже имеется в кэше, выдать из кеша
            print(f'{currency}: from cache')
            return self.rates[currency]
        else:
            // если ещё нет, то запросить реальный (медленный) сервис
            print(f'{currency}: from service')
            rate = self.currencyRateService.get_currency_rate(currency)
            self.rates.update({currency: rate})
            return rate
```

Пример использования:

```
// создаём сервис
currency_rate_service = ProxyCurrencyRateService()

// получаем курс из кеша или от ЦБ (это решает прокси)
yen_rate_request_1 = currency_rate_service.get_currency_rate('yen')
print(yen_rate_request_1)

yen_rate_request_2 = currency_rate_service.get_currency_rate('yen')
print(yen_rate_request_2)
```

Вывод:

```
yen: from service
0.57
yen: from cache
0.57
```

Заключение

Адаптер предоставляет классу альтернативный интерфейс, *Заместитель* — тот же интерфейс, *Декоратор* — расширенный интерфейс.

Фасад похож на *Заместитель* тем, что замещает сложную подсистему и может сам её инициализировать. В отличие от *Фасада*, *Заместитель* имеет тот же интерфейс, что его служебный объект, благодаря чему их можно взаимозаменять.

Декоратор и *Заместитель* имеют похожие структуры, но разные назначения. Они похожи тем, что оба построены на композиции и делегировании работы другому объекту. Паттерны отличаются тем, что *Заместитель* сам управляет жизнью сервисного объекта, а обертывание *Декораторов* контролирует клиент.

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Выбирать подходящий структурный шаблон.
2. Применять структурные шаблоны в своём коде.

Смысл:

Для использования структурных шаблонов в своём коде.

Последовательность действий:

Можно сделать всё или одно на выбор, применив при этом один из структурных паттернов, либо аргументировать, почему те или иные паттерны не были использованы:

1. Создать декоратор для получения связки url-view в приложение, чтобы можно было добавлять url, как во фреймворке Flask @app('/some_url/').
2. Добавить декоратор @debug, для view; если мы указываем его над view, то в терминал выводятся название функции и время её выполнения.
3. Добавить подкатегории. Т. е., категория курса может входить в другую категорию, а может не входить, и вложенность будет любая. Например, «Программирование -> Веб -> Python -> Django». После на страницу списка категорий добавить вывод количества курсов в каждой из категорий. Например, Программирование — 10, Web — 5, Python — 3, …
4. Добавить два новых вида WSGI-application. Первый — логирующий (такой же, как основной, только для каждого запроса выводит информацию (тип запроса и параметры) в консоль. Второй — фейковый (на все запросы пользователя отвечает: 200 OK, Hello from Fake).
5. По желанию можно добавить любой другой полезный функционал.

Дополнительные материалы

1. Structural Patterns.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2015.