Структурные шаблоны проектирования (Компоновщик, Фасад, Мост, Декоратор, Прокси, Адаптер)

Введение (15 мин)

Общие сведения о шаблонах проектирования

Шаблоны проектирования (Design Patterns) представляют собой решения для типичных задач, которые возникают при разработке программного обеспечения. Они основаны на опыте и практике лучших разработчиков и помогают создавать гибкие, эффективные и легко поддерживаемые системы.

Преимущества использования шаблонов проектирования

Использование шаблонов проектирования имеет ряд преимуществ:

1. Ускорение разработки: шаблоны предоставляют готовые решения для типичных задач, что позволяет экономить время и усилия.
2. Улучшение качества кода: шаблоны основаны на лучших практиках и помогают создавать гибкие, эффективные и легко поддерживаемые системы.
3. Облегчение обучения и сопровождения: шаблоны представляют собой общий язык для описания решений, что облегчает обучение новых разработчиков и сопровождение существующих систем.
4. Улучшение взаимодействия в команде: использование шаблонов способствует лучшему пониманию кода между членами команды и уменьшает количество конфликтов.

Категории шаблонов проектирования: поведенческие, создание объектов, структурные

Шаблоны проектирования можно разделить на три основные категории:

1. Поведенческие шаблоны (Behavioral Patterns) описывают способы взаимодействия между объектами и алгоритмы решения типичных задач.
2. Шаблоны создания объектов (Creational Patterns) предоставляют механизмы для создания и инициализации объектов, а также для управления их жизненным циклом.
3. Структурные шаблоны (Structural Patterns) определяют способы композиции и взаимодействия классов и объектов для формирования более крупных структур.

Определение структурных шаблонов проектирования

Структурные шаблоны определяют способы композиции и взаимодействия классов и объектов для формирования более крупных структур. Они позволяют создавать сложные иерархические структуры из простых объектов, а также обеспечивают единообразный доступ к ним.

Примерами из реальной жизни могут служить:

1. Структура файлов и папок в операционной системе. Она представляет собой иерархическую структуру, в которой папки могут содержать другие папки и файлы. Структура файлов и папок может быть представлена с помощью шаблона Композит (Composite).
2. Древовидная структура организации. Она также представляет собой иерархическую структуру, в которой сотрудники могут иметь подчиненных сотрудников. Древовидная структура организации может быть представлена с помощью шаблона Композит (Composite).
3. Адаптеры для различных типов розеток. Они позволяют использовать одно и то же устройство в разных странах, несмотря на различия в типах розеток. Адаптеры для различных типов розеток могут быть представлены с помощью шаблона Адаптер (Adapter).
4. Декораторы для добавления дополнительных опций к пицце или бургеру в фастфуде. Они позволяют создавать различные варианты пиццы или бургера, добавляя дополнительные ингредиенты или соусы. Декораторы для добавления дополнительных опций к пицце или бургеру могут быть представлены с помощью шаблона Декоратор (Decorator).
5. Прокси-серверы для обеспечения безопасности и контроля доступа к ресурсам в сети. Они выступают в роли посредников между клиентом и сервером, и могут выполнять дополнительную функциональность, такую как фильтрацию трафика, шифрование данных и кэширование. Прокси-серверы могут быть представлены с помощью шаблона Прокси (Proxy).

Адаптер (Adapter) (30 мин)

Определение шаблона Адаптер

Шаблон Адаптер (Adapter) предоставляет способ объединения интерфейсов разных классов, позволяя им взаимодействовать друг с другом. Адаптер выступает в роли посредника между двумя классами, преобразуя интерфейс одного класса в интерфейс другого.

_Приспособление для подключения устройств с разными типами разъемов. Например, адаптер для подключения USB-устройства к разъему Lightning на iPhone. Адаптер выступает в роли посредника между устройствами и преобразует сигнал с одного типа разъема на другой.

- Преобразование формата данных из одного формата в другой для их последующего использования в другом приложении.
- Адаптер для подключения сторонних устройств к игровой консоли.

Принципиальная схема PlantUML

```
@startuml

interface Target {
```

```
  + request()
}

class Adapter {
  + Adapter(Adaptee)
  + request()
  - adaptee: Adaptee
}

class Adaptee {
  + specificRequest()
}

Target <|-- Adapter
Adapter -- Adaptee

@enduml
```

Когда использовать шаблон Адаптер

Шаблон Адаптер следует использовать в следующих случаях:

1. Необходимо использовать существующий класс, но его интерфейс не соответствует требуемому.
2. Необходимо создать объект, который будет совмещать в себе функциональность нескольких классов.

- Примеры из реальной жизни: различные типы розеток, переходники для аудио- и видеоустройств
- Адаптер для подключения электроприборов с разными типами розеток.
- Адаптер для подключения сторонних устройств к игровой консоли.

Пример кода с использованием шаблона Адаптер:

```
from abc import ABC, abstractmethod

# Интерфейс, который должен быть реализован
class ITarget(ABC):
    @abstractmethod
    def request(self):
        pass

# Существующий класс, интерфейс которого необходимо адаптировать
class Adaptee:
    def specific_request(self):
        print("Adaptee: Specific Request")

# Класс-адаптер, реализующий интерфейс ITarget и адаптирующий интерфейс Adaptee
```

```
class Adapter(ITarget):
    def __init__(self, adaptee: Adaptee):
        self._adaptee = adaptee

    def request(self):
        self._adaptee.specific_request()

# Пример использования
if __name__ == "__main__":
    adaptee = Adaptee()
    target = Adapter(adaptee)
    target.request()
```

Подробное описание кода

В этом примере определены классы `ITarget`, `Adaptee` и `Adapter`. Класс `ITarget` представляет собой целевой интерфейс, который должен быть реализован. Класс `Adaptee` содержит конкретную реализацию, которую необходимо адаптировать. Класс `Adapter` выступает в роли посредника между `ITarget` и `Adaptee`.

Класс `Adapter` наследует интерфейс `ITarget` и реализует его метод `request()`. В конструкторе `Adapter` принимает экземпляр класса `Adaptee` и сохраняет его в качестве приватного поля `_adaptee`. В методе `request()` класса `Adapter` вызывается метод `specific_request()` экземпляра класса `Adaptee`.

В конце примера создаются экземпляры классов `Adaptee` и `Adapter`, и вызывается метод `request()` экземпляра класса `Adapter`.

Дополнительный пример кода с менее абстрактным кодом

Пример кода с использованием шаблона Адаптер для работы с разными типами файлов:

```
from abc import ABC, abstractmethod

# Интерфейс, который должен быть реализован для работы с файлами
class IFile(ABC):
    @abstractmethod
    def read(self):
        pass

    @abstractmethod
    def write(self, data):
        pass

# Класс для работы с текстовыми файлами
class TextFile:
    def __init__(self, filename):
        self._filename = filename
```

```
    def read(self):
        with open(self._filename, "r") as file:
            return file.read()

    def write(self, data):
        with open(self._filename, "w") as file:
            file.write(data)

# Класс для работы с бинарными файлами
class BinaryFile:
    def __init__(self, filename):
        self._filename = filename

    def read(self):
        with open(self._filename, "rb") as file:
            return file.read()

    def write(self, data):
        with open(self._filename, "wb") as file:
            file.write(data)

# Класс-адаптер для работы с текстовыми файлами через интерфейс IFile
class TextFileAdapter(IFile):
    def __init__(self, filename):
        self._file = TextFile(filename)

    def read(self):
        return self._file.read()

    def write(self, data):
        self._file.write(data)

# Класс-адаптер для работы с бинарными файлами через интерфейс IFile
class BinaryFileAdapter(IFile):
    def __init__(self, filename):
        self._file = BinaryFile(filename)

    def read(self):
        return self._file.read()

    def write(self, data):
        self._file.write(data)

# Пример использования
if __name__ == "__main__":
    text_file = TextFileAdapter("example.txt")
    binary_file = BinaryFileAdapter("example.bin")

    data = "Hello, World!"
```

```
    text_file.write(data)
    print(text_file.read())

    binary_file.write(data.encode("utf-8"))
    print(binary_file.read().decode("utf-8"))
```

В этом примере определены классы `IFile`, `TextFile`, `BinaryFile`, `TextFileAdapter` и `BinaryFileAdapter`. Класс `IFile` представляет собой интерфейс для работы с файлами. Классы `TextFile` и `BinaryFile` содержат конкретные реализации для работы с текстовыми и бинарными файлами соответственно. Классы `TextFileAdapter` и `BinaryFileAdapter` выступают в роли посредников между `IFile` и `TextFile`/`BinaryFile` соответственно.

Классы `TextFileAdapter` и `BinaryFileAdapter` наследуют интерфейс `IFile` и реализуют его методы `read()` и `write()`. В конструкторах `TextFileAdapter` и `BinaryFileAdapter` принимаются имена файлов и создаются экземпляры классов `TextFile` и `BinaryFile` соответственно. В методах `read()` и `write()` классов `TextFileAdapter` и `BinaryFileAdapter` вызываются соответствующие методы экземпляров классов `TextFile` и `BinaryFile`.

В конце примера создаются экземпляры классов `TextFileAdapter` и `BinaryFileAdapter`, и вызываются методы `read()` и `write()` экземпляров классов `TextFileAdapter` и `BinaryFileAdapter`.

Декоратор, Обёртка (Decorator, Wrapper) (30 мин)

Определение шаблона Декоратор

Шаблон Декоратор (Decorator) предоставляет способ динамического изменения поведения объекта, обертывая его в дополнительные объекты-декораторы. Декораторы позволяют добавлять новые возможности к существующим объектам, не изменяя их код.

Принципиальная схема PlantUML

```
@startuml

interface Component {
  + operation()
}

class ConcreteComponent {
  + operation()
}

abstract class Decorator {
  + operation()
  - component: Component
}
```

```
class ConcreteDecoratorA {
  + operation()
}

class ConcreteDecoratorB {
  + operation()
}

Component <|-- ConcreteComponent
Component <|-- Decorator
Decorator -- Component
ConcreteDecoratorA -- Decorator
ConcreteDecoratorB -- Decorator

@enduml
```

Когда использовать шаблон Декоратор

Шаблон Декоратор следует использовать в следующих случаях:

1. Необходимо динамически изменять поведение объекта, не изменяя его код.
2. Необходимо добавлять новые возможности к существующим объектам, не нарушая принцип единой ответственности.

Примеры из реальной жизни

1. Дополнительные опции в фастфуде. Они позволяют создавать различные варианты пиццы или бургера, добавляя дополнительные ингредиенты или соусы. Дополнительные опции в фастфуде могут быть представлены с помощью шаблона Декоратор (Decorator).

Например, в фастфуде есть базовый бургер, который можно дополнить различными ингредиентами, такими как сыр, бекон, салат, помидоры и прочее. Каждый из этих ингредиентов может быть представлен с помощью конкретного класса-декоратора, который добавляет свой функционал к базовому бургеру.

Конкретные классы-декораторы могут быть объединены в различные комбинации, чтобы создавать различные варианты бургера. Например, можно создать бургер с сыром и беконом, или бургер с салатом и помидорами.

2. Украшения для рождественской елки. Они позволяют создавать различные варианты рождественской елки, добавляя различные украшения, такие как игрушки, гирлянды, свечи и прочее. Украшения для рождественской елки могут быть представлены с помощью шаблона Декоратор (Decorator).

Например, можно представить рождественскую елку с помощью конкретного класса, который реализует интерфейс `IComponent`. Этот класс может содержать базовый функционал рождественской елки, такой как размер, форма, цвет и прочее.

Затем можно создать конкретные классы-декораторы, которые добавляют различные украшения к рождественской елке. Например, класс-декоратор `ConcreteDecoratorA` может добавлять игрушки к рождественской елке, класс-декоратор `ConcreteDecoratorB` может добавлять гирлянды, а класс-декоратор `ConcreteDecoratorC` может добавлять свечи.

Конкретные классы-декораторы могут быть объединены в различные комбинации, чтобы создавать различные варианты рождественской елки. Например, можно создать рождественскую елку с игрушками и гирляндами, или рождественскую елку с гирляндами и свечами.

Пример кода с использованием шаблона Декоратор:

```
from abc import ABC, abstractmethod

# Интерфейс, который должен быть реализован
class IComponent(ABC):
    @abstractmethod
    def operation(self):
        pass

# Конкретная реализация интерфейса IComponent
class ConcreteComponent(IComponent):
    def operation(self):
        print("ConcreteComponent: Operation")

# Базовый класс-декоратор, реализующий интерфейс IComponent и содержащий ссылку на
объект-компонент
class Decorator(IComponent):
    def __init__(self, component: IComponent):
        self._component = component

    def operation(self):
        self._component.operation()

# Конкретные классы-декораторы, реализующие дополнительную функциональность
class ConcreteDecoratorA(Decorator):
    def operation(self):
        print("ConcreteDecoratorA: Before")
        super().operation()
        print("ConcreteDecoratorA: After")

class ConcreteDecoratorB(Decorator):
    def operation(self):
        print("ConcreteDecoratorB: Before")
        super().operation()
        print("ConcreteDecoratorB: After")

# Пример использования
if __name__ == "__main__":
    component = ConcreteComponent()
```

```
    decorator_a = ConcreteDecoratorA(component)
    decorator_b = ConcreteDecoratorB(decorator_a)
    decorator_b.operation()
```

Подробное описание кода

В этом примере определены классы `IComponent`, `ConcreteComponent`, `Decorator`, `ConcreteDecoratorA` и `ConcreteDecoratorB`. Класс `IComponent` представляет собой интерфейс, который должен быть реализован. Класс `ConcreteComponent` содержит конкретную реализацию интерфейса `IComponent`. Класс `Decorator` выступает в роли базового класса-декоратора, реализующего интерфейс `IComponent` и содержащего ссылку на объект-компонент. Классы `ConcreteDecoratorA` и `ConcreteDecoratorB` представляют собой конкретные классы-декораторы, реализующие дополнительную функциональность.

Класс `Decorator` наследует интерфейс `IComponent` и реализует его метод `operation()`. В конструкторе `Decorator` принимает экземпляр класса, реализующего интерфейс `IComponent`, и сохраняет его в качестве приватного поля `_component`. В методе `operation()` класса `Decorator` вызывается метод `operation()` экземпляра класса, сохраненного в поле `_component`.

Классы `ConcreteDecoratorA` и `ConcreteDecoratorB` наследуют класс `Decorator` и переопределяют его метод `operation()`. В методе `operation()` классов `ConcreteDecoratorA` и `ConcreteDecoratorB` выполняется дополнительная функциональность перед и после вызова метода `operation()` экземпляра класса, сохраненного в поле `_component`.

В конце примера создаются экземпляры классов `ConcreteComponent`, `ConcreteDecoratorA` и `ConcreteDecoratorB`, и вызывается метод `operation()` экземпляра класса `ConcreteDecoratorB`.

Дополнительный пример кода с менее абстрактным кодом

Пример кода с использованием шаблона Декоратор для логирования и кэширования вызовов функций:

```
import functools

# Декоратор для логирования вызовов функции
def log_calls(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

# Декоратор для кэширования вызовов функции
def cache_calls(func):
    @functools.wraps(func)
```

```
    def wrapper(*args, **kwargs):
        cache_key = args + tuple(sorted(kwargs.items()))
        if cache_key in wrapper.cache:
            print(f"{func.__name__} cache hit")
            return wrapper.cache[cache_key]
        else:
            print(f"{func.__name__} cache miss")
            result = func(*args, **kwargs)
            wrapper.cache[cache_key] = result
            return result
    wrapper.cache = {}
    return wrapper

# Функция, которую необходимо декорировать
@log_calls
@cache_calls
def fibonacci(n):
    if n ≤ 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Пример использования
if __name__ == "__main__":
    print(fibonacci(10))
    print(fibonacci(10))
```

В этом примере определены функции `log_calls()` и `cache_calls()`, представляющие собой декораторы для логирования и кэширования вызовов функции соответственно. Функция `fibonacci()` представляет собой функцию, которую необходимо декорировать.

Функция `log_calls()` принимает в качестве аргумента функцию `func` и возвращает вложенную функцию `wrapper()`, которая вызывает функцию `func` с переданными аргументами `*args` и `**kwargs`, и выполняет логирование перед и после вызова функции `func`.

Функция `cache_calls()` принимает в качестве аргумента функцию `func` и возвращает вложенную функцию `wrapper()`, которая вызывает функцию `func` с переданными аргументами `*args` и `**kwargs`, и выполняет кэширование результата вызова функции `func` в словаре `wrapper.cache` с ключом `cache_key`, составленным из переданных аргументов `*args` и `**kwargs`.

Функция `fibonacci()` декорируется функциями `log_calls()` и `cache_calls()`, и вызывается дважды с аргументом `10`.

Композит, Компоновщик, Дерево (Composite) (30 мин)

Определение шаблона Композит

Шаблон Композит (Composite) предоставляет способ объединения объектов в древовидную структуру, позволяя обращаться к ним как к единому целому. Композит позволяет создавать сложные иерархические структуры из простых объектов, а также обеспечивает единообразный доступ к ним.

Структура организации, которая состоит из нескольких подразделений. Например, компания, которая состоит из отделов продаж, маркетинга, разработки и администрирования. Каждое подразделение может содержать свои собственные подразделения, создавая иерархическую структуру. Все подразделения и их сотрудники являются частью единой организации и могут взаимодействовать друг с другом для достижения общих целей.

- Структура файловой системы, которая состоит из папок и файлов.
- Структура веб-сайта, которая состоит из страниц, разделов и меню.

Принципиальная схема PlantUML

```
@startuml

interface Component {
  + operation()
}

class Leaf {
  + operation()
}

abstract class Composite {
  + operation()
  + add(Component)
  + remove(Component)
  + getChild(int)
  - components: ArrayList<Component>
}

class ConcreteComposite {
  + operation()
}

Component <|-- Leaf
Component <|-- Composite
Composite -- Component
ConcreteComposite -- Composite

@enduml
```

Когда использовать шаблон Композит

Шаблон Композит следует использовать в следующих случаях:

1. Необходимо создать сложные иерархические структуры из простых объектов.
2. Необходимо обеспечить единообразный доступ к объектам, независимо от их уровня в иерархии.

Структура организации, которая состоит из нескольких подразделений. Например, компания, которая состоит из отделов продаж, маркетинга, разработки и администрирования. Каждое подразделение может содержать свои собственные подразделения, создавая иерархическую структуру. Все подразделения и их сотрудники являются частью единой организации и могут взаимодействовать друг с другом для достижения общих целей.

Пример кода с использованием шаблона Композит:

```
from abc import ABC, abstractmethod

# Интерфейс, который должен быть реализован
class IComponent(ABC):
    @abstractmethod
    def operation(self):
        pass

# Конкретная реализация интерфейса IComponent для листовых объектов
class Leaf(IComponent):
    def operation(self):
        print("Leaf: Operation")

# Базовый класс-композит, реализующий интерфейс IComponent и содержащий список дочерних объектов-компонентов
class Composite(IComponent):
    def __init__(self):
        self._components = []

    def operation(self):
        for component in self._components:
            component.operation()

    def add(self, component: IComponent):
        self._components.append(component)

    def remove(self, component: IComponent):
        self._components.remove(component)

    def get_child(self, index: int):
        return self._components[index]

# Конкретная реализация интерфейса IComponent для объектов-композитов
class ConcreteComposite(Composite):
    def operation(self):
```

```
        print("ConcreteComposite: Operation")
        super().operation()

# Пример использования
if __name__ == "__main__":
    leaf_1 = Leaf()
    leaf_2 = Leaf()
    composite_1 = ConcreteComposite()
    composite_2 = ConcreteComposite()
    composite_1.add(leaf_1)
    composite_1.add(composite_2)
    composite_2.add(leaf_2)
    composite_1.operation()
```

Подробное описание кода

В этом примере определены классы `IComponent`, `Leaf`, `Composite` и `ConcreteComposite`. Класс `IComponent` представляет собой интерфейс, который должен быть реализован. Класс `Leaf` содержит конкретную реализацию интерфейса `IComponent` для листовых объектов. Класс `Composite` выступает в роли базового класса-композита, реализующего интерфейс `IComponent` и содержащего список дочерних объектов-компонентов. Класс `ConcreteComposite` представляет собой конкретную реализацию интерфейса `IComponent` для объектов-композитов.

Класс `Composite` наследует интерфейс `IComponent` и реализует его метод `operation()`. В конструкторе `Composite` инициализируется пустой список `_components` для хранения дочерних объектов-компонентов. В методе `operation()` класса `Composite` вызывается метод `operation()` для всех дочерних объектов-компонентов, сохраненных в списке `_components`.

Класс `Composite` также содержит методы `add()`, `remove()` и `get_child()` для управления дочерними объектами-компонентами. Метод `add()` добавляет объект-компонент в список `_components`. Метод `remove()` удаляет объект-компонент из списка `_components`. Метод `get_child()` возвращает дочерний объект-компонент по его индексу в списке `_components`.

Класс `ConcreteComposite` наследует класс `Composite` и переопределяет его метод `operation()`. В методе `operation()` класса `ConcreteComposite` выполняется дополнительная функциональность перед вызовом метода `operation()` базового класса `Composite`.

В конце примера создаются экземпляры классов `Leaf` и `ConcreteComposite`, добавляются экземпляры класса `Leaf` и `ConcreteComposite` в экземпляр класса `ConcreteComposite` с помощью метода `add()`, и вызывается метод `operation()` экземпляра класса `ConcreteComposite`.

Дополнительный пример кода с менее абстрактным кодом

Пример кода с использованием шаблона Композит для работы с файлами и папками:

```
from abc import ABC, abstractmethod
import os
```

```
# Интерфейс для работы с файлами и папками
class IFileSystemObject(ABC):
    @abstractmethod
    def get_name(self):
        pass

    @abstractmethod
    def get_size(self):
        pass

# Конкретная реализация интерфейса IFileSystemObject для файлов
class File(IFileSystemObject):
    def __init__(self, filename):
        self._filename = filename

    def get_name(self):
        return os.path.basename(self._filename)

    def get_size(self):
        return os.path.getsize(self._filename)

# Конкретная реализация интерфейса IFileSystemObject для папок
class Directory(IFileSystemObject):
    def __init__(self, directoryname):
        self._directoryname = directoryname
        self._objects = []
        for obj in os.scandir(self._directoryname):
            if obj.is_file():
                self._objects.append(File(obj.path))
            elif obj.is_dir():
                self._objects.append(Directory(obj.path))

    def get_name(self):
        return os.path.basename(self._directoryname)

    def get_size(self):
        return sum([obj.get_size() for obj in self._objects])

# Пример использования
if __name__ == "__main__":
    directory = Directory("/home/user/documents")
    print(f"Directory '{directory.get_name()}' size={directory.get_size()}")
```

В этом примере определены классы `IFileSystemObject`, `File` и `Directory`. Класс `IFileSystemObject` представляет собой интерфейс для работы с файлами и папками. Класс `File` содержит конкретную реализацию интерфейса `IFileSystemObject` для файлов. Класс `Directory` содержит конкретную реализацию интерфейса `IFileSystemObject` для папок, и выступает в роли класса-композита, содержащего список дочерних объектов-файлов и папок.

Класс `Directory` наследует интерфейс `IFileSystemObject` и реализует его методы `get_name()` и `get_size()`. В конструкторе `Directory` принимается имя папки `directoryname`, инициализируется пустой список `_objects` для хранения дочерних объектов-файлов и папок, и выполняется обход папки `directoryname` с помощью функции `os.scandir()`. В ходе обхода папки создаются экземпляры классов `File` и `Directory` для каждого файла и папки соответственно, и добавляются в список `_objects`.

В методе `get_size()` класса `Directory` вызывается метод `get_size()` для всех дочерних объектов-файлов и папок, сохраненных в списке `_objects`, и вычисляется суммарный размер папки.

В конце примера создается экземпляр класса `Directory` для папки `/home/user/documents`, и вызываются методы `get_name()` и `get_size()` экземпляра класса `Directory`.

Фасад (Facade) (30 мин)

Определение шаблона Фасад

Шаблон Фасад (Facade) предоставляет единообразный интерфейс к сложной системе, скрывая ее внутреннюю архитектуру и обеспечивая простой доступ к ее функционалу. Фасад позволяет упростить взаимодействие с сложными системами, а также обеспечить гибкость и расширяемость системы.

Программное обеспечение, которое предоставляет простой и удобный интерфейс для взаимодействия с сложной системой. Например, приложение для бронирования отелей, которое предоставляет простой интерфейс для поиска и бронирования номеров в различных отелях. Приложение взаимодействует с сложными системами бронирования отелей, чтобы предоставить пользователям простой и удобный способ бронирования.

- Приложение для управления домашними устройствами, которое предоставляет простой интерфейс для управления сложными системами, такими как отопление, кондиционирование и освещение.
- API для взаимодействия с сложной библиотекой или фреймворком.

Принципиальная схема PlantUML

```
@startuml

class Facade {
  + operation()
  - subsystemA: SubsystemA
  - subsystemB: SubsystemB
  - subsystemC: SubsystemC
}

class SubsystemA {
  + operation1()
```

```
  + operation2()
}

class SubsystemB {
  + operation1()
  + operation2()
}

class SubsystemC {
  + operation1()
  + operation2()
}

Facade -- SubsystemA
Facade -- SubsystemB
Facade -- SubsystemC

@enduml
```

Когда использовать шаблон Фасад

Шаблон Фасад следует использовать в следующих случаях:

1. Необходимо предоставить простой и единообразный интерфейс к сложной системе.
2. Необходимо скрыть внутреннюю архитектуру системы от клиентов, обеспечив гибкость и расширяемость системы.

- Примеры из реальной жизни: пульт управления бытовой техники, API для сложных систем
- Реализация шаблона Фасад на Python

Пример кода с использованием шаблона Фасад:

```python
# Сложная система, состоящая из нескольких подсистем
class SubsystemA:
    def operation1(self):
        print("SubsystemA: Operation1")

    def operation2(self):
        print("SubsystemA: Operation2")

class SubsystemB:
    def operation1(self):
        print("SubsystemB: Operation1")

    def operation2(self):
        print("SubsystemB: Operation2")

class SubsystemC:
```

```
    def operation1(self):
        print("SubsystemC: Operation1")

    def operation2(self):
        print("SubsystemC: Operation2")

# Класс-фасад, предоставляющий единообразный интерфейс к сложной системе
class Facade:
    def __init__(self):
        self._subsystem_a = SubsystemA()
        self._subsystem_b = SubsystemB()
        self._subsystem_c = SubsystemC()

    def operation(self):
        print("Facade: Operation")
        self._subsystem_a.operation1()
        self._subsystem_b.operation2()
        self._subsystem_c.operation1()

# Пример использования
if __name__ == "__main__":
    facade = Facade()
    facade.operation()
```

Подробное описание кода

В этом примере определены классы `SubsystemA`, `SubsystemB`, `SubsystemC` и `Facade`. Классы `SubsystemA`, `SubsystemB` и `SubsystemC` содержат конкретные реализации для работы с различными подсистемами сложной системы. Класс `Facade` выступает в роли класса-фасада, предоставляющего единообразный интерфейс к сложной системе.

Класс `Facade` содержит приватные поля `_subsystem_a`, `_subsystem_b` и `_subsystem_c` для хранения экземпляров классов `SubsystemA`, `SubsystemB` и `SubsystemC` соответственно. В конструкторе `Facade` создаются экземпляры классов `SubsystemA`, `SubsystemB` и `SubsystemC`, и сохраняются в приватных полях `_subsystem_a`, `_subsystem_b` и `_subsystem_c`.

Класс `Facade` также содержит метод `operation()`, предоставляющий единообразный интерфейс к сложной системе. В методе `operation()` класса `Facade` выполняется дополнительная функциональность, и вызываются методы `operation1()` и `operation2()` экземпляров классов `SubsystemA`, `SubsystemB` и `SubsystemC`, сохраненных в приватных полях `_subsystem_a`, `_subsystem_b` и `_subsystem_c`.

В конце примера создается экземпляр класса `Facade`, и вызывается метод `operation()` экземпляра класса `Facade`.

Дополнительный пример кода с менее абстрактным кодом

Пример кода с использованием шаблона Фасад для работы с базой данных:

```
import sqlite3

# Класс для работы с базой данных
class Database:
    def __init__(self, dbname):
        self._conn = sqlite3.connect(dbname)
        self._cursor = self._conn.cursor()

    def execute_query(self, query, params=()):
        self._cursor.execute(query, params)
        self._conn.commit()

    def execute_read_query(self, query, params=()):
        self._cursor.execute(query, params)
        return self._cursor.fetchall()

# Класс-фасад для работы с базой данных
class DataAccessObject:
    def __init__(self):
        self._db = Database("mydatabase.db")

    def create_user(self, username, password):
        self._db.execute_query("INSERT INTO users (username, password) VALUES (?, ?)",
(username, password))

    def get_user(self, username):
        result = self._db.execute_read_query("SELECT * FROM users WHERE username=?",
(username,))
        if len(result) == 0:
            return None
        else:
            return result[0]

# Пример использования
if __name__ == "__main__":
    dao = DataAccessObject()
    dao.create_user("user", "password")
    user = dao.get_user("user")
    print(user)
```

В этом примере определены классы `Database` и `DataAccessObject`. Класс `Database` содержит конкретную реализацию для работы с базой данных SQLite. Класс `DataAccessObject` выступает в роли класса-фасада, предоставляющего единообразный интерфейс для работы с базой данных.

Класс `DataAccessObject` содержит приватное поле `_db` для хранения экземпляра класса `Database`. В конструкторе `DataAccessObject` создается экземпляр класса `Database` для базы данных `mydatabase.db`, и сохраняется в приватном поле `_db`.

Класс `DataAccessObject` также содержит методы `create_user()` и `get_user()`, предоставляющие единообразный интерфейс для работы с базой данных. В методе `create_user()` класса `DataAccessObject` выполняется вставка новой записи в таблицу `users` базы данных `mydatabase.db` с помощью метода `execute_query()` экземпляра класса `Database`, сохраненного в приватном поле `_db`. В методе `get_user()` класса `DataAccessObject` выполняется выборка записи из таблицы `users` базы данных `mydatabase.db` по имени пользователя с помощью метода `execute_read_query()` экземпляра класса `Database`, сохраненного в приватном поле `_db`.

В конце примера создается экземпляр класса `DataAccessObject`, и вызываются методы `create_user()` и `get_user()` экземпляра класса `DataAccessObject`.

Мост (Bridge) (30 мин)

Определение шаблона Мост

Шаблон Мост (Bridge) предоставляет способ разделения интерфейса и его реализации, позволяя изменять их независимо друг от друга. Мост позволяет создавать гибкие и расширяемые системы, а также обеспечить единообразный доступ к различным реализациям интерфейса.

Способ соединения двух частей системы, которые должны взаимодействовать друг с другом, но имеют разные интерфейсы или реализованы на разных платформах. Например, мост между двумя сетями, которые используют разные протоколы связи. Мост преобразует пакеты данных из одного формата в другой, чтобы они могли быть переданы между сетями.

- Мост между двумя программами, которые используют разные форматы данных.
- Мост между двумя микросервисами, которые реализованы на разных языках программирования или платформах.
- Разделение графического интерфейса приложения и его логики, чтобы изменения в одном не влияли на другой.
- Разделение абстракции и реализации алгоритма сортировки, чтобы можно было использовать различные алгоритмы сортировки без изменения клиентского кода.

Принципиальная схема PlantUML

```
@startuml

interface Abstraction {
  + operation()
  - implementation: Implementor
}

class RefinedAbstraction {
  + operation()
}

interface Implementor {
```

```
  + operationImpl()
}

class ConcreteImplementorA {
  + operationImpl()
}

class ConcreteImplementorB {
  + operationImpl()
}

Abstraction <|-- RefinedAbstraction
Abstraction -- Implementor
Implementor <|-- ConcreteImplementorA
Implementor <|-- ConcreteImplementorB

@enduml
```

Когда использовать шаблон Мост

Шаблон Мост следует использовать в следующих случаях:

1. Необходимо разделить интерфейс и его реализацию, чтобы изменять их независимо друг от друга.
2. Необходимо обеспечить единообразный доступ к различным реализациям интерфейса.

- Примеры из реальной жизни: различные типы устройств и форматы файлов, платформенно-независимые приложения

Чем отличается мост от адаптера

Мост и адаптер - это структурные паттерны проектирования, которые используются для преобразования одного интерфейса в другой. Однако они имеют некоторые существенные отличия.

Адаптер преобразует интерфейс одного класса в другой, чтобы они могли взаимодействовать друг с другом. Он позволяет использовать существующий класс без изменения его кода, предоставляя новый интерфейс, который соответствует требуемому. Адаптер обычно используется для преобразования интерфейса одного класса в интерфейс другого класса, который ожидает клиент.

Мост же разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга. Он предоставляет интерфейс для абстракции и реализации, позволяя им взаимодействовать друг с другом. Мост используется для разделения класса на несколько классов, чтобы избежать тесной связи между ними. Это позволяет изменять реализацию независимо от абстракции, не влияя на клиентский код.

Основное отличие между адаптером и мостом заключается в том, что адаптер преобразует один интерфейс в другой, а мост разделяет интерфейс и реализацию. Адаптер используется для

преобразования существующего интерфейса в требуемый, а мост используется для разделения интерфейса и реализации, чтобы они могли изменяться независимо друг от друга.

Пример кода с использованием шаблона Мост:

```
from abc import ABC, abstractmethod

# Интерфейс, который должен быть реализован
class IImplementor(ABC):
    @abstractmethod
    def operation_impl(self):
        pass

# Конкретные реализации интерфейса IImplementor
class ConcreteImplementorA(IImplementor):
    def operation_impl(self):
        print("ConcreteImplementorA: OperationImpl")

class ConcreteImplementorB(IImplementor):
    def operation_impl(self):
        print("ConcreteImplementorB: OperationImpl")

# Базовый класс, реализующий интерфейс и содержащий ссылку на объект-реализацию
class Abstraction:
    def __init__(self, implementor: IImplementor):
        self._implementor = implementor

    def operation(self):
        self._implementor.operation_impl()

# Конкретные классы, реализующие дополнительную функциональность и наследующие базовый
класс
class RefinedAbstractionA(Abstraction):
    def operation(self):
        print("RefinedAbstractionA: Before")
        super().operation()
        print("RefinedAbstractionA: After")

class RefinedAbstractionB(Abstraction):
    def operation(self):
        print("RefinedAbstractionB: Before")
        super().operation()
        print("RefinedAbstractionB: After")

# Пример использования
if __name__ == "__main__":
    implementor_a = ConcreteImplementorA()
    implementor_b = ConcreteImplementorB()
    abstraction_a = RefinedAbstractionA(implementor_a)
```

```
    abstraction_b = RefinedAbstractionB(implementor_b)
    abstraction_a.operation()
    abstraction_b.operation()
```

Подробное описание кода

В этом примере определены классы `IImplementor`, `ConcreteImplementorA`, `ConcreteImplementorB`, `Abstraction`, `RefinedAbstractionA` и `RefinedAbstractionB`. Класс `IImplementor` представляет собой интерфейс, который должен быть реализован. Классы `ConcreteImplementorA` и `ConcreteImplementorB` содержат конкретные реализации интерфейса `IImplementor`. Класс `Abstraction` выступает в роли базового класса, реализующего интерфейс и содержащего ссылку на объект-реализацию. Классы `RefinedAbstractionA` и `RefinedAbstractionB` представляют собой конкретные классы, реализующие дополнительную функциональность и наследующие базовый класс `Abstraction`.

Класс `Abstraction` содержит приватное поле `_implementor` для хранения экземпляра класса, реализующего интерфейс `IImplementor`. В конструкторе `Abstraction` принимает экземпляр класса, реализующего интерфейс `IImplementor`, и сохраняет его в приватном поле `_implementor`. Класс `Abstraction` также содержит метод `operation()`, вызывающий метод `operation_impl()` экземпляра класса, сохраненного в приватном поле `_implementor`.

Классы `RefinedAbstractionA` и `RefinedAbstractionB` наследуют базовый класс `Abstraction` и переопределяют его метод `operation()`. В методе `operation()` классов `RefinedAbstractionA` и `RefinedAbstractionB` выполняется дополнительная функциональность перед и после вызова метода `operation()` базового класса `Abstraction`.

В конце примера создаются экземпляры классов `ConcreteImplementorA` и `ConcreteImplementorB`, и экземпляры классов `RefinedAbstractionA` и `RefinedAbstractionB` с передачей экземпляров классов `ConcreteImplementorA` и `ConcreteImplementorB` в конструкторы. Затем вызывается метод `operation()` экземпляров классов `RefinedAbstractionA` и `RefinedAbstractionB`.

Дополнительный пример кода с менее абстрактным кодом

Пример кода с использованием шаблона Мост для работы с различными типами устройств:

```
from abc import ABC, abstractmethod

# Интерфейс для работы с устройствами
class IDevice(ABC):
    @abstractmethod
    def enable(self):
        pass

    @abstractmethod
    def disable(self):
        pass
```

```python
# Конкретные реализации интерфейса IDevice для различных типов устройств
class ConcreteDeviceA(IDevice):
    def enable(self):
        print("ConcreteDeviceA: Enable")

    def disable(self):
        print("ConcreteDeviceA: Disable")

class ConcreteDeviceB(IDevice):
    def enable(self):
        print("ConcreteDeviceB: Enable")

    def disable(self):
        print("ConcreteDeviceB: Disable")

# Базовый класс, реализующий интерфейс и содержащий ссылку на объект-реализацию
class Device:
    def __init__(self, device: IDevice):
        self._device = device

    def enable(self):
        self._device.enable()

    def disable(self):
        self._device.disable()

# Конкретные классы, реализующие дополнительную функциональность и наследующие базовый
класс
class AdvancedDeviceA(Device):
    def enable(self):
        print("AdvancedDeviceA: Before")
        super().enable()
        print("AdvancedDeviceA: After")

    def disable(self):
        print("AdvancedDeviceA: Before")
        super().disable()
        print("AdvancedDeviceA: After")

class AdvancedDeviceB(Device):
    def enable(self):
        print("AdvancedDeviceB: Before")
        super().enable()
        print("AdvancedDeviceB: After")

    def disable(self):
        print("AdvancedDeviceB: Before")
        super().disable()
        print("AdvancedDeviceB: After")
```

```
# Пример использования
if __name__ == "__main__":
    device_a = ConcreteDeviceA()
    device_b = ConcreteDeviceB()
    advanced_device_a = AdvancedDeviceA(device_a)
    advanced_device_b = AdvancedDeviceB(device_b)
    advanced_device_a.enable()
    advanced_device_a.disable()
    advanced_device_b.enable()
    advanced_device_b.disable()
```

В этом примере определены классы `IDevice`, `ConcreteDeviceA`, `ConcreteDeviceB`, `Device`, `AdvancedDeviceA` и `AdvancedDeviceB`. Класс `IDevice` представляет собой интерфейс для работы с устройствами. Классы `ConcreteDeviceA` и `ConcreteDeviceB` содержат конкретные реализации интерфейса `IDevice` для различных типов устройств. Класс `Device` выступает в роли базового класса, реализующего интерфейс `IDevice` и содержащего ссылку на объект-реализацию. Классы `AdvancedDeviceA` и `AdvancedDeviceB` представляют собой конкретные классы, реализующие дополнительную функциональность и наследующие базовый класс `Device`.

Класс `Device` содержит приватное поле `_device` для хранения экземпляра класса, реализующего интерфейс `IDevice`. В конструкторе `Device` принимает экземпляр класса, реализующего интерфейс `IDevice`, и сохраняет его в приватном поле `_device`. Класс `Device` также содержит методы `enable()` и `disable()`, вызывающие соответствующие методы экземпляра класса, сохраненного в приватном поле `_device`.

Классы `AdvancedDeviceA` и `AdvancedDeviceB` наследуют базовый класс `Device` и переопределяют его методы `enable()` и `disable()`. В методах `enable()` и `disable()` классов `AdvancedDeviceA` и `AdvancedDeviceB` выполняется дополнительная функциональность перед и после вызова соответствующих методов базового класса `Device`.

В конце примера создаются экземпляры классов `ConcreteDeviceA` и `ConcreteDeviceB`, и экземпляры классов `AdvancedDeviceA` и `AdvancedDeviceB` с передачей экземпляров классов `ConcreteDeviceA` и `ConcreteDeviceB` в конструкторы. Затем вызываются методы `enable()` и `disable()` экземпляров классов `AdvancedDeviceA` и `AdvancedDeviceB`.

Прокси, Заместитель (Proxy) (30 мин)

Определение шаблона Прокси

Шаблон Прокси (Proxy) предоставляет способ управления доступом к объекту, выступая в роли посредника между клиентом и объектом. Прокси позволяет обеспечить безопасность, контроль доступа, кэширование и другие возможности, не изменяя код объекта.

Принципиальная схема PlantUML

```
@startuml

interface Subject {
  + request()
}

class RealSubject {
  + request()
}

class Proxy {
  + request()
  - real_subject: RealSubject
}

Subject <|-- RealSubject
Subject <|-- Proxy
Proxy -- RealSubject

@enduml
```

Когда использовать шаблон Прокси

Шаблон Прокси следует использовать в следующих случаях:

1. Необходимо обеспечить безопасность и контроль доступа к объекту.
2. Необходимо обеспечить кэширование, удаленный доступ и другие возможности, не изменяя код объекта.

Примеры из реальной жизни

1. Кэширующие серверы. Они выступают в роли посредников между клиентом и сервером, и могут кэшировать ответы сервера, чтобы уменьшить нагрузку на сервер и ускорить доставку контента клиенту. Кэширующие серверы могут быть представлены с помощью шаблона Прокси (Proxy).

Например, можно представить сервер с помощью конкретного класса, который реализует интерфейс `ISubject`. Этот класс может содержать базовый функционал сервера, такой как обработку запросов, доступ к базе данных и прочее.

Затем можно создать класс-прокси `Proxy`, который реализует тот же интерфейс `ISubject`, что и сервер. Этот класс-прокси может содержать ссылку на экземпляр сервера и выступать в роли посредника между клиентом и сервером.

Класс-прокси `Proxy` может кэшировать ответы сервера на запросы клиентов, чтобы уменьшить нагрузку на сервер и ускорить доставку контента клиенту. При получении запроса от клиента, класс-прокси `Proxy` сначала проверяет, есть ли ответ на этот запрос в кэше. Если ответ есть в кэше,

класс-прокси `Proxy` возвращает его клиенту, не обращаясь к серверу. Если ответа нет в кэше, класс-прокси `Proxy` обращается к серверу, получает ответ, сохраняет его в кэше и возвращает клиенту.

2. Защита доступа к ресурсам. Она может быть реализована с помощью прокси-серверов, которые выступают в роли посредников между клиентом и сервером, и могут выполнять дополнительную функциональность, такую как аутентификацию и авторизацию клиента, фильтрацию трафика, шифрование данных и прочее. Защита доступа к ресурсам может быть представлена с помощью шаблона Прокси (Proxy).

Например, можно представить сервер с помощью конкретного класса, который реализует интерфейс `ISubject`. Этот класс может содержать базовый функционал сервера, такой как обработку запросов, доступ к базе данных и прочее.

Затем можно создать класс-прокси `Proxy`, который реализует тот же интерфейс `ISubject`, что и сервер. Этот класс-прокси может содержать ссылку на экземпляр сервера и выступать в роли посредника между клиентом и сервером.

Класс-прокси `Proxy` может выполнять дополнительную функциональность, такую как аутентификацию и авторизацию клиента, фильтрацию трафика, шифрование данных и прочее. При получении запроса от клиента, класс-прокси `Proxy` сначала проверяет, является ли клиент авторизованным и аутентифицированным для доступа к ресурсу. Если клиент не прошел аутентификацию или авторизацию, класс-прокси `Proxy` отклоняет запрос и возвращает клиенту соответствующее сообщение об ошибке. Если клиент прошел аутентификацию и авторизацию, класс-прокси `Proxy` может выполнить дополнительную функциональность, такую как фильтрацию трафика или шифрование данных, перед тем, как передать запрос серверу.

Пример кода с использованием шаблона Прокси:

```
from abc import ABC, abstractmethod

# Интерфейс, который должен быть реализован
class ISubject(ABC):
    @abstractmethod
    def request(self):
        pass

# Конкретная реализация интерфейса ISubject
class RealSubject(ISubject):
    def request(self):
        print("RealSubject: Request")

# Класс-прокси, реализующий интерфейс ISubject и выступающий в роли посредника между
клиентом и объектом
class Proxy(ISubject):
    def __init__(self, real_subject: RealSubject):
        self._real_subject = real_subject
```

```
    def request(self):
        # Выполнение дополнительной функциональности перед вызовом метода request
объекта
        print("Proxy: Before")
        self._real_subject.request()
        # Выполнение дополнительной функциональности после вызова метода request
объекта
        print("Proxy: After")

# Пример использования
if __name__ == "__main__":
    real_subject = RealSubject()
    proxy = Proxy(real_subject)
    proxy.request()
```

Подробное описание кода

В этом примере определены классы `ISubject`, `RealSubject` и `Proxy`. Класс `ISubject` представляет собой интерфейс, который должен быть реализован. Класс `RealSubject` содержит конкретную реализацию интерфейса `ISubject`. Класс `Proxy` выступает в роли класса-прокси, реализующего интерфейс `ISubject` и выступающего в роли посредника между клиентом и объектом.

Класс `Proxy` содержит приватное поле `_real_subject` для хранения экземпляра класса `RealSubject`. В конструкторе `Proxy` принимает экземпляр класса `RealSubject` и сохраняет его в приватном поле `_real_subject`. Класс `Proxy` также содержит метод `request()`, вызывающий метод `request()` экземпляра класса `RealSubject`, сохраненного в приватном поле `_real_subject`.

В методе `request()` класса `Proxy` может выполняться дополнительная функциональность перед и после вызова метода `request()` экземпляра класса `RealSubject`. В этом примере выводится сообщение "Proxy: Before" перед вызовом метода `request()` экземпляра класса `RealSubject`, и сообщение "Proxy: After" после вызова метода `request()` экземпляра класса `RealSubject`.

В конце примера создаются экземпляры классов `RealSubject` и `Proxy`, и вызывается метод `request()` экземпляра класса `Proxy`.

Дополнительный пример кода с менее абстрактным кодом

Пример кода с использованием шаблона Прокси для кэширования результатов вызовов функций:

```
import functools

# Декоратор-прокси для кэширования результатов вызовов функции
def cache_proxy(func):
    @functools.wraps(func)
    def proxy(*args, **kwargs):
        cache_key = args + tuple(sorted(kwargs.items()))
```

```
        if cache_key in proxy.cache:
            print("Cache hit")
            return proxy.cache[cache_key]
        else:
            print("Cache miss")
            result = func(*args, **kwargs)
            proxy.cache[cache_key] = result
            return result
    proxy.cache = {}
    return proxy

# Функция, которую необходимо декорировать
@cache_proxy
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Пример использования
if __name__ == "__main__":
    print(fibonacci(10))
    print(fibonacci(10))
```

В этом примере определена функция `cache_proxy()`, представляющая собой декоратор-прокси для кэширования результатов вызовов функции. Функция `fibonacci()` представляет собой функцию, которую необходимо декорировать.

Функция `cache_proxy()` принимает в качестве аргумента функцию `func` и возвращает вложенную функцию `proxy()`, выступающую в роли посредника между клиентом и функцией `func`. Вложенная функция `proxy()` принимает произвольное количество аргументов `*args` и `**kwargs`, и вызывает функцию `func` с переданными аргументами.

Перед вызовом функции `func` вложенная функция `proxy()` вычисляет ключ `cache_key` на основе переданных аргументов `*args` и `**kwargs`, и проверяет, содержится ли этот ключ в словаре `proxy.cache`. Если ключ `cache_key` содержится в словаре `proxy.cache`, то вложенная функция `proxy()` выводит сообщение "Cache hit" и возвращает значение, сохраненное в словаре `proxy.cache` по ключу `cache_key`.

Если ключ `cache_key` не содержится в словаре `proxy.cache`, то вложенная функция `proxy()` выводит сообщение "Cache miss", вызывает функцию `func` с переданными аргументами `*args` и `**kwargs`, сохраняет результат вызова функции `func` в словаре `proxy.cache` по ключу `cache_key`, и возвращает этот результат.

В конце примера вызывается функция `fibonacci()` дважды с аргументом `10`. При первом вызове функции `fibonacci()` происходит "Cache miss", и вычисляется значение функции `fibonacci()` для аргумента `10`. При втором вызове функции `fibonacci()` происходит "Cache hit", и возвращается значение, сохраненное в словаре `proxy.cache` при первом вызове функции `fibonacci()`.

Здесь все. Удачи в изучении структурных шаблонов проектирования!