Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

РЕФЕРАТ

На тему: «Паттерн Фасад в системе веб-разработки на Python: упрощение взаимодействия с веб-сервисами.»

Выполнил:
Горшков Виталий Игоревич
3 курс, группа ИВТ-б-о-21-1,
09.03.01 – Информатика и вычислительная техника, профиль (профиль)
09.03.01 – Информатика и вычислительная техника, профиль «Автоматизированные системы обработки информации и управления», очная форма обучения

_____
(подпись)

Проверил:
Воронкин Р.А., канд. тех. наук, доцент,
доцент кафедры инфокоммуникаций
Института цифрового развития,

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Содержание

Введение

Веб-разработка является быстро развивающейся областью программирования, где взаимодействие с внешними сервисами и API стало неотъемлемой частью процесса разработки. Когда проект включает в себя несколько веб-сервисов со своими собственными интерфейсами и методами взаимодействия, становится сложно поддерживать структуру и обеспечивать легкость взаимодействия между компонентами системы. Именно здесь на помощь приходит паттерн Фасад, который позволяет упростить сложные системы путем предоставления унифицированного интерфейса для взаимодействия с веб-сервисами.

Актуальность этой темы является, что в современной веб-разработке существует множество сервисов и API, на которых основываются приложения. Взаимодействие с каждым из них требует понимания и использования специфических методов и протоколов. В этом контексте паттерн Фасад становится актуальным, так как он позволяет абстрагироваться от сложностей каждого веб-сервиса и предоставляет унифицированный интерфейс, что упрощает разработку, тестирование и поддержку системы.

Целью данной работы является изучение паттерна Фасад и его применение в системе веб-разработки на языке Python для упрощения взаимодействия с веб-сервисами. Реферат будет посвящен исследованию основных принципов и механизмов паттерна Фасад, а также его практическому применению для создания унифицированного интерфейса взаимодействия с веб-сервисами.

Задачи:

1. Изучить основные принципы и описание паттерна Фасад.
2. Рассмотреть особенности веб-разработки на языке Python и взаимодействия с веб-сервисами в этой среде.
3. Исследовать примеры применения паттерна Фасад в системе веб-

разработки на Python.

Глава 1. Основные понятия и паттерны проектирования

1.1 Общее описание паттерна и его цель

Паттерн Фасад (Facade) является структурным паттерном проектирования, который предоставляет простой интерфейс для взаимодействия с более сложной подсистемой объектов. Он позволяет сократить и упростить код, скрывая сложную логику за одним простым интерфейсом, чтобы клиентам было легче использовать систему.

В контексте веб-разработки на Python, паттерн Фасад может быть великолепным инструментом для упрощения взаимодействия с различными подсистемами, такими как базы данных, внешние API, аутентификация и другие. Рассмотрим пример применения паттерна Фасад для системы веб-разработки на Python.

Допустим, у нас есть веб-приложение, которое взаимодействует с базой данных, отправляет запросы по API и обрабатывает данные. Вместо того чтобы иметь распределенный код, который непосредственно взаимодействует с каждой из подсистем, мы можем создать Фасад, который будет представлять собой простой интерфейс для работы с системой.

Наш Фасад веб-разработки на Python может иметь методы для выполнения основных операций, таких как создание, чтение, обновление и удаление (CRUD) объектов в базе данных. Он может также предоставлять методы для отправки запросов по API и обработки полученных данных. Внутри Фасада будут скрыты сложности взаимодействия с соответствующими модулями или библиотеками.

Преимущества применения паттерна Фасад в системе веб-разработки на Python заключаются в следующем:

1. Упрощение кода: Фасад предоставляет простой интерфейс, который скрывает сложности и детали взаимодействия с подсистемами. Это делает код более читабельным, легким для понимания и поддержки.

2. Сокрытие сложности: Фасад абстрагирует сложность каждой подсистемы, предоставляя клиентам единый и удобный интерфейс взаимодействия. Клиентам не нужно знать все детали каждой подсистемы, что повышает уровень абстракции и упрощает использование системы.

3. Гибкость и границы: Фасад может служить точкой входа для взаимодействия с системой и определять границы доступа. Это позволяет изменять внутреннюю реализацию подсистем без влияния на код клиентов, что делает систему более гибкой и расширяемой.

4. Улучшение модульности: Паттерн Фасад способствует улучшению модульности системы, разделяя сложность на более мелкие и независимые подсистемы. Это упрощает разработку, тестирование и сопровождение каждой компоненты системы

1.2 Принцип работы паттерна "Фасад" и его основные компоненты

Паттерн “Фасад” предоставляет унифицированный интерфейс для работы с набором объектов или для выполнения группы операций. В Python, он может быть реализован с помощью класса, который предоставляет методы для доступа и манипуляции другими объектами.

Вот пример реализации паттерна “Фасад” на Python:

```
class Repository:
    def __init__(self, connection_string):
        self.connection_string = connection_string

    def execute(self, query):
        # Реализация доступа к базе данных
        pass


class UserRepository(Repository):
    def get_all_users(self):
        query = "SELECT * FROM users;"
        return self.execute(query)


class ProductRepository(Repository):
    def get_products(self):
        query = "SELECT * FROM products;"
        return self.execute(query)


class Facade:
    def __init__(self, user_repo, product_repo):
        self.user_repo = user_repo
        self.product_repo = product_repo


    def getAllUsers(self):
        return self.user_repo.get_all_users()
def getProducts(self):
return self.product_repo.get_products()
```

```
if name == “main”:
connection_string_user = “…”
connection_string_product = “…”
user_repo = UserRepository(connection_string=connection_string_user)
product_repo = ProductRepository(connection_string=connection_string_product)

facade = Facade(user_repo=user_repo, product_repo=product_repo)
users = facade.getAllUsers()
products = facade.getProducts()

print(f"Всего пользователей: {len(users)}")
print("Первые 10 пользователей:")
for user in users[:10]:
    print(user)

print()
print(f"Всего продуктов: {len(products)}")
print("Первые 5 продуктов:")
for product in products[:5]:
    print(product)
```

В этом примере у нас есть два класса (UserRepository и ProductRepository), реализующие паттерн Repository. Они оба наследуют от класса Repository и имеют свои собственные методы для выполнения SQL-запросов к базам данных.

Класс Facade выступает в роли “Фасада”, предоставляя унифицированный доступ к этим репозиториям. Методы getAllUsers и getProducts обращаются к соответствующим методам внутри UserRepository и ProductRepository.

Таким образом, Facade скрывает детали реализации доступа к данным и предоставляет простой интерфейс для управления данными.

Глава 2. Реализация паттерна "Фасад"

2.1 Реализация паттерна "фасад" с использованием Псевдокода

В псевдокоде паттерн Фасад можно описать следующим образом:

Фасад:

- Создать экземпляр каждого адаптера
- При необходимости, получить доступ к субъектам через адаптеры

Адаптер:

- Преобразовать запрос от фасада в запрос, понятный субъекту
- Передать запрос субъекту и обработать его ответ
- Вернуть ответ фасаду

Субъект:

- Обработать запрос адаптера и вернуть ответ

Пример использования паттерна:

```
# создаем адаптеры и субъекты

adapter1 = Adapter1()

adapter2 = Adapter2()

subject1 = Subject1()

subject2 = Subject2()


# создаем фасад

facade = Facade()


# подключаем адаптеры к фасаду

facade.addAdapter(adapter1)
```

```
facade.addAdapter(adapter2)


# отправляем запрос через фасад

result = facade.processRequest()
```

Структура:
Паттерн «Фасад» состоит из трех основных элементов:

Фасад (Facade) — представляет собой класс или модуль, который скрывает сложность системы и предоставляет простой и удобный интерфейс для взаимодействия с ней.

Субъекты (или адаптеры) (Adapters или Subjects) — представляют собой объекты или классы, которые реализуют основные функции системы. Каждый адаптер адаптирует интерфейс фасада к интерфейсу конкретного субъекта.

Объекты или субъекты (Entities or Subjects) — это основные элементы системы, с которыми работает фасад. Они могут быть сложными и иметь свои собственные интерфейсы и методы работы.

Для использования паттерна «Фасад», необходимо выполнить следующие шаги:

1. Определить интерфейс фасада. Фасад должен иметь методы для управления субъектами и для предоставления доступа к ним.
2. Определить адаптеры для каждого из субъектов. Адаптеры должны преобразовывать вызовы фасада в вызовы субъектов.
3. Создать экземпляры адаптеров и подключить их к фасаду.
4. При использовании фасада, клиентский код будет взаимодействовать только с ним, не зная о сложности системы и о конкретных субъектах.

Данные требования немного выходят за рамки реализации "фасада", но полагаю, что так будет немного интереснее.

И прежде чем перейти к реализации давайте разберемся, как вообще такая функциональность реализуется "в лоб". В boto3 есть много способов ее имплементировать, но для примера возьмем один. Также будем считать, что

ключи доступа к AWS хранятся в переменных окружения как AWS_ACCESS_KEY_ID и AWS_SECRET_ACCESS_KEY.

В первую очередь получим s3 как ресурс:

```
import boto3

import os



AWS_ACCESS_KEY_ID = os.environ['AWS_ACCESS_KEY_ID']

AWS_SECRET_ACCESS_KEY = os.environ['AWS_SECRET_ACCESS_KEY']



resource = boto3.resource(

    's3',

    aws_access_key_id=AWS_ACCESS_KEY_ID,

    aws_secret_access_key=AWS_SECRET_ACCESS_KEY

)
```

Теперь определим объект хранилища, на котором будет тестировать работу программы. Для этого нам необходим тестовый bucket и имя используемого объекта.

```
bucket_name = 'test_bucket'

path_to_file = 'test_folder/test_object.txt'
```

Сохранить файл путем передачи контента:

```
content = b'test_object_data'



bucket = resource.Bucket(bucket_name)

file_object = bucket.Object(path_to_file)



file_object.put(Body=content)
```

Получить контент:

```
content = file_object.get()['Body'].read()
```

Удалить файл:

```
file_object.delete()
```

Выглядит не сложно, а теперь представим, что нам необходимо постоянно работать с данными, хранящимися в s3 в разных модулях, в разных методах. И если надо будет как-то изменить реализацию, сменить библиотеку или подключить другое хранилище, то все становится совсем уж плохо.

Приступим к реализации архитектуры.

Так как мы знаем, что у нас может быть несколько типов хранилищ, реализуем сначала абстрактный класс для объекта хранилища.

class StorageObject:

```
    def __init__(self, path: str, base_path: str, resource: Any = None) -> None:
        """
        path: путь к файлу или же какой-либо другой идентификатор
        base_path: базовый путь
        resource: некий исходный объект хранилища, реализуемый сторонней библиотекой,
                необходим для дальнейшего вызова методов
        """
        raise NotImplementedError

    def read(self) -> bytes:
        raise NotImplementedError

    def write(self, content: bytes) -> None:
        raise NotImplementedError

    def delete(self) -> None:
        raise NotImplementedError
```

Далее перейдем уже непосредственно к реализации объекта файла с s3.

```
class S3StorageObject(StorageObject):
    _object: BotoS3Object

    def __init__(self, path: str, base_path: str, resource: BotoS3Resource) -> None:
        """
        в данном случае base_path - это название бакета
        """
        self._object = resource.Bucket(base_path).Object(path)

    def read(self) -> bytes:
        return self._object.get()['Body'].read()

    def write(self, content: bytes) -> None:
        self._object.put(Body=content)

    def delete(self) -> None:
        self._object.delete()
```

Небольшое замечание насчет типа ресурса: `BotoS3Resource`. Это результат выполнения функции `boto3.resource('s3', *args, **kwargs)`

Но так как она явно никакой конкретный тип не возвращает, для лучшего понимания мы определили наследника typing.Protocol. И там же нам нужен

нативный тип объекта s3: `BotoS3Object` . Опять же явно в boto3 такого типа нет, потому что он формируется динамически при помощи фабрики, поэтому пишем свой.

```
class BotoS3Resource(Protocol):
    """Результат вызова boto3.resource('s3', ...)"""
    def Bucket(self, bucket_name: str): ...


class BotoS3Object(Protocol):
    """Результат boto3.resource('s3', ...).Bucket(...).Object(...)"""
    def get(self) -> dict: ...
    def put(self, Body: bytes) -> dict: ...
    def delete(self) -> dict: ...
```

Таким образом мы инкапсулировали в S3StorageObject все вызовы к более низкоуровневой библиотеке и закрыли **первое архитектурное требование**. С таким классом уже можно работать, то есть частично функциональные требования выполнены:

```
resource = boto3.resource(
    's3',
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)
```

```
file_object = S3StorageObject(path_to_file, bucket_name, resource)



content = file_object.read()

file_object.write(content)

file_object.delete()
```

Что дальше? Далее нам необходимо масштабироваться до использования разных типов хранилищ. Соответственно, нужно для каждого написать свой StorageObject.

К примеру, для доступа к файлам операционной системы можно написать что-то вроде этого:

Конечно, можно было бы эти классы использовать и так: один для одного типа, другой для второго и так далее. Это гораздо лучше варианта в лоб, но тем не менее могут быть случаи, когда даже такой вариант сложно будет масштабировать. Допустим, в случае смены хранилища для всех файлов, чтобы решить данную проблему, еще немного поднимем уровень абстракции, создав proxy-класс, реализующий те же методы, что и StorageObject, но:

- во-первых, proxy будет сам решать, объект какого типа ему создавать, а так же выступать для пользователя одной точкой входа для работы с файлами
- во-вторых, добавлять необходимую дополнительную логику в работу с файлами

Выглядеть он будет примерно так:

```
class File:

    storage: Storage
```

```
    def __init__(
        self,
        path: str,
        base_path: str | None = None,
        storage: Storage | None = None
    ) -> None:
        # хранилище можно задать при инициализации, либо заранее добавить в класс
        if storage: self.storage = storage

        self._object = self.storage.build_object(path, base_path)

    def read(self) -> bytes:
        return self._action('read')

    def write(self, content: bytes) -> None:
        self._action('write', content)

    def delete(self) -> None:
```

```
        self._action('delete')


    def _action(self, action: str, *args, **kwargs) -> Any:

        return getattr(self._object, action)(*args, **kwargs)
```

С его помощью мы закрываем **четвёртое архитектурное требование**.

Но тут еще надо разобраться с несколькими вопросами. Во-первых, что такое Storage? До этого такого класса у нас не было. И правильно, потому что каждый StorageObject мог принимать какой-то resource для формирования объекта и нам было не особо важно, откуда этот resource берётся. Сейчас же мы предполагаем, что хранилищ может быть множество и они могут меняться. Соответственно, работу по их инициализации и построению StorageObject есть смысл вынести непосредственно в хранилища Storage. Интерфейс у такого класса очень простой. Фактически нам требуется только один метод для создания StorageObject: `build_object`:

```
class Storage:

    base_path: str

    resource: Any

    object_type: type[StorageObject]


    def __init__(self, base_path: str | None = None, *args, **kwargs) -> None:

        self.resource = self._build_resource(*args, **kwargs)
```

```
        if base_path: self.base_path = base_path


    def build_object(self, path: str, base_path: str | None = None) -> StorageObject:
        return self.object_type(path, base_path or self.base_path, self.resource)


    def _build_resource(*args, **kwargs) -> Any:
        return None



class S3Storage(Storage):
    object_type = S3StorageObject


    def _build_resource(self, *args, **kwargs) -> BotoS3Resource:
        return boto3.resource('s3', *args, **kwargs)  # type: ignore
```

Подход с хранилищем хорош тем, что можно определить его на уровне конфигурации приложения так:

```
# данный способ следует использовать с осторожностью
File.storage = S3Storage(
```

```
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)

file = File(path_to_file, bucket_name)
```

или так:

```
# при необходимости меняется класс хранилища, но всё продолжает работать
storage = S3Storage(
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)

file = File(path_to_file, bucket_name, storage)
```

Таким образом закрыто **второе архитектурное требование**.

И последний момент, касающийся File, это метод `_action`.

Мы используем его, чтобы определить единственную точку обращения непосредственно к методам хранилища. Благодаря этому есть возможность только в одном месте установить логгер, блок обработки ошибок или декорировать любым другим образом. При этом затрагивая сразу всё методы.

Им мы закрываем **третье архитектурное требование.**

Итак, на данный момент все требования к проекту были выполнены и можно разработку на этом закончить.

```
storage = S3Storage(
    bucket_name,
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)


file = File(path_to_file, storage=storage)


content = file.read()
file.write(content)
file.delete()
```

Шаги реализации:

Шаги реализации паттерна «Фасад»:

1. Определение субъектов (субъектом может быть класс, функция или другой элемент, который выполняет определенную задачу).

2. Создание адаптера для каждого субъекта (адаптер преобразует вызовы фасада в вызовы субъекта).

3. Создание фасада (класс или функция, которая управляет адаптерами и предоставляет унифицированный доступ к субъектам).

4. Подключение адаптеров к фасаду (создание экземпляров адаптеров и их передача фасаду).

5. Использование фасада для управления субъектами (клиентский код взаимодействует только с фасадом, не зная о конкретных субъектах).

2.2 Использования паттерна "фасад" в других языках программирования

Паттерн "фасад" может быть использован в различных языках программирования. Вот примеры его реализации на других языках:

1. Java:

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.init();
        System.out.println(facade.multiply(5, 6));
    }
}

class Operation {
    protected int a;
    protected int b;

    Operation(int a, int b) {
        this.a = a;
        this.b = b;
    }
```

```
    int execute() {
        return a * b;
    }
}

class Addition extends Operation {
    Addition(int a, int b) {
        super(a, b);
    }
}

class Multiplication extends Operation {
    Multiplication(int a, int b) {
        super(a, b);
    }
}

class Subtraction extends Operation {
    Subtraction(int a, int b) {
        super(a, b);
    }
}
```

2. C++:

```
#include <iostream>
#include "operation.h"

class Calculator
{
  public:
    void init()
    {
      operations.push_back(new Addition());
      operations.push_back(new Multiplication());
      operations.push_back(new Subtraction());
    }

    double execute(double a, double b, int operation)
    {
      Operation* op = operations[operation];
      return op->execute(a, b);
    }

  private:
    List<Operation*> operations;
};
```

```
int main()
{
  Calculator facade;
  facade.init();
  std::cout << facade.execute(5, 6, 0) << std::endl;
  std::cout << facade.execute(10, 2, 1) << std::endl;
  std::cout << facade.execute(4, 1, 2) << std::endl;

  return 0;
}
```

Это лишь несколько примеров реализации паттерна "фасад" на различных языках программирования. В каждом из этих примеров основные концепции паттерна остаются теми же, но синтаксис и способы работы с объектами могут отличаться в зависимости от конкретного языка.

Глава 4. Преимущества и недостатки использования паттерна "фасад"

1.1 Преимущества модульности и расширяемости кода

Паттерн “Фасад” предоставляет следующие преимущества в системе веб-разработки на Python:

1. Упрощение интерфейса: Фасад упрощает взаимодействие с системой, скрывая сложность и предоставляя простой и понятный интерфейс.
2. Инкапсуляция: Паттерн “Фасад” позволяет инкапсулировать сложность системы, делая ее более гибкой и легко изменяемой.
3. Улучшение модульности: Фасад позволяет разделить систему на модули, облегчая ее тестирование и поддержку.
4 .Управление зависимостями: Фасад может использоваться для управления зависимостями между компонентами системы, позволяя избежать проблем с зависимостями и обеспечить более стабильное и надежное решение.
5. Уменьшение дублирования кода: Фасад позволяет избежать дублирования кода, обеспечивая единый интерфейс для работы с различными компонентами системы.

1.2 Недостатки, связанные с увеличением количества классов и сложности реализации

Паттерн “Фасад” в системе веб-разработки на Python имеет некоторые недостатки, такие как:

1. Увеличивает уровень абстракции, что может усложнить понимание системы.
2. Может привести к увеличению количества кода и сложности системы, если не используется правильно.
3. Требует больше времени на разработку и тестирование, так как необходимо создать и настроить несколько уровней абстракции.

Заключение

В заключении можно сказать, что паттерн “Фасад” является полезным инструментом для реализации в системе веб-разработки на Python. Он упрощает взаимодействие с системой, инкапсулирует сложность, улучшает модульность, управляет зависимостями и снижает дублирование кода, что делает систему более гибкой, простой в поддержке и стабильной. Однако, его использование также может увеличить уровень абстракции, привести к увеличению сложности системы и требовать больше времени на разработку и тестирование. Поэтому, его следует использовать с осторожностью и знанием того, как он работает, чтобы получить максимальную отдачу от его применения.

Список используемой литературы

1. Мартин Фаулер Шаблоны корпоративных приложений = Patterns of Enterprise Application Architecture (Addison-Wesley Signature Series). — М.: «Вильямс», 2009. — С. 544.

2. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес Приемы объектно-ориентированного проектирования. Паттерны проектирования = Design Patterns: Elements of Reusable Object- Oriented Software. — СПб: «Питер», 2007. — С. 366.

3. Марк Гранд Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML = Patterns in Java, Volume 1. A Catalog of Reusable Design Patterns Illustrated with UML. — М.: «Новое знание», 2004. — С. 560.

4. Крэг Ларман Применение UML 2.0 и шаблонов проектирования = Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. — М.: «Вильямс», 2006. — С. 736.

5. Джошуа Кериевски Рефакторинг с использованием шаблонов (паттернов проектирования) = Refactoring to Patterns (Addison-Wesley Signature Series). — М.: «Вильямс», 2006. — С. 400.