GeekBrains

Архитектура и шаблоны проектирования на Python

Архитектурные системные паттерны

На этом уроке

Архитектурные системные паттерны. Обзор базовых паттернов. Объектно-реляционные паттерны.

Оглавление

Архитектурные системные паттерны

Архитектурные системные паттерны, или *Промышленные шаблоны*, описывают найденные типовые решения задач по реализации приложений уровня предприятия. Они описывают хорошо зарекомендовавшие себя архитектурные подходы к проектированию крупных частей приложений, например, слой предметной области или слой доступа к данным. Простые наивные решения зачастую приводят к возникновению архитектурных антипаттернов (подробно рассмотрены в восьмом уроке).

Мартин Фаулер в книге «Шаблоны корпоративных приложений» описал более 50 таких паттернов.

Рассмотрим некоторые из них.

Базовые паттерны

Value Object // Объект-значение

Объект-значение — это небольшие простые объекты наподобие денежных значений или диапазонов дат, равенство которых основано не на равенстве идентификаторов, а на тождественности этих объектов по значению.

Например, два разных объекта, представляющих точку с координатами (1,2) в декартовом пространстве, «равны» по значению, но равенство не основано на равенстве ссылок двух объектов.

Объекты-значения должны быть неизменными (immutable object) — это необходимо для реализации неявного контракта: созданные равными, два объекта-значения должны оставаться равными. Кроме того, благодаря неизменности, код клиента не может поместить объект-значение в недопустимое состояние или привести к ошибочному поведению после инициализации.

Пример

Рассмотрим упрощённую реализацию паттерна на основе библиотеки simple-value-object. Создаём классы ValueObject и ArgsSpec, а также класс исключения CannotBeChangeException. Ответственность класса ArgsSpec — вернуть список атрибутов, прописанных в инициализаторе класса создаваемого объекта. В методе .assign_instance_arguments() при помощи функции zip связываем имена атрибутов со значениями, заданными при создании объекта, и назначаем их. Неизменность (Immutable) объекта реализуется перегрузкой метода .__setattr__(). Сравниваем объекты по их словарям — используем атрибут .__dict__.

```
from inspect import getfullargspec, getmembers, getsourcelines


class CannotBeChangeException(Exception):
    def __init__(self, *args, **kwargs):
        super().__init__('You cannot change values, create a new one')


class ValueObject(object):
    def __new__(cls, *args, **kwargs):
        self = super().__new__(cls)
        args_spec = ArgsSpec(self.__init__)

        def assign_instance_arguments():
            self.__dict__.update(
                dict(list(zip(args_spec.args[1:], args)) + list(kwargs.items()))
            )

        assign_instance_arguments()

        return self

    def __setattr__(self, name, value):
        raise CannotBeChangeException

    def __eq__(self, other):
        return self.__dict__ == other.__dict__

    def __ne__(self, other):
        return self.__dict__ != other.__dict__

    @property
    def hash(self):
        return hash(self.__class__) and hash(frozenset(self.__dict__.items()))


class ArgsSpec(object):
    def __init__(self, method):
        self._args = getfullargspec(method)[0]

    @property
    def args(self):
        return self._args
```

Создаём класс точки Point и проверяем работу:

```
class Point(ValueObject):
    def __init__(self, x, y):
        pass


point_1 = Point(1, 2)
point_2 = Point(y=2, x=1)

print(point_1.x)
print(point_1.y)

print(point_1 == point_2)
print(point_1 != point_2)
print(point_1 is point_2)

point_1.x = 5
```

Как и следовало ожидать, point_1 == point_2 дает True (считаем объекты равными по значениям), а point_1 is point_2 — False (это разные экземпляры объектов с точки зрения Python).

Registry // Реестр

Глобальный объект, который используется другими объектами для поиска общих объектов или служб.

Когда нужно найти один объект, обычно начинают с другого, связанного с целевым. Например, если нужно найти все счета конкретного покупателя, можно начать с него и использовать его методы для получения списка счетов. Однако в некоторых случаях может отсутствовать подходящий начальный объект. Например, известен ID покупателя, но нет ссылки на него. Тогда потребуется объект *Поисковик* и возникнет вопрос: как его найти?

Решение — глобально видимый объект — *Реестр*, используемый для поиска других объектов.

Реестр с единственным экземпляром объекта представляет собой одну из реализаций паттерна Singleton. Благодаря глобальной области видимости Singleton, его очень легко инстанцировать, зная лишь имя класса.

Пример

Рассмотрим уточнённый вариант реализации паттерна:

```
class RegistryHolder(type):
    REGISTRY = {}

    def __new__(cls, name, bases, attrs):
        new_cls = type.__new__(cls, name, bases, attrs)
        cls.REGISTRY[new_cls.__name__] = new_cls
        return new_cls

    @classmethod
    def get_registry(cls):
```

```
        return dict(cls.REGISTRY)
```

В этом классе будем хранить сведения обо всех объявленных в коде классах. Создаем базовый класс BaseRegisteredClass и задаём для него RegistryHolder как метакласс:

```
class BaseRegisteredClass(metaclass=RegistryHolder):
    pass
```

Клиентский код:

```
print("до создания подклассов: ")
[print(f'\t{k}') for k in RegistryHolder.REGISTRY]


class FirstClass(BaseRegisteredClass):
    def __init__(self, *args, **kwargs):
        pass


class SecondClass(BaseRegisteredClass):
    def __init__(self, *args, **kwargs):
        pass


print("\n после создания подклассов: ")
[print(f'\t{k}') for k in RegistryHolder.REGISTRY]
```

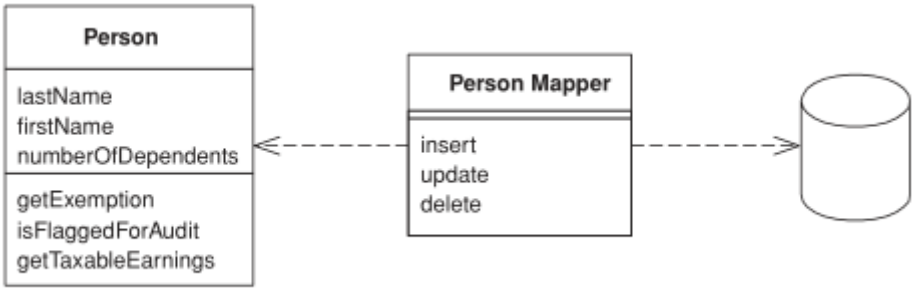
Пример вывода:

```
до создания подклассов:
    BaseRegisteredClass

после создания подклассов:
    BaseRegisteredClass
    FirstClass
    SecondClass
```

Объектно-реляционные паттерны

Data Mapper // Преобразователь данных

Это слой преобразователей, который передаёт данные между объектами и базой данных, сохраняя последние независимыми друг от друга и от самого преобразователя.



У объектов и реляционных баз данных разные механизмы структурирования данных. Например, в реляционных базах данных нет агрегирования в объектном смысле и наследования.

Паттерн Data Mapper — это слой программного кода, отделяющий объекты в памяти приложения от их отображения в базе данных. Его ответственность состоит в том, чтобы передавать данные между ними в обоих направлениях, а также изолировать их друг от друга. Предметная область не должна зависеть от особенностей реализации БД. Что касается БД, она по определению не знает о методах обработки содержащейся в ней информации.

Готовые реализации паттерна: Django ORM и SqlAlchemy.

Пример

Объект модели:

```
class Person:
    def __init__(self, id_person, first_name, last_name):
        self.id_person = id_person
        self.last_name = last_name
        self.first_name = first_name
```

Создадим в sqlite базу **patterns.sqlite** и в ней структуру таблицы:

```
CREATE TABLE person
(
idperson INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
lastname VARCHAR (32),
firstname VARCHAR (32)
);
```

Код преобразователя (без кода классов исключений):

```
class PersonMapper:
    """
    Паттерн DATA MAPPER
    Слой преобразования данных
    """

    def __init__(self, connection):
        self.connection = connection
        self.cursor = connection.cursor()

    def find_by_id(self, id_person):
        statement = "SELECT IDPERSON, FIRSTNAME, LASTNAME FROM PERSON WHERE
IDPERSON=?"

        self.cursor.execute(statement, (id_person,))
        result = self.cursor.fetchone()
        if result:
            return Person(*result)
        else:
            raise RecordNotFoundException(f'record with id={id_person} not
found')

    def insert(self, person):
```

```
        statement = "INSERT INTO PERSON (FIRSTNAME, LASTNAME) VALUES (?, ?)"
        self.cursor.execute(statement, (person.first_name, person.last_name))
        try:
            self.connection.commit()
        except Exception as e:
            raise DbCommitException(e.args)

    def update(self, person):
        statement = "UPDATE PERSON SET FIRSTNAME=?, LASTNAME=? WHERE IDPERSON=?"
        self.cursor.execute(statement, (person.first_name, person.last_name,
person.id_person))
        try:
            self.connection.commit()
        except Exception as e:
            raise DbUpdateException(e.args)

    def delete(self, person):
        statement = "DELETE FROM PERSON WHERE IDPERSON=?"
        self.cursor.execute(statement, (person.id_person,))
        try:
            self.connection.commit()
        except Exception as e:
            raise DbDeleteException(e.args)
```

Применение:

```
import sqlite3

connection = sqlite3.connect('patterns.sqlite')
person_mapper = PersonMapper(connection)
person_1 = person_mapper.find_by_id(1)
print(person_1.__dict__)
```

Unit of Work // Единица работы

Содержит список охватываемых бизнес-транзакцией объектов, координирует запись изменений в базе данных и разрешает проблемы параллелизма.

Извлекая или записывая данные в БД, важно отслеживать изменения, иначе они не будут записаны. Точно так же вы должны вставлять новые объекты, которые создаёте, и убирать любые объекты, которые удаляете.

Можно сбрасывать изменения модели в БД при каждой модификации данных модели — это приведёт к множеству очень мелких запросов к базе данных, что займёт время и снизит отзывчивость системы. Кроме того, изменяя часть данных, для поддержки непротиворечивости придётся блокировать открытой транзакцией все данные, входящие в охватываемую моделью область.

Паттерн *Единица работы* отслеживает изменения данных, которые мы производим с доменной моделью в рамках бизнес-транзакции. После того, как бизнес-транзакция закрывается, все изменения попадают в БД в виде единой транзакции.

Паттерн *Единица работы* отслеживает изменения в модели, которые могут повлиять на БД. Когда все будет готово, паттерн определит все необходимые изменения в БД, которые нужно сделать для поддержания непротиворечивости данных в модели, и БД применит их одной транзакцией.

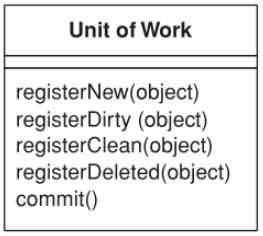


Диаграмма классов Unit of Work.

Учитывая потребность в глобальной доступности, обычно реализуется на основе *Одиночки*.

Пример

Для хранения набора изменений мы используем три списка: новые (new_objects), изменённые (dirty_objects) и удалённые (removed_objects) объекты домена модели. Здесь мы используем паттерн Супертип слоя (Суть паттерна достаточно проста, поэтому в курсе не описывается) — базовый супертип объектов модели, поддерживающий общий интерфейс для всех классов модели операции, например, операции с идентификатором.

```
import threading

class UnitOfWork:
    current = threading.local()

    def __init__(self):
        self.new_objects = []
        self.dirty_objects = []
        self.removed_objects = []

    def register_new(self, obj):
        self.new_objects.append(obj)

    def register_dirty(self, obj):
        self.dirty_objects.append(obj)

    def register_removed(self, obj):
        self.removed_objects.append(obj)

    def commit(self):
        self.insert_new()
        self.update_dirty()
        self.delete_removed()

    def insert_new(self):
        for obj in self.new_objects:
            MapperRegistry.get_mapper(obj).insert(obj)

    def update_dirty(self):
```

```
        for obj in self.dirty_objects:
            MapperRegistry.get_mapper(obj).update(obj)

    def delete_removed(self):
        for obj in self.removed_objects:
            MapperRegistry.get_mapper(obj).delete(obj)

    @staticmethod
    def new_current():
        __class__.set_current(UnitOfWork())

    @classmethod
    def set_current(cls, unit_of_work):
        cls.current.unit_of_work = unit_of_work

    @classmethod
    def get_current(cls):
        return cls.current.unit_of_work
```

Методы регистрации .register_new(), .register_dirty() и .register_removed() поддерживают состояние этих списков. Они должны выполнять базовые проверки: что идентификатор не равен None, что изменённый объект не регистрируется как новый. Метод .commit() выполняет последовательность внесения изменений в объекты БД. Каждый метод изменения находит DataMapper для каждого объекта при помощи статического метода .get_mapper() класса MapperRegistry и вызывает соответствующий метод: вставки (.insert()), обновления (.update()) или удаления (.delete()) из БД.

```
class MapperRegistry:
    @staticmethod
    def get_mapper(obj):
        if isinstance(obj, Person):
            return PersonMapper(connection)
```

Если с потоком выполнения бизнес-транзакции уже связан какой-либо объект сеанса, текущую *Единицу работы* следует поместить именно в этот объект. С логической точки зрения *Единица работы* принадлежит этому сеансу. Поэтому используем модуль threading и создаём локальную (thread-local) для потока переменную в статическом атрибуте current класса UnitOfWork. При помощи метода .new_current() создаём в потоке экземпляр объекта класса UnitOfWork, с которым дальше и работаем через геттер .get_current().

Затем следует добавить к абстрактному классу супертипа предметной области методы, позволяющие объекту модели предметной области регистрироваться в текущей единице работы.

```
class DomainObject:
    def mark_new(self):
        UnitOfWork.get_current().register_new(self)

    def mark_dirty(self):
        UnitOfWork.get_current().register_dirty(self)

    def mark_removed(self):
        UnitOfWork.get_current().register_removed(self)
```

Теперь наследуем от этого класса все классы предметной области:

```
class Person(DomainObject):
    def __init__(self, id_person, first_name, last_name):
        self.id_person = id_person
        self.last_name = last_name
        self.first_name = first_name
```

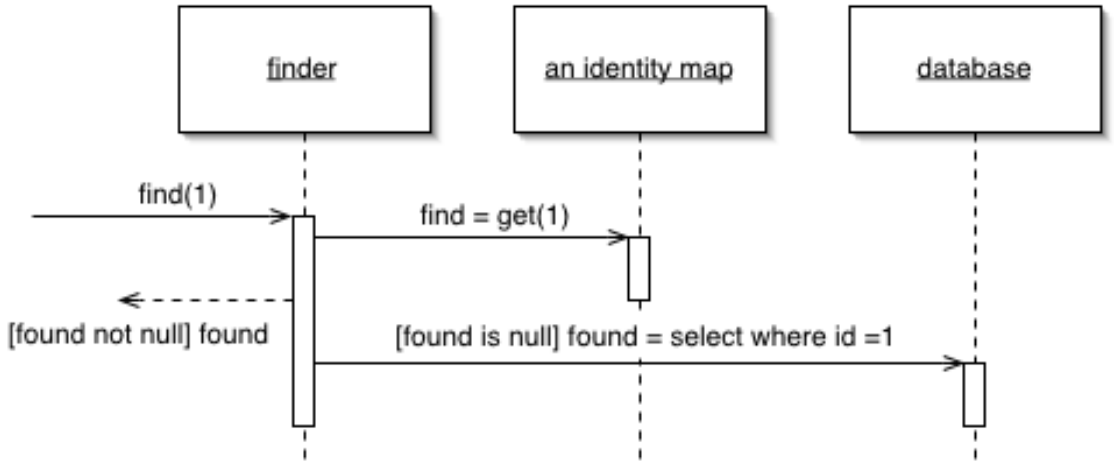
Пример клиентского кода:

```
try:
    UnitOfWork.new_current()

    new_person_1 = Person(None, 'Igor', 'Igorev')
    new_person_1.mark_new()

    new_person_2 = Person(None, 'Fedor', 'Fedorov')
    new_person_2.mark_new()

    person_mapper = PersonMapper(connection)
    exists_person_1 = person_mapper.find_by_id(1)
    exists_person_1.mark_dirty()
    print(exists_person_1.first_name)
    exists_person_1.first_name += ' Senior'
    print(exists_person_1.first_name)

    exists_person_2 = person_mapper.find_by_id(2)
    exists_person_2.mark_removed()

    print(UnitOfWork.get_current().__dict__)

    UnitOfWork.get_current().commit()
except Exception as e:
    print(e.args)
finally:
    UnitOfWork.set_current(None)

print(UnitOfWork.get_current())
```

Здесь мы создали новых пользователей, получили уже существующих и изменили их атрибуты, а также удалили одну из записей при помощи *Единицы работы*. И только в конце сохранили изменения в БД.

Identity Map // Коллекция объектов

Гарантирует, что каждый объект будет загружен из базы данных только один раз, сохраняя его в специальной коллекции. При получении запроса просматривает коллекцию в поисках нужного объекта.



Как правило, *Коллекция объектов* применяется для управления любыми объектами, которые были загружены из базы данных и затем подверглись изменениям. Основное назначение *Коллекции объектов* — не допустить ситуации, когда два разных объекта приложения будут соответствовать одной и той же записи базы данных, поскольку их изменение может происходить несогласованно и, следовательно, вызывать трудности с отображением в базе данных.

Преимущество *Коллекции объектов* — возможность её использования в качестве кеша записей, считываемых из базы данных. Это избавляет от необходимости повторно обращаться к БД, если снова понадобится какой-нибудь объект.

```
class UnitOfWork:
    ...
    person_map = {}

    @classmethod
    def add_person(cls, person):
        if person.get_id() not in cls.person_map.keys():
            cls.person_map[person.get_id()] = person

    @classmethod
    def get_person(cls, key):
        if key in cls.person_map.keys():
            return cls.person_map[key]
        else:
            return None
```

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Применять архитектурные системные паттерны.
2. Применять архитектурные системные паттерны в своём коде.

Зачем:

Для использования архитектурных системных паттернов в своём коде.

Последовательность действий:

1. Добавить базу данных к своему проекту, используя паттерн Data Mapper
2. Использовать паттерн Unit of Work.
3. Можно попробовать дополнительно реализовать Identity Map.

Дополнительные материалы

1. ValueObject:
    - simple-value-object.
    - inspect Python module.
    - ValueObject.
2. Register example.
3. UoW:
    - Domain-Driven Design: Tackling Complexity in the Heart of Software, Эрик Эванс (синяя книга).
    - Implementing Domain Driven Design, Vaughn Vernon (красная книга).
    - threading.local.
4. github.com/faif/python-patterns.

Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. Фаулер М. Шаблоны корпоративных приложений. М.: Вильямс, 2016.