GeekBrains

Архитектура и шаблоны проектирования на Python

Порождающие паттерны

На этом уроке

Обзор. Реализация. Недостатки. Abstract Factory. Factory Method. Builder. Singleton. Prototype.

Оглавление

На этом уроке

Что такое паттерн проектирования

- Обзор паттернов проектирования
- Классический каталог паттернов GoF

Порождающие паттерны

- Abstract Factory // Абстрактная фабрика
- Factory Method // Фабричный метод // уровень класса
  - Пример использования паттернов Абстрактная фабрика и Фабричный метод
- Builder // Строитель
- Singleton // Одиночка
- Prototype // Прототип

Практическое задание

Дополнительные материалы

Используемая литература

Что такое паттерн проектирования

Паттерны проектирования — это подходы к решению практических задач, **выявленные** при анализе полученных решений и применяемые многократно. Паттерны не открывают или изобретают — ихи выявляют как повторяющиеся конструкции в коде, структуре или архитектуре при разработке программ.

Кристофер Александер писал: «Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип её решения, причём таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново».

Паттерны указывают направление решения проблемы, но не дают окончательного результата — это не входит в их задачу. Ценность паттерна состоит в том, что, будучи идентифицированным однажды, он позволяет решить большое количество схожих проблем с меньшими усилиями.

Чаще всего одну задачу можно решить разными способами с применением разных паттернов: некоторые подходят лучше, другие — хуже. Неоправданное использование не подходящего в данном контексте паттерна само по себе — антипаттерн. Паттерны — не панацея, а лишь крупные строительные блоки, которые не заменят алгоритм или конкретную логику приложения.

Обзор паттернов проектирования

В начале 1990-х Эрих Гамма, вдохновлённый книгой К. Александера, обдумывал каталог паттернов проектирования программного обеспечения как тему своей докторской. В дальнейшем к работе над каталогом присоединились Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Результатом их работы стала широко известная книга, включающая в себя каталог и 23 основных шаблона проектирования, более известных как GoF («Банда четырёх»).

В 1996 году группа инженеров Siemens опубликовала свой набор паттернов, известный как POSA.

Мартин Фаулер в 2001 году в книге «Шаблоны корпоративных приложений» описал каталог паттернов проектирования корпоративных приложений.

Существует каталог паттернов параллельного программирования, впервые описанный в 2000 году.

Паттерны для языка Python рассмотрены в книгах М. Саммерфилд «Python на практике» и М. Лутц «Изучаем Python».

Классический каталог паттернов GoF

Классический каталог паттернов GoF описывает основу основ: базовые паттерны создания, структурирования и взаимодействия объектов на нижнем уровне проектирования — уровне классов и реальных объектов программы.

Другие паттерны более высокого уровня, например MVC, могут использовать эти базовые паттерны для своего построения. Например, MVC может использовать *Фабричный метод* для определения конкретного класса *Контроллера*, и *Декоратор* для добавления к представлению возможности прокрутки — основные отношения описываются паттернами *Наблюдатель*, *Компоновщик* и *Стратегия*.

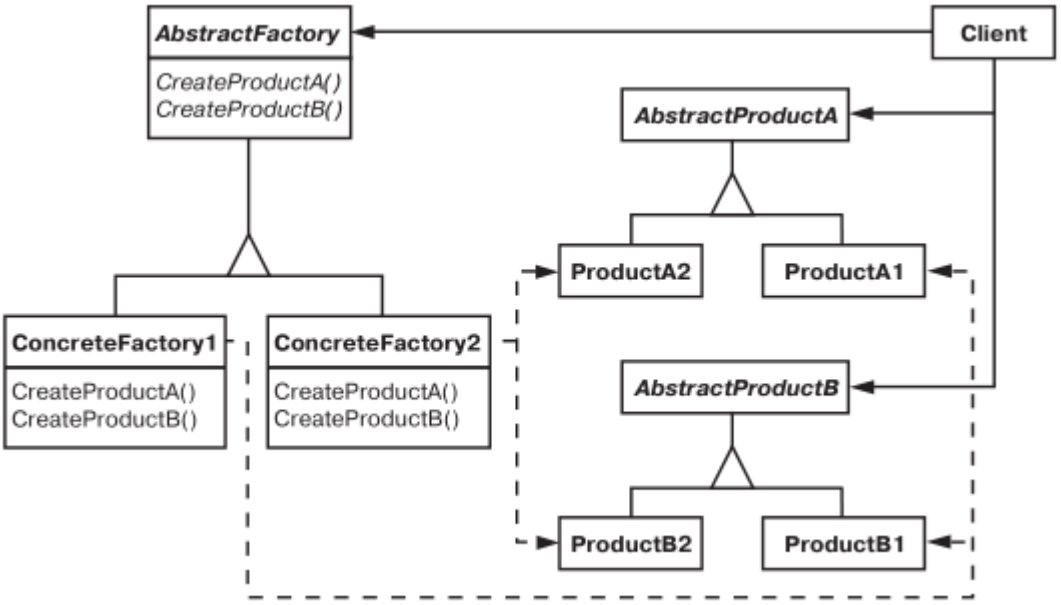
Каталог GoF классифицирует паттерны по трем целям:

1. Порождающие — абстрагируют процесс инстанцирования объектов.
2. Структурные — относятся к вопросам создания более крупных структур из классов и объектов.
3. Поведенческие — характеризуют взаимодействие классов или объектов.

Порождающие паттерны

Создание объектов оператором **new** может привести к проблемам дизайна или дополнительной сложности. Порождающие паттерны проектирования решают эту проблему, так или иначе управляя процессом создания объектов. Если оператор **new** – это арифметика чисел, то *Порождающие паттерны* — «алгебра оператора **new**».

Abstract Factory // Абстрактная фабрика



Идиома — фабрика, порождающая различные реализации объектов предопределённых интерфейсов. Позволяет легко масштабировать вширь реализации связанных семейств классов при неизменной логике клиентского кода.

Client — клиентский код, который использует исключительно интерфейсы, объявленные в классах AbstractFactory и AbstractProduct.

AbstractFactory — объявляет интерфейс для операций, создающих абстрактные объекты *Продукты*.

ConcreteFactory — конкретная фабрика, имплементирует операции создания конкретных экземпляров классов объектов *Продуктов*.

AbstractProduct — абстрактный продукт, описывает интерфейс продукта, которым пользуется клиент.

ConcreteProduct — конкретный продукт, конкретная реализация класса, имплементирует интерфейс AbstractProduct. Объекты этого класса создаются соответствующей реализацией конкретной фабрики и клиенту не видны.

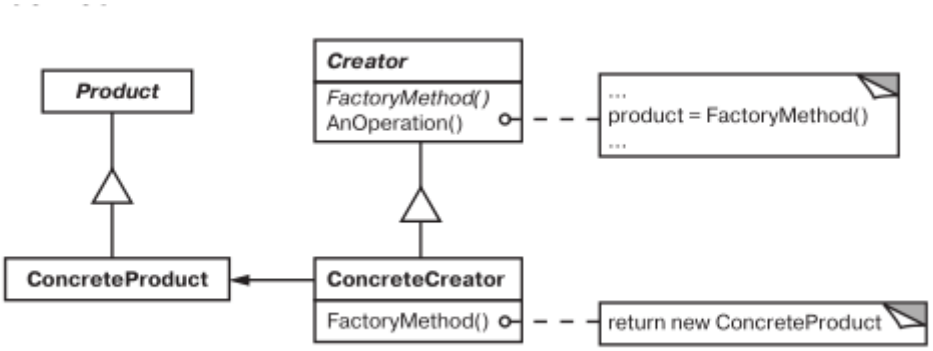
Достоинства, особенности:

- Открытая расширяемость при неизменном поведении клиентского кода.
- Клиентский код не знает ничего о классах конкретной фабрики и конкретных продуктах, код развязывается по зависимостям.
- Конкретный объект фабричного класса обычно инстациируется в единственном экземпляре, при этом часто используется паттерн Factory Method (см. ниже). В то же время возможно использование паттерна Singleton.
- Право инстанциации конкретных классов продуктов остаётся за клиентским кодом — создаётся только то, что требуется.

Недостатки:

- Жёсткий нерасширяемый фабричный интерфейс и интерфейс продуктов. При модификации интерфейса фабрики или продуктов изменения потребуется вносить во все, возможно, многочисленные, реализации.

Factory Method // Фабричный метод // уровень класса



Product — интерфейс объектов, создаваемых фабричным методом.

ConcreteProduct — конкретный продукт, имплементирует интерфейс Product.

Creator — абстрактный создатель (чаще всего каркас), объявляет *Абстрактный метод*, возвращающий объект, реализующий тип Product, вызывает *Фабричный метод*, реализованный в потомках для создания объекта, имплементирующего интерфейс Product.

ConcreteCreator — конкретный создатель, замещает *Фабричный метод*, возвращающий объект.

Создатель «полагается» на свои подклассы в определении *Фабричного метода*, который будет возвращать экземпляр подходящего конкретного продукта. Паттерн избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому может работать с любыми определёнными пользователями классов конкретных продуктов.

Основная мысль такова: ConcreteCreator (реже — неабстрактный Creator с реальным методом) инстанциирует объекты классов, реализующих известный клиентскому коду интерфейс, не завязывая клиентский код на эти конкретные реализации.

Пример использования паттернов *Абстрактная фабрика* и *Фабричный метод*

Проектируем систему, планирующую закупки у расширяющегося (и сужающегося) списка поставщиков, плюс документооборот (накладные), плюс отчёты для маркетинговых мероприятий поставщиков. Построить всех поставщиков под одну гребёнку вряд ли получится, а вот инкапсулировать логику их разнообразных информационных систем можно, применив два рассмотренных паттерна.

Определяем интерфейсы, описывающие функционал информационного обмена с поставщиками.

Очевидно, что поставщик обычно предоставляет цену на товары:

```
import abc

class PriceProvider(metaclass=abc.ABCMeta):
```

```
    @abc.abstractmethod
    def get_price(self, article):
        pass
```

и некоторый документооборот.

```
class DocProvider(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def get_doc(self, id):
        pass

    @abc.abstractmethod
    def send_payment(self, payment):
        pass
```

А также отбирает данные о продажах и выплачивает бонусы особо талантливым реселлерам.

```
class MarketingProvider(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def claim_sales(self):
        pass

    @abc.abstractmethod
    def get_bonus(self):
        pass
```

Эти три интерфейса описывают интерфейс продуктов (`AbstractProduct` на диаграмме классов).

Кроме того, нам потребуется описать интерфейс *Абстрактной фабрики*:

```
class ExchangeFactory(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def create__price_provider(self):
        pass

    @abc.abstractmethod
    def create__doc_provider(self):
        pass

    @abc.abstractmethod
    def create__marketing_provider(self):
        pass
```

Различные реализации интерфейсов конкретными поставщиками учитывают их специфику. При рассмотрении паттернов это второстепенная деталь. Важно, что они имплементируют общее, заданное заранее поведение, но каждый по-своему. Пример реализации одного из провайдеров:

```
class CitilinkPriceProvider(PriceProvider):
    ...
    def get_price(self, article):
        return self.catalog.find_by_article(article).get_price()
```

Имплементация *Конкретной фабрики* очевидна:

```
class CitilinkExchangeFactory(ExchangeFactory):
    def create_price_provider(self):
        return CitilinkPriceProvider()

    def create_doc_provider(self):
        return CitilinkDocProvider()

    def create_marketing_provider(self):
        return CitilinkMarketingProvider()
```

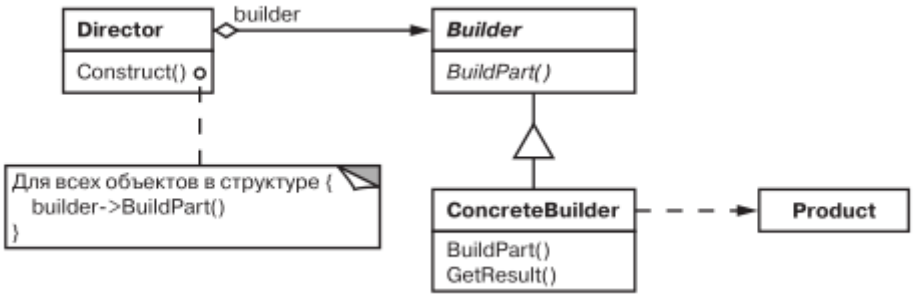
В игру вступает *Фабричный метод*. Ответственность класса Fabric — знать о конкретных реализациях *Абстрактной фабрики*, задача метода create_factory(...) — вернуть конкретную реализацию конкретной фабрики на основе внешней конфигурационной информации, например строки.

```
class Fabric:
    SUPPLIER_ONE = 'Citilink'
    SUPPLIER_TWO = 'Ulmart'

    """создать объект, реализующий интерфейс на основе внешней информации"""
    @staticmethod
    def create_factory(name):
        if name == __class__.SUPPLIER_ONE:
            return CitilinkExchangeFactory()
        elif name == __class__.SUPPLIER_TWO:
            return UlmartExchangeFactory()
        else:
            return None
```

Сводим воедино. Клиентский код:

```
def get_supplier_price(supplier_name, article):
    """создать Абстрактную фабрику сервисов конкретного поставщика"""
    exchange_factory = Fabric.create_factory(supplier_name)

    """создать Фабрикой конкретного поставщика его провайдер услуг"""
    price_provider = exchange_factory.create_price_provider()

    """получить цену"""
    price = price_provider.get_price(article)

    return price
```

Builder // Строитель



Паттерн *Строитель* даёт гибкость при построении сложных объектов, когда заранее неизвестны возможные опции построения, которые могут быть расширены без переделки кода собственно построителя.

Сегодня на практике чаще используется упрощённая схема: интерфейс *Builder* становится реальным классом, а вариации ConcreteBuilder реализованы как множество функций, принимающих аргументы для «постройки» сложного продукта и возвращающих указатель на «строителя». Построение происходит при вызове цепочки методов, крайний из которых возвращает нужный клиенту продукт стройки. Паттерн часто используется в современном коде, и его легко узнать по цепочке вызовов.

Пример. Создание сообщения электронной почты. Есть обязательные опции (кому) и множество факультативных, использование которых, скорее всего, будет различным в каждой конкретной ситуации.

```
"""построитель сообщения электронной почты"""
class MimeMessageBuilder:
    def __init__(self, session):
        self.message = MimeMessage(session)

    def from_addr(self, address):
        self.message._from_addr = address
        return self

    def to_addr(self, address):
        self.message._to_addr = address
        return self

    def cc_addr(self, address):
        self.message._cc_addr = address
        return self

    def subject(self, subject):
        self.message._subject = subject
        return self

    def body(self, body):
        self.message._body = body
        return self

    def build(self):
        return self.message
```

Клиентский код:

```
class Client:
    def send_mail(session):
        message = MimeMessageBuilder(session).\
            from_addr('me').to_addr('you').cc_addr('someone').\
            subject('test').body('hello').\
            build()
```

В этом примере Director реализован в клиентском методе send_mail() в виде цепочки методов.

Рассмотрим второй вариант реализации Builder на примере создания столов.

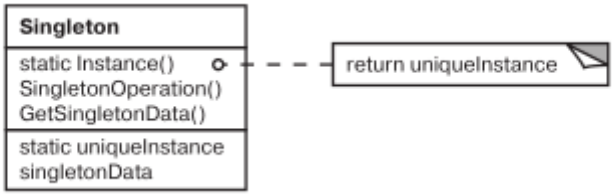
```
import abc


class TableDirector:
    def __init__(self):
        self._builder = None

    def construct(self, builder):
        self._builder = builder
```

```
        self._builder._build_tabletop()
        self._builder._build_legs()
        self._builder._build_coverage()


class Table:
    tabletop = 0
    legs = 0
    coverage = ''


class AbstractTableBuilder(metaclass=abc.ABCMeta):
    def __init__(self):
        self.product = Table()

    @abc.abstractmethod
    def _build_tabletop(self):
        pass

    @abc.abstractmethod
    def _build_legs(self):
        pass

    @abc.abstractmethod
    def _build_coverage(self):
        pass


class BigTableBuilder(AbstractTableBuilder):
    def _build_tabletop(self):
        self.product.tabletop = 120

    def _build_legs(self):
        self.product.legs = 4

    def _build_coverage(self):
        self.product.coverage = 'vanish'


class SmallTableBuilder(AbstractTableBuilder):
    def _build_tabletop(self):
        self.product.tabletop = 80

    def _build_legs(self):
        self.product.legs = 3

    def _build_coverage(self):
        self.product.coverage = 'yacht lacquer'
```

Клиентский код:

```
big_table__builder = BigTableBuilder()
small_table__builder = SmallTableBuilder()

director = TableDirector()
director.construct(big_table__builder)
director.construct(small_table__builder)

big_table_1 = big_table__builder.product
small_table_1 = small_table__builder.product
```

Singleton // Одиночка



Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Класс *Одиночка* объявляет свой конструктор приватным. С одной стороны, это не позволяет клиентскому коду самовольно порождать инстансы *Одиночки*, с другой — предоставляет статический метод getInstance, возвращающий инстанс объекта *Одиночки*, инстанциация которого полностью контролируется классом-*Одиночкой*.

Преимущества:

- Только один объект этого класса.
- Глобальная область видимости, аналог глобальной переменной.
- В зависимости от реализации инстанциация «тяжёлого» объекта может быть отложена по времени до момента первого использования — Lazy initialization.

Недостаток:

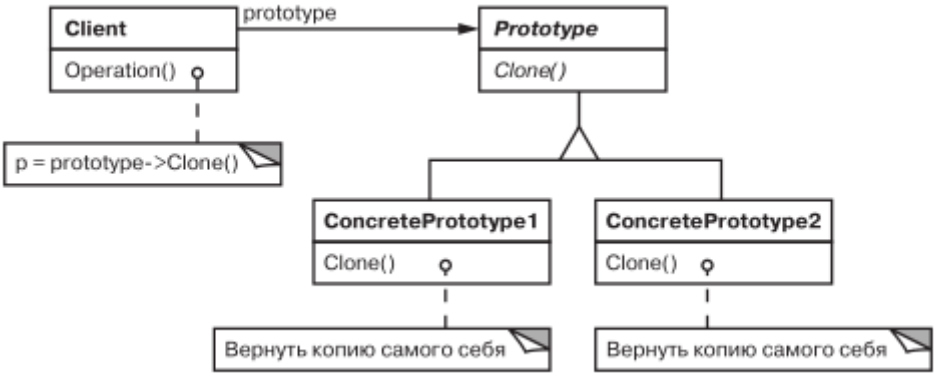
- Сложность ***правильной*** потокобезопасной реализации в многопоточном приложении.

Пример:

```
class Singleton(type):
   def __init__(cls, name, bases, attrs, **kwargs):
       super().__init__(name, bases, attrs)
       cls.__instance = None

   def __call__(cls, *args, **kwargs):
       if cls.__instance is None:
           cls.__instance = super().__call__(*args, **kwargs)
       return cls.__instance


class MySqlConnection(metaclass=Singleton):
   pass


sql_connection_1 = MySqlConnection()
sql_connection_2 = MySqlConnection()

print(sql_connection_1 is sql_connection_2)
```

Здесь мы гарантированно получаем одно соединение с БД.

Prototype // Прототип



Задаёт виды создаваемых объектов с помощью экземпляра Прототипа и новые объекты путём копирования этого Прототипа.

Проявление этого паттерна в реальной жизни — создание копии какого-либо объекта на основе имеющегося экземпляра. Например, в «1С» есть часто используемая операция:

«Создать новый элемент копированием».

При реализации этого паттерна для базового класса некоторой иерархии классов определяется операция создания новой копии объекта путём копирования — метод `copy.deepcopy()`, который определён в базовом модуле `copy` языка Python.

Пример:

```
import copy


class Original:
   pass


original = Original()
prototype = copy.deepcopy(original)
```

Можно приблизиться к более классическому виду паттерна Prototype, используя механизм наследования:

```
import copy


class PrototypeMixin:
    # прототип

    def clone(self):
        return copy.deepcopy(self)
```

В этом случае объект класса, унаследованного от PrototypeMixin, сможет копировать сам себя.

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Выбирать подходящий порождающий шаблон.
2. Применять порождающие шаблоны в своём коде.

Зачем:

Для использования порождающих шаблонов в своём коде.

Последовательность действий:

1. На базе нашего WSGI-фреймворка мы начинаем делать обучающий сайт, чтобы на нём отработать навыки применения шаблонов проектирования.
2. Тема (чему мы будем обучать) может быть любая, что вам больше нравится (например: горные лыжи, йога, администрирование, фридайвинг, продажи, …).
3. Минимальное описание работы сайта следующее:
    a. На сайте есть курсы по обучению чему-либо. Курс относится к какой-либо категории. Например, есть курсы обучения программированию на Python, Java, JavaScript. И курсы Python для новичков, Java для профи, …
    b. Также на сайте есть студенты, которые могут записаться на один или несколько курсов.

4. Это минимальный функционал, на котором мы будем отрабатывать шаблоны, можно будет его расширить.
5. В домашнем задании требуется добавить следующий функционал:
    a. Создание категории курсов.
    b. Вывод списка категорий.
    c. Создание курса.
    d. Вывод списка курсов.
6. Далее можно сделать всё или одно на выбор, применив при этом один из порождающих паттернов, либо аргументировать, почему паттерны не были использованы:
    a. На сайте могут быть курсы разных видов: офлайн (вживую), для них указывается адрес проведения, и онлайн (вебинары), для них указывается вебинарная система. Также известно, что в будущем могут добавиться новые виды курсов.
    b. Реализовать простой логгер (не используя сторонние библиотеки). У логгера есть имя. Логгер с одним и тем же именем пишет данные в один и тот же файл, а с другим именем — в другой.
    c. Реализовать страницу для копирования уже существующего курса, чтобы не создавать его снова с нуля, а скопировать существующий и немного отредактировать.

Дополнительные материалы

1. Creational patterns.
2. Порождающие паттерны проектирования.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Александер К. и др. Язык шаблонов. Города. Здания. Строительство. М.: Студия Артемия Лебедева, 2014.
2. Порождающие шаблоны проектирования.
3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2015.
4. Pattern-Oriented Software Architecture. Volume 2: Patterns for Concurrent and Networked Objects. Volume 2 Edition, Willy, 2000.
5. Саммерфилд М. Python на практике.
6. Лутц М. Изучаем Python.