*Урок 4. Порождающие паттерны*

Оглавление

[Введение 2](#_Toc90142663)

[Каталоги паттернов 2](#_Toc90142664)

[Каталог банды четырех 3](#_Toc90142665)

[Фабричный метод 3](#_Toc90142666)

[Абстрактная фабрика 7](#_Toc90142667)

[Прототип 12](#_Toc90142668)

[Строитель 14](#_Toc90142669)

# Введение

Мы разберем фундаментальное понятие паттерна и рассмотрим особенности порождающих паттернов.

Что же такое паттерны проектирования?

*«Это подходы к решению практических задач, выявленные при анализе полученных решений и применяемые многократно».*

По сути, это шаблоны решения некоторых типовых задач. На словах понятно, а на деле (в программировании) есть нюансы. Задачи с помощью паттернов вроде и решаются, но они весьма абстрактны.

Программа, это все же нечто уникальное, далеко не всегда шаблонное, и говорить, что паттерн – стопроцентное решение задачи, не приходится. Нельзя взять паттерн, просто скопировать и вставить.

Поэтому нужно знать досконально особенности каждого паттерна. Все паттерны строятся на основании анализа большого опыта большого количества разработчиков.

# Каталоги паттернов

Переходим к каталогам паттернов.

1. Самый популярный каталог – GoF (‘банда четырех’). Здесь акцент больше на код, на подходы описания объектов (ООП).
2. Также существуют паттерны Фаулера. Здесь акцент на разработку большой бизнес-системы.
3. Каталог Concurent etc. Паттерны, больше относящиеся к асинхронным системам. Т.е. когда у нас есть несколько потоков, процессов, программ. Т.е. когда есть некоторая конкуренция за ресурсы. Пример паттерна – «Очередь задач» (сообщений).

# Каталог банды четырех

Эти паттерны ориентированы на создание гибкой системы.

Мы их будем изучать три урока: порождающие (отвечают за создание объектов), структурные (как написаны классы и как из них формируются более сложные системы), поведенческие (как эти классы ведут себя в различных ситуациях – через свои объекты).

Они абстрагируют процесс инстанцирования объектов (создания объектов классов), чтобы сделать систему более независимой от способов создания и представления объектов.

То есть, когда мы создаем объект, это вносит дополнительные зависимости, **зависимости от процесса создания объекта (инстанцирования)**. Задача порождающих паттернов – избавиться от зависимостей так, чтобы система была более абстрактной и более гибкой.

Фабричный метод

*«Определяет интерфейс для создания объектов, при этом выбранный класс инстанцируется подклассами»*

**Листинг 1. Урок 4. Коды к уроку/factory\_method\_1.py**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Animal(ABC):   @abstractmethod  **def** say(self):  **pass   class** Dog(Animal):   **def** say(self):  print(**'wow-wow'**)   **class** Cat(Animal):   **def** say(self):  print(**'мяу-мяу'**) |

У нас есть некоторое абстрактное животное – класс **Animal** и оно умеет говорить. У нас есть собака, она является животным и умеет говорить. Есть кошка, тоже животное и умеет говорить. Итог, у нас есть базовый класс и наследники.

Мы хотим использовать все это в своей программе.

**Листинг 2. Урок 4. Коды к уроку/use\_factory\_method\_1.py**

|  |
| --- |
| **from** factory\_method\_1 **import** Cat, Dog  animal\_type = input()  **if** animal\_type == **'dog'**:  animal = Dog() **elif** animal\_type == **'cat'**:  animal = Cat()  animal.say() |

Мы хотим узнать у пользователя, какой вид животного создать. В зависимости от ответа пользователя, создаем животное, которое у нас будет что-то говорить.

Вроде все хорошо, но нехорошо. Когда мы создаем, например, экземпляр собаки, то привязываемся к соответствующему классу. Т.е. у нас есть жесткая зависимость от типа животного. И если потребуется добавить новое животное, его придется описать классом в файле **factory\_method\_1.py,** а затем импортировать и добавить соответствующую ветвь в файл **use\_factory\_method\_1.py**.

Как избавиться от существующей зависимости? Здесь нам поможет паттерн «Фабричный метод».

Цель – есть некоторая иерархия объектов и мы хотим не зависеть от конкретики в клиентском коде.

Нам предлагается создание объекта животного перенести в отдельный класс. И код, который отвечает за создание животного мы, по сути, перекидываем в этот метод.

**Листинг 3. Урок 4. Коды к уроку/factory\_method\_2.py**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Animal(ABC):   @abstractmethod  **def** say(self):  **pass   class** Dog(Animal):   **def** say(self):  print(**'wow-wow'**)   **class** Cat(Animal):   **def** say(self):  print(**'мяу-мяу'**)   **class** Bird(Animal):   **def** say(self):  print(**'мяу-мяу'**)   **class** AnimalCreator:   @staticmethod  **def** create\_animal(animal\_type):  **if** animal\_type == **'dog'**:  animal = Dog()  **elif** animal\_type == **'cat'**:  animal = Cat()  **elif** animal\_type == **'bird'**:  animal = Bird()   **return** animal |

Теперь посмотрим, как это будет работать.

**Листинг 4. Урок 4. Коды к уроку/use\_factory\_method\_1.py**

|  |
| --- |
| **from** factory\_method\_2 **import** AnimalCreator  animal\_type = input() animal = AnimalCreator.create\_animal(animal\_type) animal.say() |

У нас появился дополнительный уровень абстракции и код стал более гибким. Теперь модулю **use\_factory\_method\_2.py** вообще не нужно знать про типы животных. Он знает только про **AnimalCreator**. А **AnimalCreator** где-то внутри себя знает, какого типа животное нужно создать.

Здесь мы отрабатываем и принцип инверсии зависимостей. Все у нас теперь зависит от абстракции, а не от конкретики.

Попытаемся улучшить код, используя возможности фабричного метода на полную. Смысл в том, что мы вытащим метод **create\_animal()** и положим его в абстрактный класс. Теперь **Animal** создает своих потомков сам.

**Листинг 5. Урок 4. Коды к уроку/factory\_method\_3.py**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Animal(ABC):   @abstractmethod  **def** say(self):  **pass** @staticmethod  **def** create\_animal(animal\_type):  **if** animal\_type == **'dog'**:  animal = Dog()  **elif** animal\_type == **'cat'**:  animal = Cat()  **return** animal   **class** Dog(Animal):   **def** say(self):  print(**'wow-wow'**)   **class** Cat(Animal):   **def** say(self):  print(**'мяу-мяу'**) |

Теперь посмотрим, как это будет работать.

**Листинг 6. Урок 4. Коды к уроку/use\_factory\_method\_3.py**

|  |
| --- |
| **from** factory\_method\_3 **import** Animal  animal\_type = input() animal = Animal.create\_animal(animal\_type) animal.say() |

А теперь сделаем решение еще более изящным.

**Листинг 7. Урок 4. Коды к уроку/factory\_method\_4.py**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   **class** Animal(ABC):   @abstractmethod  **def** say(self):  **pass** @staticmethod  **def** create\_animal(animal\_type):  ANIMALS = {  **'dog'**: Dog,  **'cat'**: Cat,  **'bear'**: Bear  }  **return** ANIMALS[animal\_type]()   **class** Dog(Animal):   **def** say(self):  print(**'wow-wow'**)   **class** Cat(Animal):   **def** say(self):  print(**'мяу-мяу'**)   **class** Bear(Animal):   **def** say(self):  print(**'мяу-мяу'**) |

Теперь посмотрим, как это будет работать.

**Листинг 8. Урок 4. Коды к уроку/use\_factory\_method\_4.py**

|  |
| --- |
| **from** factory\_method\_4 **import** Animal  animal\_type = input() animal = Animal.create\_animal(animal\_type) animal.say() |

Абстрактная фабрика

*«Предоставляет интерфейс для создания семейств объектов, конкретные классы которых неизвестны»*

Цель – есть семейство объектов, и мы хотим от него не зависеть.

Если в фабричном методе у нас была некоторая иерархия – класс животное (абстрактный) и классы-реализации видов животных, то в абстрактной фабрике предполагается, что есть некоторое семейство объектов, в которых есть много классов.

Семейство объектов – это много разных классов, но на одну и ту же тему.

Например, у нас есть программа, работающая с социальной сетью ВК. У нас есть классы **VkParser, VkAnalizer, VkSender**.

Теперь мы решили сделать то же самое, например, для одноклассников. Делаем похожее семейство классов: **OdParser, OdAnalizer, OdSender**.

Предполагаем, что будем делать то же самое и для других социалок. В итоге может получиться сотни классов. При этом всю логику будем строить на ветвлениях (if-else) и все это будет зависеть от семейств. Т.е. опять зависим от конкретики. Как уйти от конкретики?

**Листинг 9. Урок 4. Коды к уроку/abc\_factory\_1.py**

|  |
| --- |
| *# Семейство классов для VK* **class** VkParser:  **pass   class** VkAnalizer:  **pass   class** VkSender:  **pass** *# Семейство классов для одноклассников* **class** OdParser:  **pass   class** OdAnalizer:  **pass   class** OdSender:  **pass** |

В чем же смысл паттерна? Мы создаем для каждого семейства (Одноклассники, Вконтакте и т.д.) свою фабрику. Фабрика занимается тем, что создает нужные экземпляры классов (Парсер, Сендер и т.д.).

А также еще сделаем абстрактный класс-фабрику, чтобы знать, какие должны быть прописаны методы. Эта фабрика будет родителем для фабрик-наследников.

**Листинг 10. Урок 4. Коды к уроку/abc\_factory\_2.py**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   *# Семейство классов для VK* **class** VkParser:  **pass   class** VkAnalizer:  **pass   class** VkSender:  **pass** *# Семейство классов для одноклассников* **class** OdParser:  **pass   class** OdAnalizer:  **pass   class** OdSender:  **pass   class** AbstractFactory(ABC):  @abstractmethod  **def** create\_parser(self):  **pass** @abstractmethod  **def** create\_analizer(self):  **pass** @abstractmethod  **def** create\_sender(self):  **pass   class** VkFactory(AbstractFactory):  **def** create\_parser(self):  **return** VkParser()   **def** create\_analizer(self):  **return** VkAnalizer()   **def** create\_sender(self):  **return** VkSender()   **class** OdFactory(AbstractFactory):  **def** create\_parser(self):  **return** OdParser()   **def** create\_analizer(self):  **return** OdAnalizer()   **def** create\_sender(self):  **return** OdSender() |

Сам паттерн закончен, но его пока неудобно использовать – нам нужна или одна фабрика, или другая.

Поэтому напишем еще фабричный метод, который позволит понять, какую фабрику создавать. Мы его пропишем в абстрактном классе. Он будет зависеть от типа семейства (network\_name). В нем опишем словарь фабрик. Добавим еще метод:

@staticmethod  
**def** create\_factory(network\_name):  
 NETWORKS = {  
 **'Vk'**: VkFactory,  
 **'Od'**: OdFactory,  
 **'Tw'**: TwFactory  
 }  
  
 **return** NETWORKS[network\_name]()

Итоговая реализация паттерна:

**Листинг 11. Урок 4. Коды к уроку/abc\_factory\_3.py**

|  |
| --- |
| **from** abc **import** ABC, abstractmethod   *# Семейство классов для VK* **class** VkParser:  @staticmethod  **def** parse():  print(**'Vk parser work'**)   **class** VkAnalizer:  **pass   class** VkSender:  **pass** *# Семейство классов для одноклассников* **class** OdParser:  @staticmethod  **def** parse():  print(**'Od parser work'**)   **class** OdAnalizer:  **pass   class** OdSender:  **pass** *# Семейство классов для твиттера* **class** TwParser:  @staticmethod  **def** parse():  print(**'Tw parser work'**)   **class** TwAnalizer:  **pass   class** TwSender:  **pass   class** AbstractFactory(ABC):   @staticmethod  **def** create\_factory(network\_name):  NETWORKS = {  **'Vk'**: VkFactory,  **'Od'**: OdFactory,  **'Tw'**: TwFactory  }   **return** NETWORKS[network\_name]()   @abstractmethod  **def** create\_parser(self):  **pass** @abstractmethod  **def** create\_analizer(self):  **pass** @abstractmethod  **def** create\_sender(self):  **pass   class** VkFactory(AbstractFactory):  **def** create\_parser(self):  **return** VkParser()   **def** create\_analizer(self):  **return** VkAnalizer()   **def** create\_sender(self):  **return** VkSender()   **class** OdFactory(AbstractFactory):  **def** create\_parser(self):  **return** OdParser()   **def** create\_analizer(self):  **return** OdAnalizer()   **def** create\_sender(self):  **return** OdSender()   **class** TwFactory(AbstractFactory):  **def** create\_parser(self):  **return** TwParser()   **def** create\_analizer(self):  **return** TwAnalizer()   **def** create\_sender(self):  **return** TwSender() |

Теперь, как это все будет работать!

**Листинг 12. Урок 4. Коды к уроку/use\_abc\_factory\_3.py**

|  |
| --- |
| **from** abc\_factory\_3 **import** AbstractFactory  factory = AbstractFactory.create\_factory(**'Od'**)  parser = factory.create\_parser() analizer = factory.create\_analizer() sender = factory.create\_sender()  parser.parse() |

Мы ушли от конкретики в клиентском коде (последние строки).

Яркий пример из реальной жизни – ORM. Все базы делают примерно одно и то же!

Прототип

*«Описывает виды создаваемых объектов с помощью прототипа и создаёт новые объекты путём копирования».*

Цель – просто копировать объект

Он хорошо работает, когда есть сложный объект. Например, у нас есть урок по математике, мы его описали через класс. Но тут появляется необходимость создать второй урок и многое из того, что было в первом, повторяется. В реальной жизни заново создавать не будем, а скопируем. Получается, на основании одного сложного объекта мы можем создать другие путем копирования.

По сути, есть некоторый большой объект, который уже наполнен данными и нам быстрее его скопировать, нежели создать новый.

Рассмотрим питоновский прототип и классический прототип.

**Листинг 13. Урок 4. Коды к уроку/python\_prototype.py**

|  |
| --- |
| **from** copy **import** deepcopy   **class** Original:  **pass** original = Original() prototype = deepcopy(original)   prototype.name = 2 |

Здесь мы видим модуль **copy**. Он позволяет скопировать любой сложный объект. Обычно используется, если необходимо, например, скопировать список со вложенными списками. По сути, в этом модуле **copy** реализуется паттерн Прототип.

Классический вариант позволяет делать больше, но реализация его такая же.

**Листинг 14. Урок 4. Коды к уроку/classic\_prototype.py**

|  |
| --- |
| **from** copy **import** deepcopy   **class** PrototypeMixin:  *# прототип* **def** clone(self):  **return** deepcopy(self)   **class** Original(PrototypeMixin):  **pass   class** OriginalClass(PrototypeMixin):  **pass** original = Original() original.clone()  original\_2 = OriginalClass() original\_2.clone()   **class** ModernPrototypeMixin(PrototypeMixin):   **def** clone(self):  print(**'что то еще'**)  **return** deepcopy(self)   **class** Original(ModernPrototypeMixin):  **pass** original = Original() original.clone() |

Смысл в том, чтобы объект сам себя копировал.

Строитель

*«Отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных представлений»*

Цель – у нас есть сложный объект. Который мы хотим строить по частям и не зависеть от представлений.

Например, у нас есть дом. Это сложный объект. Сначала мы строим фундамент, затем здание, крышу и т.д.

Теперь мы хотим построить тоже дом, но другой конструкции. Процесс конструирования одинаковый, но части дома различаются.

Мы хотим зафиксировать процесс строительства, а конкретные элементы использовать разные.

Или в другом контексте – фиксируем элементы, а процесс меняется. Либо все это вместе идет.

Цель – избежать дублирования.

У нас будут

1. Класс директор (он контролирует последовательность шагов процесса, т.е. говорит, что сначала строим здание, потом крышу и т.д.)
2. Класс строитель (он знает, как строить конкретную часть).

**Листинг 15. Урок 4. Коды к уроку/builder.py**

|  |
| --- |
| **from** abc **import** ABCMeta, abstractmethod   **class** TableDirector:  **def** \_\_init\_\_(self):  self.\_builder = **None   def** construct(self, builder):  self.\_builder = builder  self.\_builder.\_build\_legs()  self.\_builder.\_build\_tabletop()  self.\_builder.\_build\_coverage()   **class** Table:  tabletop = 0  legs = 0  coverage = **''   class** AbstractTableBuilder(metaclass=ABCMeta):  **def** \_\_init\_\_(self):  self.product = Table()   @abstractmethod  **def** \_build\_tabletop(self):  **pass** @abstractmethod  **def** \_build\_legs(self):  **pass** @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'** big\_table\_\_builder = BigTableBuilder() small\_table\_\_builder = SmallTableBuilder()  *# запуск процесса конструирования* director = TableDirector() director.construct(big\_table\_\_builder) director.construct(small\_table\_\_builder)  *# берем сконструированное изделие* big\_table\_1 = big\_table\_\_builder.product small\_table\_1 = small\_table\_\_builder.product  print(big\_table\_1.coverage) print(small\_table\_1.coverage)  print(big\_table\_1.legs) print(small\_table\_1.legs) |

В этом примере мы будем строить Стол (это класс). Считаем, что это сложный объект, состоящий из частей: столешница, ножки, и покрытие. У нас есть класс **TableDirector**, контролирующий процесс строительства.

У него есть параметр **self.builder = None**В него будет передаваться объект строителя.

Класс директор говорит следующее: сначала строй столешницу (build\_tabletop(), потом ножки и т.д.)

Т.к. строитель сюда заходит любой, мы можем передавать разных строителей (**builder**). А сам процесс (в директоре), он не изменится.

У нас есть абстрактный строитель с тремя методами: строй столешницу (**\_build\_tabletop**

), строй ножки (**\_build\_legs**)

И теперь мы можем создать разных строителей со своими особенностями изготовления каждой части – **BigTableBuilder**, **SmallTableBuilder**.

Далее создаем объекты строителей, потом объект директора и передаем ему разных строителей.

Т.е. каждый строитель построит по-своему, но порядок стройки четко определит директор.

В итоге получится – оба построили столы, но один – большой, а другой – маленький стол.

Что в целом можно изменить? Например, в директоре поменять порядок строительства или добавить еще одного нового строителя.

НО КАК ПОНЯТЬ, что нужен строитель?

Надо смотреть, есть ли у нас сложный объект, который нужно строить по частям, например, сложный расчет, сложный объект с данными, но мы заполняем его по частям.

Синглтон

*«Гарантирует, что класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему»*

Довольно редко используемый.

Смысл – всегда используем один экземпляр класса.

Небольшое предисловие:

**Листинг 16. Урок 4. Коды к уроку/singleton\_1.py**

|  |
| --- |
| **class** Origin:  **pass** o1 = Origin() o2 = Origin() o3 = Origin()  print(o1) print(o2)  print(o1 **is** o2)  a = [] b = a print(a **is** b)  b = a.copy() print(a **is** b) |

Применяется к специфическим объектам, чистой выдумке, например, сокет, логгер, настройки программы.

Рассмотрим реализацию на Python.

**Листинг 17. Урок 4. Коды к уроку/singleton\_2.py**

|  |
| --- |
| **class** Singleton(type):  *#\_\_prepare\_\_  #\_\_new\_\_  #\_\_init\_\_  #\_\_call\_\_* **def** \_\_init\_\_(cls, name, bases, attrs, \*\*kwargs):  *# super() - <super: <class 'Singleton'>, <Singleton object>>* super().\_\_init\_\_(name, bases, attrs)  cls.\_\_instance = **None    def** \_\_call\_\_(cls, \*args, \*\*kwargs):  *# print(cls)  # print(cls.\_\_instance)* **if** cls.\_\_instance **is None**:  *# Через магию super().\_\_call\_\_  # вызывается MySqlConnection.\_\_new\_\_и MySqlConnection.\_\_init\_\_* cls.\_\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)  **return** cls.\_\_instance   **class** MySqlConnection(metaclass=Singleton):  **pass** sql\_connection\_1 = MySqlConnection() sql\_connection\_2 = MySqlConnection() sql\_connection\_3 = MySqlConnection()  print(sql\_connection\_1 **is** sql\_connection\_3)  print(sql\_connection\_1.\_\_class\_\_.\_\_class\_\_)  **""" <class '\_\_main\_\_.MySqlConnection'> None  <class '\_\_main\_\_.MySqlConnection'> <\_\_main\_\_.MySqlConnection object at 0x000000DC41932208> """** |