# Введение

*СЛАЙД 1*

Сегодня на повестке дня порождающие паттерны, они менее абстрактные и изучим больше подходов, которые сможем применить на практике.

*СЛАЙД 2*

Мы разберем фундаментальное понятие паттерна и рассмотрим особенности порождающих паттернов.

*СЛАЙД 3*

Что же такое паттерны проектирования?

*СЛАЙД 4*

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

По сути, это шаблон 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, и говорить, что паттерн – стопроцентное решение задачи, не приходится. Нельзя взять паттерн, просто скопировать и куда-то вставить.

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

*СЛАЙД 5*

Переходим к каталогам паттернов. Для своих типов задач придумали свои паттерны.

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).

Рассмотрим каталоги подробнее.

**Каталог банды четырех**

*СЛАЙД 6*

Эти паттерны ориентированы на создание гибкой системы.

*СЛАЙД 7*

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

*СЛАЙДЫ 8-9*

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

*СЛАЙД 10*

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.

Начнем с паттерна **«Фабричный метод»** (довольно прост и применяется часто), а не «Абстрактная фабрика» (сложнее в использовании и реже применяется).

*СЛАЙД 12*

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

**Листинг 1. factory\_method.py**

|  |
| --- |
| **import** abc   **class** Animal(abc.ABC):   @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. use\_factory\_method.py**

|  |
| --- |
| **from** factory\_method **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. Запустим и получим:



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.py,** а затем импортировать и создать соответствующий объект.

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

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

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. factory\_method\_2.py**

|  |
| --- |
| **import** abc   **class** Animal(abc.ABC):   @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** AnimalCreator:   @staticmethod  **def** create\_animal(animal\_type):  **if** animal\_type == **'dog'**:  animal = Dog()  **elif** animal\_type == **'cat'**:  animal = Cat()   **return** animal |

Теперь посмотрим, как это будет работать.

**Листинг 4. use\_factory\_method\_2.py**

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

Все работает:



Но 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. factory\_method\_3.py**

|  |
| --- |
| **import** abc   **class** Animal(abc.ABC):   @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'**) |

**Листинг 6. use\_factory\_method\_3.py**

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

Все работает:



Рассмотрим еще вариант.

Если в классе один статический метод. В примере это класс AnimalCreator. То это говорит о том, что класс нам в принципе особо и не нужен. Ведь create\_animal() это просто некоторая статическая функция, находящаяся в классе.

**Листинг 7. factory\_method\_4.py**

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

**Листинг 8. use\_factory\_method\_4.py**

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

Все работает:



Но можно сделать еще красивее.

**Листинг 9. factory\_method\_5.py**

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

**Листинг 10. use\_factory\_method\_5.py**

|  |
| --- |
| **from** factory\_method\_5 **import** ANIMALS  animal\_type = input() animal = ANIMALS[animal\_type]() animal.say() |

А теперь сделаем еще красивее.

**Листинг 11. factory\_method\_6.py**

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

**Листинг 12. use\_factory\_method\_6.py**

|  |
| --- |
| **from** factory\_method\_6 **import** Animal  animal\_type = input() animal = Animal.create\_animal(animal\_type) animal.say() |

Теперь поговорим о достаточно сложном, редко применяемом, но красивом паттерне – «Абстрактная фабрика».

*СЛАЙД 11*

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

Его суть – «Предоставляет интерфейс для создания семейств объектов, конкретные классы которых неизвестны»

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 13. abc\_factory.py**

|  |
| --- |
| **import** abc  *# 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 свою фабрику. Фабрика занимается тем, что создает нужные экземпляры классов.

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

**Листинг 14. abc\_factory\_2.py**

|  |
| --- |
| **import** abc  *# 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.ABC):  @abc.abstractmethod  **def** create\_parser(self):  **pass** @abc.abstractmethod  **def** create\_analizer(self):  **pass** @abc.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 сделаем 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:

**def** parse(self):  
 print(**'Od parser work'**)

**Листинг 15. abc\_factory\_3.py**

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

Теперь, как это все будет работать!

**Листинг 16. use\_abc\_factory.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!

А с социалками – не самый яркий пример, ведь возможно одноклассники что-то умеют делать, а ВК – нет.

*СЛАЙД 15*

Самое время поговорить о прототипе!

«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 17. prototype.py**

|  |
| --- |
| **import** copy   **class** Original:   *# \_\_slots\_\_ - будет ли с ним работать deepcopy?* **pass** original = Original() prototype = copy.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 18. classic\_prototype.py**

|  |
| --- |
| **import** copy   **class** PrototypeMixin:  *# прототип* **def** clone(self):  **return** copy.deepcopy(self)   **class** Original(PrototypeMixin):  *# \_\_slots\_\_ - будет ли с ним работать deepcopy?* **pass** original = Original() original.clone()   **class** ModernPrototypeMixin(PrototypeMixin):   **def** clone(self):  print(**'что то еще'**)  **return** copy.deepcopy(self)   **class** Original(ModernPrototypeMixin):  *# \_\_slots\_\_ - будет ли с ним работать deepcopy?* **pass** original = Original() original.clone() |

Смысл в том, чтобы объект сам себя копировал. Мы делаем класс PrototypeMixin. В нем мы копируем сами себя: copy.deepcopy(self)

Чем это полезно? PrototypeMixin – это класс, мы можем добавить в него некоторые свойства, а также переопределить метод clone().

Везде, где родителем будет PrototypeMixin, получится использовать метод clone():

original.clone()

А если захотим сделать модный ModernPrototypeMixin, то можно воспользоваться старым PrototypeMixin и к этому добавить переопределение clone().

Прототип можно добавить как примесь куда угодно и куда мы добавим, сможем сделать потом вызов clone().

Переходим к паттерну «Строитель». Он довольно важен.

*СЛАЙД 13*

*«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).

Есть варианты паттернов с директором и без директора.

**Листинг 19. builder\_1.py**

|  |
| --- |
| **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'** 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  print(big\_table\_1.coverage) print(small\_table\_1.coverage) |

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**Он пока пустой. 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.

Есть еще одна интересная реализация строителя, называемая Паровозом.

**Листинг 20. builder\_2.py**

|  |
| --- |
| **class** MimeMessage:  **def** \_\_init\_\_(self, session):  self.session = session   *# построитель сообщения электронной почты* **class** MimeMessageBuilder:  **def** \_\_init\_\_(self, session):  self.message = MimeMessage(session)   *# def add\_some(self, some):  # self.message.some = some   # def add\_some(self, some):  # self.message.some = some  # return self.message* **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   **class** Client:  **def** send\_mail(session):  message = MimeMessageBuilder(session). \  from\_addr(**'me'**).to\_addr(**'you'**).cc\_addr(**'someone'**). \  subject(**'test'**).body(**'hello'**). \  build()  *# 1 ничего не возвращаем # builder = MimeMessageBuilder(session) # builder.add\_some() # builder.add\_some2() # builder.add\_some3() # builder.message # # 2 возвращаем message # builder = MimeMessageBuilder(session) # message = builder.add\_some() # message = builder.add\_some2() # message = builder.add\_some3() # return self* |

Порядок создания объектов:

from\_addr(**'me'**).to\_addr(**'you'**).cc\_addr(**'someone'**). \  
subject(**'test'**).body(**'hello'**). \  
build()

Порядок можно менять.

В конце поговорим о синглтоне

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

Довольно редко используемый.

Смысл – всегда используем один экземпляр класса.

*СЛАЙД 14*

**Листинг 20. singleton.py**

|  |
| --- |
| **class** Origin:  **pass** o1 = Origin() o2 = Origin() *# либо запрещено, либо возвращает ссылку на уже созданный экземпляр* print(o1 **is** o2)  a = [] b = a print(a **is** b)  b = a.copy() print(a **is** b) |

Пример из реальной жизни – вселенная.

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

Рассмотрим реализацию на Python.

**Листинг 21. singleton\_2.py**

|  |
| --- |
| **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**:  # записали первый полученный объект  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) |