Course Patterns

Table of Content

[1 Lesson1 3](#_Toc143207684)

[1.1 need web server + wsgi connector 4](#_Toc143207685)

[2 Lesson4 Принципы проектировния 4](#_Toc143207686)

[2.1 Методы проектирования 4](#_Toc143207687)

[2.2 Принципы проектирования 5](#_Toc143207688)

[2.3 Принцип KISS 5](#_Toc143207689)

[2.4 Принципы SOLID 5](#_Toc143207690)

[2.5 Принципы GRASP ( Устаревшие) 6](#_Toc143207691)

[3 Паттерны проектирования 6](#_Toc143207692)

[4 Порождающие паттерны (отвечают за создание объектов) 6](#_Toc143207693)

[4.1 Фабричный метод 7](#_Toc143207694)

[4.2 Абстрактная фабрика 8](#_Toc143207695)

[4.3 Прототип 10](#_Toc143207696)

[4.4 Строитель 10](#_Toc143207697)

[4.5 Одиночка 11](#_Toc143207698)

[5 Структурные паттерны 11](#_Toc143207699)

[5.1 Адаптер 12](#_Toc143207700)

[5.2 Декоратор 13](#_Toc143207701)

[5.3 Заместитель (прокси) 13](#_Toc143207702)

[5.4 Фасад 14](#_Toc143207703)

[5.5 Компоновщик 15](#_Toc143207704)

[6 Поведенческие паттерны 15](#_Toc143207705)

[6.1 Цепочка ответсвенности 16](#_Toc143207706)

[6.2 Команда 18](#_Toc143207707)

[6.3 Посредник 18](#_Toc143207708)

[6.4 Наблюдатель 19](#_Toc143207709)

[6.5 Другие поведенческие паттерны 21](#_Toc143207710)

[7 Lesson 7 Архитектурные системные паттерны 25](#_Toc143207711)

[7.1 Объектно – реляционные паттерны 25](#_Toc143207712)

|  |
| --- |
| Lesson1 |
|  |
| need web server + wsgi connectorOption1: uwsgi + ubuntu ( uwsgi only linux) ubuntu : install from Microsoft store > ubuntu 20.4  $ pip install uwsgi  запуск сервера + приложения  $ uwsgi - - http :8000 - -wsgi-file fwsgi.py |
| code fwsgi.py: def application(environ, start\_response):      start\_response('200 OK', [('Content-Type', 'text/html')])      return [b'Hello world from a simple WSGI application!'] |
| Option 2 : wsgiref + windows $ ~~pip install wsgiref~~ ( included in standard Puthon3.x)  run server + app  $ python fwsgi\_2.py |
| code fwsgi\_2.py from wsgiref.simple\_server import make\_server  def application(environ, start\_response):      start\_response('200 OK', [('Content-Type', 'text/html')]) # framework      return [b'Hello world from a simple WSGI application!'] # app  with make\_server('', 8000, application) as httpd: # server      print("Serving on port 8000...")      httpd.serve\_forever() |
| Lesson4 Принципы проектировния |
| Методы проектирования 1. Абстракция.  2. Декомпозиция.  3. Модульность.  4. Модель.  5. Тестирование. |
| Принципы проектированияПринцип DRY Don’t Repeat Yourself — «Не повторяйся»  «Каждая часть знания должна иметь единственное, непротиворечивое и  авторитетное представление в рамках системы». |
| Принцип KISS Keep It Simple Stupid — «Будь проще»  «Не следует множить сущее без необходимости».  «Бритва Оккама» |
| Принципы SOLID <https://habr.com/ru/companies/otus/articles/651753/> Принцип единственной ответственности (The Single Responsibility Principle).  “Класс должен иметь только одну причину для изменения” дословная формулировка.  «Принцип ООП, обозначающий, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс» адаптивная  формулировка.  Смысл - класс должен описывать один объект, а ф-ция делать одно дело. Принцип открытости / закрытости (The Open Closed Principle).  «Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для изменения». Принцип подстановки Барбары Лисков (The Liskov Substitution Principle).  Должна быть возможность заменить тип подтипом  *«Должна быть возможность вместо базового типа подставить любой его подтип».*  Принцип подстановки Лисков гласит: «Объекты в программе должны быть заменяемы экземплярами их подтипов без ущерба корректности работы программы». Принцип разделения интерфейса (The Interface Segregation Principle).  Клиенты не должны зависеть от методов которые не используют Принцип инверсии зависимостей (The Dependency Inversion Principle).  Все части кода должны зависеть от абстркции и не зависеть от конкретики  *Модули верхних уровней не должны зависеть от модулей нижних уровней.*  *Оба типа модулей должны зависеть от абстракций.*  *Абстракции не должны зависеть от деталей.*  *Детали должны зависеть от абстракций.* |
| Принципы GRASP ( Устаревшие) GRASP = General Responsibility Assignment Software Patterns  (Общие шаблоны распределения ответственностей в ПО).  1. Информационный эксперт (Information expert).  2. Создатель (Creator).  3. Контроллер (Controller).  4. Слабая степень связности (Low Coupling).  5. Высокое зацепление (High cohesion).  6. Полиморфизм (Polymorphism).  7. Чистая выдумка (Pure Fabrication).  8. Посредник (Indirection).  9. Устойчивость к изменениям (Protected Variations).  ` |
| Паттерны проектирования *«Это подходы к решению практических задач, выявленные при анализе полученных решений и применяемые многократно».*  GoF (‘банда четырех’)   * 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).  Порождающие паттерны (отвечают за создание объектов) Паттерны создания объектов классов  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.  Цель – есть некоторая иерархия объектов и мы хотим не зависеть от конкретики в клиентском коде.  from abc import ABC, abstractmethod   class Animal(ABC):   @abstractmethod  def say(self):  pass  # Интерфейс для создания объектов  @staticmethod  def create\_animal(animal\_type):  ANIMALS = {  'dog': Dog, # строке ‘dog’ соотв класс Dog   'cat': Cat, , # строке 'cat’ соотв класс Cat   'bear': Bear  }  return ANIMALS[animal\_type]() # Возвращает объект соответств. класса  # из списка, инстанцирует его ()  # Ниже подклассы которые создают обекты животных 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')  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.  (Предметы одинаковые : стул, стол, диван, семейства разные : кухонный набор, для гостинной, для кафе)  или  предметы одинаковые: ( парсер, sender, receiver) для ( Одноклассники, Vk, Facebook)  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()  from abc\_factory\_3 import AbstractFactory  factory = AbstractFactory.create\_factory('Od') # returns class OdFactory  parser = factory.create\_parser() # иниц.class OdFactory.create\_parcer() analizer = factory.create\_analizer() # иниц.метод OdFactory.create\_analizer() sender = factory.create\_sender() # иниц.метод OdFactory.create\_sender()  parser.parse() # |
| Прототип «Описывает виды создаваемых объектов с помощью прототипа и создаёт новые объекты путём копирования».  Цель – просто копировать объект  from copy import deepcopy   class Original:  pass   original = Original() prototype = deepcopy(original)   prototype.name = 2 |
| 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.  есть абстрктный класс продукта с методами AbstractBuilder(metaclass=ABCMeta),  AbstractBuilder.product = class Table()  создаем class (Builder (AbstractBuilder) - продукт без свойств  создаем директор (input: Builder (абстр. продукт)),  директор определяет последовательность вызова методов (присвоения атрибутов) для создания продукта  директор вызывает методы создания конкретного продукта – присваиваем атрибуты  product = builder.product |
| Одиночка Гарантирует, что класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему.  - с помошью metaclass  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() # create 1-st instance sql\_connection\_2 = MySqlConnection() # link t 1-st instance sql\_connection\_3 = MySqlConnection() # link t 1-st instance   print(sql\_connection\_1 is sql\_connection\_3)  print(sql\_connection\_1.\_\_class\_\_.\_\_class\_\_)  """ True  <class '\_\_main\_\_.Singleton'> """ |
| Структурные паттерны определяют как из классов и объектов строятся более сложные системы |
| Адаптер «Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты»  from abc import ABCMeta, abstractmethod from math import sqrt   # нечто круглое, имеющее радиус class Roundable(metaclass=ABCMeta):  @abstractmethod  def get\_radius(self):  pass   # окружность - имеет радиус class Circle(Roundable):  def \_\_init\_\_(self, radius):  self.\_radius = radius   def get\_radius(self):  return self.\_radius   # квадрат со стороной side class Square:  def \_\_init\_\_(self, side):  self.\_side = side   def get\_side(self):  return self.\_side   # адаптер квадрата к круглым фигурам class RoundableAdapter(Roundable):  def \_\_init\_\_(self, adaptee):  self.\_adaptee = adaptee  print(self.\_adaptee)   # радиус квадрата - как радиус описанной окружности  def get\_radius(self):  return self.\_adaptee.get\_side() \* sqrt(2) / 2   # список окружностей и квадратов figures\_1 = [Circle(5), Square(5), Circle(2), Square(2)] |
| Декоратор «Динамически добавляет объекту новые обязанности»  # На функциях def decorator(old\_f):  def inner(\*args, \*\*kwargs):  print('two')  return old\_f(\*args, \*\*kwargs)   return inner   @decorator def old():  print('one')   # old = decorator(old) old() |
| Заместитель (прокси) «Позволяет сослаться на объект более изощрённо, чем это возможно с простым указателем»  пример – кэширование. Мы обращаемся не к объекту, а к его кэшу |
| Фасад «Определяет интерфейс более высокого уровня, который упрощает использование подсистемы»  В Dgango – context  class Site1Checker:  def russian\_auto(self):  print('prices of russian cars on site 1')   def foreign\_auto(self):  print('prices of foreign cars on site 1')   class Site2Checker:  def russian\_auto(self):  print('prices of russian cars on site 2')   def foreign\_auto(self):  print('prices of foreign cars on site 2')   class Site3Checker:  def russian\_auto(self):  print('prices of russian cars on site 2')   def foreign\_auto(self):  print('prices of foreign cars on site 2')   class FacadeSiteChecker:  def \_\_init\_\_(self):  self.\_subsys\_1 = Site1Checker()  self.\_subsys\_2 = Site2Checker()  self.\_subsys\_3 = Site3Checker()   def russian\_auto(self):  self.\_subsys\_1.russian\_auto()  self.\_subsys\_2.russian\_auto()  self.\_subsys\_3.russian\_auto()   def foreign\_auto(self):  self.\_subsys\_1.foreign\_auto()  self.\_subsys\_2.foreign\_auto()  self.\_subsys\_3.russian\_auto()   facade\_site\_checker = FacadeSiteChecker() facade\_site\_checker.russian\_auto() facade\_site\_checker.foreign\_auto() |
| Компоновщик «Компонует объекты в древовидные иерархические структуры для представления иерархий часть — целое»  from abc import ABCMeta, abstractmethod   class Component(metaclass=ABCMeta):  @abstractmethod  def operation(self):  pass   class MachineOperation(Component): # имитация файла  def \_\_init\_\_(self, name):  self.name = name   def operation(self):  print(self.name)   class CompositeOperation(Component): # имитация папки  def \_\_init\_\_(self):  self.\_child = set()   def operation(self):  print('folder')  for child in self.\_child:  child.operation()   def append(self, component):  self.\_child.add(component)   def remove(self, component):  self.\_child.discard(component)   # инициализация операций operation\_1 = MachineOperation('drill 5 mm') operation\_2 = MachineOperation('drill 15 mm') composite\_1 = CompositeOperation() composite\_1.append(operation\_1) composite\_1.append(operation\_2)  operation\_3 = MachineOperation('assemble') operation\_4 = MachineOperation('paint') composite\_2 = CompositeOperation() composite\_2.append(composite\_1) composite\_2.append(operation\_3) composite\_2.append(operation\_4) print(composite\_2.\_child)  # использование разных по структуре операций идентично composite\_2.operation() # operation\_1.operation()  ё |
| Поведенческие паттерны |
| Цепочка ответсвенности chain of responsibility  Дать шанс обработать запрос нескольким участникам, связанным последовательно.  from abc import ABCMeta, abstractmethod from random import choice, random   class Handler(metaclass=ABCMeta):  @abstractmethod  def handle(self, request):  if self.next is not None:  self.next.handle(request)   def link(self, next):  self.next = next  return self.next   class Request:  data = [  'вопрос по возврату товара',  'вопрос по скидке',  'вопрос по стоимости товара',  'вопрос по дефекту',  'вопрос по новинке', ]   def get\_data(self):  # return random.sample(\_\_class\_\_.data, 1)[0]  return choice(\_\_class\_\_.data)   class Operator(Handler):  # вероятность занятости оператора  probability = 0.95   def \_\_init\_\_(self, name):  self.name = name   def handle(self, request):  if self.is\_busy():  print(f'Оператор {self.name} занят')  super().handle(request)  else:  print(f'Оператор {self.name} обрабатывает: "{request.get\_data()}"')   def is\_busy(self):  return random() < \_\_class\_\_.probability   class BusyHandler(Handler):  def \_\_init\_\_(self):  self.request = None   def handle(self, request):  if self.request == request:  print('Все операторы заняты, пожалуйста подождите')  else:  self.request = request   super().handle(request) |
| handler = BusyHandler()  handler.link(Operator("#1")). \  link(Operator("#2")). \  link(Operator("#3")). \  link(Operator("#4")). \  link(handler)  # генерируем поток из 3 запросов for \_ in range(1):  handler.handle(Request())  Connected to pydev debugger (build 221.5080.212)  Оператор #1 занят  Оператор #2 занят  Оператор #3 занят  Оператор #4 занят  Все операторы заняты, пожалуйста подождите  Оператор #1 занят  Оператор #2 занят  Оператор #3 занят  Оператор #4 занят  Все операторы заняты, пожалуйста подождите  Оператор #1 занят  Оператор #2 занят  Оператор #3 занят  Оператор #4 занят  Все операторы заняты, пожалуйста подождите  Оператор #1 занят  Оператор #2 занят  Оператор #3 занят  Оператор #4 занят  Все операторы заняты, пожалуйста подождите  Оператор #1 занят  Оператор #2 обрабатывает: "вопрос по возврату товара"  Process finished with exit code 0 |
| Команда Действие как объект. Позволит:  ● Передавать как объект.  ● Логировать действия.  ● Ставить в очередь.  ● Откатывать операции. |
| Посредник Определяет и координирует взаимодействие объектов.  model(Python) <-> ORM(Посредник) <-> DataBase  | |  ---------------- Not interact --------------------  пукупатель <-> риэлтер <-> продавец  | |  ---------------- Not interact -------------------- |
| Наблюдатель Создает механизм оповещения объектов об изменениях в других интересующих их объектах.  (подписка на новости)  from abc import ABCMeta, abstractmethod   class Subject:  def \_\_init\_\_(self):  self.\_observers = set()  self.\_subject\_state = None   def attach(self, observer):  observer.\_subject = self  self.\_observers.add(observer)   def detach(self, observer):  observer.\_subject = None  self.\_observers.discard(observer)   def \_notify(self):  for observer in self.\_observers:  observer.update(self.\_subject\_state)   class Observer(metaclass=ABCMeta):  def \_\_init\_\_(self):  self.\_subject = None  self.\_observer\_state = None   @abstractmethod  def update(self, arg):  pass   class Sensor(Subject):  @property  def t(self):  return self.\_subject\_state   @t.setter  def t(self, t):  self.\_subject\_state = t  self.\_notify()    |  |  \ / |
| class SmsNotifier(Observer):   def update(self, arg):  if arg > 50:  print('send sms', 'куда так горячо!')   class DisplayObserver(Observer):  def update(self, arg):  print(f'{self.\_\_class\_\_.\_\_name\_\_} temperature {arg}')   class HeaterObserver(Observer):  def \_\_init\_\_(self, low\_threshold, step):  super().\_\_init\_\_()  self.low\_threshold = low\_threshold  self.step = step   def update(self, arg):  if isinstance(self.\_subject, Sensor):  sensor = self.\_subject   t = sensor.t  delta\_low = t - self.low\_threshold   if delta\_low < 0:  t += self.step  print(f'{self.\_\_class\_\_.\_\_name\_\_} heat impulse +{self.step}')  sensor.t = t   # сенсором sensor = Sensor()  # подключаем наблюдателей за сенсором sensor.attach(DisplayObserver()) sensor.attach(HeaterObserver(40, 20)) sensor.attach(SmsNotifier())  # изменяем состояние сенсора sensor.t = 20  DisplayObserver temperature 20  HeaterObserver heat impulse +20  DisplayObserver temperature 40 |
| Другие поведенческие паттерны● Итератор. Предоставляет способ последовательного доступа ко всем элементам составного объекта, не  раскрывая его внутреннего представления.  for el in range ● Интерпретатор. Задаёт язык, определяет представление его грамматики, а также интерпретатор предложений  этого языка. ● Хранитель. Сохраняет и восстанавливает состояние объекта, не нарушая его инкапсуляции.  JSON, PICLE, CSV ● Состояние. Позволяет объекту варьировать своё поведение в зависимости от внутреннего состояния.  if … do … ● Стратегия. Позволяет использовать взаимозаменяемые алгоритмы.  Оплата (картой, биткоином, наличные, переводом….)  from abc import ABCMeta, abstractmethod   class PaymentStrategy(metaclass=ABCMeta):  @abstractmethod  def pay(self, amount):  pass   class PayPalPaymentStrategy(PaymentStrategy):  # требуем учетку от paypal  def \_\_init\_\_(self, email, token):  self.email = email  self.token = token   def pay(self, amount):  print(f'processing {amount} via PayPal account {self.email}')   class CreditCard:  def \_\_init\_\_(self, number):  self.\_number = number   def get\_number(self):  return self.\_number |
| class CreditCardPaymentStrategy(PaymentStrategy):  # требуется кредитка  def \_\_init\_\_(self, card):  self.card = card   def pay(self, amount):  print(f'processing {amount} via credit card {self.card.get\_number()}')   class Item:  def \_\_init\_\_(self, name, price):  self.name = name  self.price = price   class Order:  def \_\_init\_\_(self):  self.\_items = []   def pay(self, strategy):  total = self.get\_total()  strategy.pay(total)   def get\_total(self):  total = 0  for \_item in self.\_items:  total += \_item.price   return total   def add\_item(self, item):  self.\_items.append(item)   # товары item1 = Item("Book", 515) item2 = Item("Magazine", 298)  # создаём и наполняем заказ order = Order() order.add\_item(item1) order.add\_item(item2)  # выбор конкретной стратегии и оплата заказа paypal\_payment\_strategy = PayPalPaymentStrategy("patterns@geekbrains.com", "token") order.pay(paypal\_payment\_strategy)  # выбор конкретной стратегии и оплата заказа credit\_card = CreditCard("1234 5678 9101 2131 4156") credit\_card\_payment\_strategy = CreditCardPaymentStrategy(credit\_card) order.pay(credit\_card\_payment\_strategy)  или логгирование (в файл, в soket, в терминал) |
| ● Шаблонный метод. Определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги  алгоритма, не изменяя его структуру в целом.  DJANGO – Class Based Views:  UpdateView, DeleteView, ListView….  GetContextData, GetQuerySet …  Классы созданы, при желании можно доработать или переопределить  # from django.views.generic import ListView # # # class MyListView(ListView): # # def get\_queryset(self): # return ... # # def get\_context\_data(self, \*, object\_list=None, \*\*kwargs): # pass |
| from abc import ABCMeta, abstractmethod   class Notifier(metaclass=ABCMeta):  def \_\_init\_\_(self):  self.\_log\_list = []   def notify(self, address, subject, message):  self.\_login()  self.\_send(address, subject, message)  self.\_logout()  self.\_log(address, subject, message)   # войти в ситему доставки сообщений  @abstractmethod  def \_login(self):  pass   # отправка сообщения  @abstractmethod  def \_send(self, address, subject, message):  pass   # выход  @abstractmethod  def \_logout(self):  pass   # внутреннее логирование, задаем поведение по умолчанию  def \_log(self, address, subject, message):  self.\_log\_list.append([address, subject, message])   class EmailNotifier(Notifier):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.mail\_from = ''   def \_login(self):  # no need to login  pass   def \_send(self, mail\_to, subject, message):  # send\_mail(self.mail\_from, mail\_to, subject, message)  print(f'send\_mail: {mail\_to}, {subject}, {message}')   def \_logout(self):  # no need to logout  pass |
| class FacebookNotifier(Notifier):  def \_login(self):  # login to facebook  print('login to facebook')   def \_send(self, address, subject, message):  # send facebook message  print(f'send facebook message: {address}, {subject}, {message}')   def \_logout(self):  # logout from facebook  print('logout from facebook')   # переопределяем поведение шага внутреннего логирования  def \_log(self, address, subject, message):  # не будем внутренне логировать нотификацию по FB, ибо это избыточно.  pass   class NotifierFabric:  @staticmethod  def get\_notifier(communication\_type):  if communication\_type == 'EMAIL':  return EmailNotifier()  elif communication\_type == 'FACEBOOK':  return FacebookNotifier()  # ...   # инстанциируем объект конкретного нотификатора, используя паттерн Фабричный метод notifier\_1 = NotifierFabric.get\_notifier('EMAIL') notifier\_1.notify('patterns@geekbrains.ru', 'notify\_1', 'hello world') print(notifier\_1.\_\_dict\_\_)  # инстанциируем объект конкретного нотификатора notifier\_2 = NotifierFabric.get\_notifier('FACEBOOK') notifier\_2.notify('patterns\_facebook', 'notify\_2', 'hi') print(notifier\_2.\_\_dict\_\_) |
| ● Посетитель.  Позволяет определить новую операцию для иерархии классов, не изменяя сами классы. |
| Lesson 7 Архитектурные системные паттерны |
| Объектно – реляционные паттерныData Mapper Слой преобразователей, котрый передает данные между объектами и базой,  сохраняя последние независимыми друг от друга и от самого преобразователя |
| pros : it works  cons : person\_mapper, UserMapper, course mapper etc  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 = f"SELECT IDPERSON, FIRSTNAME," \              f" 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}'                                            f' not found')      def insert(self, person):          statement = f"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 = f"UPDATE PERSON SET FIRSTNAME=?, " \              f"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 = f"DELETE FROM PERSON WHERE IDPERSON=?"          self.cursor.execute(statement, (person.id\_person,))          try:              self.connection.commit()          except Exception as e:              raise DbDeleteException(e.args)  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 |
| # person = Person(1, 'иван', 'иванов')  connection = connect('patterns.sqlite')  person\_mapper = PersonMapper(connection)  person\_1 = person\_mapper.find\_by\_id(1)  print(person\_1.\_\_dict\_\_)  print(type(person\_1))  person\_1.last\_name = '1233'  person\_mapper.update(person\_1)  print(person\_1.\_\_dict\_\_) |
| Единица работы (Unit of Work): ● Отслеживает изменения данных в доменной модели в рамках  бизнес-транзакции.  ● После закрытия бизнес-транзакции все изменения модели  попадают в БД в виде единой транзакции. |