# Введение

*СЛАЙД 1-3*

Сегодня поговорим об архитектурных системных паттернах.

Это будет вновь отсылка к Мартину Фаулеру. Разберем базовые паттерны и объектно-реляционные.

*СЛАЙД 4-5*

Он написал книгу «Шаблоны корпоративных приложений», в 2002 году, в которой разбирает паттерны, необходимые для разработки бизнес-систем (энтерпрайз-приложений).

Объектно-реляционные паттерны являются отсылкой к реализации ORM, но она весьма сложна и полностью ее мы разобрать не сможем. Поэтому познакомимся с двумя паттернами, которые лежат в основе ORM - data mapper и unity of work.

**БАЗОВЫЕ ПАТТЕРНЫ**

*СЛАЙД 6*

**«Объект-Значение (Value Object)»**

*СЛАЙД 7*

**@Небольшие простые объекты, равенство которых не основано на равенстве идентификаторов@**

Все наверняка слышали, что использование глобальных переменных – это плохо. Потому что эту глобальную переменную можно изменить в разных местах, и мы не можем это контролировать. Глобальные переменные опасны только в том случае, если они изменяются.

*СЛАЙД 8*

То же самое с глобальным списком. Тут лучше взять копию.

А вот глобальные константы допускаются – GLOBAL-CONST. То, что всем доступно, но никто не меняет, то нормально.

Чем объект локальнее, тем лучше. Самое плохое – глобальный изменяемый объект.

Но мы не сможем везде использовать только локальные переменные.

Идея паттерна в том, чтобы мы могли свободно использовать объекты, но сделать их неизменяемыми, потому что с неизменяемыми мы реже можем совершить ошибку.

Еще есть паттерн immutable, который говорит, что неизменяемые объекты проще сравнить.

**Листинг 1. value\_object.py**

|  |
| --- |
| **from** inspect **import** getfullargspec   **class** CannotBeChangeException(Exception):  **def** \_\_init\_\_(self, \*args, \*\*kwargs):  super().\_\_init\_\_(**'You cannot change values, create a new one'**)   **class** ValueObject:   **def** \_\_new\_\_(cls, \*args, \*\*kwargs):  self = super().\_\_new\_\_(cls)  args\_spec = ArgsSpec(self.\_\_init\_\_)  print(args\_spec.args[1:])   **def** assign\_instance\_arguments():  *# print(dict(list(zip(args\_spec.args[1:], args)) + list(kwargs.items())))* self.\_\_dict\_\_.update(  dict(list(zip(args\_spec.args[1:], args)) + list(kwargs.items()))  )   assign\_instance\_arguments()   print(**'created'**)  **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):  *# print('getfullargspec', getfullargspec(method))  # print('getfullargspec 0', getfullargspec(method)[0])* self.\_args = getfullargspec(method)[0]   @property  **def** args(self):  **return** self.\_args   **class** Point(ValueObject):  *# class Point:* **def** \_\_init\_\_(self, x, y):  **pass** print(**'\_\_new\_\_'**) 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* **class** Smile(ValueObject):   **def** \_\_init\_\_(self, name, img):  **pass** fun = Smile(**'fun'**, **':)'**) fun1 = Smile(**'fun'**, **':)'**) sad = Smile(**'sad'**, **':('**)  print(fun == fun1) *# True* print(fun == sad) *# False  # fun.name = 'some' # Exception* print(fun.img) |

В Python 3.7 появилась очень эффективная вещь – DataClass, позволяющая создать класс более быстро.

У нас есть класс Point. Это пример ValueObject.

**class** Point(ValueObject):  
 *# class Point:* **def** \_\_init\_\_(self, x, y):  
 **pass**

Мы можем сделать его неизменяемым. И равенство будет по координатам.

Создаем экземпляры и делаем проверки на: равенство, неравенство и то, что это не один и тот же объект.

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)

При запуске получим:

True

False

False

Но стоит убрать из родителей ValueObject, как результат изменится:

В первом случае будет False.

Почему? Потому что, если мы не определяем метод для сравнения, то операция ==

будет то же самое, что и проверка на is.

Мы хотим сделать класс неизменяемым и добавить сравнение.

Есть интересный момент:

В инициализаторе стоит pass, но вот этот print он срабатывает:

print(point\_1.x)

Есть какая-то магия!

Мы по идее должны эти атрибуты определять в момент создания объекта:

point\_1 = Point(1, 2)

Рассмотрим класс ValueObject!

У него переопределен метод \_\_new\_\_.

И два метода, которые сравнивают атрибуты объектов в словарях:

**def** \_\_eq\_\_(self, other):  
 **return** self.\_\_dict\_\_ == other.\_\_dict\_\_  
  
**def** \_\_ne\_\_(self, other):  
 **return** self.\_\_dict\_\_ != other.\_\_dict\_\_

И есть переопределение:

**def** \_\_setattr\_\_(self, name, value):  
 **raise** CannotBeChangeException

То есть мы не можем изменить объект.

Самая главная магия скрывается в методе \_\_new\_\_

Нужно понимать разницу. Что \_\_init\_\_ - это инициализатор, а \_\_new\_\_ - конструктор.

Когда срабатывает \_\_new\_\_ объекта еще нет, а \_\_init\_\_ - уже есть.

Благодаря перегрузке \_\_new\_\_ мы можем что-то сделать до создания объекта.

Здесь воспользуемся спец. Модулем – inspect

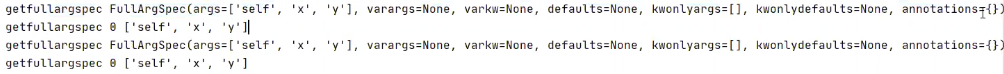
В нем есть спец ф-ция getfullargspec, она позволяет для любой ф-ции определить, какие в нее входят аргументы.

И вот важный момент:

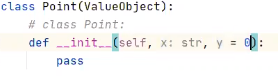
args\_spec = ArgsSpec(self.\_\_init\_\_)

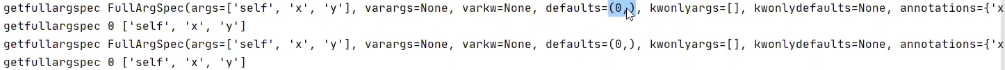
**class** ArgsSpec(object):  
 **def** \_\_init\_\_(self, method):  
 *print('getfullargspec', getfullargspec(method))  
 print('getfullargspec 0', getfullargspec(method)[0])* self.\_args = getfullargspec(method)[0]  
  
 @property  
 **def** args(self):  
 **return** self.\_args

Вот эти принты дают нам такой результат:



Если передадим в инит некоторые kwargs, то получим:







Получаем значения аргументов и собираем из них словарь:

self.\_\_dict\_\_.update(  
 dict(list(zip(args\_spec.args[1:], args)) + list(kwargs.items()))  
)

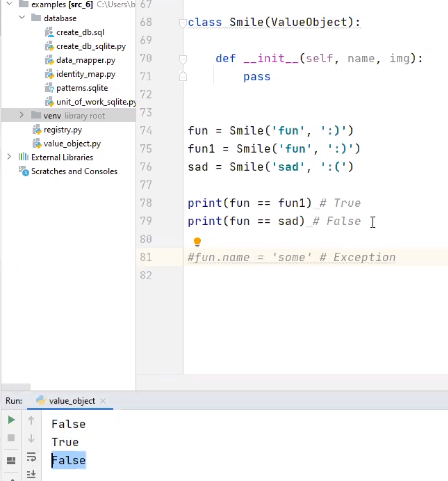
И если запринтить этот объект, то получим:



Получается апдейтим словарь объекта класса тем, что придет сюда:

point\_1 = Point(1, 2)  
point\_2 = Point(y=2, x=1)

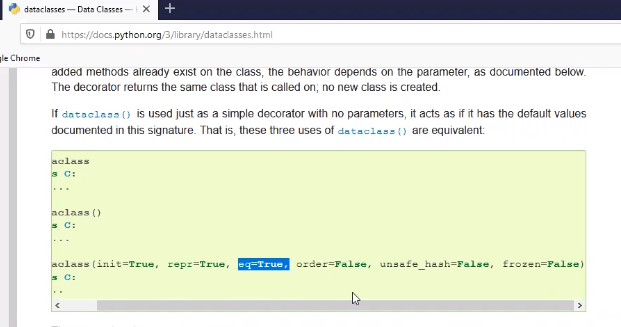
И еще пример:



Переходим к дата-классам.

**Листинг 2. valueobj\_dataclass.py**

|  |
| --- |
| **from** dataclasses **import** dataclass   @dataclass(frozen=**True**) **class** Point:  x: int  y: int   p = Point(1, 2) p1 = Point(1, 2)  print(p == p1) |

Дата-классы позволяют создать класс. При этом сразу добавляется набор методов:

Вместо того, чтобы делать класс классический с инитом, он все за нас сам делает.

Все моменты value\_obect отрабатывают. Но изменить объект нельзя, будет ошибка:

p.x = 1

*СЛАЙД 9-10*

Плюсы и минусы Value Object

**Реестр (Registry)»**

*СЛАЙД 11*

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

Очень простой паттерн.

**Листинг 3. registry\_class.py**

|  |
| --- |
| **class** RegistryHolder(type):  count = -1   **def** \_\_new\_\_(cls, name, bases, attrs):  new\_cls = type.\_\_new\_\_(cls, name, bases, attrs)  cls.count += 1  **return** new\_cls   **class** Animal(metaclass=RegistryHolder):   count = 0   **def** \_\_init\_\_(self):  Animal.count += 1   **class** Bear(Animal):  **pass   class** Cat(Animal):  **pass** animal = Animal() animal1 = Animal() animal2 = Animal()  print(Animal.count)  print(RegistryHolder.count)   **class** Dog(Animal):  **pass** print(RegistryHolder.count) |

У нас есть класс Animal. И в нем нужно посчитать количество животных. Класс будет выполнять роль реестра, где в переменной count мы будем хранить количество животных.

И после каждого создания объекта-животного будем увеличивать значение счетчика на 1.

В примере выше мы используем метакласс для регистрации и хранения информации о классах.

**class** RegistryHolder(type):  
 count = -1  
  
 **def** \_\_new\_\_(cls, name, bases, attrs):  
 new\_cls = type.\_\_new\_\_(cls, name, bases, attrs)  
 cls.count += 1  
 **return** new\_cls

В реестр складываем наш класс:

new\_cls = type.\_\_new\_\_(cls, name, bases, attrs)

Еще пример:

**Листинг 4. registry.py**

|  |
| --- |
| **class** RegistryHolder(type):  REGISTRY = {}  count = 0   **def** \_\_new\_\_(cls, name, bases, attrs):  new\_cls = type.\_\_new\_\_(cls, name, bases, attrs)  cls.REGISTRY[new\_cls.\_\_name\_\_] = new\_cls  cls.count += 1  **return** new\_cls   @classmethod  **def** get\_registry(cls):  **return** dict(cls.REGISTRY)   **class** BaseRegisteredClass(metaclass=RegistryHolder):  **pass** 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] print(RegistryHolder.count) |

**ОБЪЕКТНО-РЕЛЯЦИОННЫЕ ПАТТЕРНЫ**

*СЛАЙД 12*

Постепенно подберемся к тому, как реализовано ORM.

**«Преобразователь данных (Data Mapper)»**

*СЛАЙД 13*

**@Слой преобразователей, который передаёт данные между объектами и базой, сохраняя последние независимыми друг от друга и от самого преобразователя@**

Представим пример, что разработчики делали много-много программ, связывали их с БД, делали запросы и стали думать, как отделить бизнес-логику от БД и как вообще их подружить.

Возникла идея о реализации посредника. Тогда мы сможем отделить модель от запросов в БД.

**Листинг 5. data\_mapper.py**

|  |
| --- |
| **import** sqlite3   **class** RecordNotFoundException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Record not found: {**message**}'**)   **class** DbCommitException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Db commit error: {**message**}'**)   **class** DbUpdateException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Db update error: {**message**}'**)   **class** DbDeleteException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Db delete error: {**message**}'**)   **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, 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 = **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=?, 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   connection = sqlite3.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.name = **'1233'** person\_mapper.update(person\_1) |

Представим, что у нас есть класс Person.

**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

Это наша бизнес-логика. В нашем примере здесь будут курсы, категории и т.д. Мы хотим эти данные хранить в базе.

В чем здесь ключевая идея:

Мы создаем посредника, и он пусть все переносит на структуру БД.

**class** PersonMapper:

Его задача – брать модель и сохранять в БД. И наоборот, брать данные из БД и превращать в объект модели.

В этом классе есть целый набор методов: insert, update, delete

Они принимают объект модели и что-то делают с базой.

Метод find\_by\_id наоборот, опираясь на id, делает запрос в БД и получает объект.

Можно выполнить и изменение в БД.



В чем плюсы подхода?

Теперь модель не знает про базу, а база не знает про модель. Все знает PersonMapper/

Минусы: для каждой модели нужно писать свой Mapper. Или как-то его сгенерировать. Вот это уже будет близко к ORM.

**«Единица работы (Unit of Work)»**

*СЛАЙД 13*

* **@Отслеживает изменения данных в доменной модели в рамках бизнес-транзакции.**
* **После закрытия бизнес-транзакции все изменения модели попадают в БД в виде единой транзакции@**

Теперь нам нужно следить, что в БД мы изменили, например, person\_1



Я изменил какой-то атрибут и мне тут же нужно маппером это зафиксировать.

Хотелось бы, чтобы это отслеживание было автоматическим. Чтобы был автоматический коммит и роллбэк.

Для этого предназначен паттерн unit\_of\_work

**Листинг 6. unit\_of\_work\_sqlite.py**

|  |
| --- |
| **import** sqlite3 **import** threading  connection = sqlite3.connect(**'patterns.sqlite'**)   **class** RecordNotFoundException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Record not found: {**message**}'**)   **class** DbCommitException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Db commit error: {**message**}'**)   **class** DbUpdateException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Db update error: {**message**}'**)   **class** DbDeleteException(Exception):  **def** \_\_init\_\_(self, message):  super().\_\_init\_\_(**f'Db delete error: {**message**}'**)   **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, 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 = **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=?, 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** MapperRegistry:  @staticmethod  **def** get\_mapper(obj):  **if** isinstance(obj, Person):  **return** PersonMapper(connection)  *# elif isinstance(obj, Category):  # return CategoryMapper* **class** UnitOfWork:  *"""  Паттерн UNIT OF WORK  """  # Работает с конкретным потоком* 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   **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)   **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   **class** Category(DomainObject):  **def** \_\_init\_\_(self, name):  self.name = name   **class** CategoryMapper:  **pass   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()) |

Начнем с применения:

У нас есть такая штука, как единица работы:

Создаем сессию работы с БД

UnitOfWork.new\_current()

Далее создаем новые модели и помечаем, что мы их создали

new\_person\_1 = Person(**None**, **'Igor'**, **'Igorev'**)  
 new\_person\_1.mark\_new()  
  
 new\_person\_2 = Person(**None**, **'Fedor'**, **'Fedorov'**)  
 new\_person\_2.mark\_new()

Далее нашли некоторого персона в базе

exists\_person\_1 = person\_mapper.find\_by\_id(1)

Изменили его и помечаем, что он изменен

exists\_person\_1.mark\_dirty()  
 print(exists\_person\_1.first\_name)  
 exists\_person\_1.first\_name += **' Senior'** print(exists\_person\_1.first\_name)

В итоге у нас три команды:

mark\_new – создать объект

mark\_dirty – изменить

mark\_removed – удалить

Далее делаем операцию:

UnitOfWork.get\_current().commit()

Она как бы смотрит, что нового у нас произошло с разными моделями (не только Person)

У нас есть класс UnitOfWork

Он хранит списки разных объектов:

self.new\_objects = []  
 self.dirty\_objects = []  
 self.removed\_objects = []

У нас есть методы для работы с этими списками:

**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)

Когда мы делаем commit, происходит следующее:

**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)

Что происходит в этих методах:

Мы перебираем объекты: помеченные, удаленные, измененные

Для каждого из них мы находим его маппер.

И через маппер делаем insert, update, delete

У маппера есть эти методы, мы знаем.

Так у нас данные попадут в БД.

Теперь о

MapperRegistry.get\_mapper(obj)

Это просто фабричный метод. НО как он здесь работает?

У нас есть некоторый класс

MapperRegistry

У него есть метод get\_mapper

**class** MapperRegistry:  
 @staticmethod  
 **def** get\_mapper(obj):  
 **if** isinstance(obj, Person):  
 **return** PersonMapper(connection)

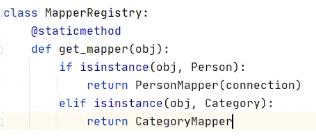
Дальше мы сюда передаем не сам класс конкретный, а объект:

**def** get\_mapper(obj):

и просто проверяем его тип:

**if** isinstance(obj, Person):

И мы легко можем добавить привязку к мапперу новой модели:



НЕБОЛЬШОЕ РЕВЬЮ:

Итак, когда мы сделали коммит, в списках лежат объекты, которые мы пометили: они добавились, изменились или нужно удалить. После этого мы просто в цикле блоки эти проходим и у каждого объекта получаем его маппер и вызываем соответствующий метод. После этого объект попадает в базу.

ТЕПЕРЬ САМАЯ сложная часть. Как у нас объекты попадут в эти списки:

self.new\_objects = []  
self.dirty\_objects = []  
self.removed\_objects = []

Почему эта часть самая сложная, потому что объект (персон) никак не взаимодействует с Unit\_of\_work.

То есть каким-то образом, когда мы вызываем нужный метод, объект должен попадать в нужный список. Как это происходит?

Мы создаем класс:

**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)

Если мы будем наследоваться от этого класса

**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

То у нас добавятся как раз эти три метода: mark\_new, mark\_dirty, mark\_removed

Как они реализуются?

Мы берем

UnitOfWork.get\_current().register\_new(self)  
UnitOfWork.get\_current().register\_dirty(self)  
UnitOfWork.get\_current().register\_removed(self)

Получается так работает: если это доменный объект, у него появляются эти три метода:

**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)

И далее, при вызове этих методов, т.к. у нас UnitOfWOrk – глобальный, мы берем и регистрируем сами себя внутри него.

Теперь самая сложная конструкция:

UnitOfWork.get\_current()

Метод get\_current делает весьма сложную операцию:

@classmethod  
**def** get\_current(cls):  
 **return** cls.current.unit\_of\_work

Т.е. он берет класс, у класса берет current и в нем берет unit\_of\_work

Что такое current?

current = threading.local()

Это некоторый текущий поток.

И unit\_of\_work должен откуда-то там появиться:

cls.current.unit\_of\_work

А вот откуда. Метод set\_current позволяет взять текущий поток и положить туда какое-то свойство – у нас это unit\_of\_work

@classmethod  
**def** set\_current(cls, unit\_of\_work):  
 cls.current.unit\_of\_work = unit\_of\_work

И еще есть метод new\_current, который берет класс, вызывает метод set\_current, создает экземпляр класса unit\_of\_work и складывает его в этот поток.

НЕБОЛЬШОЕ РЕВЬЮ.

*СЛАЙД 15*

Итак, у нас есть наш класс Unit\_of\_Work

Если мы сделаем new\_current, то мы берем и создаем объект этого класса и складываем этот объект внутрь самого класса в поток current.

Получается класс в текущем потоке хранит в себе свой же экземпляр.

И если мы делаем метод get\_current, то мы по сути берем поток, из него извлекаем текущий экземпляр класса Unit\_of\_work и можем что-то делать с этим экземпляром.

Такая сложная штука нужна для того, чтобы все хранить в экземпляре класса, а он лежит в текущем потоке.

Берем объект и у него делаем коммит.

UnitOfWork.get\_current().commit()

И остается вопрос, зачем нам этот объект класть в свой же класс.

Ответ кроется здесь:

**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)

Такая замороченная вещь нам нужна, чтобы мы могли в DomainObject глобально использовать Unit\_of\_Work

Т.е. мы здесь нигде не передаем текущий объект Unit\_of\_Work. А мы его берем из самого класса:

UnitOfWork.get\_current().register\_new(self)

UnitOfWork.get\_current().register\_dirty(self)  
 UnitOfWork.get\_current().register\_removed(self)

А класс можно написать везде. Нигде нет жесткой привязки к объекту. Мы динамически получаем этот объект.

Получается доменный объект не знает про объект Unit\_of\_Work. Хотим избавиться от зависимости от объекта и помещаем объект внутрь класса.

Теперь как сделать то же самое с другой моделью?

1. **class** Category(DomainObject):  
    **def** \_\_init\_\_(self, name):  
    self.name = name
2. добавляем маппер к этому класса
3. идем в MapperRegistry и ставим объекту маппер

И остальная сопутствующая логика.

**«Коллекция объектов (Identity Map)»**

*СЛАЙД 16*

* **@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@**

**Листинг 7. identity\_map.py**

|  |
| --- |
| **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** |

По сути он отвечает просто за кеширование объектов на стороне питона, т.е. на стороне модели, а не на стороне базы.

Мы создаем словарь для кэширования объектов. И мы сначала идем в словарь, смотрим, есть ли там объекты, потом идем в базу.

Применяется, чтобы не обращаться постоянно к базе.

НАПОСЛЕДОК, рассмотрим еще один пример использования unit\_of\_work

**Листинг 7. identity\_map.py**

|  |
| --- |
| **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** |