*Урок 7. Архитектурные системные паттерны*

Оглавление

[Введение 2](#_Toc90653472)

[Базовые паттерны 2](#_Toc90653473)

[Реестр (Registry) 2](#_Toc90653474)

[Объектно-реляционные паттерны 4](#_Toc90653475)

[Преобразователь данных (Data Mapper) 4](#_Toc90653476)

[Единица работы (Unit of Work) 6](#_Toc90653477)

# Введение

Сегодня поговорим об архитектурных системных паттернах.

Вновь будет отсылка к Мартину Фаулеру, который описал несколько десятков таких паттернов в своей книге «Шаблоны корпоративных приложений», в 2002 году, в которой разобрал паттерны, необходимые для разработки бизнес-систем (энтерпрайз-приложений).

Объектно-реляционные паттерны являются основой к реализации ORM, но она весьма сложна и полноценную ORM-систему мы реализовать не сможем, но упрощенную попытаемся. Поэтому познакомимся с несколькими паттернами, которые лежат в основе ORM.

# Базовые паттерны

Их насчитывается несколько десятков, некоторые очень сложные и мало используемые. Мы разберем один из таких паттернов, необходимый при реализации собственной ORM.

## Реестр (Registry)

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

Очень простой паттерн.

**Листинг 1. Урок 7. Коды к уроку/registry.py**

|  |
| --- |
| **class** RegistryHolder(type):  count = 0  lst = []   **def** \_\_new\_\_(cls, name, bases, attrs):  new\_cls = type.\_\_new\_\_(cls, name, bases, attrs)   **'''  <class '\_\_main\_\_.Animal'>  <class '\_\_main\_\_.Bear'>  <class '\_\_main\_\_.Cat'>  class '\_\_main\_\_.Dog'>  '''** cls.count += 1  cls.lst.append(new\_cls.\_\_name\_\_)  **return** new\_cls   **class** Animal(metaclass=RegistryHolder):   count = 0   **def** \_\_init\_\_(self):  Animal.count += 1   **class** Bear(Animal):  **pass   class** Cat(Animal):  **pass   class** Dog(Animal):  **pass** print(RegistryHolder.lst) |

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

И после каждого добавления нового класса-животного будем увеличивать значение счетчика на 1.

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

**class** RegistryHolder(type):  
 count = 0  
 lst = []  
  
 **def** \_\_new\_\_(cls, name, bases, attrs):  
 new\_cls = type.\_\_new\_\_(cls, name, bases, attrs)  
  
 **'''  
 <class '\_\_main\_\_.Animal'>  
 <class '\_\_main\_\_.Bear'>  
 <class '\_\_main\_\_.Cat'>  
 class '\_\_main\_\_.Dog'>  
 '''** cls.count += 1  
 cls.lst.append(new\_cls.\_\_name\_\_)  
 **return** new\_cls

В реестр мы помещаем наш регистрируемый класс:

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

cls.lst.append(new\_cls.\_\_name\_\_)

# Объектно-реляционные паттерны

## Преобразователь данных (Data Mapper)

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

Давайте вспомним, что Python «умеет» взаимодействовать с базами данных посредством механизма DB-API. Механизм вполне рабочий, но требует знания языка запросов SQL, что для разработчика может представлять большую проблему. Язык весьма непростой, особенно если речь идет о сложных запросах.

Но мы ведь знаем, что смысл ORM заключается в том, чтобы работать с базой данных используя привычный Python-синтаксис без необходимости составлять сложные SQL-запросы.

Нам нужен посредник, которому мы будем «отдавать» команды используя Python-синтаксис, но эти команды «кто-то» будет транслировать в SQL-команды.

Возникла идея о реализации посредника.

**Листинг 2. Урок 7. Коды к уроку/data\_mapper.py**

|  |
| --- |
| **from** sqlite3 **import** connect   **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,"** \  **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\_\_) |

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 класс-посредник (маппер) PersonMapper и именно он будет непосредственно выполнять операции с БД.

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

Они принимают объект модели и через него выполняют операции по изменению БД.

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. Но обо всех знает **PersonMapper**.

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

## 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»*

По сути при создании новых объектов классов-моделей (новых записей БД) мы фиксируем, что нам нужно создать новую запись.

Также мы фиксируем, что нам нужно изменить параметры какой-то записи БД или удалить.

Мы фиксируем это в некотором плане, а потом шаг за шагом выполняем этот план, но одной транзакцией.

**Листинг 3. Урок 7. Коды к уроку/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.

Создаем сессию работы с БД:

UnitOfWork.new\_current()

Далее создаем новые объекты классов-моделей (новые записи) 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()

Кроме создания мы можем извлекать нужную нам запись по ее идентификатору и содержимое пробрасывать в объект-класса модели, чтобы дальше работать с этой записью БД именно как с объектом класса-модели.

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

Меняем параметры записи и указываем, что мы хотим внести изменения:

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() – помечаем, что нужно создать объект

mark\_dirty() – помечаем, что нужно изменить объект

mark\_removed() – помечаем, что нужно удалить объект

Далее делаем операцию:

UnitOfWork.get\_current().commit()

Эта пользовательская функция (commit()) как бы запускает сканирование, что нового у нас произошло с разными моделями (не только Person).

У нас есть класс UnitOfWork

Он хранит списки разных объектов – создаваемых, изменяемых, удаляемых:

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 выполняем commit(),

**def** commit(self):  
 self.insert\_new()  
 self.update\_dirty()  
 self.delete\_removed()

Происходит следующее:

**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 помеченные объекты: создаваемые, удаленные, измененные.

Для каждого из них мы находим его маппер, в зависимости от того, к какому классу принадлежит объект.

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

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

У него есть метод get\_mapper().

В него мы передаем объект класса-модели:

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

И просто проверяем его тип:

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

В зависимости от того, к какому классу принадлежит наш объект, получаем нужный класс-маппер.

Мы забыли про еще один класс – DomainObject.

Он нужен для того, чтобы мы могли выполнять пометки новых записей, измененных и удаленных. Т.е. помечать наши объекты классов-моделей.

**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)