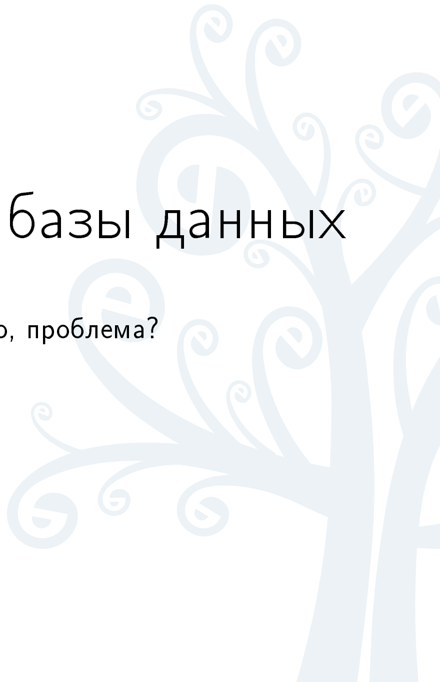
ORM как средство реализации бизнес-логики

Отдел R&D

Маркетинговая группа Текарт

19 ноября 2009 г.



Бизнес-логика и базы данных

А в чем, собственно, проблема?

Слой модели приложения

Почему объекты, а не встроенные структуры данных?

- позволяют добавить к данным поведение;
- типизация повышает надежность кода (даже в PHP);
- структурирование, меньше дублирующегося кода;
- набор классов формирует внутренний API приложения.

Проблема реализации слоя хранения

Состояние объектов необходимо сохранять.

В идеальном мире:

- объектная база данных – отдельный слой приложения;
- объекты бизнес-логики могут вообще ничего не знать о слое хранения.

В реальности:

- наиболее распространены реляционные базы (но ситуация меняется);
- необходим посредник между объектами и таблицами реляционной базы.

Object Relational Impedance Mismatch

В теории – две модели, основанные на совершенно разных принципах.

На практике:

- разные системы типов свойств объектов и колонок таблиц;
- различный подход к определению идентичности сущностей;
- проблема отображения классов, связанных отношением наследования;
- проблема частичной загрузки информации о сущности;
- проблема отложенной загрузки ассоциированных объектов;
- проблема построения выборок.

The Vietnam of Computer Science

Различия в системах типов

Простые типы колонок таблиц – аналогичные типы языка реализации.

Сложные типы атрибутов могут потребовать дополнительного преобразования или состава колонок, отличного от набора свойств объекта. например:

- поля типа DATE, DATETIME, ... (по крайней мере для PHP);
- атрибуты, представляющие собой композицию нескольких простых значений;
- атрибуты, представляющие собой коллекцию значений, не отображаемых на таблицы.

Идентичность сущностей

В объектной модели как правило:

- Identity определяется экземпляром объекта в памяти;
- Equality – состоянием объекта;

В реляционной модели однозначно определяется значением PRIMARY KEY.

Разница подходов – источник потенциальных проблем, например, при организации кеширования.

Проблема отображения иерархий наследования

У наследуемых классов часть структуры состояния одинаковая, часть – разная. Как отобразить это в таблицы?

Inheritance:

- **Single Table** одна таблица на иерархию, часть полей всегда пустые, специальное поле указывает класс объекта строки.
- **Class Table** одна таблица на каждый класс в иерархии наследования, в т.ч. абстрактный.
- **Concrete Table** одна таблица на каждый класс в иерархии наследования, объект которого можно создать.

У всех свои недостатки: Single table – много пустых полей, остальные – много JOIN-ов.

Проблема частичной загрузки информации о сущности

В объектной модели состояние объекта атомарно, в реляционной мы можем получить любое необходимое подмножество атрибутов.

В статических языках неразрешимо, только обходные пути:

- допускаем возможность пустых значений для некоторых полей – неполноценные объекты с неполным состоянием
- композиция объектов: обязательная часть и дополнительно подгружаемая необязательная;

В динамических проще, но все равно остается проблема определения неполной загрузки, проблемы связанные с кешированием, определением равенства и т.д.

Отложенная загрузка ассоциированных объектов

- ассоциированные объекты нужно загружать только по необходимости;
- ассоциированные объекты нужно по возможности кешировать;
- с другой стороны, для списка объектов отдельный запрос на ассоциации для каждого – катастрофа;

Многие решения пытаются создать видимость прозрачной работы с ассоциированными объектами через proxy-объекты для свойств и так далее. Очень сложно в реализации, тяжело контролировать и оптимизировать.

Построение выборок

SQL – лучшее решение для выборок из RDBMS.

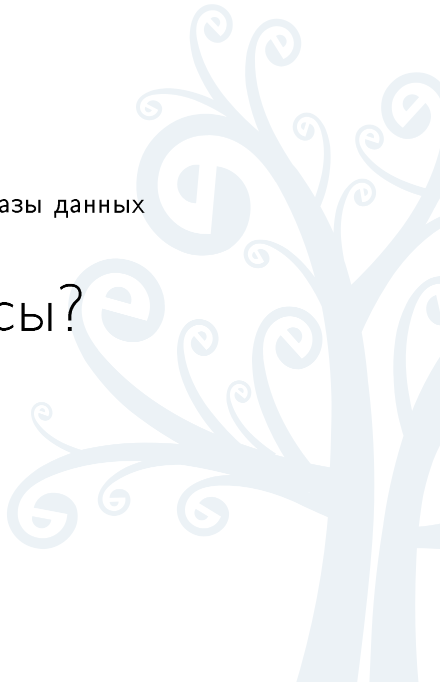
Многие фреймворки пытаются использовать другие подходы:

- **QBE** Query by Example: сопоставление с эталонным объектом. Накладывает ограничения на классы сущностей, недостаточно выразителен;
- **QBA** Query by API: специальные объекты для построения критериев выборки. Многословно и сложно по сравнению с SQL-запросом.
- **QBL** Query by Language: пишем свой SQL, но объектный. Только не это.

Процесс генерации SQL скрыт от разработчики ⇒ проблемы с оптимизацией запросов и избыточность кода.

Бизнес-логика и базы данных

Вопросы?



Типовые решения

Наиболее распространенные паттерны проектирования.

Архитектурные шаблоны

Идеального решения нет, всегда приходится выбирать, что ближе – объектное или реляционное представление.

Relational

Table Data Gateway базовые операции для каждой таблицы в отдельной классе;

Row Data Gateway объектное представление строки таблицы;

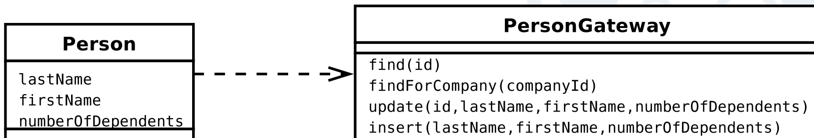
Active Record работа с базой данных непосредственно в объекте бизнес-логики;

Data Mapper отдельный независимый слой хранения

Object

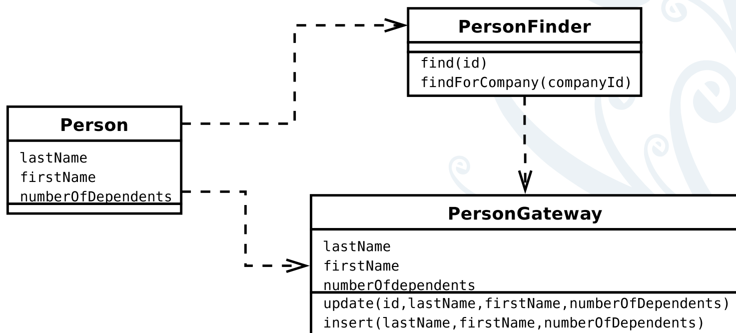
Table Data Gateway

Объект, выполняющий роль шлюза к таблице БД.
Один экземпляр работает для всех записей в таблице.



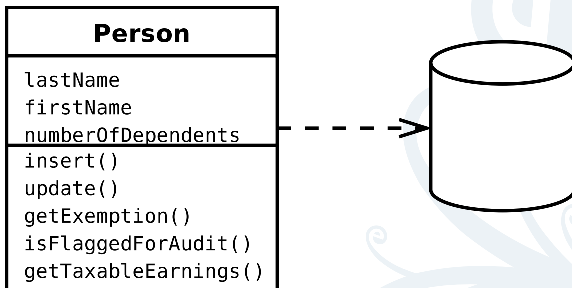
Row Data Gateway

Объект, выполняющий роль шлюза к записи БД. Как правило, используется в Transaction Script, с объектами бизнес-логики используется редко.



Active Record

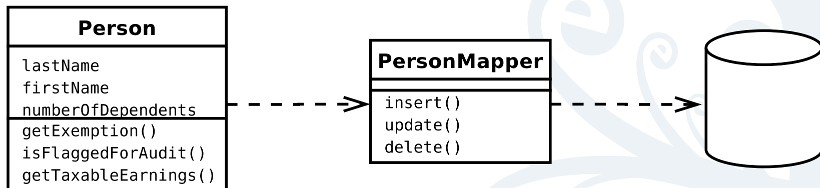
Бизнес-логика и работа с базой в одном объекте.



Простое и самое известное решение, удобно работать с одним объектом, хуже – с выборками. Самый известный фреймворк – ActiveRecord в Ruby On Rails.

Data Mapper

Идеальный ORM. Объекты бизнес-логики не зависят от слоя хранения, создается иллюзия работы с объектами без использования базы.



Самая известная реализация – Hibernate.

Поведенческие паттерны

Unit of Work изменение состояние объектов регистрируется, набор выполненных изменений записывается в базу одной операцией.

Identity Map кеширование уже загруженных бизнес-объектов для уменьшения количества простых запросов;

Lazy Load подгрузка дополнительной информации только тогда, когда в ней появилась необходимость.

- **Lazy Initialization** в качестве значения поля используется специальный маркер (null);
- **Virtual Proxy** объект с совпадающим интерфейсом, после загрузки делегирует вызовы реальному объекту.
- **Value Holder** промежуточный объект с методом getValue, при первом вызове подгружается объект.

Структурные паттерны

Identity Field primary key записи в таблице – поле бизнес-объекта;

Serialized LOB сложную структуру связанных объектов можно хранить в сериализованном виде в поле таблицы;

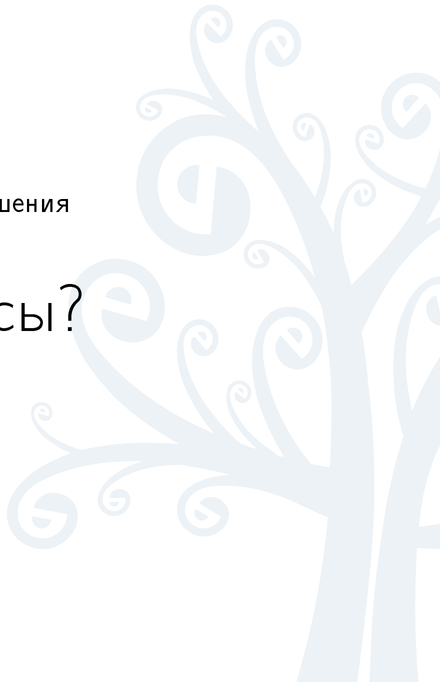
Single Table Inheritance одна таблица на иерархию классов;

Class Table Inheritance одна таблица на каждый класс иерархии, в т.ч. абстрактный;

Concrete Class Table Inheritance одна таблица на каждый класс в иерархии, для которого можно создать объект.

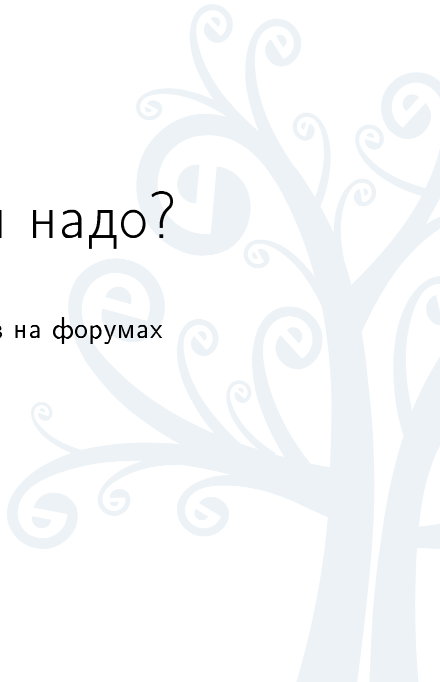
Типовые решения

Вопросы?



А оно нам надо?

Вечная тема споров на форумах



Религиозные войны

«Системные Архитекторы Высоконагруженных Приложений» (говорим ORM, подразумеваем Hibernate):

> *ORM генерирует кривые запросы и не дает писать прямые, оптимизированные вручную!*
> *ORM не дает нам использовать хранимые процедуры!*
> *ORM – это ООП, а ООП усложняет приложение и не нужно!*
> *Уже сейчас видно, что всё это будет глючить и тормозить.*

«Пионеры с ORM»

> *А вдруг придется поменять базу?*
> *Разработчику не нужно знать SQL, ORM все сделает сам!*
> *ORM это ООП, а ООП упрощает приложение, у нас N:M одной строкой, а схема базы – автоматом!*
> *Будет тормозить – как нибудь оптимизируем.*

Наша точка зрения

Не нужно крайностей, нужен компромисс.

- Идеальный сферический ORM, полностью скрывающий от разработчика работу с базой – невозможен, сложность и неэффективность решения обесценивает все его теоретические плюсы;
- Объекты бизнес логики нужны, разумно теряем в производительности – выигрываем в удобстве разработке и простоте сопровождения;
- Использование возможностей конкретного сервера БД – правильно, смена сервера – маловероятное событие.
- Как можно более простой стандартный ORM-слой между базой и объектами бизнес-логики необходим.

Наша реализация

Модули DB и DB.ORM

Что мы хотим от слоя бизнес-логики

- Представление сущностей предметной области в виде объектов;
- Удобное построение различных выборок в терминах предметной области;
- Отсутствие необходимости генерации SQL-запросов вручную;
- Возможность использования SQL при необходимости;
- Легковесность и поменьше магии: чрезмерная интеллектуальность ORM – залог последующей борьбы с ним.

DB – тонкий объектный слой над PDO

- Два базовых класса: соединение и курсор;
- Курсор является итератором по результатам выборки;
- Даты преобразуются в объекты при выборке и подстановке параметров;
- Каждая запись результата может быть преобразована в объект указанного класса;
- Значения параметров для подстановки могут быть получены из произвольного объекта;
- Поддержка пользовательских обработчиков событий.

DB: пример

```
$c = DB::Connection($dsn_string)->
$c->listener($listener);

$cursor = $c->prepare("SELECT * FROM categories");
$categories = $cursor->execute()->fetch_all();

foreach ($c->
  prepare("SELECT * FROM stories ".
          "WHERE pub_date < :date AND status = :status")->
  bind(Time::now(), $stats)->as_object('Story') as $story)
    print $story->id;

$c->execute(
  "UPDATE stories SET title = :title WHERE id = :id",
  $story);
```

DB.ORM.Entity – базовый класс объекта домена

Две стороны объекта домена

[ ]	$\longrightarrow$
скалярные значения атрибутов	объектное представление свойств
`row_get_{property}` `row_set_{property}`	`get_{property}` `set_{property}`

- набор атрибутов доступен как свойство `attrs`;
- набор атрибутов не фиксирован $\Rightarrow$ частично загруженные сущности или дополнительные атрибуты.

DB.ORM – простой иерархический Data Mapper

- относительно легковесный;
- не использует Active Record, скорее продвинутый Table Gateway;
- условия выборок в терминах предметной области;
- специализированные мапперы выводятся из более общих;
- все мапперы образуют динамически формируемую иерархию;
- все базовые классы мапперов абстрактные, реализация поведения – через наследование.

DB.ORM.Mapper – базовый класс маппера

- абстрактный класс;
- содержит ссылку на родительский маппер;
- наследует значения свойств от родительского маппера.

---

spawn порождение дочернего маппера своего класса;

pmap отображение свойства в дочерний маппер;

cmap отображение вызова в дочерний маппер.

Отображение свойств и методов – путем реализации методов с префиксами `pmap_`, `cmap_` и `map_`.

Группирующие мапперы

DB.ORM.MapperSet → DB.ORM.Mapper

- логическая группировка дочерних мапперов;
- кеширует результаты выполнения `pmap`;
- не кеширует результаты `cmap`.

DB.ORM.ConnectionMapper → DB.ORM.MapperSet

- группировка + хранение объекта подключения;
- чаще всего используется как корневой маппер;
- объект подключения доступен всем дочерним мапперам через механизм наследования свойств;

Группирующие мапперы: пример

```
class App_DB_NewsMapperSet extends DB_ORM_MapperSet {
  protected function map_stories() {
    return App_DB::StoriesMapper($this)->immutable();
  }
  protected function map_categories() {
    return App_DB::CategoriesMapper($this)->immutable();
  }
}
```

- мапперы статей и рубрик доступны как `$db->news->stories` и `$db->news->categories`;
- вызов `immutable()` гарантирует возможность изменения опций только при явном использовании `spawn()`.

SQL-мапперы

DB.ORM.SQLMapper → DB.ORM.Mapper

- абстрактный класс
- содержит набор опций генерации SQL-запросов
- наследует опции генерации родительских мапперов
- работает как итератор

Базовые операции

select	**select_for**	**select_first**
find	**insert**	**update**
delete	**stat**	**stat_all**

SQL-мапперы: базовые операции

```
$all = $db->news->categories->select();
foreach ($db->news->categories as $category)
  echo $category->title."\n";

$category = $db->news->categories->find($id);
$category->title = 'New title';
$db->news->categories->update($category);

$db->news->categories->insert($new_category);
printf("New category id is %d", $category->id);

$db->news->categories->delete($category);
$num_of_categories = $db->news->categories->stat();
```

SQL-мапперы: изменение параметров запроса

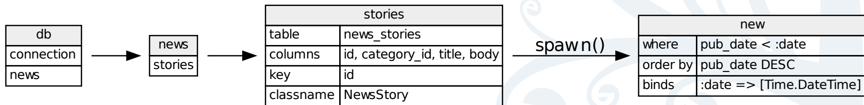
Набор опций формирования SQL-запроса

table	**join**	**calculate**
columns	**group_by**	**exclude**
classname	**order_by**	**where**
key	**range**	**having**

- выполняем spawn();
- дочерний маппер содержит модифицированные опции, остальное – из родительского маппера;
- реализация не содержит циклических ссылок.

Изменение параметров запроса: пример

```
$db->news->stories->spawn()->
  where('pub_date < :date', $date)->
  order_by('pub_date DESC')->
    select()
```



Выборки в терминах предметной области

Переопределяем вызов или свойство:

```
class StoriesMapper extends DB_ORM_SQLMapper {
  protected function map_for_category(Category $category) {
    return $this->where('category_id = :id', $category);
  }
}
```

`$db->news->stories->for_category($c)`:

1. неявно выполняется spawn();
2. map-метод вызывается для дочернего маппера;
3. для дальнейшего изменения параметров явный `spawn()` уже не нужен.

Выборки в терминах предметной области (продолжение)

```
protected function map_without_body() {
  return $this->exclude('body');
}

protected function map_most_popular($limit = 5) {
  return $this->order_by('num_of_views')->
  range(5);
}

$db->stories->
  most_popular(10)->
  for_category($category)->
  without_body()->
  select();
```

Кеширование сущностей

`$mapper[$id]` = `$mapper->find($id)` + **кеширование**

Позволяет удобно реализовать выборку связанного объекта:

```
class Story {
  protected function get_category() {
    return App::db()->news->categories[$this['category_id']];
  }
  protected function set_category($category) {
    if ($category->id) $this['category_id'] = $category->id;
  }
}
```

Кеш может быть загружен заранее: `$mapper->preload()`.

Дополнительные возможности

- валидация объектов путем определения объекта-валидатора для маппера;
- хранение информации о пользователе в корневом маппере позволяет дочерним проверять права доступа для различных операций;
- класс сущности может определить набор обработчиков событий:

before_save	before_insert	before_update	before_delete
after_save	after_insert	after_update	after_delete

Отношение 1:N

Используем специализированный маппер:

```
$stories = $db->news->stories->for_category($category);
```

Или возвращаем маппер из метода сущности:

```
class Category {
  protected function get_stories() {
    return App::db()->news->stories->for_category($this);
  }
}
class StoriesMapper {
  protected function map_for_category($category) {
    return $this->
      where('category_id = :category_id', $category->id)->
      force(array('category_id' => $category->id));
  }
}
```

Отношение M:N

Отдельный маппер для связующей таблицы:

```
class StoriesTagsMapper {
  protected function setup() {
    return $this->table('news_tag_refs')->
                  columns('story_id', 'tag_id', 'ord')->
                  key('story_id', 'tag_id');
  }
  public function associate($tag, $story, $ord) {
    return $this->insert(
      array($tag->id, $story->id, $ord), 'ignore');
  }
  public function dissociate($tag, $story) {
    return $this->delete(array($tag->id, $story->id));
  }
}
```

Интеграция с Sphinx

Search.Sphinx – интерфейс к Sphinx, Search.Sphinx.ORM – выборка результатов поиска с помощью DB.ORM

```
$sphinx->
  select($query)->
  using($index)->
  resolve_with(Search_Sphinx_ORM::Resolver()->
    mappers($db->news->stories,
            $db->photo->galleries,
            $db->video->clips));
```

- для найденных документов автоматически подгружаются соответствующие объекты;
- поддерживаются разнотипные документы.

Иерархия мапперов как REST-сервис

```
$db->news->stories->most_popular->select()
GET /api/news/stories/most_popular/
```

```
$db->news->stories->insert($story);
POST /api/news/stories/
```

```
$story = $db->news->stories[15];
$db->news->stories->update($story);
$db->news->stories->delete($story);

PUT /api/news/stories/15/
DELETE /api/news/stories/15/
```

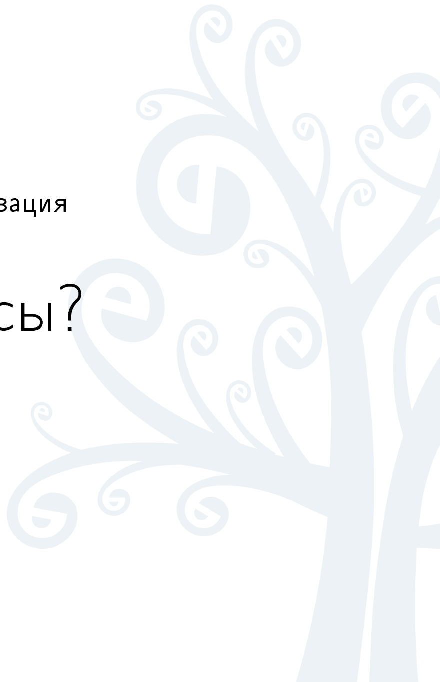
```
$db->news->categories[5]->stories;
GET /api/news/categories/5/stories/
```

Подведем итог

- Несложно реализовать легковесную альтернативу тяжелым ORM-библиотекам;
- Даже простое решение может помочь писать компактный и выразительный код;
- Даже простое решение может помочь писать компактный и выразительный код;
- Часто проще непосредственно реализовать действие в коде, чем конфигурировать сложный автоматический механизм;
- Иерархическая структура мапперов – хорошая основа для API приложения в стиле REST.

Наша реализация

Вопросы?



Что читать

Теория:

- The Vietnam of Computer Science
- Martin Fowler: Patterns of Enterprise Application Architecture
- Patterns of Enterprise Application Architecture Catalog
- Crossing the Chasm from Object to Relational Databases

Реализации:

- Hibernate: Relational Persistence for Java and .Net
- Active Record – Object-relational mapping put on rails
- Storm – object relational mapper for Python
- The Python SQL Toolkit and Object Relational Mapper
- iBatis – Object-relational mapping and persistence framework