1 ОГЛАВЛЕНИЕ

1 ОГЛАВЛЕНИЕ 2
2 ВВЕДЕНИЕ 4
2.1 Понятие объектно-реляционного отображения 4
2.2 Цели объектно-реляционного отображения 5
2.3 Задачи объектно-реляционного отображения 5
2.4 Критерии сравнения 7
2.5 Примеры библиотек объектно-реляционного отображения 7
2.6 Обзор Doctrine ORM 11
2.7 Постановка задачи 12
3 АРХИТЕКТУРА БИБЛИОТЕКИ 13
3.1 Сравнительный анализ возможных подходов 13
3.2 Преимущества выбранного подхода 16
3.3 Описание концепции 17
3.4 Структура разработанных классов 19
3.5 Управление доступом к внутренним данным объектов 20
3.6 Управление схемой базы данных 22
3.7 Загрузка объекта по первичному ключу 25
3.8 Преобразование типов данных 25
4 ОПИСАНИЕ СЦЕНАРИЕВ ИСПОЛЬЗОВАНИЯ 27
4.1 Конфигурация и подключение в базе данных 27
4.2 Создание и сохранение объектов 27
4.3 Получение пользователя по первичному ключу 29

4.4 Выполнение пользовательского SQL запроса 29

4.5 Использование собственной функции отображения строки в объект 30

5 ЗАКЛЮЧЕНИЕ 32

6 СПИСОК ЛИТЕРАТУРЫ 33

2 ВВЕДЕНИЕ

2.1 Понятие объектно-реляционного отображения

ORM (англ. Object-Relation Mapping, рус. Объектно-реляционное отображение) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

При создании программного обеспечения при помощи объектно-ориентированного подхода, возникает задача долговременного хранения объектов. Использование реляционной БД для хранения объектно-ориентированных данных приводит к семантическому разрыву между способами обработки объектов и их хранения [1]. Программисты вынуждены использовать различные концепции при разработке бизнес логики приложения (объектно-ориентированный подход) и при взаимодействии с хранилищем объектов (реляционная модель). Переход между концепциями происходит каждый раз при обращении к БД, что приводит либо к многократному дублированию кода, либо к созданию библиотеки «прослойки» между базой данных и приложением. Вариант с разработкой библиотеки с точки зрения технологии программирования представляется предпочтительным. Класс библиотек, предоставляющих методы для прямого и обратного преобразования объектов в реляционную модель, называется ORM.

Альтернативой использования объектно-реляционного отображения является хранение объектов в объектно-ориентированных базах данных [2]. Такой подход устраняет семантический разрыв между представлениями объектов. Тем не менее, из-за высокой распространенности реляционных БД, качественное объектно-ориентированное отображение остается на сегодняшний день актуальной задачей.

2.2 Цели объектно-реляционного отображения

Цели использования ORM библиотек:

1. Снижение количества ошибок при преобразованиях данных между реляционной и объектной моделями.
2. Разделение логики приложения и логики хранения данных.
3. Независимость кода приложения от конкретной реализации СУБД.
4. Снижение рисков нарушения целостности данных.
5. Повышение безопасности работы с данными.
6. Уменьшение размеров кода приложения.

2.3 Задачи объектно-реляционного отображения

Разработка универсального подхода к автоматическому прямому и обратному отображению объектно-ориентированных и реляционных данных связана с трудностями (в англ. литературе - «object-relational impedance mismatch» [1], [3]):

1. *Степень детализации*

   Объектная модель может содержать больше классов, чем количество соответствующих таблиц в базе данных. Объектная модель является более детализированной чем реляционная модель.

2. *Наследование*

   Наследование является одной из особенностей объектно-ориентированных языков программирования. Современные реляционные СУБД не имеют ничего подобного наследованию.

3. *Идентичность*

Реляционные СУБД определяют идентичность с помощью первичного ключа. В ООП объекты не имеют аналога первичного ключа, и идентичность для объектов определяется самим программистом.

4. *Ассоциации*

   Ассоциации в объектно-ориентированных языках представлены в виде однонаправленных ссылок. В реляционных СУБД используются понятие внешнего ключа. Если в объектно-ориентированном языке нужно определить двунаправленную связь, то необходимо определять её дважды.

5. *Доступ к данным*

   В объектно-ориентированных языках для доступа к данным происходит переход от одной ассоциации к другой по графу объектов. Такой способ не является эффективным при получении данных из реляционной БД. Как правило, количество SQL запросов полезно минимизировать. Загрузка нескольких объектов осуществляется с помощью JOIN операций по всем целевым объектам сразу.

Преодоление этих трудностей в процессе достижения целей из предыдущего параграфа приводит к необходимости решения следующих задач:

1. по объектной модели сгенерировать схему реляционной базы данных,
2. поддержка любой реляционной базы данных,
3. описание объектной модели, типов полей, ассоциаций,
4. загрузка объектов по первичному ключу,
5. выполнение составленных программистом SQL запросов,
6. предоставление механизмов описания отображений возвращенных данных в объекты,
7. автоматическая генерация SQL запросов,
8. защита базы данных от атак вида SQL-injection,
9. ранняя/отложенная загрузка данных.

2.4 Критерии сравнения

Главными критериями сравнения для объектно-реляционного отображения являются:

1. функциональность:
   a. составление SQL запросов,
   b. выполнение составленных программистом SQL запросов,
   c. механизм описания отображения,
   d. управление схемой БД,
2. простота использования,
3. модульность библиотеки,
4. полнота тестового покрытия,
5. наличие зависимостей от других библиотек.
6. следование стандартам PSR,
7. шаблон проектирования.

2.5 Примеры библиотек объектно-реляционного отображения

- Eloquent

1. функциональность:
   a. составление SQL запросов: присутствует.
   b. выполнение составленных программистом SQL запросов: отсутствует.
   c. механизм описания отображения: отсутствует,
   d. управление схемой БД: присутствует ручное управление схемой базы данных,
2. простота использования: предоставляет достаточно простое API
3. модульность библиотеки: библиотека не является модульной, жестко встроена в фреймворк,
4. полнота тестового покрытия: хорошее покрытие тестами,

5. наличие зависимостей от других библиотек: зависит от других библиотек фреймворка Laravel,
6. следование стандартам PSR: следует стандартам,
7. шаблон проектирования: ActiveRecord [4].

- PHP ActiveRecord

1. функциональность:
    a. составление SQL запросов: присутствует,
    b. выполнение составленных программистом SQL запросов: отсутствует,
    c. механизм описания отображения: отсутствует,
    d. управление схемой БД: не умеет генерировать схему базы данных по объектной модели.
2. простота использования: достаточно просто использовать,
3. модульность библиотеки: один модуль,
4. полнота тестового покрытия: хорошее покрытие тестами,
5. наличие зависимостей от других библиотек: библиотека не зависит от других библиотек, но сама реализует множество качественных сторонних библиотек,
6. следование стандартам PSR: не следует ни одному стандарту,
7. шаблон проектирования: ActiveRecord [4].

- RedBean

  Простая, но малофункциональная ORM. Позволяет просто получать и сохранять данные из базы данных. Нет необходимости описывать схему базы данных, она генерируется библиотекой автоматически, на лету. Однако, ввиду и отсутствия объектной модели эту библиотеку нельзя отнести к ORM. Данная библиотека принципиально представляет из себя Table Gateway [4].

1. функциональность:
    a. составление SQL запросов: ограниченная функциональность,
    b. выполнение составленных программистом SQL запросов: отсутствует,
    c. механизм описания отображения: отсутствует,
    d. управление схемой БД: библиотека создаёт схему базы данных на лету.
2. простота использования: очень проста в использовании, низкий порог вхождения,
3. модульность библиотеки: состоит из одного модуля,
4. полнота тестового покрытия: хорошее покрытие тестами,
5. наличие зависимостей от других библиотек: не зависит от других библиотек,
6. следование стандартам PSR: следует части стандартов,
7. шаблон проектирования: Table Gateway [4].

- Propel

  Функциональная и мощная ORM библиотека, умеет генерировать код объектной модели по базе данных. Поддерживает все базовые операции (создание, получение, обновление, удаление) над объектами. Умеет управлять ассоциациями, транзакциями. Поддерживает наследование. Для обновления схемы базы данных используется механизм «миграций». Однако имеет достаточно большой порог вхождения и сложна в использовании.

1. функциональность:
    a. составление SQL запросов: присутствует,
    b. выполнение составленных программистом SQL запросов: отсутствует,
    c. механизм описания отображения: присутствует,

d. управление схемой БД: присутствует в виде конфигурационных файлов XML,

2. простота использования: сложна в использовании, высокий порог вхождения,
3. модульность библиотеки: библиотека состоит из одного модуля,
4. полнота тестового покрытия: отличное покрытие тестами,
5. наличие зависимостей от других библиотек: зависит от ряда сторонних библиотек,
6. следование стандартам PSR: полностью следует стандартам,
7. шаблон проектирования: Data Mapper [4].

- Doctrine ORM

1. функциональность:
   a. составление SQL запросов: присутствует,
   b. выполнение составленных программистом SQL запросов: присутствует,
   c. механизм описания отображения: присутствует,
   d. управление схемой БД: присутствует,
2. простота использования: сложна в использовании, высокий порог вхождения,
3. модульность библиотеки: состоит из независимых модулей,
4. полнота тестового покрытия: отличное покрытие тестами,
5. наличие зависимостей от других библиотек: зависит от большого числа сторонних библиотек,
6. следование стандартам PSR: следует части стандартов,
7. шаблон проектирования: Data Mapper [4].

2.6 Обзор Doctrine ORM

Doctrine ORM представляют из себя модульную библиотеку основанную на других библиотеках Doctrine Project [5]:

1. Doctrine Common

   Отвечает за конфигурирование, кеширование и различные вспомогательные действия.
2. Doctrine DBAL

   Представляет из себя слой абстракции базы данных (англ. Database Abstraction Layer). Имеет множество функций среди которых управление и самоанализ схемы базы данных.

Doctrine ORM позволяет описывать конфигурацию отображения в четырёх различных форматах:

1. Аннотации
2. XML
3. YAML
4. PHP

По конфигурации умеет генерировать код объектной модели и схему базы данных. Имеет множество функциональных возможностей по описанию различных ассоциаций и работе с ними. Позволяет описывать запросы на объектно-ориентированном диалекте SQL, а так же при помощи построителя запросов и при помощи родного SQL языка используемой СУБД. Имеет мощную систему кеширования запросов и результатов запросов. Однако высокий порог входа зачастую отталкивает разработчиков, а сложность реализации вызывает непредсказуемые сложности в использовании.

2.7 Постановка задачи

Разработать объектно-реляционное отображение, которое было бы достаточно функционально и одновременно просто в реализации. Полученное решение должно удовлетворять следующим требованиям:

- иметь простую конфигурацию,
- получать информацию об объектной модели из кода программы,
- по объектной модели генерировать схему базы данных,
- иметь простой способ получения объектов,
- возможность запроса на SQL языке СУБД,
- управлять ленивой/жадной загрузкой.

3 АРХИТЕКТУРА БИБЛИОТЕКИ

3.1 Сравнительный анализ возможных подходов

Существует несколько подходов (паттернов) к созданию библиотеки объектно-реляционного отображения (англ. Data Source Architectural Patterns): Table Gateway, Row Gateway, Active Record, Data Mapper [4]. В этом параграфе мы рассмотрим и сравним перечисленные паттерны.

1. *Table Gateway*

   Объект выполняет роль шлюза к таблице БД. Один экземпляр объекта обрабатывает все строки в таблице.

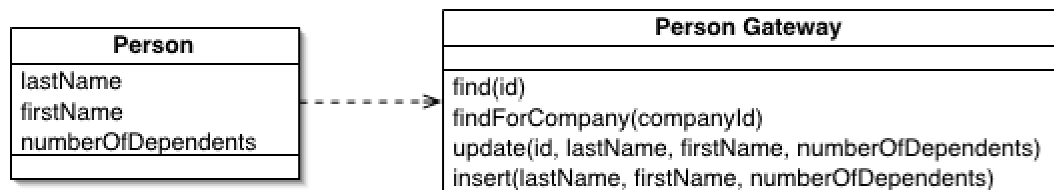

Рисунок 1

Объект реализующий Table Gateway содержит все SQL запросы для доступа к одной таблице из БД: выборки, вставки, обновления и удаления строк. Код работающий с объектом Table Gateway вызывает соответствующие методы для взаимодействия с базой данных.

Недостатки:

- Не происходит разделения логики приложения и логики хранения данных.
- От разработчиков требуется построение корректных оптимизированных SQL-запросов.

2. *Row Gateway*

Объект выполняет роль шлюза к одной строке в таблице БД. Создаётся один экземпляр объекта для каждой строки БД.

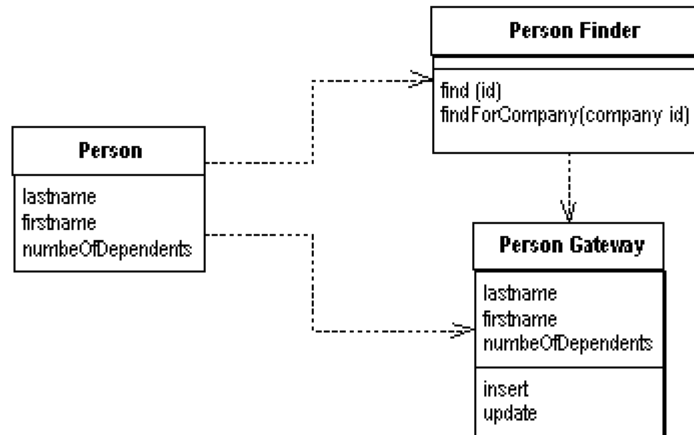

Рисунок 2

Row Gateway предоставляет объекты, имеющие ту же структуру, что и строки в таблице БД. Все детали взаимодействия с таблицей БД скрыты за этим интерфейсом.

Недостатки:

- Если объекты имеют свою бизнес логику, добавление кода для взаимодействия с базой данных, увеличивает сложность разработки.
- Замедление работы приложения (в том числе, в режиме тестирования) из-за постоянного обращения к БД при выборке объектов.

3. *Active Record*

Объект представляет обертку для строки таблицы БД и добавляет бизнес-логику к этим данным.

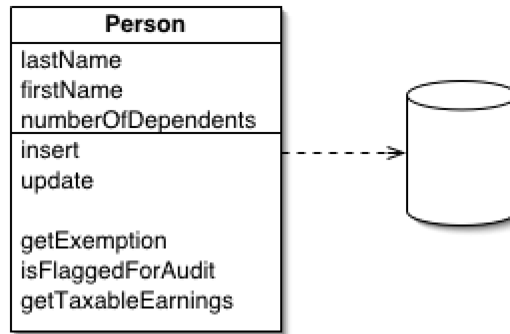

Рисунок 3

Объект реализующий паттерн Active Record содержит в себе как данные, так и поведение. Значительная часть данных содержащихся в объекте должна быть сохранена в БД. Active Record использует наиболее очевидный подход, включив логику доступа к данным в бизнес-логику объекта.

4. *Data Mapper*

Данный паттерн представляет объект, который перемещает данные между объектной моделью и базой данных, сохраняя их независимыми друг от друга.

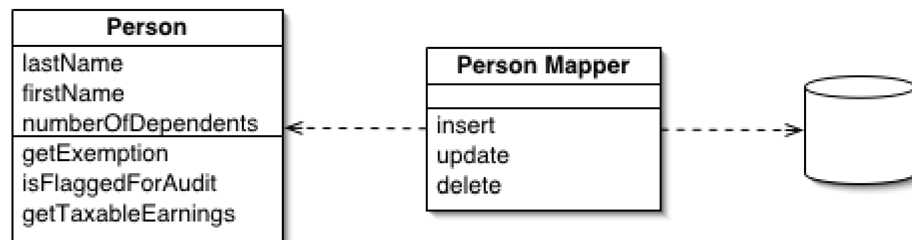

Рисунок 4

Объектная и реляционная модели данных имеют разные механизмы для работы с ними. Многие части объектной модели, такие как, коллекции и наследования, не присутствуют в реляционных данных. При организации объектной модели с большим количеством бизнес-логики полезно разделять данные и поведение. Это приводит к противоречию: объектная модель и реляционная схема не совпадают.

Data Mapper представляет собой слой программного обеспечения, который отделяет объектную модель от базы данных. Data Mapper должен передавать данные между объектной моделью и БД, а так же изолировать их друг от друга. При использовании паттерна Data Mapper, объектная модель ничего не знает о структуре БД, не содержит никаких методов получения данных, не содержит в себе SQL запросы.

3.2 Преимущества выбранного подхода

Для разработки библиотеки объектно-реляционного приложения был выбран паттерн проектирования *Active Record*. Выбор был продиктован следующими соображениями:

- Паттерн *Active Record* позволяет избежать перечисленных для *Table Gateway* и *Row Gateway* недостатков.
- Широко распространён и хорошо известен в среде веб-программистов.
- По сравнению с Data Mapper существенно выигрывает по количеству кода, который необходимо написать программисту для аналогичных операций с данными.

Листинг 1 демонстрирует пример работы с Data Mapper при помощи Doctrine ORM.

Тот же код может быть переписан при использовании Active Record с использованием меньшего количества строк кода, таким образом оставляя меньше места для потенциальных ошибок.

```php
// Пример использования Data Mapper с Doctrine ORM.

// Для работы с Data Mapper нужно всегда получать/создавать EntityManager класс.
$entityManager = new EntityManager();

// Создаём объект.
$message = new Message();
$message->user = $user;
$message->datetime = new \DateTime();
$message->text = $text;

// Размещаем объект в EntityManager.
$entityManager->persist($message);

// Производим выполнение всех необходимых запросов для синхронизации с БД.
$entityManager->flush();
```

Листинг 1

Пример того же кода при использовании Active Record с применением разрабатываемой в дипломе библиотеки Granula ORM.

```php
// Пример использования Active Record с Granula ORM.

// Создаём объект.
$message = new Message();
$message->user = $user;
$message->datetime = new \DateTime();
$message->text = $text;

// Сохраняем объект в БД.
$message->save();
```

Листинг 2

3.3 Описание концепции

В основу архитектуры разрабатываемой библиотеки Granula ORM положен паттерн Active Record. Код библиотеки следует стандартам оформления кода PSR-1 и стандарту автозагрузки классов PSR-4.

Минимально необходимая версия PHP для работы библиотеки - 5.5. Библиотека использует множество нововведений, появившихся в последних версиях PHP, такие как позднее статическое связывание, трейты/примеси, генераторы и многое другое.

Существующие реализации Active Record требуют, чтобы все объекты бизнес-логики наследовались от некоторого класса, реализующего данный паттерн. Granula ORM использует примеси для добавления требуемой функциональности. Пример использования примеси представлен в Листинге 3.

```
namespace Model;

use Granula\ActiveRecord;

class User
{
    use ActiveRecord;
}
```

Листинг 3

3.4 Структура разработанных классов

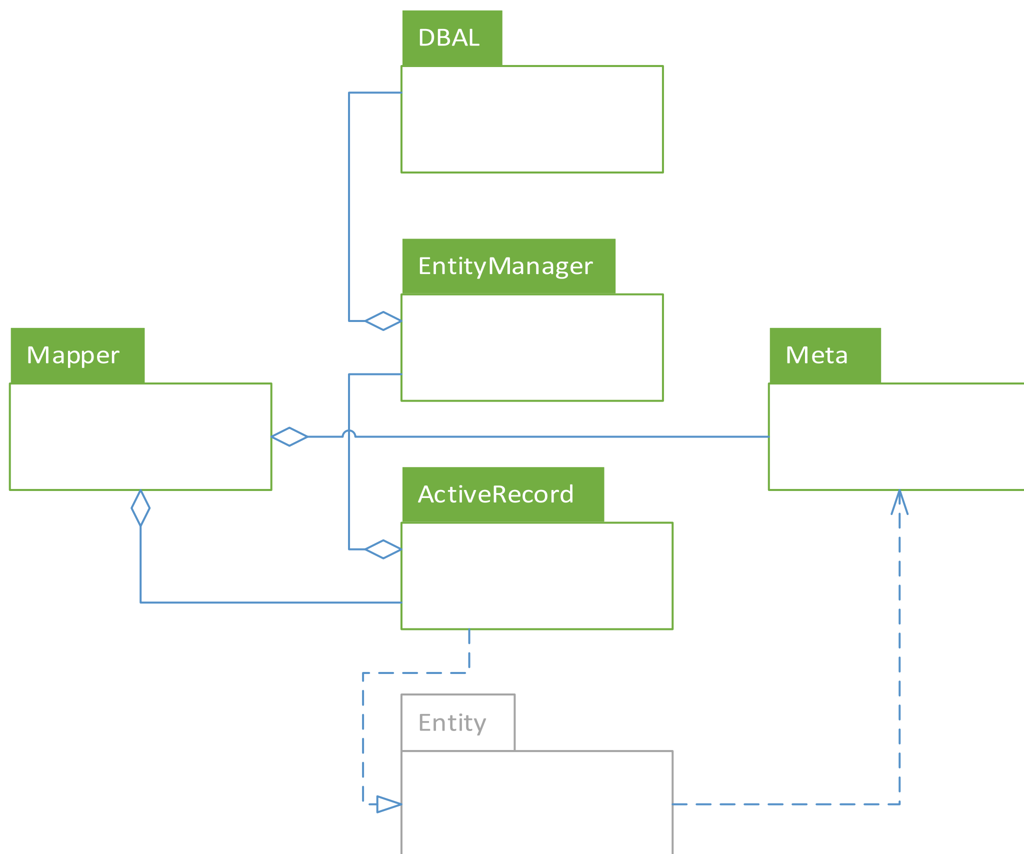

Рисунок 5

- Entity – набор классов, конкретная реализация бизнес-логики приложения, реализуется программистами при разработке ПО;
- ActiveRecord – компонент, содержащий в себе методы для работы с объектами, наиболее частые и общие методы построения запросов;
- EntityManager – управляет всеми объектами бизнес-логики, конфигурирует подключения к базе данных, кеширует объекты в памяти;
- DBAL – слой абстракции базы данных (англ. Database Abstraction Layer), управляет схемой базы данных и конвертацией типов между базой данных и приложением;

- Meta – набор классов представляющий информацию о бизнес-логике приложения;
- Mapper – компонент анализирующий SQL запросы и создающий функцию отображения;

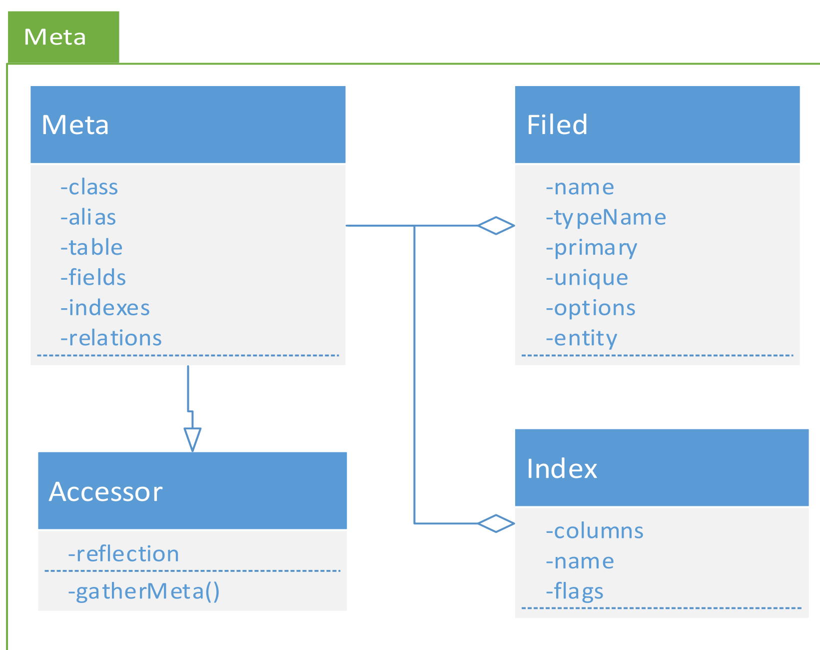

Рисунок 6

На Рисунок 6 показана структура классов компонента Meta. Класс Meta содержит в себе ссылки на классы Filed и Index отвечающие за поля и индексы объектов соответственно. Класс Accessor наследуется от класса Meta и реализует в себе функциональность, позволяющую получить мета-информацию о объектах используя DocBlock.

3.5 Управление доступом к внутренним данным объектов

В PHP отсутствуют механизмы для доступа к приватным членам класса (англ. setter|getter). В Granula ORM реализован трейт Granula\Setter, который

добавляет данную функциональность, а также добавляет поддержку обхода графа объектной модели, управление ленивой загрузкой, проверку типов подставляемых значений. При этом остаётся поддержка работы пользовательских сеттеров и геттеров.

```
namespace Model;

use Granula\ActiveRecord;
use Granula\Setter;

// Описываем класс с приватными полями и включаем трейт Setter.
class User
{
    use ActiveRecord;
    use Setter;

    private $id;
    private $name;
    private $email;
}
```

Листинг 4

Листинг 4 демонстрирует декларацию класса с приватными членами, доступ к которым может быть осуществлён при помощи трейта Granula\Setter (Листинг 5).

```
// Создаём экземпляр класса.
$user = new User();

// Устанавливаем заначения полей.
$user->name = 'User Name';
$user->email = 'mail@domain.com';
```

Листинг 5

Теперь, если нужно поменять логику работы сеттера *name*, следует определить метод *setName* (Листинг 6).

```php
// Описываем класс с приватными полями и включаем трейт Setter.
// Переопределяем сеттер для поля name.
class User
{
    use ActiveRecord;
    use Setter;

    private $id;
    private $name;
    private $email;

    public function setName($value)
    {
        $this->name = strtoupper($value);
    }
}
```

Листинг 6

Код, присваивающий значения для объекта (Листинг 5), остается без изменений, но логика работы сеттера изменена.

Так же данный трейт управляет загрузкой объектов при обходе дерева объектов.

```php
// Получение названия города проживания пользователя.
$cityName = $user->address->city->name;
```

Листинг 7

В Листинге 7 показан обычный способ обхода дерева объектов. В данном случае до выполнения оператора из БД был загружен только объект «пользователь». Для получения необходимого значения библиотека делает ещё два запроса к базе данных для получения связанных объектов «адрес» и «город», для чего автоматически генерируются и выполняются следующие SQL-запросы:

```sql
SELECT u.id AS u_id, u.name AS u_name, address.id AS address_id, address.city AS address_city FROM users u LEFT JOIN address address ON u.address = address.id WHERE u.id = ? LIMIT 1

SELECT a.id AS a_id, city.id AS city_id, city.name AS city_name FROM address a LEFT JOIN city city ON a.city = city.id WHERE a.id = ? LIMIT 1
```

Листинг 8

3.6 Управление схемой базы данных

Для управления схемой базы данных используется библиотека Doctrine DBAL. Описание схемы можно сделать двумя различными способами:

Первый способ требует, чтобы объекты бизнес-логики реализовывали статический метод *describe(Granula\Meta)*. В данный метод передаётся экземпляр класса *Granula\Meta,* который и служит описанием мета данных объектной модели и схемы базы данных. Пример описания схемы БД первым способом представлен в Листинге 8.

```
namespace Model;

use Granula\Meta;
use Granula\ActiveRecord;
use Granula\Setter;

class User
{
    use ActiveRecord;
    use Setter;

    private $id;
    private $name;
    private $password;
    private $email;
    private $avatar;
    private $profile;
    private $friend;
    private $date;

    public static function describe(Meta $meta)
    {
        $meta->table('users');
        $meta->field('id', 'integer')->primary()->options(['autoincrement' => true]);
        $meta->field('name', 'string');
        $meta->field('password', 'string');
        $meta->field('email', 'string')->unique()->options(['notnull' => true]);
        $meta->field('avatar', 'string')->options(['notnull' => false, 'default' => '']);
        $meta->field('profile', 'entity')->entity(Profile::class);
        $meta->field('friend', 'entity')->entity(User::class);
        $meta->field('date', 'datetime');
        $meta->index(['name', 'email'], 'name_email_index');
    }
}
```

Листинг 9

Экземпляр класса *Granula\Meta* позволяет описать схему базы данных: имя таблицы, поля, типы полей, опции, отношения и индексы.

Второй способ использует рефлексию классов PHP [6]. Библиотека Granula ORM анализирует названия полей, комментарии к ним, и выполняет построение мета данных схемы *Granula\Meta*, как если бы они были описаны в методе *describe*. Пример описания схемы БД вторым способом продемонстрирован в Листинге 9.

```
namespace Model;

use Granula\Meta;
use Granula\ActiveRecord;
use Granula\Setter;

/**
 * Class User
 * @property int $id
 * @property string $name
 * @property string $password
 * @property string $email
 * @property string $avatar
 * @property Profile $profile
 * @property User $friend
 * @property DateTime $date
 */
class User
{
    use ActiveRecord;
    use Setter;

    private $id;
    private $name;
    private $password;
    private $email;
    private $avatar;
    private $profile;
    private $friend;
    private $date;
}
```

Листинг 10

Granula ORM работает в двух режимах - develop и production. В режиме develop библиотека получает информацию о структуре БД, структуре схемы

данных объектной модели, выполняет сравнение полученных схем, после чего генерирует SQL-код, выполняющий изменение схемы БД для соответствия объектной модели.

3.7 Загрузка объекта по первичному ключу

Для загрузки объектов из БД по первичному ключу реализован метод *find(int)*.

Перед генерацией и выполнением SQL кода проверяется, не был ли этот объект загружен ранее. Таким образом, удается избежать повторных запросов и появления копий одного и того же объекта. Управление и хранение объектов осуществляется классом Granula*EntityManager*.

Если объект не содержится в Granula*EntityManager*, то выполняется генерация SQL кода для получения всех полей этого объекта из базы данных. Если этот объект имеет связи с другими объектами, SQL код будет дополнен JOIN конструкциями для загрузки этих объектов. Таким образом, удается избежать лишних запросов к БД при обходе дерева объектов.

3.8 Преобразование типов данных

При работе с БД возникает проблема преобразования типов данных. Для ее решения используются преобразователи типов данных, вызываемые при каждом прямом и обратном отображении. Поддерживаемые типы данных:

- integer
- float
- string
- text
- binary
- blob
- boolean
- datetime

- array
- object

Пользователь может описать свой тип данных. Пример описания пользовательского типа данных «деньги» представлен в Листинге 10.

```
namespace Model\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

// Пользовательский тип данных представляющий "деньги".
class MoneyType extends Type
{
    const name = 'money'; // Название типа данных.

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        return 'MyMoney';
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return new Money($value);
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        return $value->toDecimal();
    }

    public function getName()
    {
        return self::name;
    }
}

Type::addType(MoneyType::name, MoneyType::class);
```

Листинг 11

4 ОПИСАНИЕ СЦЕНАРИЕВ ИСПОЛЬЗОВАНИЯ

4.1 Конфигурация и подключение в базе данных

```
$params = [
    'dev' => true,
    'driver' => 'pdo_mysql',
    'host' => 'localhost',
    'user' => 'root',
    'password' => '',
    'dbname' => 'granula',
    'charset' => 'utf8'
];

$em = new EntityManager($params, [
    User::class,
    Profile::class,
]);
```

Листинг 12

В листинге 11 приведен пример конфигурации, в котором указывается:

- драйвер БД,
- параметры подключения к БД,
- режим работы библиотеки.

Если указан режим работы «develop», то будет произведёна синхронизация схем БД и объектной модели. Если схема БД отсутствует, будет сгенерирован SQL код для создания этой схемы.

В класс *EntityManager* необходимо передать список объектов объектной модели.

4.2 Создание и сохранение объектов

```
$profile = new Profile();
$profile->age = 21;
$profile->tags = ['one', 'two'];
$profile->birth = new DateTime();
$profile->city = 'Saint Petersburg';
$profile->create();

$user = new User();
$user->name = 'Anton';
$user->avatar = null;
$user->profile = $profile;
$user->friend = User::lazy($friendId = 12);
$user->create();
```

Листинг 13

Листинг 13 демонстрирует создание двух объектов и сохранение их в базе данных. Сначала создается, заполняется и сохраняется объект «Profile». Затем создаётся объект «User», с ссылкой на объект «Profile», и при помощи метода *lazy* создаётся и присваивается ссылка на несуществующий объект другого пользователя.

После выполнения этого кода будут сгенерированы и выполнены следующие SQL запросы:

```
INSERT INTO profile (id, city, birth, age, tags) VALUES (null, "Saint Petersburg", "2014-04-14 20:03:56", 21, "one,two")
INSERT INTO users (id, name, avatar, profile, friend) VALUES (null, "Anton", null, 10, 12)
```

Листинг 14

4.3 Получение пользователя по первичному ключу

```
$user = User::find(1);
$user->profile->tags = ['one', 'two'];
$user->profile->save();
```

Листинг 15

Листинг 14 демонстрирует загрузку объекта «пользователь» по первичному ключу. Объект «профиль» будет загружен вместе с объектом «пользователь», без дополнительного запроса к БД.

В листинге 15 приведены SQL-запросы, сгенерированные для данного кода.

```
SELECT u.id AS u_id, u.name AS u_name, u.password AS u_password, u.email AS u_email, u.avatar
AS u_avatar, u.profile AS u_profile, u.friend AS u_friend, profile.id AS profile_id,
profile.city AS profile_city, profile.birth AS profile_birth, profile.age AS profile_age,
profile.tags AS profile_tags, friend.id AS friend_id, friend.name AS friend_name,
friend.password AS friend_password, friend.email AS friend_email, friend.avatar AS
friend_avatar, friend.profile AS friend_profile, friend.friend AS friend_friend FROM users u
LEFT JOIN profile profile ON u.profile = profile.id LEFT JOIN users friend ON u.friend =
friend.id WHERE u.id = 1 LIMIT 1;
UPDATE profile SET tags = "one,two" WHERE id = 10;
```

Листинг 16

4.4 Выполнение пользовательского SQL запроса

```
$result = User::query('SELECT * FROM users u WHERE u.id IN (?)', [[1, 2]],
[Connection::PARAM_INT_ARRAY]);

foreach ($result as $user) {
    // Работа с экземпляром $user.
}
```

Листинг 17

Листинг 16 демонстрирует выполнение пользовательского SQL запроса к БД. Метод *query(string, array, array, mixed)* принимает SQL запрос, список

параметров, типы параметров и функцию, реализующую отображение строки в объект. В данном примере функция отображения не передаётся. Поэтому она будет создана автоматически по мета-информации об объекте.

Метод *query* возвращает итератор (созданный при помощи генератора), который может быть обойдён в цикле.

Полученный SQL запрос представлен на следующем листинге.

```
SELECT * FROM users u WHERE u.id IN (1,2)
```

Листинг 18

4.5 Использование собственной функции отображения строки в объект

```
$users = User::query('SELECT * FROM users u WHERE u.id > ?',
 [1],
 [\PDO::PARAM_INT],
 function ($result) {
    $user = new User();
    $user->id = $result['id'];
    $user->name = $result['name'];
    $user->email = $result['email'];
    $user->profile = Profile::lazy($result['profile']);
    return $user;
});

foreach($users as $user) {
    // Работа с экземпляром $user.
}
```

Листинг 19

В данном примере в функцию *query* передаётся пользовательская функция отображения. В неё передаются результаты запроса к БД. Функция может вернуть любой результат, он будет передан в итератор, возвращаемый методом *query*.

5 ЗАКЛЮЧЕНИЕ

В результате данной дипломной работы реализована библиотека Granula ORM на PHP, обладающая следующими преимуществами, по сравнению с ближайшими аналогами:

1. Простота использования;
2. Быстрота изучения;
3. Достаточно широкая функциональность;
4. Следование стандартам PSR;

В библиотеке реализованы следующие возможности:

1. Автоматическая генерация схемы БД;
2. Два типа конфигурации объектной модели: DocBlock и PHP;
3. API для работы с объектной моделью;
4. API для работы с построителем запросов;
5. Преобразование типов данных;
6. Ассоциации «один к одному» и «один ко многим»;
7. Ленивая и жадная загрузка;

В настоящий момент библиотека Granula ORM доступна на GitHub по адресу https://github.com/elfet/granula и успешно используется в следующих проектах:

1. Веб-чат ElfChat: http://elfchat.net
2. Магазин ElfChat.

6 СПИСОК ЛИТЕРАТУРЫ

[1] R. Johnson, J2EE Design and Development, Wrox, 2002, pp. 255-256.

[2] S. D. U. Suzanne W. Dietrich, Fundamentals of Object Databases: Object-Oriented and Object-Relational Design, Morgan & Claypool Publishers, 2011, pp. 66-100.

[3] E. Bernard, «The Object-Relational Impedance Mismatch,» [В Интернете]. Available: http://hibernate.org/orm/what-is-an-orm/#the-object-relational-impedance-mismatch.

[4] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002.

[5] K. Dunglas, Persistence in PHP with Doctrine ORM, Packt Publishing, 2013.

[6] «PHP: Reflection,» [В Интернете]. Available: http://www.php.net/manual/ru/book.reflection.php.