# Урок №6

## Создание класса модели. Драйвер БД

В этом уроке

- Вы узнаете, как перейти с процедурной модели на объектно-ориентированную;
- как сделать это правильно, используя паттерн проектирования «Singletoon»;
- поймете, какими преимуществами данный паттерн обладает в сравнении со статическими классами;
- изучите драйвер для работы с базой данных, позволяющий существенно сократить время написания запросов к БД

### 1. Преобразование модели от процедурного подхода к ООП.

Переход к объектно-ориентированному подходу осуществляется достаточно просто:

<table>
<tr><th>Процедурный подход</th><th>ООП</th></tr>
<tr><td colspan="2">Модель</td></tr>
<tr><td>

```
function articles_all()
function articles_get()
function articles_new()
function articles_edit()
function articles_delete()
```

</td><td>

```
class M_Articles
{
    public function All()
    public function Get()
    public function Add()
    public function Edit()
    public function Delete()
}
```

</td></tr>
<tr><td colspan="2">Контроллер</td></tr>
<tr><td>

```
class C_Articles extends C_Base
{
    public function action_index()
    {
        ...
        // обращение к модели
        $this->articles = articles_all();
        ...
    }
}
```

</td><td>

```
class C_Articles extends C_Base
{
   public function action_index()
   {
      // создание экземпляра модели
      $mArticles = new M_Articles();
      // обращение к модели
      $this->articles = $mArticles->All();
        ...
   }
}
```

</td></tr>
</table>

По сути, мы просто добавили слово class, модификаторы доступа и немного изменили обращение к методам модели в контроллере. Вроде бы всё очень просто. Однако в данной реализации кроется небольшой и труднонаходимый минус.

Если мы внимательно посмотрим на класс M_Articles, то поймём, что нам нет никакого смысла создавать более одного его экземпляра, так как объекты будут получаться абсолютно одинаковыми. С другой стороны, ситуация использования модели для работы со статьями в разных контроллерах достаточна вероятна на реальном сайте. Например, в левый блок, который относится к базовому шаблону, необходимо выводить пять свежих записей. Получается, что контроллер C_Base должен просить у модели пять последних статей, т.е., создавать экземпляр класса M_Articles. В тоже время его наследник C_Articles может просить у этой же модели одну статью, чтобы отобразить её в центральной части сайта.

Выходит, что в рамках одного запроса будут созданы два экземпляра M_Articles, хотя хватило бы и одного. Это может показаться не такой уж и страшной проблемой, однако в крупном проекте таких моделей будет намного больше, а сайт, возможно, будет высоконагруженным. В такой ситуации следует избегать любого необдуманного действия, бессмысленно напрягающего систему.

Ниже мы рассмотрим два способа избежать создания ненужных экземпляров класса.

## 2. Статический класс

Немного напомним про то, что такое статические методы и классы.

**Статические** методы и свойства – те, которые принадлежат классу, а не его экземплярам.

**Статический** класс – тот, у которого есть только статические методы и свойства.

Создание статического класса модели позволит нам не плодить его экземпляры, а просто обращаться к необходимым методом напрямую, без создания объекта:

| Обычный класс модели | Статический класс модели |
|---|---|
| Модель | |
| `class M_Articles`<br>`{`<br>`  public function All()`<br>`  public function Get()`<br>`  public function Add()`<br>`  public function Edit()`<br>`  public function Delete()`<br>`}` | `class M_Articles`<br>`{`<br>`  public static function All()`<br>`  public static function Get()`<br>`  public static function Add()`<br>`  public static function Edit()`<br>`  public static function Delete()`<br>`}` |

| Контроллер | |
|---|---|
| `class C_Articles extends C_Base`<br>`{`<br>`  public function action_index()`<br>`  {`<br>`  // создание экземпляра модели`<br>`  $mArticles = new M_Articles();`<br>`  // обращение к модели`<br>`  $this->articles = $mArticles->All();`<br>`  ...`<br>`  }`<br>`}` | `class C_Articles extends C_Base`<br>`{`<br>`  public function action_index()`<br>`  {`<br>`  // обращение к модели`<br>`  $this->articles = M_Articles::All();`<br>`  ...`<br>`  }`<br>`}` |

Что же, основную задачу мы решили, однако, у статического класса есть определённые минусы. Главный из них – невозможность наследования. Это происходит по той причине, что мы не создаёт объекта.

Очень хорошо, если сразу возник вопрос, а зачем нам вообще наследовать от кого-то модель. На самом деле, это имеет очень большой смысл, когда сущностей станет много. Например, помимо статей у нас появляются комментарии, контент-страницы, пользователи и т.п. Во всех моделях обязательно будет набор из пяти стандартных функций:

1. All
2. Get
3. Add
4. Edit
5. Delete

Методы Add и Edit имеют существенное отличие для разных сущностей, так как ориентируются на их поля в базе. А вот методы All, Get и Delete зачастую реализуют очень похожие запросы к БД, которые отличаются только именем таблицы и названием первичного ключа.

В связи с этим можно создать абстрактный класс модели, которая будет определять в себе реализацию данных методов. Все остальные модели, которые относятся к конкретным сущностям, наследуются от базовой и переопределяют имя таблицы и название первичного ключа. Благодаря такому подходу, в большинстве случаев мы сможем забыть про написание трёх стандартных методов, названных выше.

Подобных схем, которые дают преимущества с помощью наследования, можно придумать много. Поэтому, более гибким решением является использование не статического класса, а шаблона проектирования «Singleton».

## 3. Шаблон проектирования «Singleton»

<table>
<tr><th>Обычный класс модели</th><th>Singleton</th></tr>
<tr><td colspan="2">Модель</td></tr>
<tr><td>

```
class M_Articles
{
    public function All()
    public function Get()
    public function Add()
    public function Edit()
    public function Delete()
}
```

</td><td>

```
class M_Articles
{
    // ссылка на экземпляр класса
    private static $instance;
    // получение единственного экземпляра
    public static function Instance()
    {
        if (self::$instance == null)
        self::$instance = new M_Articles();

        return self::$instance;
    }
    // основные методы модели
}
```

</td></tr>
<tr><td>

```
class C_Articles extends C_Base
{
  public function action_index()
  {
    // создание экземпляра модели
    $mArticles = new M_Articles();
    // обращение к модели
    $this->articles = $mArticles->All();
        ...
  }
}
```

</td><td>

```
class C_Articles extends C_Base
{
  public function action_index()
  {
    // получение экземпляра модели
    $mArticles = M_Articles::Instance();
    // обращение к модели
    $this->articles = $mArticles->All();
        ...
  }
}
```

</td></tr>
</table>

Общая идея заключается в следующем: у класса создаётся одно статическое поле и один статический метод, с помощью которого мы сможем получить экземпляр данного класса. Функция Instance проверяет, был ли уже создан экземпляр класса: если нет – создаёт его и

записывает в поле $instance; если был – просто возвращает ссылку на объект из данного поля.

Паттерн Singleton позволяет решить проблему создания лишних экземпляров класса, не ущемляя при этом возможностей наследования.

## 4. Драйвер работы с БД

А теперь переходим к самому приятному. Предыдущая тема была посвящена принципам грамотного проектирования приложений и не несла в себе прямой выгоды для программиста. А вот создание некоторой надстройки для работы с базой данных позволит в дальнейшем существенно сократить код остальных моделей.

Общая идея заключается в том, что все SQL-запросы так или иначе похожи друг на друга. Давайте в этом убедимся.

1. INSERT INTO `таблица` (`столбцы через запятую`) VALUES (`значения через запятую`)
2. DELETE FROM `таблица` WHERE `первичный ключ` = число
3. UPDATE `таблица` SET `пары через запятую` WHERE `первичный ключ` = число

Операция SELECT отсутствуем в списке, так как выборки могут значительно отличаться при создании многотабличных запросов (JOIN). Однако, под остальные вышеперечисленные операции можно создать определённые функции-шаблоны, которые будут принимать недостающие параметры и подставлять их в основной каркас.

```
class M_MSQL
{
    // $query - текст sql-запроса
    // return - двумерный массив с данными выборки
    public function Select($query)

    // $table - имя таблицы
    // $object - ассоциативный массив. Ключи - имена столбцов.
    // return - id добавленной записи
    public function Insert($table, $object)

    // $table - имя таблицы
    // $object - ассоциативный массив. Ключи - имена столбцов.
    // $where - строка вида `первичный ключ` = `число`
    // return - количесвто изменённых строк
    public function Update($table, $object, $where)
```

```
    // $table - имя таблицы
    // $where - строка вида `первичный ключ` = `число`
    // return - количество удалённых строк
    public function Delete($table, $where)
}
```

Сама реализация данных методов является слишком громоздкой для помещения её сюда. Полный код модели M_MSQL и пример её использования находятся в исходниках в папке 5_database_driver.

## Самоконтроль

- ✓ В чём заключается смысл паттерна «Singletoon»
- ✓ Чем паттерн «Singletoon» лучше использования статических методов класса
- ✓ Зачем мы вводим драйвер для работы с базой данных
- ✓ Что принимает и что возвращает функция драйвера Select
- ✓ Delete
- ✓ Insert
- ✓ Update

## Домашнее задание

1. Привести модель к объектно-ориентированному подходу, используя паттерн «Singleton».
2. Внедрить в систему драйвер для работы с базой данных.
3. Добавить возможность написания комментариев к статьям. Под каждой статьёй должны отображаться предыдущие сообщения и форма для добавления нового. Авторизации для совершения данного действия не требуется, человек просто вводит своё имя и текст комментария.