Стиль написания кода в студии hipot studio

Последний раз редактировалось:

`24.07.2017 16:37 (вер. 3.1)`

Стандарт написания кода — это очень важно

Опыт многих проектов показывает, что при использовании стандартов кодирования управление и разработка идет легко и гладко. Но достаточно ли стандарта для успешного проекта? Конечно, нет, но они помогают. А мы должны использовать все инструменты, которые упрощают нам жизнь! Поверьте, многие протесты против стандарта кодирования исходят из чьего-то ущемленного самолюбия. Но для того, чтобы следовать стандартам необходимо минимальное усилие над собой, а также необходимо помнить, что любой проект — это коллективная работа.

Цели

Хороший стандарт кодирования важен в любом проекте, и особенно там, где множество разработчиков работают над одним проектом. Наличие стандарта кодирования помогает гарантировать, что код высокого качества, с меньшим количеством ошибок, и легко поддерживается. Правила форматирования кода должны быть едиными во всем проекте. И крайне желательно, чтобы они были очень похожими между проектами.

Содержание

#1. Форматирование PHP-файлов

1. Рекомендуемая длина строки составляет 120 символов, т.е. разработчики должны стремиться держать код как можно ближе к 120-символьной границе, когда это возможно. Однако более длинные строки также допустимы. Максимальная длина любой строки PHP-кода не должна превышать 200 символов.
2. Переводы строк должны быть как принято для текстовых файлов в Unix-системах. Строки должны заканчиваться только символом перевода на новую строку (`LF`). Символ перевода на новую строку в десятичном виде представляется как число 10, или как `0x0A` в шестнадцатеричном. Не используйте комбинацию символов возврата каретки/перевода строки (`CRLF`) как на Windows-компьютерах (`0x0D, 0x0A`). Многие редакторы кода позволяют управлять этой настройкой. *Причина*: старые FTP-серверы при передаче на них файлов с окончанием строк в `CRLF` преобразуют их в `LFLF`, в итоге при повторном открытии файлов мы имеем пустую строку после каждой строки кода.

Отступы

Использовать tab вместо пробелов, т.к. обычно редактор настраивается, сколько пробелов проставлять за tab. Делаем отступов столько, сколько необходимо, но не больше. Существует правило о максимуме числа отступов. Если вложенность в коде больше, чем 4 или 5 уровней, то следует задуматься о переработке такого кода. Конечно, это рекомендация, в реальной жизни уменьшить число вложений до 5ти не всегда возможно, поэтому будем надеяться на мудрость программиста при написании кода.

Подключение файлов: include или require?

В тех местах, где вы используете подключение файлов других классов вне зависимости от условий, используйте конструкцию `require_once()`.
Если же подключение файлов зависит от каких-либо условий, то следует использовать `include_once()`. В этом случае вы всегда будете уверены в том, что файлы подключаются только единожды.

`include_once()` и `require_once()` являются конструкциями, а не функциями, поэтому не обязательно использовать скобки вокруг имени файла, который подключается.

Использование `include()` и `require()` рекомендуется только в исключительных случаях: например, когда файл требуется подключить несколько раз.

#2. Соглашения по именованию

Хорошее именование в коде имеет определяющее значение при отладке, поиске ошибок и дальнейшей работе с кодом.

Имена таблиц и полей в БД

1. Имена наших таблиц БД должны начинаться с нашего корпоративного префикса `hi_`, быть в нижнем регистре, и могут содержать английские буквы, а также знак подчеркивания. Цифры в именах таблиц не рекомендуются. Напр.: `hi_brands, hi_action_cache,` ...
2. Имена полей в таблице БД должны быть в ВЕРХНЕМ_РЕГИСТРЕ, содержать английские буквы, а также знак подчеркивания для разделителя слов. Цифры в именах полей не рекомендуются. Напр. `DISCOUNT_ID, ACTION_CODE,` ...

Пример таблицы:

```
CREATE TABLE IF NOT EXISTS `hi_video_comments` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `USER_ID` int(11) NOT NULL,
  `PRODUCT_ID` int(11) NOT NULL,
  `CODE_VIDEO` text COLLATE utf8_unicode_ci NOT NULL,
  `DATE` datetime NOT NULL,
  `STATUS` varchar(1) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`ID`),
  KEY `PRODUCT_ID` (`PRODUCT_ID`,`STATUS`),
  KEY `PRODUCT_ID_2` (`PRODUCT_ID`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci COMMENT='Видео о товаре'
```

Имена файлов и структура папок с файлами для классов

1. Для имен файлов допустимы буквенно-числовые символы (нижнего регистра), символы нижнего подчеркивания и тире («-»). Пробелы запрещены. Имена файлов Бдолжны быть всегда **в нижнем регистре**; исключением могуть быть лишь имена простанства имен и файлов с классами, именуемыми по PSR-0 в `ПространствоИмен/ВерхниемКемелКейсе.php` (см. п.3).
2. Если файл содержит любой php-код, то он должен заканчиваться «.php»
3. Все классы нужно складывать по папкам в соответствии с пространством имен:

   ```
   namespace -> путь к файлу
   Hipot\Test\Test -> /local/php_interface/include/lib/classes/hipot/test/test.php
   Hipot\Utils\CImg -> /local/php_interface/include/lib/classes/hipot/utils/cimg.php

   // в верхнем регистре допустимо назвать папку с простанством имен и имя файла с классом
   Hipot\BitrixUtils\PhpCacher -> /local/php_interface/include/lib/classes/Hipot/BitrixUtils/PhpCacher.ph

   класс без пространства имен:
   hiTestRoot -> /local/php_interface/include/lib/classes/hitestroot.php
   ```

   ** заметьте, использовать нужно папку /local/, чтобы отделить классы без пространства имен на классы с ними.*

   Как видите, путь к файлу состоит из физических папок, определяющих пространство имен в нижнем регистре, а имя файла содержит имя класса, тоже в нижнем регистре.

Именование Пространств имен (Неймспейсов)

1. Наше базовое пространство имен Hipot. Все наши самописные классы должны лежать в папке `php_interface/include/lib/classes/hipot/<суть функционала>/`, Где <суть функционала> - это второе пространство имен в главном пространстве Hipot.

   Делать классы прямо в пространстве Hipot, следовательно они будут физически в папке php_interface/include/lib/classes/hipot/, можно, но не рекомендуется.

2. пространства имен нужно именовать через ВерхнийКемелКейс (как и имена классов).
3. Имена пространств имен классов могут содержать только буквенно-числовые символы. Числа допустимы в именах классов, но не приветствуются. Остальные символы использовать нельзя.

Именование Классов

1. Имена классов нужно писать через ВерхнийКемелКейс (как и пространства имен). Если этот класс лежит в корне пространства имен, то тогда его нужно именовать через префикс `hi`: `hiВерхнийКемелКейс` (см. п.5 ниже)
2. Имена классов могут содержать только буквенно-числовые символы. Числа допустимы в именах классов, но не приветствуются.
3. Если имя класса состоит из более, чем одного слова, то первая буква каждого слова должна быть заглавной. Последующие заглавные буквы недопустимы, например, имя класса «`WorkPDF`» — недопустимо, в то время как имя «`WorkPdf`» допустимо.
4. *Разъяснение 3)*: столкнувшись с ситуацией, когда необходимо использовать все верхние буквы аббревиатуры в названии, пишем первую прописную, а остальные строчные. Обоснование: Возьмем, к примеру, `NetworkABCKey`. Обратите внимание, как легко можно неверно понять аббревиатуру `ABCK`, и не заметить сразу, что далее идет `Key`. В данном случае, верно назвать `NetworkAbcKey`. *Например*: Используем `HtmlStatistic` вместо `HTMLStatistic`.
5. Для имен классов БЕЗ пространства имен, чтобы защититься от коллизий, нужно добавлять префикс ко всем именам классов в глобальном пространстве имен. Нужно использовать 2х буквенный префикс. Наш корпоративных префикс имен классов — `hi`. Напр, имя класса `hiAbstractBaseAdapter`.
   В случае, если этот класс будет лежать в пространстве имен (а это весьма приветствуется), то префикс hi писать не нужно, т.к. класс и так будет лежать в пространстве имен `Hipot: Hipot\DB\AbstractBaseAdapter`
6. Особые случаи именования классов применительно платформы битрикс
   - Имя сущности `HiloadBlock` требует битрикс, чтобы было с большой буквы (имя класса). Для хайлоад-блоков жетально Hi-префикс: `HiВерхнийКемелКейс`. Разрешено и просто `ВерхнийКемелКейс`, но не приветствуется
   - для классов компонент нужно проставлять суффикс `Component`, напр. `hiВерхнийКемелКейсComponent`
   - для всех остальных классов: `hiВерхнийКемелКейс` или в пространстве имен без префикса `Hipot: Hipot\DB\AbstractBaseAdapter`

Именование интерфейсов

1. Интерфейсы должны следовать той же схеме именования, как и классы (смотрите выше), однако их имя должно заканчиваться словом «`Interface`», как в следующих примерах: `WorkPdfInterface`, `ControllerDispatcherInterface`

Именование функций и методов

1. Имена функций могут содержать буквенно-числовые символы. Символы нижнего подчеркивания не разрешены. Числа разрешены в именах функций, но не приветствуются.
2. Имена функций должны всегда начинаться с буквы в верхнем регистре. Когда имя функции состоит из более, чем одного слова, первая буква каждого нового слова должна быть заглавной. Это обычно называется «верблюжьей» («**CamelCase**») нотацией. Обычно, метод или функция выполняют какое-либо действие, поэтому имя такого метода или функции должно указывать на это действие: `CheckForErrors()` вместо `ErrorCheck()`, `DumpDataToFile()` вместо `DataFile()`.
3. Многословность приветствуется. Имена функций должны быть настолько говорящими, насколько это практично для повышения понимаемости кода.

4. Если необходимо в функции что-то выяснить, то удачно дать ей имя с префиксом `Is`. Напр. `IsAuthorized()`
5. Функции в глобальной области видимости («плавающие функции») допустимы, но не приветствуются. Рекомендуется обрамлять такие функции в статические классы. Написание статических методов формирует библиотеку классов, которую можно будет повторно использовать.
6. Если функция является рекурсивной, то у нее должен быть суффикс «`_r`» (напр. `WalkBsp_r()`)

Именование методов в классах

1. Имена методов, в отличие от функций, могут начинаться с буквы в нижнем регистре. Когда имя метода состоит из более, чем одного слова, первая буква каждого нового слова должна быть заглавной. Это обычно называется «верблюжьей» нотацией.
2. Для объектно-ориентированного программирования принято, чтобы методы доступа имели префикс «`get`» или «`set`» (если необходимо что-либо вернуть, либо установить)
3. Для методов, объявленных как `private` или `protected` первый символ можно сделать нижним подчеркиванием, для обращения внимания на скрытую область видимости.

Именование переменных

1. Имена переменных могут содержать буквенно-числовые символы. Символы нижнего подчеркивания не разрешены (см. п.5). Числа разрешены в именах переменных, но не приветствуются
2. Как и имена методов (смотрите выше) имена переменных должны начинаться с буквы в нижнем регистре и следовать «верблюжьей» нотации.
3. Для переменных — членов классов, определенных с помощью префиксов области видимости «`private`» или «`protected`», первый символ имени должен быть один символ нижнего подчеркивания. Это единственное допустимое использование символа нижнего подчеркивания в имени. Переменные — члены классов определенные с помощью префикса
области видимости «`public`» никогда не должны начинаться с символа нижнего подчеркивания.
4. Многословность приветствуется. Имена переменных должны быть настолько говорящими, насколько это практично. Краткие имена переменных, такие как «`$i`» и «`$n`» не приветствуются нигде, кроме как в контексте маленьких циклов. Если цикл содержит более 20 строк кода, то переменные для индексов должны иметь более говорящие имена.
5. Имена переменных, содержащие только нижний регистр и знак подчеркивания, разрешается использовать только в локальных частях кода, содержащего не более 20 строк. В противном случае переменной необходимо давать осмысленное название.
6. Встроенные переменные PHP `true`, `false` и `null` должны быть написаны в нижнем регистре.
7. Использование говорящих префиксов/суффиксов — это хорошо: Например, `Max`— обозначает, что переменная хранит какой-либо максимум, `Cnt, Count` - обозначает, что переменная хранит кол-во чего-либо. Например: `$itemsMax` — максимальное значение в массиве; `$dateOfSomething` — дата какого-либо события; `$useHtml, $isHtml` — для флагов.

Префиксы имен переменных для удобочитаемости

Использование префиксов является строго рекомендуемым, однако в частных случаях разрешено отступать от них. PHP — не особо типизированый язык и в нем различаются по смыслу три группы типов: скалярные, массивы и объекты. Скалярные типы следует начинать с префиксов всегда, если точно известно, что они имеют заданный тип:

1. '`ar`' — Массивы
2. '`ob`', '`o`' — Объекты
3. '`b`' — тип boolen
4. '`g`' — глобальные переменные
5. '`db`' — дескриптор результата БД (напр. при
возврате в bitrix объекта `CDBResult`)
6. '`res`', '`rs`' — ресурс (напр.
дескриптор открытого файла)
7. '`r`' — для параметров функции, используемых как ссылка (см. пример). Тогда при написании кода в функции мы

наглядно знаем, меняет ли функция переданную переменную.

```
global $bTrace = true;
global $gLog;

$dbresRelatentItems = '';

class Test
{
    function DoSomething(&$rStatus)
    {
    }
}
```

Именование констант

1. Константы могут содержать буквенно-числовые символы и символы нижнего подчеркивания. Числа разрешены в именах констант.
2. Имена констант должны быть в верхнем регистре.
3. Имена констант из нескольких слов пишутся, разделяя каждое слово знаком подчеркивания (напр. `EMBED_SURPRESS`)
4. Константы должны быть определены как члены классов с использованием ключевого слова «`const`». Определение констант в глобальной области видимости с помощью «`define`» допустимо, но не рекомендуется:

   ```
   // можно
   define('MY_CONSTANT', 'foo/bar');

   // но лучше так:
   class CoffeeParams
   {
    const MY_CONSTANT = true;
   }
   var_dump(CoffeeParams::MY_CONSTANT);
   ```

#3. Стиль кодирования

Строковые литералы

1. Когда строка является литеральной (не содержит подстановок переменных), для ее обрамления должны использоваться апострофы или «одинарные кавычки» (`$a = 'Example String';`)
2. Когда строка литералов сама содержит апострофы, разрешается для обрамления строки использовать «двойные кавычки». Это особенно актуально для SQL-запросов:

   ```
   $sql = "SELECT `id`, `name` FROM `people` WHERE `name` = 'Fred' OR `name` = 'Susan'";
   ```

Конкатенация строк

1. Строки должны объединятся с помощью оператора «`.`». Пробел должен всегда добавляться до и после оператора «`.`» для улучшения читабельности: (`$company = 'Zend' . 'Technologies';`)
2. Когда производится конкатенация строк с помощью оператора «`.`», разрешается разрывать выражение на несколько строк для улучшения читабельности. В этом случае, каждая следующая строка должна быть дополнена пробелами так, чтобы оператор «`.`» был выровнен под оператором «`=`»:

   ```
   $sql = " SELECT `id`, `name` FROM `people` "
   ```

```
        . " WHERE `name` = 'Susan' "
        . " ORDER BY `name` ASC ";
```

Массивы с числовыми индексами

1. Хотя индекс массива может начинаться с отрицательного числа, но это не приветствуется и рекомендуется, чтобы все массивы начинали индексирование с `0`.
2. Когда определяется индексированный массив с помощью конструкции `array`, завершающий пробел должен быть добавлен после каждой запятой для улучшения читабельности:

```
$sampleArray = array(1, 2, 3, 'BEST', 'PHP IDE');
```

3. Также разрешается определять многострочные индексированные массивы, используя конструкцию «`array`». В этом случае, каждая следующая строка должна быть дополнена пробелами так, чтобы начало каждой строки было выравнено как показано ниже:

```
$sampleArray = array(
    1,      2,     3,    'BEST',    'PHP IDE',
    $a,     $b,    $c,
    56.44,  $d,    500
);
```

Ассоциативные массивы

1. Когда определяется ассоциативный массив с помощью конструкции «`array`», приветствуется разделение выражения на несколько строк. В этом случае, каждая следующая строка должна быть дополнена с помощью пробелов так, чтобы и ключи и значения были выровнены:

```
$sampleArray = array('firstKey'  => 'firstValue',
                     'secondKey' => 'secondValue');

// а лучше даже так
$sampleArray = array(
    'firstKey'           => 'firstValue',
    'secondKey'          => 'secondValue',
    'longLongLongKey'    => 'secondValue'
);
```

Пробелы вокруг сложных индексных выражений

В случае, если Вы обращаетесь к элементу массива по индексу и индексное выражение достаточно сложное, отделяйте его пробелами для улучшения удобочитаемости. Если выражение простое — пробелы не обязательны.

```
$arTmp['KEY'] = 0;
$arItems[$formId][ $arPage["VALUE"] * 2 - 4 ] = $arTmp;

// или даже так
$arItems[$formId][ ($arPage["VALUE"] * 2 - 4) ] = $arTmp;
```

Определение класса

Классы должны определяться по следующей схеме:

1. Фигурная скобка всегда пишется на следующей строке под именем класса.
2. Каждый класс должен иметь блок документации (doc-блок) в соответствии со стандартом PHPDocumentor.

3. Код внутри класса должен иметь отступ.
4. Все классы нужно складывать по папкам в соответствии с пространством имен:

```
namespace -> путь к файлу
Hipot\Test\Test -> /local/php_interface/include/lib/classes/hipot/test/test.php
Hipot\Utils\CImg -> /local/php_interface/include/lib/classes/hipot/utils/cimg.php

// в верхнем регистре допустимо назвать папку с простанством имен и имя файла с классом
Hipot\BitrixUtils\PhpCacher -> /local/php_interface/include/lib/classes/Hipot/BitrixUtils/PhpCacher.ph

класс без пространства имен:
hiTestRoot -> /local/php_interface/include/lib/classes/hitestroot.php
```

Каждый файл с классом должен содержать первой строкой: namespace Hipot\<суть функционала>;
И не рекомендуется в одном файле определять больше одного класса.

Пример:

```
namespace Hipot\Test;

/**
 * phpDoc-блок здесь с описанием класса
 * обратите внимание на путь \CIblockElement -
 * его обязательно использовать при записи namespace
 */
class SampleClass extends \CIblockElement
{
    // содержимое класса должно быть
    // с отступом в один tab
}
```

Использование классов, определенных через пространство имен

Само определение классов через пространства имен предполагает автозагрузку. Поэтому, в любом другом месте определенные нами классы можно использовать следующим путем:

```
// первый путь использования:
// чтобы не писать длинные записи при использовании классов
use Hipot\Test as hiT;
$t = new hiT\Test();
hiT\Test::test_static();

// второй путь использования (полный путь к классу):
$t = new \Hipot\Test\Test();
\Hipot\Test\Test::test_static();

// третий путь использования (не рекомендую, сразу создаем ссылку на класс в
// глобальной области видимости)
use Hipot\Test\Test as Test;
$t = new Test();
Test::test_static();
```

Обратите внимание на именование псевдонимов простанства имен:

```
/** мы указываем псевдоним пути как we + первая буква подпростансва имен T(est):
    получаем weT. */
use Hipot\Test as hiT;
```

```
use Hipot\Utils as hiU;
hiU\CImg::Resize(…); // CImg is deprecated
```

Переменные-члены классов

Переменные-члены классов должны определяться по следующей схеме:

1. Имена переменных-членов класса могут именоваться, как обычные переменные (см. именование переменных выше)
2. Любые переменные, определенные в классе, должны быть определены в начале класса, до определения любого метода.
3. Ключевое слово `var` не рекомендуется. Желательно всегда определять область видимости членов, используя ключевое слово `private`, `protected` или `public`.
4. Доступ к переменным-членам класса напрямую используя префикс `public` разрешено, но не приветствуется в пользу методов доступа (`set/get`)

Определение функций и методов

Функции должны определяться по следующей схеме:

1. Функции должны именоваться согласно правилам именования (см. раздел именование функций и методов)
2. Функции внутри классов (методы) должны всегда определять свою область видимости с помощью одного из префиксов `private`, `protected` или `public`.
3. Как и у классов, фигурная скобка всегда пишется на следующей строке под именем функции. Пробелы между именем функции и круглой скобкой для аргументов отсутствуют. («**one true brace**» форма)
4. Функции в глобальной области видимости крайне не приветствуются.
5. Аргументы функций со значениями по умолчанию должны находиться в конце списка аргументов.
6. Передача по ссылке во время вызова запрещена. Передача по ссылке допустима только в случае явного определения такого поведения:

```
/**
 * Doc-блок здесь
 */
class Foo
{
    /**
     * Doc-блок здесь
     */
    public function bar(&$baz)
    {
    }
}
```

7. Возвращаемое значение не должно обрамляться в круглые скобки, иначе это ухудшает читабельность, а также может поломать код, если метод позже станет возвращать результат по ссылке.

```
/**
 * Doc-блок здесь
 */
class Foo
{
    /**
     * ПЛОХО
     */
    public function bar()
```

```
    {
        return($this->bar);
    }

    /**
     * ХОРОШО
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

Использование функций и методов

Функции должны определяться по следующей схеме:

1. Аргументы функции разделяются одним завершающим пробелом после каждой запятой.
2. Вызовы функций должны быть написаны без отступов между именем функции, открывающей скобкой и первым параметром. Отступы в виде пробела должны присутствовать после каждой запятой в перечислении параметров (`$var = foo($bar, $baz, $quux);`)
3. Передача по ссылке во время вызова запрещена. Смотрите секцию определения функций для правильного способа передачи аргументов функции по ссылке.
4. Для функций, чьи аргументы допускают массив, вызов функции может включать конструкцию «`array`» и может быть разделено на несколько строк для улучшения читабельности. В этом случае, применим стандарт описания массивов:

```
threeArguments(array(1, 2, 3), 2, 3);

moreThreeArguments(
    array(1, 2, 3, 'Best', 'PHP IDE', $a, $b, $c, 56.44, $d, 500),
    2, 3,
    array(
        array(1, 2),
        array($a, $b, $c),
    )
);
```

Управляющие структуры и простановка скобок

Управляющие структуры включают в себя операторы `if`, `for`, `while`, `switch`, и др. Ниже приведен пример оформления оператора `if` , который в этом отношении является самым сложным. Его и рассмотрим, другие пишутся по аналогии. Этот стиль называется «**trailing braces**», его использовать во всех управляющих конструкциях.

1. Управляющие структуры, основанные на конструкциях `if` и `elseif`, должны иметь один пробел до открывающей круглой скобки условия, и один пробел после закрывающей круглой скобки.
2. Внутри выражения условия между круглыми скобками операторы должны разделяться пробелами для читабельности. Внутренние скобки приветствуются для улучшения логической группировки больших условий.
3. Открывающаяся фигурная скобка пишется на той же строке, что и условие. Закрывающаяся фигурная скобка пишется на отдельной строке. Все содержимое между скобками пишется с отступом в один tab.

```
if ($a != 2) {
    $a = 2;
}
```

4. Для выражения `if`, включая `elseif` или `else`, форматирование должно быть таким, как в следующем примере:

```
if ($a != 2) {
    $a = 2;
} else {
    $a = 7;
}

if ($a != 2) {
    $a = 2;
} else if ($a == 3) {
    $a = 4;
} else {
    $a = 7;
}
```

5. Ключевые слова должны быть строго в нижнем регистре: `array, if, foreach, while, true, false, null, new, class, function, ...`

PHP допускает написание таких выражений без фигурных скобок при некоторых условиях. Стандарт кодирования не делает различий — для всех `if`, `elseif` или `else` выражений необходимо использовать фигурные скобки. Настойчиво рекомендуется использовать фигурные скобки, даже в том случае, когда их использование не является необходимостью. Использование фигурных скобок увеличивает читабельность кода и уменьшает вероятность логических ошибок при изменении кода.

```
/* плохо - нет скобок, плохое расположение statement
   (каждый оператор на своей строке) */

if (expr) statement;

// плохо - нет скобок
if (expr)
    statement;

// хорошо
if (expr) {
    statement;
}
```

Использование комбинации `else if` вместо конструкции `elseif` допускается.

Стиль «**one true brace**» (каждая скобка на новой строке) нужно использовать только в декларации классов и функций. В управляющих конструкциях его использовать нельзя:

```
namespace Wexpert\Test;

/**
 * Внимание на простановку скобок!
 */
class Foo
{ // скобка на новой строке, т.к. определение класса!

    /**
     * Определение метода в классе
     */
    static function bar($ok = true)
    { // скобка на новой строке, т.к. определение метода!

        if (! $ok) { // скобка на этой же строке, управляющая конструкция
```

```
            return false;
        } else {
            return true;
        }
    }
}


// обратите внимание на простановку скобок в этом случае
if (! function_exists('bar')) { // управляющая конструкция, скобка на этой же строке
   function bar()
   { // опередделение функции/метода или класса – скобка на новой строке
      return $this->bar;
   }
}
```

Альтернативный синтаксис рекомендуется использовать только в частях, которые используют прерывания на вывод html-кусков (напр. для шаблонов вывода).

В коде, содержащем только PHP, не рекомендуется использовать альтернативный синтаксис, т.к. теряется учет открытых-закрытых скобок (к тому же многие среды разработки позволяют легко найти открывающуюся скобку по закрытой, но не позволяют найти начало `if (…):` по его окончанию — `endif;`).
Пример:

```
<? if ($a != 2):?>
    <tr class="some" >
        <td><?=$var?></td>
    </tr>
<? elseif ($a == 3):?>
    <tr>
        <td><?=$varFix?></td>
    </tr>
<? endif ;?>
```

Правила написания switch-конструкции

1. Управляющие структуры написанные с использованием «`switch`» конструкции должны иметь один пробел до открывающей круглой скобки
   условного выражения, и также один пробел после закрывающей круглой скобки.
2. Все содержимое между фигурными скобками пишется с отступом. Содержимое каждого «`case`» выражения также должно писаться с отступом
3. Ключевое слово `default` никогда не должно опускаться в выражении `switch`.
4. *ЗАМЕЧАНИЕ*: Иногда полезно писать case-выражения, которые передают управление следующему case-выражению, опуская `break` или `return`. Для того, чтобы отличать такие случаи от ошибок, каждое case-выражение, где опущен `break` или `return`, должно содержать комментарий (напр. «`// Выполняем код следующего case`»).

```
switch (...) {
    case 1:
        ...
        // Выполняем код следующего case

    case 2:
        $v = GetWeekNumber();
        ...
        break;
```

```
    default:
        break;
}

// а еще лучше каждый case тоже оформить в скобки:
switch (...) {
    case 2: {
        $v = GetWeekNumber();
        ...
        break;
    }
}
```

Написание анонимных фукнций

Анонимные функции необходимо писать в стиле скобка-на-той-же-строке «trailing braces»:

```
// обратите внимание, как use я передаю в итерацию возможно необходимые переменные
array_walk($arraySlugs, function(&$element) use ($suffixName, $suffixName2) {
    $element    = preg_replace('/[^A-Za-z0-9-]+/', '-', $element);
});

array_walk($arraySlugs, function(&$element) {
    //…
});

// создание анонимной функции:
$f = function() {
    echo __FUNCTION__;
};
$f();
```

1. Между словом `function` и открывающейся скобкой `(` пробел не является обязательным: `function() {`
2. До и после ключевого слова `use` пробелы обязательны
3. Пробрасываемые переменные в выращение анонимной фукнции через `use` определяются также, как и параметры обычной фукнции - каждая переменная через запятую с пробелом после запятой, напр: `use ($suffixName, $suffixName2) {`.

Тернарный оператор ’?:’

Проблема обычно заключается в том, что люди пытаются запихать слишком много кода между `"?"` и `":"`. Вот несколько правил:

1. Условие заключайте в скобки, тем самым отделяя его от остального кода
2. По возможности действия, производимые по условию, должны быть простыми функциями
3. Если весь блок ветвления плохо читается, будучи расположен на одной строке, то блоки else размещайте каждый на отдельной строке:

```
(условие) ? funct1() : func2();

or

(условие)
    ? длинный блок
    : ещё один длинный блок;
```

Использование коротких тегов ’<?’ и ’<?=’

Использование коротких тегов приветствуется. Для вывода в шаблоне одной переменной или результата выражения «?:» удобно пользоваться записью «<?=» вместо `<?php echo`. Пример:

```
// удобно практически, чем второй вариант
<a href= "<?=$arItem["LINK"];?>"><?=$arItem["TEXT"];?></a>

// допустимо, но удобнее воспользоваться первым вариантом
<a href= "<?php echo $arItem["LINK"];?>"><?php echo $arItem["TEXT"];?></a>
```

Для длинных выражений (более одного тернарного оператора) использование такой записи практически не оправдано. Лучше их в этом случае писать через полноценное `if...else` выражение.

Пробелы вокруг знаков операций

Любые операторы / знаки операций (например `=, ==, =>, >, <, &&, ||` и т.п.) обязательно отделяются пробелами с обоих сторон.
В арифметических выражениях количество пробелов вокруг знаков операций можно варьировать, чтобы подчеркнуть приоритет операций. Примеры:

```
$a = $b * $c + $d * $e;
$a = $b * $c  +  $d * $e;

// для читаемости уместно поставить скобки, хотя тут они и не требуются
$a = ($b * $c)  +  ($d * $e);
```

Декларативный блок желательно выравнивать по правому краю

Данный подход упрощает читаемость и понятность кода. Используем smart-tabs символы после знака присвоения для этих целей.

Пример:

```
$date     = 0;
$arDate   = NULL;
$gName    = 0;
$arName   = NULL;
```

Каждый оператор должен быть на новой строке

Необходимо, чтобы в каждой строчке присутствовало только одно выражение.

Напр. допустимо:

```
$dateBy = 0;
$date   = $test = 0;
if ($dateBy == 0) {
    $date = 13;
}
```

Не допустимо:

```
// плохо: более одно оператора на строке
$dateBy = 0; $date = 8;
```

```
// плохо: нет скобок и оператор на той же строке
if ($dateBy == 0) $date = 13;

// хорошо:
if ($dateBy == 0) {
    $date = 13;
}
```

Пустые строки для дробления кода на параграфы

Пустые строки помогаю разбивать код приложения на логические сегменты (параграфы кода). Несколькими строками могут отделяться секции в исходном файле. Одной пустой строкой отделяются друг от друга методы, логические секции внутри метода для более удобного чтения.

Встроенная документация phpDoc и комментарии

Комментарии внутри кода классов должны соответствовать синтаксису комментариев PHPDoc. За дополнительной информацией о PHPDoc обращайтесь сюда: http://www.phpdoc.org/

К примеру, редактор PDT/PhpStorm/ZendStudio 5.51-8.0 очень хорошо работает с подобной документацией: для этого следует впереди определения функции или класса набрать `/**` и нажать `Enter` - и среда разработки постоит всю требуемую шапку для корректного phpdoc-комментария.

Дополнительные комментарии, кроме тех, что предусмотрены PHPDoc, только приветствуются. Основное правило в данном случае — каждая часть кода повышенной сложности должна быть прокомментирована до того, как вы забыли, как она работает.

Подходят комментарии в стилях C (`/**/`) и C++ (`//`).
Использование комментариев в стиле Perl/shell (`#`) не рекомендуется.

1. Все блоки документации («docblocks») должны быть совместимы с форматом phpDocumentor. Описание формата phpDocumentor вне рамок данного докумета. Для дополнительно информации смотрите: http://www.phpdoc.org/
2. Всем файлам с исходными кодами, рекомендуется содержать «файловые» doc-блоки в начале каждого файла и обязательно наличие «классового» doc-блок непосредственно перед каждым классом. Ниже даны примеры таких doc-блоков.

Встроенная документация — Файлы

В каждый файл, содержащий PHP-код, желательно размещать заголовочный блок в начале файла, содержащий как минимум следующие phpDocumentor-теги:

```
/**
 * Краткое описание файла
 *
 * Длинное описание файла (если есть)...
 *
 *
 * @copyright  Copyright (c) 2005-2017
 * @version    1.0
 */
```

Встроенная документация — Классы

Аналогично файлам, однако для класса обязательно: каждый класс должен иметь doc-блок, содержащий как минимум

следующие phpDocumentor-теги: `@copyright`, `@version`, `@link`

Встроенная документация — Функции

Для функций и методов документация в виде phpDocumentator обязательна. Каждая функция, включая методы объектов, должна иметь doc-блок, содержащий как минимум:

1. Описание функции
2. Все аргументы
3. Все возможные возвращаемые значения
4. Нет надобности использовать тег «`@access`», потому что область видимости уже известна из ключевых слов «`public`», «`private`» или «`protected`». используемых при определении функции.
5. Если функция/метод может выбрасывать исключение, используйте тег `@throw`

```
/**
 * Название метода
 *
 * подробное описание метода
 *
 * @param string|array $directory Директория
 * @param mixed $module Описание параметра два
 * @return MyObj
 * @throws MyException если директория не доступна
 */
public function addControllerDirectory($directory, $module = null)
{
    return $this;
}
```

Для блоков требующих в будущем рефакторинга, использовать теги в комментариях @todo, FIXME, WTF?, TOFUTURE

`@todo` – требуется дальшейшая доработка

`FIXME` – критическая ошибка, нужно исправить

`WTF?` – непонятно, что делает этот код, выяснить, исправить

`TOFUTURE` – задел на будущее (описать, для чего в будущем это может понадобиться)

```
/**
 * FIXME убрать эту ненужную глубокую вложенность
 */
if (1) {
    if (1) {
        if (1) {
            if (1) {
            }
        }
    }
}

/**
 * TOFUTURE возвращаем для дальнейшего использования
 * тут же возможно и поменять значение
 */
return $arResult;
```

#4. Дополнительные частные случаи, на которые следует обратить внимание

Используемая версия PHP

Пиши на PHP 5.3! Все вещи, которые доступны выше 5.3 необходимо решать путем создания промежуточных вспомогательных переменных!

Автозагрузка классов

Все правила **именования класов** (ВерхнийКемелКейс) и **распределения их по простанствам имен** (ВерхнийКемелКейс) сводятся также к такой важной вещи, как автозагрузка классов из файлов (имена файлов совпадают с именами простанств имен и классов и **содержат ту же иерархию папок. Имена файлов и папок - в нижнем регистре**).

В нашей компании используется загрузчик `simple_loader.php` который выполняет автозагрузку классов, написанных по правилам данного документа.

Всегда документировать пустое выражение

Необходимо всегда комментировать пустое выражение, чтобы было ясно, что это не ошибка программиста, а задуманное программное решение.

```
while ($dest++ = $src++) {
    ;       // ПУСТО
}

if ($veryDiff) {
    ;  // ПУСТО
} else {
    // А тут уже что-то делаем
}
```

Не следует делать реальную работу в конструкторе

Напр. если мы хотим открыть соединение с БД, то делаем метод Open(), в котором и совершаем открытие. Причина: конструктор не может вернуть значение (напр. ошибку).

```
/** DOC-блок здесь */
class Device
{
    /** DOC-блок здесь */
    const FAIL_OPEN = 10;

    /** DOC-блок здесь */
    function __construct()
    {
        /* инициализация экземпляра */
    }

    /** DOC-блок здесь */
    function Open()
```

```
    {
        return FAIL;
    }
}

$dev = new Device;
if ($dev->Open() == Device::FAIL_OPEN) {
    exit(1);
}
```

Короткие методы

Желательно, чтобы метод (функция) занимал не более одной страницы кода. Большой метод необходимо делить на кучу маленьких, в таком случае мы добиваемся преждевременной оптимизации кода за счет продумывания структуры одной большой задачи и разделение их на подзадачи (структурный подход к построению решения)

Рефакторинг

Если код Вам перестаёт нравится: функция слишком сложная и длинная (более 100 строк), слишком много входных и выходных параметров, неудачное название функции и т.п. — делайте рефакторинг. После проведения рефакторинга обязательно запускайте автоматические тесты. Если для функции, над которой Вы работали, тесты не предусмотрены — создайте их.

Не живите с «разбитыми окнами»!

Не оставляйте «разбитые окна» (неудачные конструкции, неверные решения или некачественный текст программы) без внимания. Как только Вы их обнаружите — чините сразу. Часто безошибочные, функциональные системы быстро портились, как только окна начали разбиваться. Не давайте энтропии победить себя.

Старайтесь повторно использовать свой и чужой труд

Очень часто разработчики ленятся делать библиотеки классов & методов, которые упрощают жизнь и себе и команде. Очень хорошо, когда программист собирает себе библиотеку с кодом для последующего использования. Конечно, это идеальная картина, когда в студии существует хорошо задокументированная библиотека наработок и о том, что в ней находится, знают все разработчики студии.

Do not Repeat Youself — не повторяй самого себя!

Подумайте о повторном использовании кода. Зачем тратить впустую усилия мысли на одноразовые программы, если в будущем вам может потребоваться что-то подобное снова? Подумайте над обобщением вашего кода. Подумайте над написанием модуля
или класса. Подумайте, не предложить ли Ваш код другим.

Don’t Reinvent Wheel — Не изобретайте колесо!

Не переизобретайте колесо — если подобное колесо уже существует, просто адаптируйте его для своих нужд. Вероятно, для решения данной проблемы уже существует стандартный модуль в ядре PHP или API CMS, или модуль, разработанный другими людьми внутри организации. Иногда не составляет труда найти нужный код в интернете.

Куски кода и ответственность

Каждый человек должен отвечать за свои куски кода. Соответственно, если в них возникают ошибки, то человек, который написал код, идет исправлять ошибку. Тем более если из-за части чьего-то кода перестает работать вся целая система, то за эту часть «создатель» должен нести ответственность.

#5. Links и использованные материалы

1. Стандарт кодирования на PHP в Zend Framework'е
2. ZF Certification Study Guide, Сертификат специалиста в Zend Framework'е (pdf)
3. Анализ исходного кода Zend Framework'а
4. Анализ исходного кода phpMyAdmin
5. PHP Coding Standard, основанный на Todd Hoff's C++ Coding Standard , Перевод данного стандарта
6. Стандарты кодирования PEAR
7. Соглашения по написанию кода на C++ от компании Id Software (.doc)
8. Стандарт PHPDocumentor'а
9. Правила написания исходного кода на PHP от Bitrix (устарело)
10. Cake PHP Coding Standards
11. REG.RU perl coding standards
12. Правила именования функций и переменных в ядре от PHP.net
13. Добавляем префикс в именах для исключения коллизий
14. Official PHP Coding Standards
15. Идеология нового ядра D7
16. Соглашения по именованию нового ядра D7
17. Три PHP-стандарта Symfony2: PSR-0 (Стандарт автозагрузки), PSR-1 (Базовые стандарты оформления кода), PSR-2 (Руководство по Code Style)