* MyISAM поддерживает сжатие таблиц в отличии от InnoDB.
* MyISAM имеет встроенные полнотекстный поиск в отличии от InnoDB.
* InnoDB поддерживает транзакции в отличии от MyISAM.
* InnoDB поддерживает блокировки уровня строки (MyISAM - только уровня таблицы).
* InnoDB поддерживает ограничения внешних ключей (MyISAM - нет).
* InnoDB более надежна при больших объемах данных.
* InnoDB в теории немного быстрее.

До недавнего времени **InnoDB** не поддерживала полнотекстовые индексы.  
InnoDB не хранит ни количество строк в таблице, ни следующее значение автоинкрементного поля. Первое не так важно, поскольку мало кому бывает нужно считать все строки в таблице, без какого бы то ни было фильтра. А вот второе в теории чревато. Если после удаления последнего выданного id сервер будет перегружен, то id будет выделен повторно. Впрочем, я не слышал о случавшихся такого рода косяках на практике

Есть один очевидный (но не сразу) минус MyISAM, вытекающий из особенностей блокировок. Если в системе могут выполняться тяжелые SELECTы, то любой UPDATE на участвующие в нем таблицы будет ждать окончания SELECTа и блочить все дальнейшие запросы. Если с базой работают достаточно активно, то это не вариант.

**Отличия между системами хранения данных MyISAM и InnoDB простым языком.**

В данной таблице я попробовал показать разницу между MyISAM и InnoDB на простых примерах:

|  |  |  |
| --- | --- | --- |
| Описание | **MyISAM** | **InnoDB** |
| **Транзакционный движек?**Транзакция (Transaction) — блок операторов SQL , который в случае ошибки в одном запросе, возвращается к предыдущему состоянию (Rollback), и только в случае выполнения всех запросов подтверждается (Commit) | Нет | Да |
| **Поддержка внешних ключей**Внешние ключи — это способ связать записи в двух таблицах по определенным полям так, что при обновлении поля в родительской автоматически происходит определенное изменение поля в дочерней (дочернюю и родительскую выбираешь при создании ключа; точнее, создаешь ключ в дочерней, который ссылается на родительскую). | Нет | Да |
| **Блокировка.**Блокировка на уровне строк, т.е. если процессу нужно обновить строку в таблице, то он блокирует только эту строку, позволяя другим обновлять другие строки параллельно | Блокировка на уровне таблиц | Блокировка на уровне строк |
| Одновременные запросы к разным частям таблицы. | Медленнее | Быстрее |
| При смешанной нагрузке в таблице (select/update/delete/insert) | Медленнее | Быстрее |
| Операция Insert | Быстрее | Медленнее, ибо есть оверхед на транзакцию, но это цена надежности |
| Если преобладают операции чтения (SELECT) | Работает быстрее | Работает медленнее |
| **Deadlock**Deadlock — ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, захваченных самими этими процессами. | Не возникают | Возможны. |
| Поддержка полнотекстового поиска | Да | Нет (доступен начиная с версии MySQL 5.6.4) |
| Запрос Count(\*) | Быстрее | Медленнее |
| **Поддержка mysqlhotcopy**Утилита mysqlhotcopy представляет собой Perl-сценарий, использующий SQL-команды LOCK TABLES, FLUSH TABLES и Unix-утилиты cp или scp для быстрого получения резервной копии базы данных. | Да | Нет |
| Файловое хранение таблиц | Каждой таблице отдельный файл | Данные при настройках по умолчанию хранятся в больших совместно используемых файлах |
| **Бинарное копировании таблиц?**Табличные файлы можно перемещать между компьютерами разных архитектур и разными операционными системами без всякого преобразования. | Да | Нет |
| Размер таблиц в БД | Меньше | Больше |
| Поведение в случае сбоя | Крашится вся таблица | По логам можно все восстановить |
| В случае хранения «логов» и подобного | Лучше | Хуже |

**Выводы:**

* Использовать MyISAM лучше в таблицах, которых преобладает один вид доступа: чтение (новостной сайт) или запись (например, логирование) ;
* Использование InnoDB имеет смысл во всех остальных случаях и случаях повышенных требований по сохранности данных.

|  |  |
| --- | --- |
| 3голос «против» | БД - это набор таблиц. Таблицы могут быть как InnoDB, так и MyISAM (или даже другие). Т.е. нельзя сказать - "*БД у меня MyISAM*". Можно сказать "*Для таблицы A использована подсистема хранения InnoDB, а для таблицы B - подсистема хранения MyISAM*"  А узнать какая подсистема хранения используется для таблицы можно, например, через SHOW CREATE TABLE table\_name ([см. доки](http://dev.mysql.com/doc/refman/5.0/en/show-create-table.html)) |

show create table t;  
ищите слово ENGINE=

to **warma2d**Вы задаете разные вопросы.  
  
Как узнать тип конкретной таблицы - вам уже ответил **deadka**  
  
Как узнать какие типы поддерживает сервер и какой используется по умолчанию:  
SHOW ENGINES;

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

## До PHP 5.3

#### $str1{0} — синтаксис получения символа строки

Новый синтаксис обращения к символам строки:

// получение символа

$str = 'abc';

**echo** $str{0}; //> a

**echo** $str{1}; //> b

**echo** 'abc'{1}; //> b

// установка символа

$str1 = $str2 = 'ff';

$str1{0} = 'a';

**echo** $str1; //> af

$str2[0] = 'a'; // этот синтаксис не рекомендуется

**echo** $str2; //> af

$str2[0] не рекомендуется, хотя и работает точно также. Рекомендация использовать фигурные скобки {} связана с тем, чтобы при прочтении кода было сразу понятно, что обрабатывается строка, а не элемент массива. Как мы знаем квадратными скобками в PHP принято обозначать массивы, а не строки.

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

## PHP 5.3

В PHP 5.3, как и во всей пятой ветке PHP, включена новая машина-интерпретатор скриптов [Zend Engine 2.0](https://ru.wikipedia.org/wiki/Zend_Engine). Благодаря этому PHP стал работать быстрее примерно на 15-20%.

Новые возможности в PHP 5.3 ([ссылка на офф. сайт](http://php.net/manual/ru/migration53.new-features.php)):

#### ?: — сокращение тернарного оператора

С PHP 5.3 стало возможным не писать среднюю часть тернарного оператора. Выражение expr1 ?: expr3 возвращает expr1 если expr1 не пустой, и expr3 в противном случае.

Тернарный — состоящий из трёх частей, компонентов.

$a = $expr1 ?: $expr3;

// равносильно записи:

$a = $expr1 ? $expr1 : $expr3;

Пример тернарного оператора:

// полная запись

**if** ( $a > 100 )

$result = "Больше";

**else**

$result = "Меньше";

// краткая запись

$result = $a > 100 ? "Больше" : "Меньше";

В короткой записи есть еще момент производительности, например:

// полная запись

**if** ( get\_post\_meta(25, 'meta\_key', 1) )

**echo** esc\_html( get\_post\_meta(25, 'meta\_key', 1) );

**else**

**echo** 'Мета поля нет';

// краткая запись

**echo** esc\_html( get\_post\_meta(25, 'meta\_key', 1) ?: 'Мета поля нет' );

В полной записи функция get\_post\_meta() вызывается 2 раза. В короткой один раз, и если она что-то вернула, второму аргументу сразу передается полученное значение: не нужны дополнительные переменные...

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### $func = function(){}; — анонимные (лямбда) функции

Лямбда-функции еще называют «анонимными функциями», потому что для них не указывается название.

Лямбда-функции представляют собой замыкание (англ. closure) — это особый вид функции, которая определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. Насколько я понял, любая функция — это замыкание текущего контекста, т.е. контекст не будет очищен пока работает функция. Но если в функции есть лямбда-функция, то она становится замыканием, и если в неё передаются переменные из «верхней» функции, то они не будут очищены, до тех пор пока работает вложенная-функция, даже если «верхняя» функция работу закончила...

В ранних версиях, анонимные функции создавались с помощью функции create\_function().

Пример создания анонимной функции для сортировки usort():

Пример создания анонимной функции для сортировки usort():

$arr = **array**(3, 2, 5, 6, 1);

usort( $arr, **function**($a, $b) {

**if** ( $a == $b )

**return** 0;

**return** ( $a > $b ) ? -1 : 1;

});

Еще одна фишка лямбда-функций — это использование переменных из текущей области видимости, с помощью оператора use:

$var = 'Превед, Медвед!';

$func = **function**() **use** ( $var ) { echo $var; };

$func(); //> Превед, Медвед!

Переменные передаются как значение, но можно передать и ссылку на переменную, указав &:

$var = 'Превед, Медвед!';

$func = **function**() **use** ( & $var ) { $var = $var .' Мы в гости!'; };

$func(); // вызовем

**echo** $var; //> Превед, Медвед! Мы в гости!

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### method()->var — получение объекта из метода/функции

Это удобно:

$object->method()->method()->method();

$object->method()->method()->member = 5;

В PHP ниже 5.3 писали как-то так:

$tmp = & $object->method();

$tmp = & $tmp->method();

$tmp->method();

#### <<<'DOC' — поддержка NOWDOC

В php 5.3 можно использовать аналог [HEREDOC](http://ru2.php.net/manual/en/language.types.string.php#language.types.string.syntax.heredoc), который называется [NOWDOC](http://php.net/nowdoc). Особенность его в том, что внутри него переменные остаются простым текстом, как если бы мы указали её в строке с одинарными кавычками: 'текст $foo':

$foo = 'Лето';

// HEREDOC был в 5.2

$str = <<<DOC

Текст с переменной '**$foo**'

DOC;

**echo** $str; // Текст с переменной 'Лето'

// NOWDOC появился в 5.3

$str = <<<'DOC'

Текст с переменной '. **$foo** .'

DOC;

**echo** $str; // Текст с переменной '. $foo .'

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### namespace — поддержка пространств имен

Пространства имен нужны, чтобы избежать конфликтов при совпадении названий функций/классов/переменных/констант. Если коротко: одинаковые называния в разных пространствах — это разные названия.

Пример ниже должен объяснить почти все, что возможно в пространствах имен. За подробностями [идем в официальную документацию](http://php.net/namespaces).

<?php

#

# Объявлять пространство нужно в самом начале файла содержащего пространство имен,

# т.е. до любого кода, кроме зарезервированного declare(encoding='...');.

# Также ничего не должно выводиться на экран до объявления пространства

# Одно и тоже пространство имен можно определять в разных файлах. Так эти файлы будут относиться к одному пространству

# Объявляем пространство my\name

**namespace** my\name;

## Динамичное получения названия пространства --------------

$s = **\_\_NAMESPACE\_\_**; //> my\name

$s = **\_\_NAMESPACE\_\_** . '\HELLO'; //> my\name\HELLO

// namespace: есть еще специальное слово namespace, которое используется для динамичного

// получения названия текущего пространства при вызове функций/методов/констант (см. ниже)

## ГЛОБАЛЬНЫЕ функции/классы/константы в нашем пространстве --------------

$s = strlen('hi'); // вызовет my\name\strlen() - если функция есть в нашем пространстве, иначе глобальную функцию strlen()

define('HELLO', 'HI всем'); // добавит константу в глобальное пространство "\HELLO"

# Доступ к глобальным классам/функциям/константам из пространства имен

$a = \strlen('hi'); // вызывает глобальную функцию strlen()

$b = \ABSPATH; // получает доступ к глобальной константе ABSPATH

$c = **new** \WP\_Query; // создает экземпляр глобального класса WP\_Query

## ФУНКЦИЯ в нашем пространстве --------------

**function** my\_func(){ **return** 'Моя функция'; }

// Вызов

my\_func(); //> "Моя функция"

**namespace**\my\_func(); //> "Моя функция"

\my\name\my\_func(); //> "Моя функция"

// my\name\my\_func(); //> ошибка: будет вызвана функция my\name\my\name\my\_func()

// такой синтаксис можно использовать для доступа к под-пространствам нашего пространства

## ФУНКЦИЯ в нашем пространстве, которая существует в глобальном --------------

**function** trim( $str ){

**return** \trim( $str, '-' ); # если вызвать trim( $str, '-' ), то функция вызовет сама себя...

}

// Вызов

$s = trim('-foo'); // вызов trim() из текущего пространства. Выведет: foo

$s = \my\name\trim('-foo'); // тоже что в строке выше

$s = **namespace**\trim('-foo'); // тоже что в строке выше

$s = \trim('-foo'); // вызов trim() из глобального пространства. Выведет: -foo

## КОНСТАНТЫ в нашем пространстве --------------

**const** HELLO = 'HI'; // добавим константу в текущее пространство

define('my\name\HELLO', 'HI'); // тоже что в строке выше

define(**\_\_NAMESPACE\_\_** . '\HELLO', 'HI'); // тоже что в строке выше

// Вызов

$s = HELLO; //> HI - если константа есть в текущем пространстве, или значение глобальной константы

$s = \my\name\HELLO; //> HI

$s = **namespace**\HELLO; //> HI

$s = \HELLO; //> HI всем - глобальная константа HELLO

## КЛАСС в нашем пространстве --------------

**class** MyClass {

**function** method(){ **return** 'метод MyClass'; }

**static** **function** static\_method(){ **return** 'статический метод MyClass'; }

}

// Вызов

$a = **new** MyClass; // обращение к MyClass из текущего пространства

$a = **new** \my\name\MyClass; // тоже что в строке выше

$s = **namespace**\MyClass::static\_method(); //> 'статический метод MyClass' - вызывает статический метод "static\_method" класса my\name\MyClass.

$s = $a::static\_method(); // тоже что в строке выше

$s = $a->method(); //> 'метод MyClass' - вызывает метод "method" класса my\name\MyClass

// namespace\MyClass->method() - такой вызов метода, вызовет ошибку - syntax error

## ВНЕДРЕНИЕ функций/методов/констант в наше пространстве из других пространств --------------

// ЗАМЕТКА: операторы use можно комбинировать: указывать через запятую

// Например: use other\name\OtherClass as Another, other\name\NSname;

**use** other\name\OtherClass as Another;

$obj = **new** Another; // создает объект класса other\name\OtherClass

**use** other\name; // теперь name = other\name

name\other\_func(); // вызывает функцию other\name\other\_func();

// импорт глобального класса

**use** WP\_Query;

$a = **new** WP\_Query(); // создаст экземпляр класса WP\_Query

// без выражения "use WP\_Query;" создавался бы экземпляр my\name\WP\_Query

// импорт функции (PHP 5.6+)

**use** function other\name\other\_func;

$s = other\_func(); //> "Другая Функция" - работа функции other\name\other\_func()

// импорт функции под псевдонимом func (PHP 5.6+)

**use** function other\name\other\_func as func;

$s = func(); //> "Другая Функция" - работа функции other\name\other\_func()

//const other\name\HELLO2 = 'И снова здрасте!'; // вызовет ошибку синтаксиса, что странно

define('other\name\HELLO2', 'И снова здрасте!');

// импорт константы (PHP 5.6+)

**use** const other\name\HELLO2;

$s = HELLO2; //> "И снова здрасте!" - содержимое константы other\name\HELLO2

## ЕЩЕ ОДНО ПРОСТРАНСТВО в одном файле --------------

// Подробнее: http://php.net/manual/ru/language.namespaces.definitionmultiple.php

**namespace** other\name;

**class** OtherClass {}

**function** other\_func() { **return** 'Другая Функция'; }

// При описании нескольких пространств в одном файле лучше использовать синтаксис со скобками:

/\*

namespace MyProject {

function connect() {}

}

namespace AnotherProject {

function connect() {}

}

\*/

#### \_\_DIR\_\_ — новая магическая константа

\_\_DIR\_\_ содержит директорию текущего файла - файла в котором она используется. Возвращает полный путь до текущего файла без закрывающего слэша, за исключением корневой директории.

\_\_DIR\_\_ можно заменить:

dirname(**\_\_FILE\_\_**)

#### $class::$foo — динамичное указание класса

Это дает динамичный доступ к статическим методам/свойствам класса:

**class** C {

**static** $foo = 'foo';

}

$class = 'C';

**echo** $class::$foo; //> foo

#### const — ключевое слово для создания констант вне классов

Сразу пример, где все понятно:

define('SHORTINIT', 'true');

// теперь можно объявить константу и так:

**const** SHORTINIT = 'true';

В отличие define(), такие константы, должны быть объявлены в самой верхней области видимости, потому что они определяются при компилировании скрипта. Это значит, что их нельзя объявлять внутри функций/циклов/выражений if или try/ catch блоков.

#### static::method() — статическое связывание

Статическое объявление метода/свойства связывает его с классом из которого оно вызывается, а не с тем в котором оно зарегистрировано. Посмотрим на примере:

**class** A {

**static** **function** who() {

**echo** **\_\_CLASS\_\_**;

}

**static** **function** test1() {

**self**::who();

}

**static** **function** test2() {

**static**::who(); // статическое связывание

}

}

**class** B **extends** A {

**static** **function** who() {

**echo** **\_\_CLASS\_\_**;

}

}

**echo** B::test1(); //> A

**echo** B::test2(); //> B

**echo** B::who(); //> B

Подробнее про статическое связывание [читайте в документации](http://php.net/manual/ru/language.oop5.late-static-bindings.php).

#### goto hell; — оператор goto

Используется для перехода в другую часть программы. Место, куда необходимо перейти указывается с помощью метки, за которой ставится двоеточие, после оператора goto указывается желаемая метка для перехода.

Целевая метка должна находиться в том же файле, в том же контексте. Т.е. нельзя выйти за границы функции или метода, а значит нельзя перейти внутрь любой функции.

Также нельзя перейти внутрь любой циклической структуры или оператора switch. Но можно выйти из любой циклической структуры, поэтому «goto» удобен как замена многоуровневых break.

Пример использования goto:

**function** zayac(){

$i = 1;

$out = '';

start: $out .= ($i > 1 ? '-' : '' ) .$i;

**if**( $i++ < 5 ){ **goto** start; }

**return** $out . ' вышел зайчик погулять';

}

**echo** zayac(); //> 1-2-3-4-5 вышел зайчик погулять

Пример использования goto в цикле:

**for**( $i=0, $j=50; $i<100; $i++ ) {

**while**( $j-- ) {

**if**( $j==17 ) **goto** end;

}

}

**echo** "i = **$i**"; // будет пропущено

end: **echo** 'j дошло до 17';

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### \_\_callStatic(), \_\_invoke() — магические методы

\_\_callStatic() — срабатывает, когда вызывается несуществующий метод из статического контекста: Foo::bar():

**class** A {

**static** **function** \_\_callStatic( $name, $args ){

**return** $name .' '. print\_r( $args, 1 );

}

}

**echo** A::no\_matter\_what('bar');

/\* Выведет:

no\_matter\_what Array

(

[0] => bar

)

\*/

\_\_invoke() — срабатывает, когда объект выполняется как функция: $obj():

**class** A {

**function** \_\_invoke( $var ){

var\_dump( $var );

}

}

$obj = **new** A;

$obj('foo'); //> string(3) "foo"

## PHP 5.4

Возможности, добавленные в версии PHP 5.4. [Ссылка на офф. сайт](http://php.net/manual/ru/migration54.new-features.php).

#### <?= — короткая запись вывода на экран работает всегда

Короткая запись о которой идет речь это: <?= вместо <?php echo.

Для работы такой короткой записи вывода на экран в версиях ниже 5.4 нужно было, чтобы опция short\_open\_tag в php.ini была включена.

Пример длинной и короткой записи:

<a href="#"><?php **echo** $page ?></a>

<a href="#"><?= $page ?></a>

#### [1,2] — запись массива, без слова array

$a = [1, 2, 3, 4];

$a = ['one' => 1, 'two' => 2, 'three' => 3, 'four' => 4];.

#### trait Class {} — примеси (трейты)

Трейт - это аналог класса, который содержит в себе методы. Нужен он для «подмешивания» его в имеющийся класс, чтобы методы трейта стали методами класса в который он добавлен.

Несколько примесей можно задавать через запятую:

**trait** TR\_A {

**public** $var = 'var';

**function** foo() { **return** 'foo'; }

}

**trait** TR\_B {

**function** bar() { **return** 'bar'; }

}

**class** A {

**use** TR\_A, TR\_B; // подмешиваем

**function** hello() { **return** 'hello A'; }

}

$A = **new** A();

**echo** $A->foo(); // foo

**echo** $A->bar(); // bar

**echo** $A->hello(); // hello A

**echo** $A->var; // var

**class** B **extends** A {

**use** TR\_A, TR\_B;

**function** hello() { **return** 'hello B'; }

}

$B = **new** B();

**echo** $B->foo(); // foo

**echo** $B->bar(); // bar

**echo** $B->hello(); // hello B

**Приоритеты трейтов**

При совпадении названий свойств/методов приоритеты расставляются так: текущий класс имеет наивысший приоритет, затем трейт, а затем расширяемый класс. Другими словами: элементы из текущего класса переопределяют элементы в трейте, которые в свою очередь переопределяют унаследованные элементы.

**Статический доступ к методу примеси из класса**

Когда в класс подмешивается trait, то его методы становятся методами класса, включая статические и статический доступ:

**trait** A {

**static** **function** func(){ **echo** 'A'; }

}

**class** B {

**use** A;

}

B::func(); //> A

Подробнее про трейты [читайте в документации](http://php.net/manual/ru/language.oop5.traits.php)

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### foo()[0] — быстрое получение элемента массива

Теперь не нужно сохранять массив, который вернула функция/метод в переменную и получать элемент массива из этой переменной. Можно сразу получать элемент из функции/метода:

$foo = func()[0];

$foo = **Class**::func()[0];

#### (new Foo)->method() — доступ к элементу объекта при его создании

$foo = (**new** Foo)->method();

$foo = (**new** Foo)->property;

$foo = (**new** Foo)[0];

// было так

$obj = **new** Foo;

$foo = $obj->method();

#### Class::{'foo'}() — динамичное указание метода

Чтобы вызвать статический метод/свойство класса, не нужно запоминать его в отдельную переменную:

**class** A {

**static** **function** foo() {

**echo** "Hello world!";

}

}

$x = "f";

A::{ $x .'oo' }();

#### callable — новый тип для аргументов функции/метода

Авто-проверка передаваемых данных в функции/методы, известная как «контроль типа» (typehint), продолжает развиваться и теперь понимает слово callable.

Раньше для автоматической проверки типа передаваемого параметра, в аргументах функции/метода можно было указывать только: array или имя класса.

Теперь, можно указать еще: callable — значит, что передаваемый аргумент должен быть вызываемым, т.е. удовлетворяет условию is\_callable( $arg, false ).

Пример:

**function** func( callable $callback ){

**return** **true**;

}

func('trim'); //> true

func( **function**(){} ); //> true

$db = **new** wpdb();

func( **array**($db, 'query') ); //> true

func('my\_trim'); //> fatal error: Argument 1 passed to func() must be callable, string given

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### @ — улучшена производительность

Оператор @ нужен для подавления вывода ошибок любого уровня. Вообще его использовать не рекомендуется, но иногда с ним короче:

**if**( @ $\_GET['foo'] ) **echo** 'OK';

// или так

**if**( **isset**($\_GET['foo']) && $\_GET['foo'] ) **echo** 'OK';

// раньше так работало быстрее раз в 20, теперь раз в 5

Использовать @ нужно как можно реже и очень осторожно, потому что часто заметки и предупреждения дают понять, что логика кода работает неправильно. Например, у меня бывало что лезу поправить казалось бы безобидный NOTICE, но при анализе выясняется что ошибка появилась из-за неправильной логики кода, которая изменилась в процессе расширения кода...

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

## PHP 5.5

Новые возможности в PHP 5.5 ([ссылка на офф.сайт](http://php.net/manual/ru/migration55.new-features.php)):

#### [1,3,4][2], "foobar"{2} — разыменования только-созданных массивов и строк

**echo** **array**(1, 2, 3)[0]; // 1

**echo** [1,3,4][2]; // 4

**echo** "foobar"{2}; // o

**echo** "foobar"[0] // f

// это может пригодиться для быстрой генерации:

**echo** 'abcdefghijk'{ rand(0,10) }; // получим одну из букв: 'abcdefghijk'

#### empty() — можно применять к результатам функций и выражений

Раньше empty() мог принимать только переменные, теперь можно передавать сами выражения без необходимости сохранять результат в отдельную переменную:

**empty**( $object->get() );

#### list() в foreach

В foreach стало возможным использовать [list()](http://php.net/manual/ru/function.list.php):

$array = [

[1, 2],

[3, 4],

];

**foreach**( $array **as** **list**($a, $b) ){

**echo** $a;

**echo** $b;

}

// получим: 1234

#### finally — в конструкции try/catch

Выбрасывать и ловить исключения можно с PHP 5. Такой подход позволяет контролировать выполнение кода, если есть подозрение, что в нем что-то может пойти не так.

А с версии 5.5. в эту конструкцию добавили третий блок finally. Блок finally выполняется всегда после завершается конструкции try/catch. Он выполняется даже когда код try вызвал фатальную ошибку:

**try** {

**echo** 'Тут что-то деламе... ';

// выбрасываем

**throw** **new** **Exception**('Лови меня! ');

}

// ловим

**catch**( **Exception** $e ){

**echo** $e->getMessage(); // выведет: Лови меня!

}

**finally** {

**echo** 'А это выводиться всегда!';

}

Нужен finally для удобства, и дополнительных возможностей. С ним можно будет писать меньше кода и можно, например, удобно чистить память, когда это нужно.

Пару домонстрационных примеров:

**Меньше кода**

Допустим, нам нужно выполнить функцию close() в любому случае, было выброшено исключение или нет:

**try** {

my\_function();

}

**catch**( **Exception** $e ){

// close(); // эта строка нужна была бы без finally

**echo** $e->getMessage(); // выведет: Лови меня!

}

**finally** {

close();

}

//close(); // эта строка нужна была бы без finally

**Больше возможностей**

Допустим мы открыли соединение с БД до выполнения кода и есть вероятность что код вызовет ошибку и открытое соединение не закроется, а нам нужно его закрыть в любом случае. finally как раз кстати:

$db = mysqli\_connect();

**try** {

my\_function( $db ); // результат работы функции может вызвать фатальную ошибку...

}

// исключение можно не обрабатывать

**finally** {

mysqli\_close($db);

}

Подробнее про finally читайте [*статью на хабре*](https://habrahabr.ru/post/149314/).

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### Class::class — для получение имени класса в пространствах

Появилось ключевое слово class для классов, которое выводит название класса. В обычном режиме нам это не нужно, а вот при работе с пространствами (namespace) — это удобно:

**namespace** test;

**class** A {}

**echo** A::class; //> test\A

#### yield — создание генераторов

Если говорить простым языком: yield похожа на return, она также возвращает значение, но она не обрывает работу функции, а приостанавливает её до тех пор пока не будет запрошено следующее значение. Благодаря этому создавать генераторы стало удобнее.

Пример генератора:

**function** generator() {

**for**( $i = 1; $i <= 3; $i++ ){

**yield** $i; // выброс значения

}

}

**foreach**( generator() **as** $value ){

**echo** "**$value** ";

}

// выведет: '1 2 3 '

Как это работает на самом деле?

yield возвращает специальный объект — Generator. Когда функция generator() вызывается в цикле, например foreach, PHP выполнит код функции до первой встречи слова yield, на котором PHP прервет работу функции, запомнит позицию и выбросит значение (объект Generator). Затем, foreach обработает значение и вызовет метод next() у полученного объекта Generator. PHP снова выполнит код функции generator(), только начнет его не с начала, а с прошлой позиции, и опять, до слова yield, которое опять выбросит объект Generator. Работа цикла прервется тогда, когда функция generator() дойдет до конца (не вернет yield), или если она будет прервана с помощью return;.

Пример генератора который возвращает пару: ключ/значение:

**function** generator( $input ){

**foreach**( explode('.', $input) **as** $part ){

**list**( $num, $name ) = explode(' - ', $part );

**yield** $num => trim($name);

}

}

$input = '1 - один. 2 - два. 3 - три';

**foreach**( generator( $input ) **as** $num => $name ){

**echo** "**$num** (**$name**) ";

}

Кратко о генераторах

— Не добавляют нового функционала в язык  
— Быстрее  
— Возобновление работы генератора происходит с последнего «выброса» yield  
— В генератор можно отправлять значения и исключения (через метод throw())  
— Генераторы однонаправлены, т.е. нельзя вернуться назад  
— Меньше кода в большинстве случаев, более простые для понимания конструкции

Чтобы лучше понять генераторы прочитайте [эту статью на Хабре](https://habrahabr.ru/post/189796/).

[Подробно о генераторах в документации (англ.)](http://php.net/manual/en/language.generators.syntax.php)

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### API для хэширования паролей

Теперь PHP из коробки предлагает правильный способ хэшировать пароли. Новый API хэширования паролей предоставляет четыре функции:

1. password\_hash() — используется для хэширования пароля. В WP для этого есть своя функция [wp\_hash\_password()](https://wp-kama.ru/function/wp_hash_password).

$hash = password\_hash( $passwod, PASSWORD\_DEFAULT );

1. password\_verify() — используется для проверки пароля на соответствие хэшу. В WP для этого есть своя функция [wp\_check\_password()](https://wp-kama.ru/function/wp_check_password).
2. **if**( password\_verify( $password, $hash ) ){
3. // Success!

}

1. password\_needs\_rehash() — используется для проверки необходимости создать новый хэш.
2. password\_get\_info() — возвращает имя алгоритма хеширования и различные параметры, используемые при хэшировании.

Подробнее [*читайте в статье на хабре*](http://habrahabr.ru/post/184220/).

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

## PHP 5.6

Новые возможности PHP 5.6. [Ссылка на офф.сайт](http://php.net/manual/ru/migration56.new-features.php).

#### const PLUS = 1 + 2; — скалярные выражения в константах/свойствах/аргументах функции

Теперь стало возможным указывать в значения констант примитивные PHP выражения (выражения из скаляров).

Точнее, новинка касается не только констант, а всего где раньше PHP ожидал статическое значение. Теперь вместо статики можно указать выражение из чисел/строк/констант. Если точнее, то PHP выражение можно указывать: в константах/свойствах класса и в значении аргумента функции по умолчанию.

**const** ONE = 1;

**const** TWO = ONE \* 2;

**class** C {

**const** THREE = TWO + 1;

**const** ONE\_THIRD = ONE / **self**::THREE;

**const** SENTENCE = 'Значение THREE равно '. **self**::THREE;

**public** **function** f( $a = ONE + **self**::THREE ){

**return** $a;

}

}

**echo** (**new** C)->f() .' - '. C::SENTENCE; //> 4 - Значение THREE равно 3

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### const ARR = ['a', 'b']; — константа может хранить массив

Стало возможным держать в константе массивы:

**const** ARR = ['a', 'b'];

**echo** ARR[0]; //> a

#### func( ...$args ) или func( ...[2, 3] ) — неизвестное число аргументов функции или распаковка массива с помощью '...' (splat оператор)

Когда мы не знали заранее, сколько параметров может получить функция, нам приходилось внутри функции обрабатывать переданные параметры с помощью специальных функций: func\_num\_args(), func\_get\_arg(), func\_get\_args().

Теперь они не нужны и мы можем получить все параметры в одной переменной, для этого перед этой переменной нужно указать оператор ...:

**function** sum( ...$numbers ){

$plus = 0;

**foreach**( $numbers **as** $n ){

$plus += $n;

}

**return** $plus;

}

**echo** sum(1, 2, 3); //> 6

Оператор ... еще называют «Splat Оператор», например в языке Ruby

Еще пример:

**function** func( ...$numbers ){

**return** print\_r( $numbers, 1 );

}

**echo** func(1, 2, 3);

/\*

Получим:

Array

(

[0] => 1

[1] => 2

[2] => 3

)

\*/

##### **Быстрая распаковка передаваемых параметров функции**

Теперь с помощью splat оператора ..., можно указать параметры функции сразу из значений массива:

**function** plus( $a, $b, $c ){

**return** $a + $b + $c;

}

$array = [2, 3];

**echo** plus( 1, ...$array ); //> 6

// или так

**echo** plus( 1, ...[2, 3] ); //> 6

##### **Замена медленной функции call\_user\_func\_array()**

Теперь call\_user\_func\_array(callable $callback, array $param\_arr), которая обычно не самая быстрая, можно заменить так:

$params = [1, 2, 3];

$callback( ...$params );

Пример: известная своим неторопливым временем исполнения функция  
call\_user\_func\_array(callable $callback, array $param\_arr)  
может быть заменена на

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### \*\* — оператор возведения в степень

До php 5.6, чтобы возвести число в степень нужно было использовать функцию pow(2,2);, а теперь есть оператор \*\*:

// пример 1

**echo** $a = 2 \*\* 2; //> 4

// пример 2

$a = 2;

**echo** $a \*\*= 2; //> 8

// пример 3

**echo** $a = 2 \*\* 3 \*\* 2; //> 512 = 2^9

#### use function и use const — импорт функций и констант в пространство

Теперь стало возможным при помощью ключевого слова use подключать функции или константы другого пространства в наше:

**namespace** our\space {

const FOO = 42;

**function** func() { **echo** **\_\_FUNCTION\_\_**; }

}

**namespace** my\space {

use const our\space\FOO;

**use** function our\space\func;

**echo** FOO .' - '. func(); //> 42 - our\space\func

}

-

Остальные новинки PHP 5.6 не связанные с синтаксисом, смотрите [в статье на Хабре](https://habrahabr.ru/post/234899/).

## Куда делся PHP 6?

Умер не родившись... В ядро PHP 6 планировали внедрить полную поддержку юникода, но затея оказалась слишком амбициозной, а объем работ слишком велик. К тому моменту, когда это стало понятно, о PHP 6 уже было написано не мало статей. Чтобы не было путаницы, из-за того что новая версия стала преследовать совсем другие цели (производительность) и сильно отличалась по концепции от PHP 6, было решено пропустить PHP 6. Еще одной причиной стало наличие весомого количества недоделанного кода в репозитории PHP, который решили не трогать, чтобы тот в ответ тоже никого не трогал...

## PHP 7

Что нового в PHP 7. [Ссылка на офф.сайт](http://php.net/manual/ru/migration70.new-features.php).

3 декабря 2015 года было объявлено о выходе PHP 7. Новая версия основывается на экспериментальной ветке PHP, которая изначально называлась phpng (PHPNextGeneration - следующее поколение), и разрабатывалась с упором на увеличение производительности и уменьшение потребления памяти.

Самой важной новинкой стало изменение ядра интерпретатора: теперь он называется PHPNG (Next Generation). Благодаря PHPNG удалось увеличить скорость обработки скриптов почти в двое по сравнению с PHP 5.x. Так же появился более эффективный менеджер памяти.

Прирост в скорости на практике хорошо виден [на этой картинке](https://wp-kama.ru/wp-content/uploads/2016/01/php7-infographic.png). А для WordPress прирост в скорости выглядит так:

[Подробнее смотрите в тестах PHP 7](http://talks.php.net/oz15#/wpbench)

Синтаксические новинки PHP 7:

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### $a ?? '' — одновременная проверка isset и получение значения

Новый оператор слияния с NULL (NULL coalescing operator) ?? — это сокращение проверки isset и получения значения, если проверка пройдена.

Такая проверка часто была нужна в тернарном операторе ?::

// Получит значение $\_GET['foo'], если переменная установлена или не пустая, иначе получит 'default'

$foo = $\_GET['foo'] ?? 'default';

// Запись равносильна этой

$foo = **isset**($\_GET['foo']) ? $\_GET['foo'] : 'default';

// или этой

$foo = @ $\_GET['foo'] ?: 'default';

// удобная проверка при получении $\_GET параметра

**if**( $\_GET['foo'] ?? 0 ){ }

// раньше писали так

**if**( **isset**($\_GET['foo']) && $\_GET['foo'] ){ }

Так же, проверять можно по цепочке:

$foo = $\_GET['foo'] ?? $\_POST['foo'] ?? 'default';

// вернет: $\_GET['foo'], если его нет, то $\_POST['foo'], если нет, то 'default'

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### $a <=> $b — одновременное выполнение трех сравнений: больше, равно или меньше.

Новый оператор сравнения <=> — «spaceship operator». Сравнивает 2 переменные и возвращает результат сравнения в виде числа:

* -1 — если в сравнении подходит первый символ оператора <
* 0 — подходит второй символ =
* 1 — подходит третий символ >

// Числа

**echo** 1 <=> 1; // 0

**echo** 1 <=> 2; // -1

**echo** 2 <=> 1; // 1

// Дробные числа

**echo** 1.5 <=> 1.5; // 0

**echo** 1.5 <=> 2.5; // -1

**echo** 2.5 <=> 1.5; // 1

// Строки

**echo** "a" <=> "a"; // 0

**echo** "a" <=> "b"; // -1

**echo** "b" <=> "a"; // 1

Удобен для использования в колбэках для usort().

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### define('FOO', [1,2]); — передача массива константе через define()

Константы могут содержать массивы еще с PHP 5.6. Но тогда их можно было передавать только через ключевое слово const. Теперь их можно указывать еще и через define().

define('ANIMALS', ['dog', 'cat', 'bird']);

**echo** ANIMALS[2]; //> bird

#### use name\space\{A, B, C as c}; — группировка импорта при помощи use

Теперь для краткой записи, импорт данных в наше пространство можно группировать:

// PHP 7

**use** some\namespace\{ ClassA, ClassB, ClassC as C };

**use** function some\namespace\{ fn\_a, fn\_b, fn\_c };

**use** const some\namespace\{ СonstA, ConstB, ConstC };

// тоже самое до PHP 7

**use** some\namespace\ClassA;

**use** some\namespace\ClassB;

**use** some\namespace\ClassC as C;

**use** function some\namespace\fn\_a;

**use** function some\namespace\fn\_b;

**use** function some\namespace\fn\_c;

**use** const some\namespace\ConstA;

**use** const some\namespace\ConstB;

**use** const some\namespace\ConstC;

#### int, float, bool — новые типы для аргументов функции/метода

Авто-проверка типа передаваемых данных в функции/методы, известная как «контроль типа» (typehint), продолжает развиваться и теперь понимает скаляры: int, float, bool, string. Раньше понимались только типы: array, имя класса или callable (с версии 5.4).

Пример:

**function** foo( int $a, bool $b, callable $с, **array** $d, WP\_Post $e ) {

**return** var\_dump( $a, $b, $c, $d, $e );

}

foo( 1, **true**, 'trim', **array**(1), get\_post(1) );

/\* выведет:

int(1)

bool(true)

NULL

array(1) { [0]=> int(1) }

object(WP\_Post)#2660 (24) { ...данные объекта... }

\*/

// если указать неверный тип:

foo( 'foo', **true**, 'trim', **array**(1), get\_post(1) );

// Получим ошибку Fatal error: Argument 1 passed to A::foo() must be of the type integer, string given

**Режим строгой типизации**

Если указан тип int и передать строку '123' то проверка все равно будет пройдена, и php превратить строку в число.

***function*** func( int $num ){

var\_dump( $num );

}

func('123'); //> int(123)

Но что, если нужно получать именно число 123? Для этого можно включить режим строгой типизации, поместив в самое начало файла такую строку:

**declare**(strict\_types=1);

Это объявление должно быть первой строкой в файле, до выполнения какого-либо кода. Оно затрагивает только код файла и только вызовы и возвращаемые значения в этом файле.

Заметка: если строгая типизация указана в файле X, но не указана в файле Y и в файле Y вызывается функция из файла X. То вызов такой функции не будет подвержен строгой типизации!

Читайте по типизации [статью на Хабре](https://habrahabr.ru/post/248721/) и [вот еще интересная статья](https://habrahabr.ru/post/267799/).

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### int, float, bool, array — указание возвращаемых типов для функции/метода

Указывать принимаемый тип, можно еще с версии PHP 5.3. А вот указать какой тип функция/метод должна вернуть доступно только с версии PHP 7. Тут понимаются все типы: string, int, float, bool, array, callable, self (в методах), parent (в методах) , Closure, имя класса, имя интерфейса.

Синтаксис:

**function** func( $var ): int{ /\* код функции \*/ }

**function** func( $var ): string{ }

**function** func( $var ): float{ }

**function** func( $var ): bool{ }

**function** func( $var ): **array**{ }

**function** func( $var ): callable{ }

**function** func( $var ): Closure{ }

**function** func( $var ): WP\_Post{ } // может вернуть только объект класса WP\_Post

**class** A **extends** B {

**function** func( $var ): **self**{ }

**function** func( $var ): **parent**{ }

}

Рабочие примеры:

// Пример 1:

**function** func( $var ): int {

**return** $var;

}

**echo** func( 123 ); //> 123

**echo** func( 'asfd' ); //> вызовет ошибку: Fatal error: Uncaught TypeError: Return value of func() must be of the type integer, string returned

// Пример 2: Closure

**function** func(): Closure {

**return** **function**( $var ){ **return** $var .' + 2 = 3'; };

}

**echo** func()( 1 ); //> 1 + 2 = 3

**Возвращаемые типы при наследовании методов класса**

При наследовании в классах, дочерние методы должны иметь такие же возвращаемые типы как и в родительском классе/интерфейсе:

**class** A {

**function** func() : int {

**return** 123;

}

}

**class** B **extends** A {

**function** func() : string {

**return** '123';

}

// такое объявление функции вызовет ошибку:

// Fatal error: Declaration of B::func(): string must be compatible with A::func(): int

// т.е. тип int должен совпадать!

}

**Навороченный пример того, как можно писать в PHP 7**

Тут сразу несколько новинок:

1. принимаемый и возвращаемый тип;
2. объединение и распаковка параметров с помощью ...;
3. пример создания анонимной функции с указанием возвращаемого типа данных.

**function** arraysSum( **array** ...$arrays ): **array** {

**return** array\_map( **function**( **array** $array ): int {

**return** array\_sum( $array );

}, $arrays );

}

print\_r( arraysSum( [1,2,3], [4,5,6], [7,8,9] ) );

/\*

Выведет:

Array

(

[0] => 6

[1] => 15

[2] => 24

)

\*/

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### foo()(), $a::$b::$c, $$foo->bar — единый синтаксис переменных: СЛЕВА НАПРАВО

Очень важная новинка! Теперь обращения к сложносочиненным переменным разбираются последовательно СЛЕВА НАПРАВО.

Примеры новых возможностей:

// можно не указывать комбинирующие скобки

$foo()['bar']()

[$obj1, $obj2][0]->prop

getStr(){0}

// поддерживает вложенность ::

$foo['bar']::$baz // значение: ($foo['bar'])::$baz

$foo::$bar::$baz // значение: ($foo::$bar)::$baz

$foo->bar()::baz() // значение: ($foo->bar())::$baz

// поддерживает вложенные ()

foo()() // вызывает результат вызова foo(): (foo())()

$foo->bar()() // ($foo->bar())()

Foo::bar()() // (Foo::bar())()

$foo()() // ($foo())()

// Операторы над выражениями заключенными в ()

(**function**() { ... })() // IIFE синтаксис JS

($obj->closure)()

// и т.д.

(...)['foo']

(...)->foo

(...)->foo()

(...)::$foo

(...)::foo()

(...)()

// все операции по разименованию скаляров (не очень полезно)

"string"->toLower()

[$obj, 'method']()

'Foo'::$bar

Примеры разницы старого и нового распознавания:

// строка // старое понимание // новое понимание

$$foo['bar']['baz'] ${$foo['bar']['baz']} ($$foo)['bar']['baz']

$foo->$bar['baz'] $foo->{$bar['baz']} ($foo->$bar)['baz']

$foo->$bar['baz']() $foo->{$bar['baz']}() ($foo->$bar)['baz']()

Foo::$bar['baz']() Foo::{$bar['baz']}() (Foo::$bar)['baz']()

Старый код написанный с использованием {} для обработки переменных возможно не будет работать в новой версии PHP7.

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### foreach — изменена логика работы

Теперь foreach не переключает автоматически внутренний указатель перебираемого массива, т.е. next() не работает автоматически.

Так же, если значение массива передается в foreach по ссылке, то он всегда работает с оригиналом массива. Если значение не передается по ссылке, то foreach всегда работает с копией массива и оригинальный массив не затрагивается при изменении его внутри foreach.

Переключение указателей и влияние на работу цикла в PHP 7:

// Пример 1: автоматически не переключает внутренний переключатель next()

$a = [1,2,3]; **foreach**($a **as** $v) { **echo** $v ."-". current($a) ." "; } // 1-1 2-1 3-1 (php5: 1-2 2-2 3-2)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; } // 1-1 2-1 3-1 (php5: 1-2 2-3 3-)

$a = [1,2,3]; $b = $a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; } // 1-1 2-1 3-1 (php5: 1-1 2-1 3-1)

$a = [1,2,3]; **foreach**($a **as** & $v) { **echo** $v ."-". current($a) ." "; } // 1-1 2-1 3-1 (php5: 1-2 2-3 3-)

// В PHP 7 все строки выведут один результат: 1-1 2-1 3-1

// В PHP 5 он будет разный

// Пример 2: внутренний переключатель можно переключить с помощью next()

$a = [1,2,3]; **foreach**($a **as** $v) { **echo** $v ."-". current($a) ." "; next($a); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; next($b); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

$a = [1,2,3]; $b = $a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; next($b); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

$a = [1,2,3]; **foreach**($a **as** & $v) { **echo** $v ."-". current($a) ." "; next($a); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

// В PHP 7 все строки выведут один результат: 1-1 2-2 3-3

// В PHP 5 результат тоже будет один, но другой: 1-2 2-3 3-

// Пример 3:

// при &$v foreach работает с оригиналом $a и изменение массива влияет на цикл

$a = [1,2,3]; **foreach**($a **as** &$v) { **echo** "**$v** "; **unset**($a[1]); } // 1 3 (php5: 1 3)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** &$v) { **echo** "**$v** "; **unset**($a[1]); } // 1 3 (php5: 1 3)

// в PHP 5 и PHP 7 совпадают

// при $v foreach работает с копией $a и изменение массива НЕ влияет на цикл

$a = [1,2,3]; **foreach**($a **as** $v) { **echo** "**$v** "; **unset**($a[1]); } // 1 2 3 (php5: 1 2 3)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** $v) { **echo** "**$v** "; **unset**($b[1]); } // 1 2 3 (php5: 1 3)

// в PHP 5 и PHP 7 результаты отличаются

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### $class = new class{} — анонимные классы

Анонимные классы позволяют делать тоже самое что и обычные классы: передавать данные в конструктор, наследовать другие классы, использовать трейты и т.п.

$class = **new** **class** {

**public** **function** **echo**( $msg ){

**echo** $msg;

}

};

$class->echo('Привет!'); // выведет на экран: "Привет!"

Расширение классов работает как и ожидается:

**class** Foo {}

$child = **new** **class** **extends** Foo {};

var\_dump( $child **instanceof** Foo ); //> true

Использование треитов:

**trait** Foo {

**public** **function** method() {

**return** "bar";

}

}

$class = **new** **class** {

**use** Foo;

};

var\_dump( $class->method() ); //> string(3) "bar"

Подробнее про анонимные классы [читайте в документации](http://php.net/manual/ru/language.oop5.anonymous.php) и на [wiki.php.net](https://wiki.php.net/rfc/anonymous_classes).

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### yield ... return 99; — возврат (return) выражений в генераторах

Функции-генераторы появились в PHP 5.5. Но там можно было использовать return, только чтобы прервать работу генератора. Теперь return может возвращать выражение (значение/массив/другой генератор), а не только NULL. Но сделать это можно только в конце работы генератора.

Получить возвращенное значение можно методом getReturn(), но только по завершении работы генератора.

Возможность явно вернуть последнее значение упрощает работу с генераторами:  
теперь не нужно проверять является ли значение последним, а просто вызываем getReturn().

**function** gen() {

**yield** 1;

**yield** 2;

**return** 3;

}

$gen = gen();

// если генератор еще ничего не вернул, то вызов такой строки

// echo $gen->getReturn();

// вызовет ошибку: Fatal error: Uncaught Exception: Cannot get return value of a generator that hasn't returned

**foreach**( $gen **as** $val ) {

**echo** $val;

}

**echo** $gen->getReturn();

// результат работы этого кода выведет на экран: 123

Подробнее: [Generator Return Expressions](https://wiki.php.net/rfc/generator-return-expressions)

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### yield from gen() — делегирование генераторов

Позволяет разбить сложный генератор на несколько простых.

Для этого используется новый синтаксис: yield from <expr>, где <expr> может быть значением (скаляром), массивом или другим генератором.

<expr> будет работать до тех пор, пока возвращает данные, затем выполнение продолжится в генераторе откуда <expr> был вызван. Смотрите пример:

**function** gen() {

**yield** 1;

**yield** from gen2();

**yield** 4;

}

**function** gen2(){

**yield** 2;

**yield** 3;

}

$gen = gen();

**foreach** ( $gen **as** $val ) {

**echo** $val;

}

// результат работы этого кода: 1234

Пример с массивом:

**function** g() {

**yield** 1;

**yield** from [2, 3, 4];

**yield** 5;

}

$g = g();

**foreach** ( $g **as** $yielded ) {

**echo** $yielded;

}

// выведет: 12345

Пример с return из дочернего генератора:

**function** gen() {

**yield** 1;

$sub\_gen = **yield** from sub\_gen();

**yield** 4;

**return** $sub\_gen;

}

**function** sub\_gen() {

**yield** 2;

**yield** 3;

**return** 42;

}

$gen = gen();

**foreach**( $gen **as** $val ) {

**echo** $val;

}

**echo** ' - '. $gen->getReturn();

// выведет: 1234 - 42

Подробнее: [Generator Delegation](https://wiki.php.net/rfc/generator-delegation)

[к началу](https://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

#### Еще новинки PHP 7.0

1. Синтаксис конструкторов в стиле PHP 4 (имя метода конструктора совпадает с именем класса) теперь считается устаревшим.
2. Статичные вызовы :: нестатичных методов теперь считаются устаревшими.
3. list() — изменение поведения. В PHP 5, list() устанавливал значения начиная с правого крайнего значения указанного массива, в PHP 7 параметры устанавливаются начиная с левого крайнего значения массива. Так же в PHP 5 list() умела разбивать строки на символы, в PHP 7 не работает со строками вообще...
4. // Пример 1: обратное чтение
5. // Если используются обычные переменные, то разницы нет
6. **list**( $a, $b, $c ) = ['apple', 'bannana', 'cherry', 'damson'];
7. var\_dump( $a, $b, $c ); // php5 и php7 вернут: apple bannana cherry
8. // А вот если устанавливаются элементы массива, то порядок будет отличаться
9. $arr = [];
10. **list**( $arr['a'], $arr['b'], $arr['c'] ) = ['apple', 'bannana', 'cherry', 'damson'];
11. print\_r( $arr );
12. /\*
13. PHP 7
14. Array
15. (
16. [a] => apple
17. [b] => bannana
18. [c] => cherry
19. )
20. PHP 5
21. Array
22. (
23. [c] => cherry
24. [b] => bannana
25. [a] => apple
26. )
27. \*/
28. // Пример 2: разбивание строк
29. $str = 'ab';
30. **list**( $a, $b ) = $str;
31. var\_dump( $a, $b );
32. // В PHP 7: NULL NULL

// В PHP 5: string(1) "a" string(1) "b"

1. Поддержка юникод управляющих (escape-) последовательностей. Т.е. в строках "" и heredoc можно использовать конструкцию \uXXXX для создания юникод символа. Вот так:

**echo** "\u{1F602}"; //> ?

Подробнее читайте: [Unicode Codepoint Escape Syntax](https://wiki.php.net/rfc/unicode_escape)

1. Класс IntlChar. Cодержит методы и константы для работы с юникодом.
2. printf('%x', IntlChar::CODEPOINT\_MAX); // 10ffff
3. **echo** IntlChar::ord('@'); //> 64
4. **echo** IntlChar::chr( 64 ); //> @
5. **echo** "\u{1F602}"; //> ?
6. **echo** IntlChar::ord("\u{1F602}"); //> 128514

**echo** IntlChar::chr( 128514 ); //> ?

1. Функция intdiv() — делит 2 числа и возвращает только целую часть от деления:
2. **echo** intdiv(10, 3); //> 3

**echo** intdiv(5, 2); //> 2

1. session\_start() умеет получать параметры (стандартные настройки сессий из php.ini):

session\_start(['cache\_limiter' => 'private']);

1. Функция preg\_replace\_callback\_array() — альтернатива preg\_replace\_callback(). Позволяет передать в качестве обратной функции - массив ['/regex'/ => callback, ...]:
2. $str = 'a1a2a3';
3. $array = [
4. '~[0-9]~' => **function** ( $m ){ **return** $m[0] \* 2; },
5. '~a~' => **function** ( $m ){ **return** $m[0] . '-'; }
6. ];

**echo** preg\_replace\_callback\_array( $array, $str ); //> a-2a-4a-6

1. Можно использовать глобальные ключевые слова в названиях методов. Т.е. раньше нельзя было назвать метод словами: with/new/for/foreach/... — это приводило к ошибке. Теперь можно:
2. **Class**::new('Project Name');

$class->for('purpose here');

Подробнее о новинках PHP 7 читайте в [этой статье](http://www.skillz.ru/dev/php/article-php7-whats-new.html) и [вторая часть](http://www.skillz.ru/dev/php/article-php7-whats-new-part-2.html).

## PHP 7.1

#### void — возвращаемый тип

Теперь функции и методы, которые не должны ничего возвращать, можно помечать возвращаемым типом void. Оператор return при этом должен отсутствовать или должен быть пустым - return;. Вызов return null; вызовет ошибку.

**function** someMethod(): void {

// работает если return отсутствует

// работает с return;

// не работает если return null;

// не работает если return 123;

}

#### iterable — новый псевдо-тип

Введен новый тип iterable для передаваемых/возвращаемых значений. Может использоваться при передаче массивов или объектов, которые соответствуют интерфейсу Traversable.

**function** myfunc( iterable $data ){

**foreach**( $data **as** $k => $v ){

**echo** $k, ':', $v, PHP\_EOL;

}

}

// массив

myfunc([10, 20, 30]); // 0:10 1:20 2:30

// объект

myfunc( **new** SplFixedArray(3) ) // 0: 1: 2:

// генератор

**function** myGen(){

**yield** 10;

**yield** 20;

**yield** 30;

}

myfunc( myGen() ); // 0:10 1:20 2:30

Подробнее [читайте по ссылке](https://wiki.php.net/rfc/iterable).

#### null — тип передаваемых/возвращаемых значений

В PHP 7.0 стало возможным указать тип возвращаемых/передаваемых значений, но типизация не допускала использование null в качестве значения параметра.

В PHP 7.1 для разрешения null-значений перед типом параметра указывается "?":

**function** myfunc(?int $i) : ?int {

var\_dump($a);

**return** $a;

}

myfunc(20); // int(20)

myfunc(**null**); // null

myfunc(); // Ошибка: Uncaught Error: Too few arguments to function name(), 0 passed

Подробнее [читайте по ссылке](https://wp-kama.ru/id_6310/%5bhttps:/wiki.php.net/rfc/iterable%5d(https:/wiki.php.net/rfc/nullable_types)).

#### {-1} — отрицательное значение смещения в строках

Добавлена возможность использовать отрицательное значение для смещения в строках

**echo** $msg[-1]; // последний символ

**echo** $msg{-1}; // последний символ

#### ['key'=>$a] = ['key'=>'Значение'] — поддержка ключей и новый синтаксис list()

Теперь, можно использовать новый синтаксис - аналог list(), в котором разрешено использовать строковые ключи:

['test'=>$a, 'name'=>$b] = ['name'=>'Hello', 'test'=>'World!'];

var\_dump($a); // World!

var\_dump($b); // Hello

Замена list():

**list**($one, $two) = ['один', 'два']; // list style

[$one, $two] = ['один', 'два']; // [] style

**echo** "**$one**, **$two**";

#### Область видимости констант в классах

Конец публичным константам, теперь для констант можно указать видимость:

**class** ConstClass {

**const** CONST\_ONE = 1; // public

**public** **const** CONST\_TWO = 2;

**protected** **const** CONST\_THREE = 3;

**private** **const** CONST\_FOUR = 4;

}

#### Заметки по PHP 7.1

PHP движется в сторону строгой типизации данных и при переходе на 7.1 я столкнулся с ФАТАЛЬНОЙ ошибкой. И мне это показалось очень странным. Приведу пример:

$foo = '';

$foo['bar'] = 'мир'; // Warning: Illegal string offset 'bar'

$foo['bar'][] = 'мир'; // Fatal error: Uncaught Error: Cannot use string offset as an array

// фатальная ошибка: нельзя использовать отступ строки как массив...

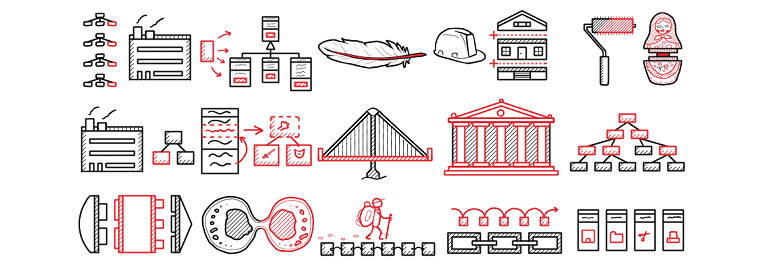
При Warning PHP еще работает, а дальше уже нет! А еще в 7.0 код просто работал, даже без предупреждений и нотисов... Похоже на недоработку в PHP 7.1.

К примеру, ошибка такого типа есть в популярном плагине WP Super Cache (отписал авторам, надеюсь скоро поправят).

Весь список изменение [смотрите по ссылке](https://github.com/php/php-src/blob/PHP-7.1.0/UPGRADING) и [по этой](http://www.skillz.ru/dev/php/article-php71-whats-new.html).

# Шаблоны проектирования простым языком. Часть вторая. Структурные шаблоны

* 4 июля 2017 в 14:34, [Переводы](https://tproger.ru/category/translations/)
* 18 минут
* 1 305

[](https://cdn.tproger.ru/wp-content/uploads/2017/05/designpatternsmini.png)

Рассказывает [*Камран Ахмед*](https://github.com/kamranahmedse)

Шаблоны проектирования — это руководства по решению повторяющихся проблем. Это не классы, пакеты или библиотеки, которые можно было бы подключить к вашему приложению и сидеть в ожидании чуда. Они скорее являются методиками решения определенных проблем в определенных ситуациях.

Википедия [описывает](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) их следующим образом:

**Шаблон проектирования,** или **паттерн**, в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

### Будьте осторожны

* шаблоны проектирования не являются решением всех ваших проблем;
* не пытайтесь использовать их в обязательном порядке — это может привести к негативным последствиям. Шаблоны — это подходы к решению проблем, а не решения для поиска проблем;
* если их правильно использовать в нужных местах, то они могут стать спасением, а иначе могут привести к ужасному беспорядку.

Также заметьте, что примеры ниже написаны на PHP 7. Но это не должно вас останавливать, ведь принципы остаются такими же.

### Типы шаблонов

Шаблоны бывают следующих трех видов:

1. [Порождающие](https://tproger.ru/translations/design-patterns-simple-words-1/).
2. Структурные**—** о них мы рассказываем в этой статье.
3. [Поведенческие](https://tproger.ru/translations/design-patterns-simple-words-3/).

**Простыми словами:**Структурные шаблоны в основном связаны с композицией объектов, другими словами, с тем, как сущности могут использовать друг друга. Ещё одним объяснением было бы то, что они помогают ответить на вопрос «Как создать программный компонент?».

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F):

**Структурные шаблоны** — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Список структурных шаблонов проектирования:

* [адаптер (Adapter)](https://tproger.ru/translations/design-patterns-simple-words-2/#21);
* [мост (Bridge)](https://tproger.ru/translations/design-patterns-simple-words-2/#22);
* [компоновщик (Composite)](https://tproger.ru/translations/design-patterns-simple-words-2/#23);
* [декоратор (Decorator)](https://tproger.ru/translations/design-patterns-simple-words-2/#24);
* [фасад (Facade)](https://tproger.ru/translations/design-patterns-simple-words-2/#25);
* [приспособленец (Flyweight)](https://tproger.ru/translations/design-patterns-simple-words-2/#26);
* [заместитель (Proxy)](https://tproger.ru/translations/design-patterns-simple-words-2/#27).

## Адаптер (Adapter)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)):

**Адаптер** — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

**Пример из жизни:** Представим, что у вас на карте памяти есть какие-то изображения и вам надо перенести их на ваш компьютер. Чтобы это сделать, вам нужен какой-то адаптер, который совместим с портами вашего компьютера. В этом случае карт-ридер — это адаптер. Другим примером будет блок питания. Вилку с тремя ножками нельзя вставить в розетку с двумя отверстиями. Для того, чтобы она подошла, надо использовать адаптер. Ещё одним примером будет переводчик, переводящий слова одного человека для другого.

**Простыми словами:** Шаблон позволяет обернуть несовместимые объекты в адаптер, чтобы сделать их совместимыми с другим классом.

Обратимся к коду. Представим игру, в которой охотник охотится на львов.

Изначально у нас есть интерфейс Lion, который реализует всех львов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | interface Lion  {      public function roar();  }    class AfricanLion implements Lion  {      public function roar()      {      }  }    class AsianLion implements Lion  {      public function roar()      {      }  } |

И Hunter охотится на любую реализацию интерфейса Lion:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Hunter  {      public function hunt(Lion $lion)      {      }  } |

Теперь представим, что нам надо добавить WildDog в нашу игру, на которую наш Hunterтакже мог бы охотиться. Но мы не можем сделать это напрямую, потому что у WildDogдругой интерфейс. Чтобы сделать её совместимой с нашим Hunter, нам надо создать адаптер:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Это надо добавить в игру  class WildDog  {      public function bark()      {      }  }    // Адаптер, чтобы сделать <code>WildDog</code> совместимой с нашей игрой class WildDogAdapter implements Lion { protected $dog; public function \_\_construct(WildDog $dog) { $this->dog = $dog; } public function roar() { $this->dog->bark(); } } |

Способ применения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | $wildDog = new WildDog();  $wildDogAdapter = new WildDogAdapter($wildDog);    $hunter = new Hunter();  $hunter->hunt($wildDogAdapter); |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/adapter" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/structural/adapter.py" \t "_blank).

## Мост (Bridge)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%81%D1%82_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Мост** — структурный шаблон проектирования, используемый в проектировании программного обеспечения чтобы разделять абстракцию и реализацию так, чтобы они могли изменяться независимо. Шаблон мост использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

**Пример из жизни:** Представим, что у вас есть сайт с разными страницами, и вам надо разрешить пользователям менять их тему. Что вы будете делать? Создавать множественные копии каждой страницы для каждой темы или просто отдельную тему, которую пользователь сможет выбрать сам? Шаблон мост позволяет вам сделать второе.

**Простыми словами:** Шаблон мост — это предпочтение композиции над наследованием. Детали реализации передаются из одной иерархии в другой объект с отдельной иерархией.

Обратимся к примеру в коде. Возьмем пример с нашими страницами. У нас есть иерархия WebPage:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | interface WebPage  {      public function \_\_construct(Theme $theme);      public function getContent();  }    class About implements WebPage  {      protected $theme;        public function \_\_construct(Theme $theme)      {          $this->theme = $theme;      }        public function getContent()      {          return "Страница с информацией в " . $this->theme->getColor();      }  }    class Careers implements WebPage  {      protected $theme;        public function \_\_construct(Theme $theme)      {          $this->theme = $theme;      }        public function getContent()      {          return "Страница карьеры в " . $this->theme->getColor();      }  } |

И отдельная иерархия Theme:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | interface Theme  {      public function getColor();  }    class DarkTheme implements Theme  {      public function getColor()      {          return 'темной теме';      }  }  class LightTheme implements Theme  {      public function getColor()      {          return 'светлой теме';      }  }  class AquaTheme implements Theme  {      public function getColor()      {          return 'голубой теме';      }  } |

Применение в коде:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $darkTheme = new DarkTheme();    $about = new About($darkTheme);  $careers = new Careers($darkTheme);    echo $about->getContent(); // "Страница информации в темной теме";  echo $careers->getContent(); // "Страница карьеры в темной теме"; |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/bridge" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/structural/bridge.py" \t "_blank).

## Компоновщик (Composite)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Компоновщик** — структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково. Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.

**Пример из жизни:** Каждая организация скомпонована из сотрудников. У каждого сотрудника есть одинаковые свойства, такие как зарплата, обязанности, отчётность и т.д.

**Простыми словами:** Шаблон компоновщик позволяет клиентам работать с индивидуальными объектами в едином стиле.

Обратимся к коду. Возьмем наш пример с рабочими. У нас есть Employee разных типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | interface Assignee {    public function canHandleTask($task): bool;    public function takeTask($task);  }    class Employee implements Assignee {    // реализуем методы интерфейса  }    class Team implements Assignee {    /\*\* @var Assignee[] \*/    private $assignees;      // вспомогательные методы для управления композитом:    public function add($assignee);    public function remove($assignee);      // метода интерфейса Employee      public function canHandleTask($task): bool {      foreach ($this->assignees as $assignee) if ($assignee->canHandleTask($task)) return true;      return false;    }    public function takeTask($task) {      // может быть разная имплементация - допустим, некоторые задания требуют нескольких человек из команды одновременно      // в простейшем случае берем первого незанятого работника среди this->assignees      $assignee = ...;      $assignee->takeTask($task);    }  } |

Теперь у нас есть TaskManager:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class TaskManager {    private $assignees;    public function performTask($task) {      foreach ($this->assignees as $assignee) {         if ($assignee->canHandleTask($task)) {           $assignee->takeTask($task);           return;         }      }        throw new Exception('Cannot handle the task - please hire more people');    }  } |

Способ применения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $employee1 = new Employee();  $employee2 = new Employee();  $employee3 = new Employee();  $employee4 = new Employee();  $team1 = new Team([$employee3, $employee4);    // ВНИМАНИЕ: передаем команду в taskManager как единый композит.  // Сам taskManager не знает, что это команда и работает с ней без модификации своей логики.  $taskManager = new TaskManager([$employee1, $employee2, $team1]);  $taskManager->preformTask($task); |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/composite" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/structural/composite.py" \t "_blank).

## Декоратор (Decorator)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Декоратор** — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

**Пример из жизни:** Представим, что у вас есть свой автосервис. Как вы будете рассчитывать сумму в счете за услуги? Вы выбираете одну услугу и динамически добавляете к ней цены на предоставляемые услуги, пока не получите окончательную стоимость. Здесь каждый тип сервиса является декоратором.

**Простыми словами:** Шаблон декоратор позволяет вам динамически изменять поведение объекта во время работы, оборачивая их в объект класса декоратора.

Перейдем к коду. Возьмем пример с кофе. Изначально у нас есть простой Coffeeи реализующий его интерфейс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | interface Coffee  {      public function getCost();      public function getDescription();  }    class SimpleCoffee implements Coffee  {      public function getCost()      {          return 10;      }        public function getDescription()      {          return 'Простой кофе';      }  } |

Мы хотим сделать код расширяемым, чтобы при необходимости можно было изменять его. Давайте сделаем некоторые дополнения (декораторы):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | class MilkCoffee implements Coffee  {      protected $coffee;        public function \_\_construct(Coffee $coffee)      {          $this->coffee = $coffee;      }        public function getCost()      {          return $this->coffee->getCost() + 2;      }        public function getDescription()      {          return $this->coffee->getDescription() . ', молоко';      }  }    class WhipCoffee implements Coffee  {      protected $coffee;        public function \_\_construct(Coffee $coffee)      {          $this->coffee = $coffee;      }        public function getCost()      {          return $this->coffee->getCost() + 5;      }        public function getDescription()      {          return $this->coffee->getDescription() . ', сливки';      }  }    class VanillaCoffee implements Coffee  {      protected $coffee;        public function \_\_construct(Coffee $coffee)      {          $this->coffee = $coffee;      }        public function getCost()      {          return $this->coffee->getCost() + 3;      }        public function getDescription()      {          return $this->coffee->getDescription() . ', ваниль';      }  } |

А теперь приготовим Coffee:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | $someCoffee = new SimpleCoffee();  echo $someCoffee->getCost(); // 10  echo $someCoffee->getDescription(); // Простой кофе    $someCoffee = new MilkCoffee($someCoffee);  echo $someCoffee->getCost(); // 12  echo $someCoffee->getDescription(); // Простой кофе, молоко    $someCoffee = new WhipCoffee($someCoffee);  echo $someCoffee->getCost(); // 17  echo $someCoffee->getDescription(); // Простой кофе, молоко, сливки    $someCoffee = new VanillaCoffee($someCoffee);  echo $someCoffee->getCost(); // 20  echo $someCoffee->getDescription(); // Простой кофе, молоко, сливки, ваниль |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/decorator" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/structural/decorator.py" \t "_blank).

## Фасад (Facade)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D1%81%D0%B0%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Фасад** — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

**Пример из жизни:** Как вы включаете компьютер? Нажимаю на кнопку включения, скажете вы. Это то, во что вы верите, потому что вы используете простой интерфейс, который компьютер предоставляет для доступа снаружи. Внутри же должно произойти гораздо больше вещей. Этот простой интерфейс для сложной подсистемы называется фасадом.

**Простыми словами:** Шаблон фасад предоставляет упрощенный интерфейс для сложной системы.

Перейдем к примерам в коде. Возьмем пример с компьютером. Изначально у нас есть класс Computer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | class Computer  {      public function getElectricShock()      {          echo "Ай!";      }        public function makeSound()      {          echo "Бип-бип!";      }        public function showLoadingScreen()      {          echo "Загрузка..";      }        public function bam()      {          echo "Готов к использованию!";      }        public function closeEverything()      {          echo "Буп-буп-буп-бззз!";      }        public function sooth()      {          echo "Zzzzz";      }        public function pullCurrent()      {          echo "Аах!";      }  } |

Затем у нас есть фасад:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class ComputerFacade  {      protected $computer;        public function \_\_construct(Computer $computer)      {          $this->computer = $computer;      }        public function turnOn()      {          $this->computer->getElectricShock();          $this->computer->makeSound();          $this->computer->showLoadingScreen();          $this->computer->bam();      }        public function turnOff()      {          $this->computer->closeEverything();          $this->computer->pullCurrent();          $this->computer->sooth();      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3 | $computer = new ComputerFacade(new Computer());  $computer->turnOn(); // Ай! Бип-бип! Загрузка.. Готов к использованию!  $computer->turnOff(); // Буп-буп-буп-бззз! Аах! Zzzzz |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/facade" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/structural/facade.py" \t "_blank).

## Приспособленец (Flyweight)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%BB%D0%B5%D0%BD%D0%B5%D1%86_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Приспособленец** — структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

**Пример из жизни:** Вы когда-нибудь заказывали чай в уличном ларьке? Там зачастуют готовят не одну чашку, которую вы заказали, а гораздо большую емкость. Это делается для того, чтобы экономить ресурсы (газ/электричество). Газ/электричество в этом примере и являются приспособленцами, ресурсы которых делятся (sharing).

**Простыми словами:** Приспособленец используется для минимизации использования памяти или вычислительной стоимости путем разделения ресурсов с наибольшим количеством похожих объектов.

Перейдем к примерам в коде. Возьмем наш пример с чаем. Изначально у нас есть различные виды Tea и TeaMaker:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // Все, что будет закешировано, является приспособленцем.  // Типы чая здесь будут приспособленцами.  class KarakTea  {  }    //Ведет себя как фабрика и сохраняет чай  class TeaMaker  {      protected $availableTea = [];        public function make($preference)      {          if (empty($this->availableTea[$preference])) {              $this->availableTea[$preference] = new KarakTea();          }            return $this->availableTea[$preference];      }  } |

Теперь у нас есть TeaShop, который принимает заказы и выполняет их:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class TeaShop  {      protected $orders;      protected $teaMaker;        public function \_\_construct(TeaMaker $teaMaker)      {          $this->teaMaker = $teaMaker;      }        public function takeOrder(string $teaType, int $table)      {          $this->orders[$table] = $this->teaMaker->make($teaType);      }        public function serve()      {          foreach ($this->orders as $table => $tea) {              echo "Serving tea to table# " . $table;          }      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $teaMaker = new TeaMaker();  $shop = new TeaShop($teaMaker);    $shop->takeOrder('меньше сахара', 1);  $shop->takeOrder('больше молока', 2);  $shop->takeOrder('без сахара', 5);    $shop->serve();  // Подаем чай на первый стол  // Подаем чай на второй стол  // Подаем чай на пятый стол |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/flyweight" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/structural/flyweight.py" \t "_blank).

## Заместитель (Proxy)

Википедия [гласит](https://ru.wikipedia.org/wiki/Proxy_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Заместитель** — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

**Пример из жизни:** Вы когда-нибудь использовали карту доступа, чтобы пройти через дверь? Есть несколько способов открыть дверь: например, она может быть открыта при помощи карты доступа или нажатия кнопки, которая обходит защиту. Основная функциональность двери — это открытие, но заместитель, добавленный поверх этого, добавляет функциональность. Но лучше я объясню это на примере кода чуть ниже.

**Простыми словами:** Используя шаблон заместитель, класс отображает функциональность другого класса.

Перейдем к коду. Возьмем наш пример с безопасностью. Сначала у нас есть интерфейс Door и его реализация:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | interface Door  {      public function open();      public function close();  }    class LabDoor implements Door  {      public function open()      {          echo "Открытие дверь лаборатории";      }        public function close()      {          echo "Закрытие двери лаборатории";      }  } |

Затем у нас есть заместитель Security для защиты любых наших дверей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class Security  {      protected $door;        public function \_\_construct(Door $door)      {          $this->door = $door;      }        public function open($password)      {          if ($this->authenticate($password)) {              $this->door->open();          } else {              echo "Нет! Это невозможно.";          }      }        public function authenticate($password)      {          return $password === '$ecr@t';      }        public function close()      {          $this->door->close();      }  } |

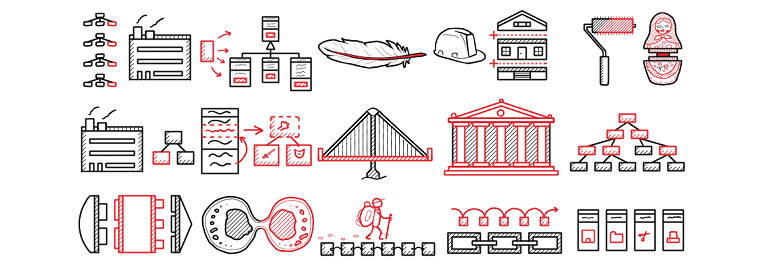
Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5 | $door = new Security(new LabDoor());  $door->open('invalid'); // Нет! Это невозможно.    $door->open('$ecr@t'); // Открытие двери лаборатории  $door->close(); // Закрытие двери лаборатории |

Другим примером будет реализация маппинга данных. Например, недавно я создал ODM (Object Data Mapper) для MongoDB, используя этот шаблон, где я написал заместитель вокруг классов mongo и использовал магический метод \_\_call(). Все вызовы методов были замещены оригинальным классом mongo, и полученный результат возвращался без изменений, но в случае find или findOne данные сопоставлялись необходимому классу, и возвращались в объект вместо Cursor.

# Шаблоны проектирования простым языком. Часть первая. Порождающие шаблоны

* 1 июня 2017 в 12:14, [Переводы](https://tproger.ru/category/translations/)
* 16 минут
* 5 716

[](https://cdn.tproger.ru/wp-content/uploads/2017/05/designpatternsmini.png)

Рассказывает [*Камран Ахмед*](https://github.com/kamranahmedse)

Шаблоны проектирования — это руководства по решению повторяющихся проблем. Это не классы, пакеты или библиотеки, которые можно было бы подключить к вашему приложению и сидеть в ожидании чуда. Они скорее являются методиками, как решать определенные проблемы в определенных ситуациях.

Википедия [описывает](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) их следующим образом:

**Шаблон проектирования,** или **паттерн**, в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

### Будьте осторожны

* шаблоны проектирования не являются решением всех ваших проблем;
* не пытайтесь использовать их в обязательном порядке — это может привести к негативным последствиям. Шаблоны — это подходы к решению проблем, а не решения для поиска проблем;
* если их правильно использовать в нужных местах, то они могут стать спасением, а иначе могут привести к ужасному беспорядку.

Также заметьте, что примеры ниже написаны на PHP 7. Но это не должно вас останавливать, ведь принципы остаются такими же.

### Типы шаблонов

Шаблоны бывают следующих трех видов:

1. Порождающие.
2. [Структурные](https://tproger.ru/translations/design-patterns-simple-words-2/).
3. [Поведенческие](https://tproger.ru/translations/design-patterns-simple-words-3/).

Если говорить простыми словами, то это шаблоны, которые предназначены для создания экземпляра объекта или группы связанных объектов.

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F):

**Порождающие шаблоны** — шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Существуют следующие порождающие шаблоны:

* [простая фабрика (Simple Factory)](https://tproger.ru/translations/design-patterns-simple-words-1/#11);
* [фабричный метод (Factory Method)](https://tproger.ru/translations/design-patterns-simple-words-1/#12);
* [абстрактная фабрика (Abstract Factory)](https://tproger.ru/translations/design-patterns-simple-words-1/#13);
* [строитель (Builder)](https://tproger.ru/translations/design-patterns-simple-words-1/#14);
* [прототип (Prototype)](https://tproger.ru/translations/design-patterns-simple-words-1/#15);
* [одиночка (Singleton)](https://tproger.ru/translations/design-patterns-simple-words-1/#16).

## Простая фабрика (Simple Factory)

Википедия [гласит](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)):

В объектно-ориентированном программировании (ООП), **фабрика** — это объект для создания других объектов. Формально фабрика — это функция или метод, который возвращает объекты изменяющегося прототипа или класса из некоторого вызова метода, который считается «новым».

**Пример из жизни:**Представьте, что вам надо построить дом, и вам нужны двери. Было бы глупо каждый раз, когда вам нужны двери, надевать вашу столярную форму и начинать делать дверь. Вместо этого вы делаете её на фабрике.

**Простыми словами:**Простая фабрика генерирует экземпляр для клиента, не раскрывая никакой логики.

Перейдем к коду. У нас есть интерфейс Door и его реализация:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | interface Door  {      public function getWidth(): float;      public function getHeight(): float;  }    class WoodenDoor implements Door  {      protected $width;      protected $height;        public function \_\_construct(float $width, float $height)      {          $this->width = $width;          $this->height = $height;      }        public function getWidth(): float      {          return $this->width;      }        public function getHeight(): float      {          return $this->height;      }  } |

Затем у нас есть наша DoorFactory, которая делает дверь и возвращает её:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class DoorFactory  {      public static function makeDoor($width, $height): Door      {          return new WoodenDoor($width, $height);      }  } |

И затем мы можем использовать всё это:

|  |  |
| --- | --- |
| 1  2  3 | $door = DoorFactory::makeDoor(100, 200);  echo 'Width: ' . $door->getWidth();  echo 'Height: ' . $door->getHeight(); |

**Когда использовать:** Когда создание объекта — это не просто несколько присвоений, а какая-то логика, тогда имеет смысл создать отдельную фабрику вместо повторения одного и того же кода повсюду.

[Пример](https://github.com/iluwatar/java-design-patterns/tree/master/factory-kit) на Java.

## Фабричный метод (Fabric Method)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)):

**Фабричный метод** — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

**Пример из жизни:**Рассмотрим пример с менеджером по найму. Невозможно одному человеку провести собеседования со всеми кандидатами на все вакансии. В зависимости от вакансии он должен распределить этапы собеседования между разными людьми.

**Простыми словами:**Менеджер предоставляет способ делегирования логики создания экземпляра дочерним классам.

Перейдём к коду. Рассмотрим приведенный выше пример про HR-менеджера. Изначально у нас есть интерфейс Interviewer и несколько реализаций для него:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | interface Interviewer  {      public function askQuestions();  }    class Developer implements Interviewer  {      public function askQuestions()      {          echo 'Спрашивает про шаблоны проектирования!';      }  }    class CommunityExecutive implements Interviewer  {      public function askQuestions()      {          echo 'Спрашивает о работе с сообществом';      }  } |

Теперь создадим нашего HiringManager:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | abstract class HiringManager  {        // Фабричный метод      abstract public function makeInterviewer(): Interviewer;        public function takeInterview()      {          $interviewer = $this->makeInterviewer();          $interviewer->askQuestions();      }  } |

И теперь любой дочерний класс может расширять его и предоставлять необходимого интервьюера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class DevelopmentManager extends HiringManager  {      public function makeInterviewer(): Interviewer      {          return new Developer();      }  }    class MarketingManager extends HiringManager  {      public function makeInterviewer(): Interviewer      {          return new CommunityExecutive();      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5 | $devManager = new DevelopmentManager();  $devManager->takeInterview(); // Вывод: Спрашивает о шаблонах проектирования!    $marketingManager = new MarketingManager();  $marketingManager->takeInterview(); // Вывод: Спрашивает о работе с сообществом |

**Когда использовать:** Полезен, когда есть некоторая общая обработка в классе, но необходимый подкласс динамически определяется во время выполнения. Иными словами, когда клиент не знает, какой именно подкласс ему может понадобиться.

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/factory-method" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/creational/factory_method.py" \t "_blank).

## Абстрактная фабрика (Abstract Factory)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B0%D1%8F_%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Абстрактная фабрика** — порождающий шаблон проектирования, предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

**Пример из жизни:**Расширим наш пример про двери из простой фабрики. В зависимости от ваших нужд вам понадобится деревянная дверь из одного магазина, железная дверь — из другого или пластиковая — из третьего. Кроме того, вам понадобится соответствующий специалист: столяр для деревянной двери, сварщик для железной двери и так далее. Как вы можете заметить, тут есть зависимость между дверьми.

**Простыми словами:**Фабрика фабрик. Фабрика, которая группирует индивидуальные, но связанные/зависимые фабрики без указания их конкретных классов.

Обратимся к коду. Используем пример про двери. Сначала у нас есть интерфейс Doorи несколько его реализаций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | interface Door  {      public function getDescription();  }    class WoodenDoor implements Door  {      public function getDescription()      {          echo 'Я деревянная дверь';      }  }    class IronDoor implements Door  {      public function getDescription()      {          echo 'Я железная дверь';      }  } |

Затем у нас есть несколько DoorFittingExpert для каждого типа дверей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | interface DoorFittingExpert  {      public function getDescription();  }    class Welder implements DoorFittingExpert  {      public function getDescription()      {          echo 'Я работаю только с железными дверьми';      }  }    class Carpenter implements DoorFittingExpert  {      public function getDescription()      {          echo 'Я работаю только с деревянными дверьми';      }  } |

Теперь у нас есть DoorFactory, которая позволит нам создать семейство связанных объектов. То есть фабрика деревянных дверей предоставит нам деревянную дверь и эксперта по деревянным дверям. Аналогично для железных дверей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | interface DoorFactory  {      public function makeDoor(): Door;      public function makeFittingExpert(): DoorFittingExpert;  }    // Деревянная фабрика вернет деревянную дверь и столяра  class WoodenDoorFactory implements DoorFactory  {      public function makeDoor(): Door      {          return new WoodenDoor();      }        public function makeFittingExpert(): DoorFittingExpert      {          return new Carpenter();      }  }    // Железная фабрика вернет железную дверь и сварщика  class IronDoorFactory implements DoorFactory  {      public function makeDoor(): Door      {          return new IronDoor();      }        public function makeFittingExpert(): DoorFittingExpert      {          return new Welder();      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | $woodenFactory = new WoodenDoorFactory();    $door = $woodenFactory->makeDoor();  $expert = $woodenFactory->makeFittingExpert();    $door->getDescription();  // Вывод: Я деревянная дверь  $expert->getDescription(); // Вывод: Я работаю только с деревянными дверями    // Аналогично для железной двери  $ironFactory = new IronDoorFactory();    $door = $ironFactory->makeDoor();  $expert = $ironFactory->makeFittingExpert();    $door->getDescription();  // Вывод: Я железная дверь  $expert->getDescription(); // Вывод: Я работаю только с железными дверями |

Как вы можете заметить, фабрика деревянных дверей инкапсулирует столяра и деревянную дверь, а фабрика железных дверей инкапсулирует железную дверь и сварщика. Это позволило нам убедиться, что для каждой двери мы получим нужного нам эксперта.

**Когда использовать:** Когда есть взаимосвязанные зависимости с не очень простой логикой создания.

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/abstract-factory" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/creational/abstract_factory.py" \t "_blank).

## Строитель (Builder)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Строитель** — порождающий шаблон проектирования, который предоставляет способ создания составного объекта. Предназначен для решения проблемы антипаттерна «Телескопический конструктор».

**Пример из жизни:**Представьте, что вы пришли в McDonalds и заказали конкретный продукт, например, БигМак, и вам готовят его без лишних вопросов. Это пример простой фабрики. Но есть случаи, когда логика создания может включать в себя больше шагов. Например, вы хотите индивидуальный сэндвич в Subway: у вас есть несколько вариантов того, как он будет сделан. Какой хлеб вы хотите? Какие соусы использовать? Какой сыр? В таких случаях на помощь приходит шаблон «Строитель».

**Простыми словами:**Шаблон позволяет вам создавать различные виды объекта, избегая засорения конструктора. Он полезен, когда может быть несколько видов объекта или когда необходимо множество шагов, связанных с его созданием.

Давайте я покажу на примере, что такое «Телескопический конструктор». Когда-то мы все видели конструктор вроде такого:

|  |  |
| --- | --- |
| 1  2  3 | public function \_\_construct($size, $cheese = true, $pepperoni = true, $tomato = false, $lettuce = true)  {  } |

Как вы можете заметить, количество параметров конструктора может резко увеличиться, и станет сложно понимать расположение параметров. Кроме того, этот список параметров будет продолжать расти, если вы захотите добавить новые варианты. Это и есть «Телескопический конструктор».

Перейдем к примеру в коде. Адекватной альтернативой будет использование шаблона «Строитель». Сначала у нас есть Burger, который мы хотим создать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Burger  {      protected $size;        protected $cheese = false;      protected $pepperoni = false;      protected $lettuce = false;      protected $tomato = false;        public function \_\_construct(BurgerBuilder $builder)      {          $this->size = $builder->size;          $this->cheese = $builder->cheese;          $this->pepperoni = $builder->pepperoni;          $this->lettuce = $builder->lettuce;          $this->tomato = $builder->tomato;      }  } |

Затем мы берём «Строителя»:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | class BurgerBuilder  {      public $size;        public $cheese = false;      public $pepperoni = false;      public $lettuce = false;      public $tomato = false;        public function \_\_construct(int $size)      {          $this->size = $size;      }        public function addPepperoni()      {          $this->pepperoni = true;          return $this;      }        public function addLettuce()      {          $this->lettuce = true;          return $this;      }        public function addCheese()      {          $this->cheese = true;          return $this;      }        public function addTomato()      {          $this->tomato = true;          return $this;      }        public function build(): Burger      {          return new Burger($this);      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5 | $burger = (new BurgerBuilder(14))                      ->addPepperoni()                      ->addLettuce()                      ->addTomato()                      ->build(); |

**Когда использовать:** Когда может быть несколько видов объекта и надо избежать «телескопического конструктора». Главное отличие от «фабрики» — это то, что она используется, когда создание занимает один шаг, а «строитель» применяется при множестве шагов.

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/builder" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/creational/builder.py" \t "_blank).

## Прототип (Prototype)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Он позволяет уйти от реализации и позволяет следовать принципу «программирование через интерфейсы». В качестве возвращающего типа указывается интерфейс / абстрактный класс на вершине иерархии, а классы-наследники могут подставить туда наследника, реализующего этот тип.

**Пример из жизни:**Помните Долли? Овечка, которая была клонирована. Не будем углубляться, главное — это то, что здесь все вращается вокруг клонирования.

**Простыми словами:**Прототип создает объект, основанный на существующем объекте при помощи клонирования.

То есть он позволяет вам создавать копию существующего объекта и модернизировать его согласно вашим нуждам, вместо того, чтобы создавать объект заново.

Обратимся к коду. В PHP это может быть легко реализовано с использованием clone:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class Sheep  {      protected $name;      protected $category;        public function \_\_construct(string $name, string $category = 'Горная овечка')      {          $this->name = $name;          $this->category = $category;      }        public function setName(string $name)      {          $this->name = $name;      }        public function getName()      {          return $this->name;      }        public function setCategory(string $category)      {          $this->category = $category;      }        public function getCategory()      {          return $this->category;      }  } |

Затем он может быть клонирован следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $original = new Sheep('Джолли');  echo $original->getName(); // Джолли  echo $original->getCategory(); // Горная овечка    // Клонируем и модифицируем то что нужно  $cloned = clone $original;  $cloned->setName('Долли');  echo $cloned->getName(); // Долли  echo $cloned->getCategory(); // Горная овечка |

Также вы можете использовать волшебный метод \_\_clone для изменения клонирующего поведения.

**Когда использовать:** Когда необходим объект, похожий на существующий объект, либо когда создание будет дороже клонирования.

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/prototype" \t "_blank) и [Python](https://github.com/faif/python-patterns/blob/master/creational/prototype.py" \t "_blank).

## Одиночка (Singleton)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%B8%D0%BD%D0%BE%D1%87%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Одиночка** — порождающий шаблон проектирования, гарантирующий, что в однопроцессном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.

**Пример из жизни:**В стране одновременно может быть только один президент. Один и тот же президент должен действовать, когда того требуют обстоятельства. Президент здесь является одиночкой.

**Простыми словами:**Обеспечивает тот факт, что создаваемый объект является единственным объектом своего класса.

Вообще шаблон одиночка признан антипаттерном, необходимо избегать его чрезмерного использования. Он необязательно плох и может иметь полезные применения, но использовать его надо с осторожностью, потому что он вводит глобальное состояние в ваше приложение и его изменение в одном месте может повлиять на другие части приложения, что вызовет трудности при отладке. Другой минус — это то, что он делает ваш код связанным.

**Прим. перев.** Подробнее о подводных камнях шаблона одиночка читайте в [нашей статье](https://tproger.ru/translations/singleton-pitfalls/" \t "_blank).

Перейдем к коду. Чтобы создать одиночку, сделайте конструктор приватным, отключите клонирование и расширение и создайте статическую переменную для хранения экземпляра:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | final class President  {      private static $instance;        private function \_\_construct()      {          // Прячем конструктор      }        public static function getInstance(): President      {          if (!self::$instance) {              self::$instance = new self();          }            return self::$instance;      }        private function \_\_clone()      {          // Отключаем клонирование      }        private function \_\_wakeup()      {          // Отключаем десериализацию      }  } |

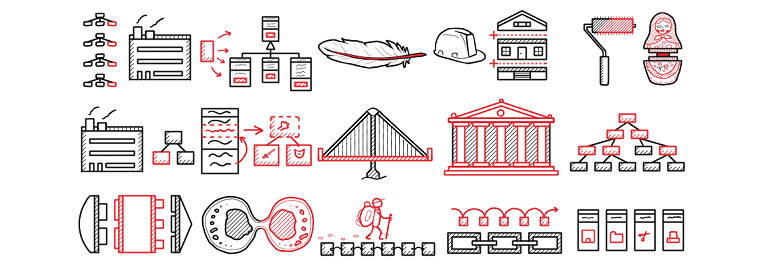
Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4 | $president1 = President::getInstance();  $president2 = President::getInstance();    var\_dump($president1 === $president2); // true |

Пример на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/singleton" \t "_blank).

# Шаблоны проектирования простым языком. Часть третья. Поведенческие шаблоны

* 4 июля 2017 в 14:40, [Переводы](https://tproger.ru/category/translations/)
* 27 минут
* 1 181

[](https://cdn.tproger.ru/wp-content/uploads/2017/05/designpatternsmini.png)

Рассказывает [*Камран Ахмед*](https://github.com/kamranahmedse)

Шаблоны проектирования — это руководства по решению повторяющихся проблем. Это не классы, пакеты или библиотеки, которые можно было бы подключить к вашему приложению и сидеть в ожидании чуда. Они скорее являются методиками, как решать определенные проблемы в определенных ситуациях.

Википедия [описывает](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) их следующим образом:

**Шаблон проектирования,** или **паттерн**, в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

### Будьте осторожны

* шаблоны проектирования не являются решением всех ваших проблем;
* не пытайтесь насильно использовать их, из-за этого могут произойти плохие вещи. Шаблоны — решения проблем, а не решения для поиска проблем;
* если их правильно использовать в нужных местах, то они могут стать спасением, а иначе могут привести к ужасному беспорядку.

Также заметьте, что примеры ниже написаны на PHP 7. Но это не должно вас останавливать, ведь принципы остаются такими же.

### Типы шаблонов

Шаблоны бывают следующих трех видов:

1. [Порождающие](https://tproger.ru/translations/design-patterns-simple-words-1/).
2. [Структурные](https://tproger.ru/translations/design-patterns-simple-words-2/).
3. Поведенческие**—** о них мы рассказываем в этой статье.

**Простыми словами:**Поведенческие шаблоны связаны с распределением обязанностей между объектами. Их отличие от структурных шаблонов заключается в том, что они не просто описывают структуру, но также описывают шаблоны для передачи сообщений / связи между ними. Или, другими словами, они помогают ответить на вопрос «Как запустить поведение в программном компоненте?»

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F):

**Поведенческие шаблоны** — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

Поведенческие шаблоны:

* [цепочка обязанностей (Chain of Responsibility)](https://tproger.ru/translations/design-patterns-simple-words-3/#31);
* [команда (Command)](https://tproger.ru/translations/design-patterns-simple-words-3/#32);
* [итератор (Iterator)](https://tproger.ru/translations/design-patterns-simple-words-3/#33);
* [посредник (Mediator)](https://tproger.ru/translations/design-patterns-simple-words-3/#34);
* [хранитель (Memento)](https://tproger.ru/translations/design-patterns-simple-words-3/#35);
* [наблюдатель (Observer)](https://tproger.ru/translations/design-patterns-simple-words-3/#36);
* [посетитель (Visitor)](https://tproger.ru/translations/design-patterns-simple-words-3/#37);
* [стратегия (Strategy)](https://tproger.ru/translations/design-patterns-simple-words-3/#38);
* [состояние (State)](https://tproger.ru/translations/design-patterns-simple-words-3/#39);
* [шаблонный метод (Template Method)](https://tproger.ru/translations/design-patterns-simple-words-3/#30).

## Цепочка обязанностей (Chain of Responsibility)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BF%D0%BE%D1%87%D0%BA%D0%B0_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B5%D0%B9):

**Цепочка обязанностей** — поведенческий шаблон проектирования предназначенный для организации в системе уровней ответственности.

**Пример из жизни:**например, у вас есть три платежных метода (A, B и C), настроенных на вашем банковском счёте. На каждом лежит разное количество денег. На A есть 100 долларов, на B есть 300 долларов и на C — 1000 долларов. Предпочтение отдается в следующем порядке: A, B и C. Вы пытаетесь заказать что-то, что стоит 210 долларов. Используя цепочку обязанностей, первым на возможность оплаты будет проверен метод А, и в случае успеха пройдет оплата и цепь разорвется. Если нет, то запрос перейдет к методу B для аналогичной проверки. Здесь A, B и C — это звенья цепи, а все явление — цепочка обязанностей.

**Простыми словами:**цепочка обязанностей помогает строить цепочки объектов. Запрос входит с одного конца и проходит через каждый объект, пока не найдет подходящий обработчик.

Обратимся к коду. Приведем пример с банковскими счетами. Изначально у нас есть базовый Account с логикой для соединения счетов цепью и некоторые счета:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | abstract class Account  {      protected $successor;      protected $balance;        public function setNext(Account $account)      {          $this->successor = $account;      }        public function pay(float $amountToPay)      {          if ($this->canPay($amountToPay)) {              echo sprintf('Оплата %s, используя %s' . PHP\_EOL, $amountToPay, get\_called\_class());          } elseif ($this->successor) {              echo sprintf('Нельзя заплатить, используя %s. Обработка ..' . PHP\_EOL, get\_called\_class());              $this->successor->pay($amountToPay);          } else {              throw new Exception('Ни на одном из аккаунтов нет необходимого количества денег');          }      }        public function canPay($amount): bool      {          return $this->balance >= $amount;      }  }    class Bank extends Account  {      protected $balance;        public function \_\_construct(float $balance)      {          $this->balance = $balance;      }  }    class Paypal extends Account  {      protected $balance;        public function \_\_construct(float $balance)      {          $this->balance = $balance;      }  }    class Bitcoin extends Account  {      protected $balance;        public function \_\_construct(float $balance)      {          $this->balance = $balance;      }  } |

Теперь приготовим цепь, используя объявленные выше звенья (например, Bank, Paypal, Bitcoin):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // Подготовим цепь  //      $bank->$paypal->$bitcoin  //  // Первый по приоритету банк  //      Если нельзя через банк, то Paypal  //      Если нельзя через Paypal, то Bitcoin    $bank = new Bank(100);          // Банк с балансом 100  $paypal = new Paypal(200);      // Paypal с балансом 200  $bitcoin = new Bitcoin(300);    // Bitcoin с балансом 300    $bank->setNext($paypal);  $paypal->setNext($bitcoin);    // Попробуем оплатить через банк  $bank->pay(259);    // Вывод  // ==============  // Нельзя заплатить, используя Банк. Обработка ..  // Нельзя заплатить, используя Paypal. Обработка ..:  // Оплата 259, используя Bitcoin! |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/chain)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/chain.py).

## Команда (Command)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)):

**Команда** — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

**Пример из жизни:**Типичный пример: вы заказываете еду в ресторане. Вы (т.е. Client) просите официанта (например, Invoker) принести еду (то есть Command), а официант просто переправляет запрос шеф-повару (то есть Receiver), который знает, что и как готовить. Другим примером может быть то, что вы (Client) включаете (Command) телевизор (Receiver) с помощью пульта дистанционного управления (Invoker).

**Простыми словами:**Позволяет вам инкапсулировать действия в объекты. Основная идея, стоящая за шаблоном — это предоставление средств, для разделения клиента и получателя.

Обратимся к коду. Изначально у нас есть получатель Bulb, в котором есть реализация каждого действия, которое может быть выполнено:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Получатель  class Bulb  {      public function turnOn()      {          echo "Лампочка загорелась";      }        public function turnOff()      {          echo "Темнота!";      }  } |

Затем у нас есть интерфейс Command, которая каждая команда должна реализовывать, и затем у нас будет набор команд:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | interface Command  {      public function execute();      public function undo();      public function redo();  }    // Команда  class TurnOn implements Command  {      protected $bulb;        public function \_\_construct(Bulb $bulb)      {          $this->bulb = $bulb;      }        public function execute()      {          $this->bulb->turnOn();      }        public function undo()      {          $this->bulb->turnOff();      }        public function redo()      {          $this->execute();      }  }    class TurnOff implements Command  {      protected $bulb;        public function \_\_construct(Bulb $bulb)      {          $this->bulb = $bulb;      }        public function execute()      {          $this->bulb->turnOff();      }        public function undo()      {          $this->bulb->turnOn();      }        public function redo()      {          $this->execute();      }  } |

Затем у нас есть Invoker, с которым клиент будет взаимодействовать для обработки любых команд:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Invoker  class RemoteControl  {      public function submit(Command $command)      {          $command->execute();      }  } |

Наконец, мы можем увидеть, как использовать нашего клиента:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $bulb = new Bulb();    $turnOn = new TurnOn($bulb);  $turnOff = new TurnOff($bulb);    $remote = new RemoteControl();  $remote->submit($turnOn); // Лампочка загорелась!  $remote->submit($turnOff); // Темнота! |

Шаблон команда может быть использован для реализации системы, основанной на транзакциях, где вы сохраняете историю команд, как только их выполняете. Если окончательная команда успешно выполнена, то все хорошо, иначе алгоритм просто перебирает историю и продолжает выполнять отмену для всех выполненных команд.

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/command" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/command.py" \t "_blank).

## Итератор (Iterator)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Итератор** — поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

**Пример из жизни:**Старый радионабор будет хорошим предметом итератора, где пользователь может начать искать сигнал на каком-то канале и затем использовать кнопки переключения на следующий и предыдущий канал для перехода между соответствующими каналами. Или используем пример телевизора, где вы можете нажимать кнопки следующего или предыдущего канала для перехода через последовательные каналы, или, иными словами, они предоставляют интерфейс для итерирования между соответствующими каналами, песнями или радиостанциями.

**Простыми словами:**Представляет способ доступа к элементам объекта без показа базового представления.

Обратимся к примерам в коде. В PHP очень просто реализовать это, используя SPL (Standard PHP Library). Приводя наш пример с радиостанциями, изначально у нас есть Radiostation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class RadioStation  {      protected $frequency;        public function \_\_construct(float $frequency)      {          $this->frequency = $frequency;      }        public function getFrequency(): float      {          return $this->frequency;      }  } |

Затем у нас есть итератор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | use Countable;  use Iterator;    class StationList implements Countable, Iterator  {      /\*\* @var RadioStation[] $stations \*/      protected $stations = [];        /\*\* @var int $counter \*/      protected $counter;        public function addStation(RadioStation $station)      {          $this->stations[] = $station;      }        public function removeStation(RadioStation $toRemove)      {          $toRemoveFrequency = $toRemove->getFrequency();          $this->stations = array\_filter($this->stations, function (RadioStation $station) use ($toRemoveFrequency) {              return $station->getFrequency() !== $toRemoveFrequency;          });      }        public function count(): int      {          return count($this->stations);      }        public function current(): RadioStation      {          return $this->stations[$this->counter];      }        public function key()      {          return $this->counter;      }        public function next()      {          $this->counter++;      }        public function rewind()      {          $this->counter = 0;      }        public function valid(): bool      {          return isset($this->stations[$this->counter]);      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | $stationList = new StationList();    // Добавление станций  $stationList->addStation(new RadioStation(89));  $stationList->addStation(new RadioStation(101));  $stationList->addStation(new RadioStation(102));  $stationList->addStation(new RadioStation(103.2));    foreach($stationList as $station) {      echo $station->getFrequency() . PHP\_EOL;  }    $stationList->removeStation(new RadioStation(89)); // Удалит 89 станцию |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/iterator" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/iterator.py" \t "_blank).

## Посредник (Mediator)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D1%80%D0%B5%D0%B4%D0%BD%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Посредник** — поведенческий шаблон проектирования, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность, и избавляя объекты, от необходимости явно ссылаться друг на друга.

**Пример из жизни:**Общим примером будет, когда вы говорите с кем-то по мобильнику, то между вами и собеседником находится мобильный оператор. То есть сигнал передаётся через него, а не напрямую. В данном примере оператор — посредник.

**Простыми словами:**Шаблон посредник подразумевает добавление стороннего объекта (посредника) для управления взаимодействием между двумя объектами (коллегами). Шаблон помогает уменьшить связанность (coupling) классов, общающихся друг с другом, ведь теперь они не должны знать о реализациях своих собеседников.

Разберем пример в коде. Простейший пример: чат (посредник), в котором пользователи (коллеги) отправляют друг другу сообщения.

Изначально у нас есть посредник ChatRoomMediator:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | interface ChatRoomMediator  {      public function showMessage(User $user, string $message);  }    // Посредник  class ChatRoom implements ChatRoomMediator  {      public function showMessage(User $user, string $message)      {          $time = date('M d, y H:i');          $sender = $user->getName();            echo $time . '[' . $sender . ']:' . $message;      }  } |

Затем у нас есть наши User (коллеги):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class User {      protected $name;      protected $chatMediator;        public function \_\_construct(string $name, ChatRoomMediator $chatMediator) {          $this->name = $name;          $this->chatMediator = $chatMediator;      }        public function getName() {          return $this->name;      }        public function send($message) {          $this->chatMediator->showMessage($this, $message);      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $mediator = new ChatRoom();    $john = new User('John Doe', $mediator);  $jane = new User('Jane Doe', $mediator);    $john->send('Привет!');  $jane->send('Привет!');    // Вывод  // Feb 14, 10:58 [John]: Привет!  // Feb 14, 10:58 [Jane]: Привет! |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/mediator" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/mediator.py" \t "_blank).

## Хранитель (Memento)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A5%D1%80%D0%B0%D0%BD%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Хранитель** — поведенческий шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в этом состоянии.

**Пример из жизни:** В качестве примера можно привести калькулятор (создатель), у которого любая последняя выполненная операция сохраняется в памяти (хранитель), чтобы вы могли снова вызвать её с помощью каких-то кнопок (опекун).

**Простыми словами:** Шаблон хранитель фиксирует и хранит текущее состояние объекта, чтобы оно легко восстанавливалось.

Обратимся к коду. Возьмем наш пример текстового редактора, который время от времени сохраняет состояние, которое вы можете восстановить.

Изначально у нас есть наш объект EditorMemento, который может содержать состояние редактора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class EditorMemento  {      protected $content;        public function \_\_construct(string $content)      {          $this->content = $content;      }        public function getContent()      {          return $this->content;      }  } |

Затем у нас есть наш Editor (создатель), который будет использовать объект хранитель:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Editor  {      protected $content = '';        public function type(string $words)      {          $this->content = $this->content . ' ' . $words;      }        public function getContent()      {          return $this->content;      }        public function save()      {          return new EditorMemento($this->content);      }        public function restore(EditorMemento $memento)      {          $this->content = $memento->getContent();      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | $editor = new Editor();    // Печатаем что-нибудь  $editor->type('Это первое предложение.');  $editor->type('Это второе.');    // Сохраняем состояние для восстановления : Это первое предложение. Это второе.  $saved = $editor->save();    // Печатаем ещё  $editor->type('И это третье.');    // Вывод: Данные до сохранения  echo $editor->getContent(); // Это первое предложение. Это второе. И это третье.    // Восстановление последнего сохранения  $editor->restore($saved);    $editor->getContent(); // Это первое предложение. Это второе. |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/memento" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/memento.py" \t "_blank).

## Наблюдатель (Observer)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B1%D0%BB%D1%8E%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Наблюдатель** — поведенческий шаблон проектирования, также известен как «подчинённые» (Dependents). Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

**Пример из жизни:** Хороший пример: люди, ищущие работу, подписываются на публикации на сайтах вакансий и получают уведомления, когда появляются вакансии подходящие по параметрам.

**Простыми словами:** Шаблон определяет зависимость между объектами, чтобы при изменении состояния одного из них зависимые от него узнавали об этом.

Обратимся к коду. Приводя наш пример. Изначально у нас есть JobSeeker, которые ищут работы JobPost и должны быть уведомлены о её появлении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class JobPost  {      protected $title;        public function \_\_construct(string $title)      {          $this->title = $title;      }        public function getTitle()      {          return $this->title;      }  }    class JobSeeker implements Observer  {      protected $name;        public function \_\_construct(string $name)      {          $this->name = $name;      }        public function onJobPosted(JobPost $job)      {          // Делаем что-то с публикациями вакансий          echo 'Привет ' . $this->name . '! Появилась новая работа: '. $job->getTitle();      }  } |

Затем мы делаем публикации JobPostings на которые соискатели могут подписываться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class JobPostings implements Observable  {      protected $observers = [];        protected function notify(JobPost $jobPosting)      {          foreach ($this->observers as $observer) {              $observer->onJobPosted($jobPosting);          }      }        public function attach(Observer $observer)      {          $this->observers[] = $observer;      }        public function addJob(JobPost $jobPosting)      {          $this->notify($jobPosting);      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // Создаем соискателей  $johnDoe = new JobSeeker('John Doe');  $janeDoe = new JobSeeker('Jane Doe');    // Создаем публикацию и добавляем подписчика  $jobPostings = new JobPostings();  $jobPostings->attach($johnDoe);  $jobPostings->attach($janeDoe);    // Добавляем новую работу и смотрим получит ли соискатель уведомление  $jobPostings->addJob(new JobPost('Software Engineer'));    // Вывод  // Привет John Doe! Появилась новая работа: Software Engineer  // Привет Jane Doe! Появилась новая работа: Software Engineer |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/observer" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/observer.py" \t "_blank).

## Посетитель (Visitor)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%B5%D1%82%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Посетитель** — поведенческий шаблон проектирования, описывающий операцию, которая выполняется над объектами других классов. При изменении visitor нет необходимости изменять обслуживаемые классы.

**Пример из жизни:** Туристы собрались в Дубай. Сначала им нужен способ попасть туда (виза). После прибытия они будут посещать любую часть города, не спрашивая разрешения ходить где вздумается. Просто скажите им о каком-нибудь месте — и туристы могут там побывать. Шаблон посетитель помогает добавлять места для посещения.

**Простыми словами:** Шаблон посетитель позволяет добавлять будущие операции для объектов без их модифицирования.

Перейдем к примерам в коде. Возьмём зоопарк: у нас есть несколько видов Animal, и нам нужно послушать издаваемые ими звуки.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Посещаемый  interface Animal  {      public function accept(AnimalOperation $operation);  }    // Посетитель  interface AnimalOperation  {      public function visitMonkey(Monkey $monkey);      public function visitLion(Lion $lion);      public function visitDolphin(Dolphin $dolphin);  } |

Затем у нас есть реализация для животных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | class Monkey implements Animal  {      public function shout()      {          echo 'У-у-а-а!';      }        public function accept(AnimalOperation $operation)      {          $operation->visitMonkey($this);      }  }    class Lion implements Animal  {      public function roar()      {          echo 'рррр!';      }        public function accept(AnimalOperation $operation)      {          $operation->visitLion($this);      }  }    class Dolphin implements Animal  {      public function speak()      {          echo '\*звуки дельфина\*!'; // Я понятия не имею как описать их звуки      }        public function accept(AnimalOperation $operation)      {          $operation->visitDolphin($this);      }  } |

Давайте реализуем посетителя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Speak implements AnimalOperation  {      public function visitMonkey(Monkey $monkey)      {          $monkey->shout();      }        public function visitLion(Lion $lion)      {          $lion->roar();      }        public function visitDolphin(Dolphin $dolphin)      {          $dolphin->speak();      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $monkey = new Monkey();  $lion = new Lion();  $dolphin = new Dolphin();    $speak = new Speak();    $monkey->accept($speak);    // У-у-а-а!  $lion->accept($speak);      // Рррр!  $dolphin->accept($speak);   // \*звуки дельфина\*! |

Это можно было сделать просто с помощью иерархии наследования, но тогда пришлось бы модифицировать животных при каждом добавлении к ним новых действий. А здесь менять их не нужно. Например, мы можем добавить животным прыжки, просто создав нового посетителя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Jump implements AnimalOperation  {      public function visitMonkey(Monkey $monkey)      {          echo 'Прыгает на 20 футов!';      }        public function visitLion(Lion $lion)      {          echo 'Прыгает на 7 футов!';      }        public function visitDolphin(Dolphin $dolphin)      {          echo 'Появился над водой и исчез!';      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $jump = new Jump();    $monkey->accept($speak);   // У-у-а-а!  $monkey->accept($jump);    // Прыгает на 20 футов!    $lion->accept($speak);     // Рррр!  $lion->accept($jump);      // Прыгает на 7 футов!    $dolphin->accept($speak);  // \*звуки дельфинов\*!  $dolphin->accept($jump);   // Появился над водой и исчез |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/visitor" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/visitor.py" \t "_blank).

## Стратегия (Strategy)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B0%D1%82%D0%B5%D0%B3%D0%B8%D1%8F_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Стратегия** — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

**Пример из жизни:** Возьмём пример с пузырьковой сортировкой. Мы её реализовали, но с ростом объёмов данных сортировка работа стала выполняться очень медленно. Тогда мы сделали быструю сортировку. Алгоритм работает быстрее на больших объёмах, но на маленьких он очень медленный. Тогда мы реализовали стратегию, при которой для маленьких объёмов данных используется пузырьковая сортировка, а для больших объёмов — быстрая.

**Простыми словами:** Шаблон стратегия позволяет переключаться между алгоритмами или стратегиями в зависимости от ситуации.

Перейдем к коду. Возьмем наш пример. Изначально у нас есть наша SortStrategy и разные её реализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | interface SortStrategy  {      public function sort(array $dataset): array;  }    class BubbleSortStrategy implements SortStrategy  {      public function sort(array $dataset): array      {          echo "Сортировка пузырьком";            // Сортировка          return $dataset;      }  }    class QuickSortStrategy implements SortStrategy  {      public function sort(array $dataset): array      {          echo "Быстрая сортировка";            // Сортировка          return $dataset;      }  } |

И у нас есть Sorter, который собирается использовать какую-то стратегию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Sorter  {      protected $sorter;        public function \_\_construct(SortStrategy $sorter)      {          $this->sorter = $sorter;      }        public function sort(array $dataset): array      {          return $this->sorter->sort($dataset);      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $dataset = [1, 5, 4, 3, 2, 8];    $sorter = new Sorter(new BubbleSortStrategy());  $sorter->sort($dataset); // Вывод : Сортировка пузырьком    $sorter = new Sorter(new QuickSortStrategy());  $sorter->sort($dataset); // Вывод : Быстрая сортировка |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/strategy" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/strategy.py" \t "_blank).

## Состояние (State)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Состояние** — поведенческий шаблон проектирования. Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

**Пример из жизни:** Допустим, в графическом редакторе вы выбрали кисть. Она меняет своё поведение в зависимости от настройки цвета, т. е. рисует линию выбранного цвета.

**Простыми словами:** Шаблон позволяет менять поведение класса при изменении состояния.

Перейдем к примерам в коде. Возьмем пример текстового редактора, он позволяет вам менять состояние напечатанного текста. Например, если у вас выбран курсив, то он будет писать курсивом и так далее.

Изначально у нас есть интерфейс WritingState и несколько его реализаций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | interface WritingState  {      public function write(string $words);  }    class UpperCase implements WritingState  {      public function write(string $words)      {          echo strtoupper($words);      }  }    class LowerCase implements WritingState  {      public function write(string $words)      {          echo strtolower($words);      }  }    class Default implements WritingState  {      public function write(string $words)      {          echo $words;      }  } |

Затем TextEditor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class TextEditor  {      protected $state;        public function \_\_construct(WritingState $state)      {          $this->state = $state;      }        public function setState(WritingState $state)      {          $this->state = $state;      }        public function type(string $words)      {          $this->state->write($words);      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | $editor = new TextEditor(new Default());    $editor->type('Первая строка');    $editor->setState(new UpperCase());    $editor->type('Вторая строка');  $editor->type('Третья строка');    $editor->setState(new LowerCase());    $editor->type('Четвертая строка');  $editor->type('Пятая строка');    // Output:  // Первая строка  // ВТОРАЯ СТРОКА  // ТРЕТЬЯ СТРОКА  // четвертая строка  // пятая строка |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/state" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/state.py" \t "_blank).

## Шаблонный метод (Template Method)

Википедия [гласит](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \t "_blank):

**Шаблонный метод** — поведенческий шаблон проектирования, определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

**Пример из жизни:** Допустим, вы собрались строить дома. Этапы будут такими:

1. Подготовка фундамента.
2. Возведение стен.
3. Настил крыши.
4. Настил перекрытий.

Порядок этапов никогда не меняется. Вы не настелите крышу до возведения стен и т. д. Но каждый этап модифицируется: стены, например, можно возвести из дерева, кирпича или газобетона.

**Простыми словами:** Шаблонный метод определяет каркас выполнения определённого алгоритма, но реализацию самих этапов делегирует дочерним классам.

Обратимся к коду. Допустим, у нас есть программный инструмент, позволяющий тестировать, проводить контроль качества кода, выполнять сборку, генерировать отчёты сборки (отчёты о покрытии кода, о качестве кода и т. д.), а также развёртывать приложение на тестовом сервере.

Изначально у нас есть наш Builder, который описывает скелет для построения алгоритма:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | abstract class Builder  {        // Шаблонный метод      final public function build()      {          $this->test();          $this->lint();          $this->assemble();          $this->deploy();      }        abstract public function test();      abstract public function lint();      abstract public function assemble();      abstract public function deploy();  } |

Затем у нас есть его реализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | class AndroidBuilder extends Builder  {      public function test()      {          echo 'Запуск Android тестов';      }        public function lint()      {          echo 'Копирование Android кода';      }        public function assemble()      {          echo 'Android сборка';      }        public function deploy()      {          echo 'Развертывание сборки на сервере';      }  }    class IosBuilder extends Builder  {      public function test()      {          echo 'Запуск iOS тестов';      }        public function lint()      {          echo 'Копирование iOS кода';      }        public function assemble()      {          echo 'iOS сборка';      }        public function deploy()      {          echo 'Развертывание сборки на сервере';      }  } |

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | $androidBuilder = new AndroidBuilder();  $androidBuilder->build();    // Вывод:  // Запуск Android тестов  // Копирование Android кода  // Android сборка  // Развертывание сборки на сервере    $iosBuilder = new IosBuilder();  $iosBuilder->build();    // Вывод:  // Запуск iOS тестов  // Копирование iOS кода  // iOS сборка  // Развертывание сборки на сервере |

Примеры на [Java](https://github.com/iluwatar/java-design-patterns/tree/master/template-method" \t "_blank)и [Python](https://github.com/faif/python-patterns/blob/master/behavioral/template.py" \t "_blank).

# Шпаргалка по SOLID-принципам с примерами на PHP

[Разработка веб-сайтов](https://habrahabr.ru/hub/webdev/)\*, [ООП](https://habrahabr.ru/hub/oop/)\*, [PHP](https://habrahabr.ru/hub/php/)\*

Тема SOLID-принципов и в целом чистоты кода не раз поднималась на Хабре и, возможно, уже порядком изъезженная. Но тем не менее, не так давно мне приходилось проходить собеседования в одну интересную IT-компанию, где меня попросили рассказать о принципах SOLID с примерами и ситуациями, когда я не соблюл эти принципы и к чему это привело. И в тот момент я понял, что на каком-то подсознательном уровне я понимаю эти принципы и даже могут назвать их все, но привести лаконичные и понятные примеры для меня стало проблемой. Поэтому я и решил для себя самого и для сообщества обобщить информацию по SOLID-принципам для ещё лучшего её понимания. Статья должна быть полезной, для людей только знакомящихся с SOLID-принципами, также, как и для людей «съевших собаку» на SOLID-принципах.  
  
  
Для тех, кто знаком с принципами и хочет только освежить память о них и их использовании, можно обратиться сразу к шпаргалке в конце статьи.  
  
Что же такое SOLID-принципы? Если верить определению Wikipedia, это:

аббревиатура пяти основных принципов дизайна классов в объектно-ориентированном проектировании — **S**ingle responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation и **D**ependency inversion.

Таким образом, мы имеем 5 принципов, которые и рассмотрим ниже:

* Принцип единственной ответственности (Single responsibility)
* Принцип открытости/закрытости (Open-closed)
* Принцип подстановки Барбары Лисков (Liskov substitution)
* Принцип разделения интерфейса (Interface segregation)
* Принцип инверсии зависимостей (Dependency Invertion)

#### Принцип единственной ответственности (Single responsibility)

Итак, в качества примера возьмём довольно популярный и широкоиспользуемый пример — интернет-магазин с заказами, товарами и покупателями.  
  
Принцип единственной ответственности гласит — *«На каждый объект должна быть возложена одна единственная обязанность»*. Т.е. другими словами — конкретный класс должен решать конкретную задачу — ни больше, ни меньше.  
  
Рассмотрим следующее описание класса для представления заказа в интернет-магазине:

**class** **Order**

{

**public** **function** **calculateTotalSum**(){*/\*...\*/*}

**public** **function** **getItems**(){*/\*...\*/*}

**public** **function** **getItemCount**(){*/\*...\*/*}

**public** **function** **addItem**($item){*/\*...\*/*}

**public** **function** **deleteItem**($item){*/\*...\*/*}

**public** **function** **printOrder**(){*/\*...\*/*}

**public** **function** **showOrder**(){*/\*...\*/*}

**public** **function** **load**(){*/\*...\*/*}

**public** **function** **save**(){*/\*...\*/*}

**public** **function** **update**(){*/\*...\*/*}

**public** **function** **delete**(){*/\*...\*/*}

}

Как можно увидеть, данный класс выполняет операций для 3 различный типов задач: работа с самим заказом(calculateTotalSum, getItems, getItemsCount, addItem, deleteItem), отображение заказа(printOrder, showOrder) и работа с хранилищем данных(load, save, update, delete).  
К чему это может привести?  
Приводит это к тому, что в случае, если мы хотим внести изменения в методы печати или работы хранилища, мы изменяем сам класс заказа, что может привести к его неработоспособности.  
Решить эту проблему стоит разделением данного класса на 3 отдельных класса, каждый из которых будет заниматься своей задачей

**class** **Order**

{

**public** **function** **calculateTotalSum**(){*/\*...\*/*}

**public** **function** **getItems**(){*/\*...\*/*}

**public** **function** **getItemCount**(){*/\*...\*/*}

**public** **function** **addItem**($item){*/\*...\*/*}

**public** **function** **deleteItem**($item){*/\*...\*/*}

}

**class** **OrderRepository**

{

**public** **function** **load**($orderID){*/\*...\*/*}

**public** **function** **save**($order){*/\*...\*/*}

**public** **function** **update**($order){*/\*...\*/*}

**public** **function** **delete**($order){*/\*...\*/*}

}

**class** **OrderViewer**

{

**public** **function** **printOrder**($order){*/\*...\*/*}

**public** **function** **showOrder**($order){*/\*...\*/*}

}

Теперь каждый класс занимается своей конкретной задачей и для каждого класса есть только 1 причина для его изменения.

#### Принцип открытости/закрытости (Open-closed)

Данный принцип гласит — "программные сущности должны быть открыты для расширения, но закрыты для модификации". На более простых словах это можно описать так — все классы, функции и т.д. должны проектироваться так, чтобы для изменения их поведения, нам не нужно было изменять их исходный код.  
Рассмотри на примере класса OrderRepository.

**class** **OrderRepository**

{

**public** **function** **load**($orderID)

{

$pdo = **new** PDO($this->config->getDsn(), $this->config->getDBUser(), $this->config->getDBPassword());

$statement = $pdo->prepare('SELECT \* FROM `orders` WHERE id=:id');

$statement->execute(**array**(':id' => $orderID));

**return** $query->fetchObject('Order');

}

**public** **function** **save**($order){*/\*...\*/*}

**public** **function** **update**($order){*/\*...\*/*}

**public** **function** **delete**($order){*/\*...\*/*}

}

В данном случае хранилищем у нас является база данных. например, MySQL. Но вдруг мы захотели подгружать наши данные о заказах, например, через API стороннего сервера, который, допустим, берёт данные из 1С. Какие изменения нам надо будет внести? Есть несколько вариантов, например, непосредственно изменить методы класса OrderRepository, но этот не соответствует *принципу открытости/закрытости*, так как класс закрыт для модификации, да и внесение изменений в уже хорошо работающий класс нежелательно. Значит, можно наследоваться от класса OrderRepository и переопределить все методы, но это решение не самое лучше, так как при добавлении метода в OrderRepository нам придётся добавить аналогичные методы во все его наследники. Поэтому для выполнения *принципа открытости/закрытости* лучше применить следующее решение — создать интерфейc IOrderSource, который будет реализовываться соответствующими классами MySQLOrderSource, ApiOrderSource и так далее.

Интерфейс IOrderSource и его реализация и использование

Таким образом, мы можем изменить источник и соответственно поведение для класса OrderRepository, установив нужный нам класс реализующий IOrderSource, без изменения класса OrderRepository.

#### Принцип подстановки Барбары Лисков (Liskov substitution)

Пожалуй, принцип, который вызывает самые большие затруднения в понимании.   
Принцип гласит — *«Объекты в программе могут быть заменены их наследниками без изменения свойств программы»*. Своими словами я бы это сказал так — при использовании наследника класса результат выполнения кода должен быть предсказуем и не изменять свойств метод.  
К сожалению, придумать доступного примера для это принципа в рамках задачи интернет-магазина я не смог, но есть классический пример с иерархией геометрических фигур и вычисления площади. Код примера ниже.

Пример иерархии прямоугольника и квадрата и вычислении их площади

Очевидно, что такой код явно выполняется не так, как от него этого ждут.   
Но в чём проблема? Разве «квадрат» не является «прямоугольником»? Является, но в геометрических понятиях. В понятиях же объектов, квадрат не есть прямоугольник, поскольку поведение объекта «квадрат» не согласуется с поведением объекта «прямоугольник».  
  
Тогда же как решить проблему?  
Решение тесно связано с таким понятием как *проектирование по контракту*. Описание проектирования по контракту может занять не одну статью, поэтому ограничимся особенностями, которые касаются *принципа Лисков*.  
Проектирование по контракту ведет к некоторым ограничениям на то, как контракты могут взаимодействовать с наследованием, а именно:

* Предусловия не могут быть усилены в подклассе.
* Постусловия не могут быть ослаблены в подклассе.

«Что ещё за пред- и постусловия?» — можете спросите Вы.   
**Ответ**: *предусловия* – это то, что должно быть выполнено вызывающей стороной перед вызовом метода, *постусловия* – это то, что, гарантируется вызываемым методом.  
  
Вернёмся к нашему примеру и посмотрим, как мы изменили пред- и постусловия.  
Предусловия мы никак не использовали при вызове методов установки высоты и ширины, а вот постусловия в классе-наследнике мы изменили и изменили на более слабые, чего по принципу Лисков делать было нельзя.   
Ослабили мы их вот почему. Если за постусловие метода setWidth принять (($this->width == $width) && ($this->height == $oldHeight)) ($oldHeight мы присвоили вначале метода setWidth), то это условие не выполняется в дочернем классе и соответственно мы его ослабили и принцип Лисков нарушен.  
  
Поэтому, лучше в рамках ООП и задачи расчёта площади фигуры не делать иерархию «квадрат» наследует «прямоугольник», а сделать их как 2 отдельные сущности:

**class** **Rectangle**

{

**protected** $width;

**protected** $height;

**public** setWidth($width)

{

$this->width = $width;

}

**public** setHeight($height)

{

$this->height = $height;

}

**public** **function** **getWidth**()

{

**return** $this->width;

}

**public** **function** **getHeight**()

{

**return** $this->height;

}

}

**class** **Square**

{

**protected** $size;

**public** setSize($size)

{

$this->size = $size;

}

**public** **function** **getSize**()

{

**return** $this->size;

}

}

Хороший **реальный** пример несоблюдения принципа Лискоу и решения, принятого в связи с этим, рассмотрен в книге Роберта Мартина «Быстрая разработка программ» в разделе «Принцип подстановки Лискоу. Реальный пример».

#### Принцип разделения интерфейса (Interface segregation)

Данный принцип гласит, что *«Много специализированных интерфейсов лучше, чем один универсальный»*  
Соблюдение этого принципа необходимо для того, чтобы классы-клиенты использующий/реализующий интерфейс знали только о тех методах, которые они используют, что ведёт к уменьшению количества неиспользуемого кода.  
  
Вернёмся к примеру с интернет-магазином.  
Предположим наши товары могут иметь промокод, скидку, у них есть какая-то цена, состояние и т.д. Если это одежда то для неё устанавливается из какого материала сделана, цвет и размер.  
Опишем следующий интерфейс

**interface** **IItem**

{

**public** **function** **applyDiscount**($discount);

**public** **function** **applyPromocode**($promocode);

**public** **function** **setColor**($color);

**public** **function** **setSize**($size);

**public** **function** **setCondition**($condition);

**public** **function** **setPrice**($price);

}

Данный интефейс плох тем, что он включает слишком много методов. А что, если наш класс товаров не может иметь скидок или промокодов, либо для него нет смысла устанавливать материал из которого сделан (например, для книг). Таким образом, чтобы не реализовывать в каждом классе неиспользуемые в нём методы, лучше разбить интерфейс на несколько мелких и каждым конкретным классом реализовывать нужные интерфейсы.

Разбиение интерфейса IItem на несколько

#### Принцип инверсии зависимостей (Dependency Invertion)

Принцип гласит — *«Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций»*. Данное определение можно сократить — *«зависимости должны строится относительно абстракций, а не деталей»*.  
  
Для примера рассмотрим оплату заказа покупателем.

**class** **Customer**

{

**private** $currentOrder = **null**;

**public** **function** **buyItems**()

{

**if**(is\_null($this->currentOrder)){

**return** **false**;

}

$processor = **new** OrderProcessor();

**return** $processor->checkout($this->currentOrder);

}

**public** **function** **addItem**($item){

**if**(is\_null($this->currentOrder)){

$this->currentOrder = **new** Order();

}

**return** $this->currentOrder->addItem($item);

}

**public** **function** **deleteItem**($item){

**if**(is\_null($this->currentOrder)){

**return** **false**;

}

**return** $this->currentOrder ->deleteItem($item);

}

}

**class** **OrderProcessor**

{

**public** **function** **checkout**($order){*/\*...\*/*}

}

Всё кажется вполне логичным и закономерным. Но есть одна проблема — класс Customer зависит от класса OrderProcessor (мало того, не выполняется и принцип открытости/закрытости).  
Для того, чтобы избавится от зависимости от конкретного класса, надо сделать так чтобы Customerзависел от абстракции, т.е. от интерфейса IOrderProcessor. Данную зависимость можно внедрить через сеттеры, параметры метода, или Dependency Injection контейнера. Я решил остановится на 2 методе и получил следующий код.

Инвертирование зависимости класса Customer

Таким образом, класс Customer теперь зависит только от абстракции, а конкретную реализацию, т.е. детали, ему не так важны.

#### Шпаргалка

Резюмируя всё выше изложенное, хотелось бы сделать следующую шпаргалку

* **Принцип единственной ответственности (Single responsibility)**  
  *«На каждый объект должна быть возложена одна единственная обязанность»*  
  Для этого проверяем, сколько у нас есть причин для изменения класса — если больше одной, то следует разбить данный класс.
* **Принцип открытости/закрытости (Open-closed)**  
  *«Программные сущности должны быть открыты для расширения, но закрыты для модификации»*  
  Для этого представляем наш класс как «чёрный ящик» и смотрим, можем ли в таком случае изменить его поведение.
* **Принцип подстановки Барбары Лисков (Liskov substitution)**  
  *«Объекты в программе могут быть заменены их наследниками без изменения свойств программы»*  
  Для этого проверяем, не усилили ли мы предусловия и не ослабили ли постусловия. Если это произошло — то принцип не соблюдается
* **Принцип разделения интерфейса (Interface segregation)**  
  *«Много специализированных интерфейсов лучше, чем один универсальный»*  
  Проверяем, насколько много интерфейс содержит методов и насколько разные функции накладываются на эти методы, и если необходимо — разбиваем интерфейсы.
* **Принцип инверсии зависимостей (Dependency Invertion)**  
  *«Зависимости должны строится относительно абстракций, а не деталей»*  
  Проверяем, зависят ли классы от каких-то других классов(непосредственно инстанцируют объекты других классов и т.д) и если эта зависимость имеет место, заменяем на зависимость от абстракции.

Надеюсь, моя «шпаргалка» поможет кому-нибудь в понимании принципов SOLID и даст толчок к их использованию в своих проектах.  
Спасибо за внимание.  
  
P.S. В комментариях посоветовали хорошую книгу — Роберт Мартин «Быстрая разработка программ». Там очень подробно и с примерами описаны принципы SOLID.

# Почему я выбрал Yii2 [из песочницы](https://habrahabr.ru/sandbox/)

[Yii](https://habrahabr.ru/hub/yii/)\*, [PHP](https://habrahabr.ru/hub/php/)\*

Перевод некоторых глав из книги Bill Keck. [Yii 2 For Beginners](http://leanpub.com/yii2forbeginners).

#### Предисловие

Добро пожаловать в Yii2 для начинающих. Эта книга проведёт вас шаг за шагом сквозь установку и развёртывание, а затем программирование на сегодняшний день самом увлекательном фреймворке Yii2.  
  
Yii2 поставляется в двух вариантах, basic и advanced, и может показаться неестественно, использовать для начинающих шаблон advanced(продвинутый), но этот шаблон, по иронии судьбы, облегчает работу с приложением, если оно использует модели, которые хранятся в базе данных. Современные веб-приложения нуждаются в этой функциональности и шаблон advanced имеет в своём распоряжении для этого готовые решения.  
  
Другое большое преимущество у шаблона advanced, это то, что приложение делится на frontend(пользовательская часть) и backend(модуль управления), который дает исчерпывающий ответ на постоянно возникающий вопрос: «куда же поместить модуль управления?»  
Я бы хотел не только ввести вас в удивительный мир php фреймворка, но и так, же дать вам точку опоры, включающую все, в чем нуждается надежное веб-приложение управляемое базой данных. Хотя advanced шаблон содержит чрезвычайно полезные возможности, в нем, конечно же, отсутствуют важные части, которые мы будем пополнять с помощью этой книги.  
  
Цель — предоставить вам модель шаблона, которую вы сможете использовать для всех ваших будущих проектов.

#### Возможности

С инсталляцией advanced шаблона вы получите некоторые возможности веб-приложения, включающие в себя:

* предопределённую схему таблицы пользователей
* форму входа и регистрации пользователей
* функциональность восстановления забытого пароля
* разделённые области пользователей (frontend) и администраторов(backend)
* автоматическую генерацию кода для моделей, контроллеров и представлений
* встроенную интеграцию с Twitter Bootstrap и приоритетную верстку для мобильных устройств
* надежные виджеты и помощники для представления данных

Если вам что-то не понятно в этом списке, не волнуйтесь, мы раскроем их детально в дальнейшем. Только, знайте — это действительно изумительно, что делает Yii2 для вас. Но дело не в том, как велик тот или иной фреймворк, а в том, как вы все, же сможете это поддерживать и развивать в реальном приложении.  
  
Для этого будем добавлять в приложение все, что в нем отсутствует из коробки:

* красивые усовершенствования пользовательского интерфейса подобно jquery календарю
* методы для зависимых моделей, которые облегчат отображения связанных данных
* методы для контроллеров, которые разграничивают доступ
* расширяемая структура данных, которые будут общими для всех ваших будущих приложений
* управление доступом на основе ролей (RBAC)
* ограничение содержания на основе типа пользователей, таких как «бесплатный» и «платный».
* социальный модуль, который позволит поделиться
* регистрация и вход с помощью Facebook

Это всё, вероятно потребуются для вашего приложения, не взирая на то, кого типа он будет. Поэтому если вы будете изучать Yii2 с помощь этой книги, вы будете в состоянии создать шаблон, который сможете расширять для всех ваших будущих приложений.  
  
Эта книга идеальна для начинающих программистов, кто готов перейти на разработку с помощью фреймворка. Yii2 PHP фреймворк широко масштабируемый и расширяемый, со встроенными возможностями. Мы познакомим вас с этим удивительным фреймворком и объясним детально все, в чём вы будете нуждаться, чтобы знать, как создать и запустить. И вы полюбите Yii2!  
  
Продвинутые PHP специалисты смогут стремительно пронестись сквозь книгу и будут в состоянии быстро создавать и запускать на феноменальном фреймворке Yii2. Что позволит не только сберечь им время на работе, но так, же использовать эффективные рычаги извлечения выгод от фреймворка с открытым кодом, имеющего цельное сообщество за собой.  
  
Тем не менее, стиль изложения в книге рассчитан для начинающих. В книге много подробного описания, чтобы помочь людям, которые имеют небольшой опыт работы с PHP и ещё не могут реально перейти в продвинутое объектно-ориентированное программирование. Мы старались, несомненно, полностью объяснять, как работает фреймворк, как он использует ООП, чтобы создать интуитивный уровень разработки, который позволял бы программистам различного уровня достигать результата, к которому они стремятся.  
  
В любом случае, изучение Yii2 даст вам опыт работы с ООП, с практическим результатом. Вы завершите изучение книги с работающим веб-приложением.

#### Что делает Yii2 фреймворк особенным?

Программисты должны сделать выбор, что будет лежать в основе программирования. Поэтому одно из самых больших решений, которое вы должны сделать как программист, а, может, скорее всего, решение вы будете делать, как часть команды программистов, использовать или нет фреймворк, и если да, то какой.  
  
Что касается вопроса использования PHP фреймворк, есть так много преимуществ, чтобы поступить так, что это становится и ежу понятно.  
  
**Плюсы**  
  
Вот несколько очевидных преимуществ использования фреймворка, который:  
— Использует стандартные способы решения задач, что уменьшает или устраняет запутанность кода.  
— Сокращает время, затрачиваемое на рутинные задачи, такие как проверка форм и безопасность.  
— Облегчает командную работу с помощью соблюдения стандартов.  
— Облегчает поддержку кода с использованием общей архитектуры и методов.  
— Имеет активное сообщество разработчиков, которые поддерживают фреймворк, обобщенные задачи и новые возможности.  
  
**Минусы**  
  
Существует пара недостатков использования фреймворка, которые следует отметить. Во-первых, весь код, который включает в себя фреймворк, создает нагрузку на сервер и это может стать реальной проблемой. К счастью, доступны варианты кэширование, которые позволяют снизить последствия перегрузки, и для корпоративных приложений, вы можете использовать нативный SQL, чтобы минимизировать время выполнения запроса. Поэтому проблема перегрузки сервера не должна остановить вас от использования фреймворка.  
  
Другое дело, очевидно, когда вы работаете с фреймворком, то работаете с огромным количеством кода, который не вы писали и придётся потратить время, чтобы понять, как это все работает. Некоторые части кода могут быть довольно непонятны в зависимости от вашего уровня мастерства и опыта, поэтому не следует ожидать, что вам будет сразу все понятно. Это не произойдёт мгновенно.  
  
Конечно же, вы знаете, что существует путь обучения, благодаря которому вы сейчас читаете эту книгу. И хотя это занимает время, чтобы разобраться с чужим кодом, что может быть мучительно больно, но куда больнее, написать собственную пользовательскую систему с нуля. Учитывая все обстоятельства, использование фреймворка для корпоративной разработки является разумным выбором.  
  
Итак, самое простое — это понять, что использование фреймворка поможет вам разрабатывать более стройный и надежный проект, но теперь самое трудное. Вы должны решить, какой фреймворк использовать.

#### Почему я выбрал Yii2

Я не могу сказать какой фреймворк для вас лучший, что вам интересно и вам лично решать, но я могу немного рассказать о дороге, которая привела меня к Yii2. Это не было решением руководимым необходимостью найти легкий путь изучения PHP, это точно. В моей компании еще в 2012 году, я был в команде разработчиков, которые рассматривали различные фреймворки и должен был решить, какой из них использовать. В то время я даже не мечтал, что напишу книги об одном из них.  
  
В любом случае, мы коллективно исследовали все, что могли найти из основных PHP-фреймворков. Я лично прочитал всю документацию и мы имели долгие инженерные дискуссии о том, что мы думали, что будет работать. Вы не представляете мое разочарование, когда я прочитал всю эту документацию и отошел от этого с чувством менее осведомленным, чем я прежде начал читать все это.  
  
Однако наша команда программистов сделала предпочтение. Они чувствовали, что Yii 1.1.14 был лучшим выбором. Это была версия фреймворка, которая была доступна в то время, мы выбрали её. Поэтому команда приняла этот фреймворк и никогда не оглядывалась назад. Они полюбили её.  
Я, с другой стороны, остался разочарованным. Так как я был только начинающим программистом, и действительно изо всех сил пытался этому научиться. Я не находил этот фреймворк очень интуитивным и понятным. Особенно после сравнения его с другими платформами, где старались все интегрировать красиво, а архитектура Yii казалась мне просто уродливой.  
  
Я так расстроился в один прекрасный момент, что начал искать другой вариант.

#### Другие варианты

Я хотел найти красиво написанную документацию для нового фреймворка и запустить её мимо команды. Но всегда получал одинаковый ответ. Команда была довольна фреймворком Yii.   
  
Они мне говорили, что может быть трудно её изучать, но она легка в использовании, как только поймете, как она работает. Поэтому я посвятил себя изучению выбранного командой фреймворка. Дело продвигалось медленно и езда была ухабистая. Я не получал удовольствия. Я прорабатывал главу 10 в книге по Yii 1.1.14, полагая, что никогда по-настоящему не смогу самостоятельно создать приложение менее чем за сто лет. Слишком много дорог, казалось, уходили в никуда.  
  
Затем произошло чудо.

#### Появление Yii2

Я нашел Yii2 Альфа. Мне было любопытно увидеть, какие различия были в Yii2, по сравнению с бывшей 3 года на тот момент платформой. Так что я нырнул в неё и к моему глубокому и полному удивлению, я мгновенно получил с ней контакт. Я понимал структуры. Я мог бы написать код, который фактически работал!  
  
Какое это было великое чувство. Я лично нашел, что Yii 2 будет наиболее интуитивно понятной и элегантной из всех PHP-фреймворков, которые я изучал. У меня столько энтузиазма, что я хочу поделиться им, с каждым программистом которого знаю, и даже тех, кого я не знаю, что побудило меня написать эту книгу.  
  
С помощью Yii2, даже будучи новичком, я смог поднять на ноги рабочий сайт, который имел управляемую базой данных модель пользователей, с пользовательской и администраторской частью. Прямо из коробки, я получил рабочую модель пользователей, с функцией восстановления забытого пароля, которая также интегрирована с начальной загрузки для мобильного, отзывчивого дизайна, без программирования вообще.   
  
Как это здорово! Хотя я был начинающий программист, когда изучал PHP-фреймворки, я не имел опыта работы с базами данных и это одна из областей, на мой взгляд, где Yii2 действительно блещет.