Стандарт оформления и документирования исходного кода

Структура стандарта

Содержание данного раздела имеет следующую структуру:

• **Правилом** называется минимальная единица стандарта, частично или полностью описывающая форму представления элемента языка PHP в рамках данного стандарта. *Правило* должно отвечать всем или большинству (в случае аргументированного отступления от конкретного *критерия*) *критериев оценки*, описанным ниже. Каждое *правило* должно дополняться аргументами, приведшими к появлению данного *правила* (со ссылкой на *критерии оценки*) и примерами реализации в виде PHP кода;

• Все *правила*, относящиеся к одному элементу языка PHP (на пример к именам переменных или строковым литералам) называются **формой представления** этого элемента и должны записываться на отдельной странице стандарта;

• *Формы представления* логически связанных элементов языка PHP группируются в **разделы**;

• Стандарт имеет иерархическую структуру с тремя уровнями иерархии:

1. Корневые узлы, имена которых соответствуют языку, элементы которого описываются стандартом;
2. Узлы *разделов*, группирующих логически связынне элементы языка;
3. Страницы стандарта, содержащие *формы представления* конкретных элементов языка, относящихся к данному *разделу*. Имена этих узлов должны соответствовать именам элементов, описываемых в них.

Критерии оценки

Критерии оценки позволяют определить верность используемого *правила*. Если два *правила* описывают одну и ту же часть элемента, предпочтение отдается тому, который отвечает наибольшему числу *критериев*. *Критерии* расположены в порядке убывания приоритета, это значит, что если один критерий вступает в конфликт с другим (что называется *конфликтом критериев*), предпочтение отдается тому, номер которого ниже. В случае, если существует два *правила* описывающих одну и ту же часть элемента, при этом одно правила создает *конфликт критериев*, а другое нет, предпочтение отдается второму правилу.

1. **Читабельность** - *правило* должно подчеркивать логическое назначение элемента. Данный критерий требует писать программный код так, чтобы его было легко читать и понимать в дальнейшем;
2. **Простота** - программист должен выполнить минимальное число операций для записи элемента. Данный критерий позволяет программисту сэкономить время для интеллектуальной работы, связанной с проектированием кода, за счет уменьшения количества времени, отводимого на написание кода. Кроме того, простые *формы представления* элемента проще реализовать в других языках, что снижает время переучивания при переходе между языками (стандартами);
3. **Эффективность** - *правило* не должно снижать эффективность элемента, то есть алгоритм, в котором используется данный элемент или группа элементов, не должен выполнять больше работы при использовании *правила*. Данный критерий призван уберечь программиста от излишнего "жертвования" аппаратными ресурсами.

Можно заметить, что все критерии конфликтуют между собой. Часто для достижения максимальной эффективности, приходится жертвовать читабельностью, а повышение читабельности, снижает простоту и т.д. Для разрешения такого рода конфликтов необходимо учитывать приоритетность критериев, а так же критически относиться к каждому *правилу* данного стандарта.

PHP

Литералы

Строковые литералы

1. Строковый литерал записывается в одинарных кавычках (')

В связи с тем, что в языке PHP возможна запись строковых литералов с использованием двойных (с интерпретацией) и одинарных (без интерпретации) кавычек, предпочтение отдается одинарым, так как это позволяет избежать интерпретации строковых литералов, когда этого явно не нужно (критерий *эффективности*). Так же включение переменных в строковые литералы более очевидно в случае использования конкатенации, чем в случае интерпретации (критерий *читабельности*). Данное правило не отвечает критерию *простоты*, так как требует использования большего числа символов, в случае включения переменных в строковый литерал, но так как других альтернатив в языке PHP нет, данное решение отвечает большему числу критериев.

```
$str = 'String';
$char = 'A';
```

2. Если строковый литерал следует объединить с данными из других источников, его следует явно конкатенировать с этими данными по средствам оператора Точка, а не использовать двойные кавычки

Данное правило прямо следует из правила 1, так как при использовании иного подхода к включению переменных в строковые литералы, это привело бы к противоречию правил.

```
$var = ' ';
$message = 'Hello' . $var . 'world';
```

3. Двойные кавычки (") для обозначения строковых литералов могут применяться только в тех случаях, когда необходимо использовать специальные символы языка PHP (возврат каретки, перенос строки и т.д.). В этом случае специальные символ записывается отдельно и объединяется со строкой с помощью конкатенации

Данное правило следует из ограничений самого языка PHP. Другого способа создания специальных символов в виде литералов в нем просто не существует.

```
$message = 'Hello world'."\n";
```

4. Строковый литерал не переносится между строками файла

В случае применения автоматического переноса строк файла (средствами текстового редактора), приводит к "потере" конкретной строки в файле при чтении программистом (критерий *читабельности*), а в случае ручного выставления переносов, требует от программиста дополнительных действий (критерий *простоты*). Оба подхода никак не сказываются на эффективность алгоритма, потому критерий *эффективности* не берется в расчет. Однако следует отметить, что применение механизма автоматического переноса строк, выходящих за границы окна текстового редактора, позволяется с целью **временного** чтения его содержимого, не требующего прокрутки окна по горизонтали.

```
$badStr = 'Hello
world'; // Пример нарушения правила.
```

5. Не следует использовать строковый литерал в качестве операнда логического выражения, вместо этого его следует присвоить константе класса, к которому этот литерал относится

Данное правило призвано исключить использование "магических строк". *Читабельность* кода, в котором используются не связанные с конкретной сущностью строки, резко снижается. Так же это исключает возможность дальнейшего изменения строкового литерала без изменения некоторой логики кода, зависящего от него. Эффективность кода, не использующего "магические строки" не снижается, но критерием *простоты* приходится принебречь в связи с приоритетом критерия *читабельности*. Наиболее приоритетным местом размещения "магических строк" являются константы классов, к которым они относятся.

```
class Message{
  const INFO = 'info';
  const WARNING = 'warning';
  const ERROR = 'error';

  ...
}
```

Логические литералы

1. Логические литералы (true, false) записываются в нижнем регистре

Так, как регистр логического литерала не имеет значение, данное правило требует от программиста меньше действий при реализации (исключение операции перевода регистра), что отвечает критерию *простоты*. *Читабельность* логических литералов при использовании нижнего регистра, соответствует читабельности при записи литерала с заглавной буквы, но второй вариант конфликтует с критерием *простоты*, за что был исключен. Эффективность скрипта, записанного с любой формой представления логических литералов не меняется, потому критерий *эффективности* не учитывается.

```
$t = true;
$f = false;
```

Литерал NULL

1. Литерал NULL записывается в нижнем регистре

Так, как регистр литерала null не имеет значение, данное правило требует от программиста меньше действий при реализации (исключение операции перевода регистра), что отвечает критерию *простоты*. *Читабельность* литерала null при использовании нижнего регистра, соответствует читабельности при записи литерала с заглавной буквы, но второй вариант конфликтует с критерием *простоты*, за что был исключен. Эффективность скрипта, записанного с любой формой представления литералов null не меняется, потому критерий *эффективности* не учитывается.

```
$n = null;
```

Операторы

Инициализация перемнных и свойств класса

1. Оператор инициализации (=) записывается с использованием обрамляющих пробелов

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
$var = 1;
```

*Арифметические операторы + - * / %*

1. Арифметические операторы (+,-,*,/,%) записываются с использованием обрамляющих пробелов

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
$var = 1 + 1;
```

Операторы инкремента и декремента

1. Операторы инкремента и декремента (++, --) не зависимо от формы (префиксные и постфиксные) записываются без использования разделяющего пробела

Данное правило основывается на предположении, что операторы инкремента и декремента более точно выражают свою урнарную природу (использование одного операнда) при совмещении их с операндом (критерий *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
$var++;
--$var;
```

Логические операторы И ИЛИ НЕ

1. Логический оператор И записывается с использованием слова &&

Данная запись включает меньше символов, чем AND, что отвечает критерию *простоты*. Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if($a && $b){
  ...
}
```

2. Логический оператор ИЛИ записывается с использованием слова ||

Данное правило исходит из правила 1 и призвано использовать единый формат записи (либо AND, OR, либо &&, ||) с целью повышения *читабельности*. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if($a || $b){
  ...
}
```

3. Логический оператор НЕ записывается с использованием слова !

Данная запись включает меньше символов, чем NOT, что отвечает критерию *простоты*. Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if(!$a){
  ...
}
```

4. Логические операторы записывается с использованием обрамляющих пробелов за исключением оператора НЕ, который не отбивается пробелом от выражения

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

Данное правило основывается на предположении, что оператор НЕ более точно выражают свою урнарную природу (использование одного операнда) при совмещении их с операндом (критерий *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if($a && $b || !$c){
}
```

```
if($a != $b){
  ...
}
```

2. Операторы сравнения записывается с использованием обрамляющих пробелов

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
if($a >= $b || $b != $c){
}
```

Логические выражения

1. В случае, если логическое выражение включает более двух подвыражений, следует оценить его читабельность и, возможно, выделить подвыражения в переменные с "говорящими" названиями

Данное правило призвано повысить *читабельность* сложных логических выражений. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
// Пример нарушения правила. Слишком низкая читабельность выражения.
if(elementIndex < 0 || elementIndex > MAX_ELEMENTS || elementIndex == lastElementIndex){
  ...
}
```

```
// Пример следования правилу.
$isOutOfBounds = elementIndex < 0 || elementIndex > MAX_ELEMENTS;
$finished = elementIndex == lastElementIndex;
if($isOutOfBounds || $finished){
  ...
}
```

Тернарный логический оператор ?:

1. Тернарный логический оператор записывается без использования разделяющего пробела слева для символа ?, но с использованием разделяющего пробела с права от этого символа

Данное правило призвано повысить читабельность оператора за счет отделения условного выражения от тела. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Тернарный логический оператор записывается с использованием обрамляющих пробелов для с имвола :

Данное правило позволяет визуально разделить ветви оператора, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск значений ветвей в операторе. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Условие тернарного логического оператора записывается с круглых скобках

Данное правило позволяет визуально выделить логическое выражение оператора (критерий читабельности) и избежать ошибок, связанных с приоритетом выполнения. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
($a == $b)? true : false;
```

Управляющие структуры

Ветвление if

1. Управляющая структура if записывается без отделения от открывающей круглой скобки логического выражения

Данное правило упрощает процесс набора структуры (критерий *простоты*). Перенос выражения на новую строку не выполняется с целью сохранения логической структуры выражения: if, then, else. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Логическое выражение записывается без использования обрамляющих

пробелов

Отбиение логического выражения от круглых скобок условия нисколько не повышает читабельность кода, а лишь добавляет сложность.

3. Код, относящийся к ветвям, всегда группируется в блоки и отбивается одим стандартным отступом относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска ветви структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

4. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после закрывающей круглой скобки или слова else. За ней сразу должен следовать символ новой строки

Данное правило следует из логической структуры ветвления. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является конструкция if(...) и else, которые проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

5. Закрывающая фигурная скобка, заканчивающая блок кода, ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода

Данное правило позволяет четко выделить окончание блока кода структуры (критерий *читабельности*). В связи с тем, что альтернативная версия правила, при котором закрывающая фигурная скобка записывается рядом с else, мало отличается от данного варианта, принятое правило является спорным и может быть изменено в будущем.

6. В качестве альтернативного условия используется слово elseif

Данный вариант записи проще else if (критерий простоты). Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

7. Слово else ставится с новой строки

Аналогично правилу 5.

8. Слово elseif ставится с новой строки

Аналогично правилу 5.

```
if($a > $b){
  ...
}
elseif($a < $b){
  ...
}
else{
  ...
}
```

9. Нормальный путь алгоритма следует располагать в первом блоке структуры, а не в блоке else

Данное правило позволяет программисту быстро выделить нормальный ход алгоритма (начало условия) и обработчики ошибок (конец условия), что повышает читабельность. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if(...){
  if(...){
    ... // Нормальный ход.
  }
  else{
    ... // Альтернативный ход.
  }
}
else{
  ... // Альтернативный ход.
}
```

Ветвление switch

1. Управляющая структура switch записывается без отделения стуктуры от открывающей круглой скобки, содержащей проверяемое значение

Данное правило упрощает процесс набора структуры (критерий *простоты*). Перенос выражения на новую строку не выполняется с целью сохранения логической структуры выражения: switch, case. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Проверяемое значение записывается без использования обрамляющих пробелов

Отбиение проверяемого значения от круглых скобок условия нисколько не повышает читабельность кода, а лишь добавляет сложность.

3. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после закрывающей круглой скобки. За ней сразу должен следовать символ новой строки

Данное правило следует из логической структуры ветвления. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является конструкция switch(...) и case, которые проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

4. Закрывающая фигурная скобка ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода

Данное правило позволяет четко выделить окончание структуры (критерий *читабельности*).

5. Слово case, с проверяемым условием, записывается с новой строки и одим стандартным отступом относительно структуры. После слова case должен следовать символ новой строки

Отделение case от ветви призвано упростить поиск конкретного значения вариации условия (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

6. Код ветви записывается с двумя стандартными отступами относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска блока структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

7. Слово break записывается с новой строки в блоке кода ветви

Данное правило соответствует логической структуре switch, case, в которой началом блока является case, а согласно правилу 6, тело блока выделяется отбиением, что позволяет не выделять окончание блока словом break (критерий *простоты* и *читабельности*).

8. Слово default записывается с новой строки и одим стандартным отступом относительно структуры

Аналогично правилам 5 и 6.

```
switch($a){
  case 1:
    ...
    break;
  case 2:
    ...
  case 3:
    ...
    break;
  default:
    ...
}
```

9. Если варианты можно упорядочить по частоте использования, необходимо это сделать установив наиболее используемый вариант в начале, а наименее используемый в конце

Данное правило позволяет повысить эффективность кода и выделить нормальный путь (принцип *читабельности*). От критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

10. Если варианты нельзя упорядочить по частоте, их необходимо упорядочить по алфавиту или численно

Данное правило позволяет ускорить поиск вариантов (принцип *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

Выбор цикла

1. Если заранее известно число итераций цикла, необходимо использовать цикл for

Данное правило позволяет наиболее эффективно использовать цикл.

```
$min = 5;
$max = 10;
for($i = $min; $i < $max; $i++){
  ...
}
```

2. Если цисло итераций цикла вычисляется в процессе работы цикла, используется цикл while

Данное правило позволяет наиболее эффективно использовать цикл.

```
while($ping){
  $ping = ping();
}
```

3. Если для цикла while нужно обязательно выполнить первую итерацию, используется цикл do, while

Данное правило позволяет наиболее эффективно использовать цикл.

```
do{
  $name = getNextName();
}while($name == 'Test');
```

4. Если необходимо выполнить одну итерацию для каждого элемента коллекции, используется цикл foreach

Данное правило позволяет наиболее эффективно использовать цикл.

```
foreach($nodeList as $node){
  ...
}
```

5. Если заранее невозможно определить тип цикла, необходимо реализовать цикл "Изнутри"

Для реализации цикла "Изнутри" необходимо сначало реализовать тело цикла без индексов, затем добавить индексы цикла, а затем реализовать сам цикл выбрав наиболее подходящий.

Цикл for

1. Цикл for записывается без отделения стуктуры от открывающей круглой скобки итератора

Данное правило упрощает процесс набора структуры (критерий *простоты*). Перенос выражения на новую

строку не выполняется с целью сохранения логической структуры выражения for. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Итератор записывается записывается без использования обрамляющих пробелов

Отбиение итератора от круглых скобок цикла нисколько не повышает читабельность кода, а лишь добавляет сложность.

3. Разделяющий символ условия (;) записывается без разделяющего пробела слева, но с разделяющим пробелом справа. Если компонент условия не содержит кода, разделяющий пробел слева можно опустить

Данное правило позволяет выделить элементы итератора (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

4. Код, относящийся к циклу, всегда группируется в блок и отбивается одим стандартным отступом относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска блока кода структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

5. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после закрывающей круглой скобки. За ней сразу должен следовать символ новой строки

Данное правило следует из логической структуры цикла. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является конструкция for(...), которую проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

6. Закрывающая фигурная скобка, заканчивающая блок кода, ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода

Данное правило позволяет четко выделить окончание структуры (критерий *читабельности*).

```
for($a = 5; $a--;){
  ...
}
```

7. Недопустимо изменять значение индекса внутри тела цилка

Данное правило служит для защиты от логических ошибок (принцип *эффективности*).

8. Имя индекса цикла должно быть информативным

Данное правило позволяет легко определить назначение индекса цикла (принцип *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
for($recordIndex = 0; $recordIndex < $countRecord; $recordIndex++){
  ...
}
```

Цикл while

1. Цикл while записывается без отделения стуктуры от открывающей круглой скобки условий

Данное правило упрощает процесс набора структуры (критерий *простоты*). Перенос выражения на новую строку не выполняется с целью сохранения логической структуры выражения while. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Логическое выражение записывается без использования обрамляющих пробелов

Отбиение логического выражения от круглых скобок цикла нисколько не повышает читабельность кода, а лишь

добавляет сложность.

3. Код, относящийся к циклу, всегда группируется в блок, и отбивается одим стандартным отступом относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска блока кода структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

4. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после закрывающей круглой скобки. За ней сразу должен следовать символ новой строки

Данное правило следует из логической структуры цикла. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является конструкция while(...), которую проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

5. Закрывающая фигурная скобка, заканчивающая блок кода, ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода

Данное правило позволяет четко выделить окончание структуры (критерий *читабельности*).

```
while($a < $b){
  ...
}
```

6. Инициализационный код цикла должен размещаться перед началом цикла, если это не влияет на работу кода в целом

Данное правило позволяет быстрее найти место инициализации итератора и сгруппировать цикл (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

7. Инкрементация итератора должна располагаться в конце цилка, если это не влияет на работу кода в целом

Данное правило позволяет быстрее найти место инкрементации итератора и сгруппировать цикл (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
$row = getFirstRow();
while($row->getName() != '...'){
  ...
  $row = nextRow();
}
```

Цикл do/while

1. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после слова do. За ней сразу должен следовать символ новой строки

Данное правило следует из логической структуры цикла. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является слово do, которую проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Закрывающая фигурная скобка, заканчивающая блок кода, ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода перед словом while

Данное правило следует из логической структуры цикла. При переносе слова while на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что закрывающим элементом блока кода является конструкция while(...), которую проще найти в коде, нежели одиночные символы закрывающей фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

3. Логическое выражение записывается без использования обрамляющих пробелов

Отбиение логического выражения от круглых скобок цикла нисколько не повышает читабельность кода, а лишь добавляет сложность.

4. Код, относящийся к циклу, всегда группируется в блок и отбивается одим стандартным отступом относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска блока кода структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
do{
  ...
}while($a < $b);
```

5. Инкрементация итератора должна располагаться в конце цилка, если это не влияет на работу кода в целом

Данное правило позволяет быстрее найти место инкрементации итератора и сгруппировать цикл (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

Перехват исключений try/catch

1. Код, относящийся к ветвям, всегда группируется в блоки и отбивается одим стандартным отступом относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска ветви структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после слова try и catch. Сразу за ней ставится символ новой строки

Данное правило следует из логической структуры обработчика. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода являются слова try и catch, которые проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

3. Закрывающая фигурная скобка, заканчивающая блок кода, ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода

Данное правило позволяет четко выделить окончание блока кода структуры (критерий *читабельности*). В связи с тем, что альтернативная версия правила, при котором закрывающая фигурная скобка записывается рядом с case, мало отличается от данного варианта, принятое правило является спорным и может быть изменено в будущем.

4. Слово catch записывается с новой строки

Аналогично правилу 3.

```
try{
  ...
}
catch(...){
  ...
}
catch(...){
  ...
}
```

Пакеты и физическая структура

Оператор namespace и именование пакетов

1. Оператор namespace должен располагаться во второй строке файла исходного кода (после <?php). За ним должен следовать символ новой строки

Данное правило позволяет выделить пакет, в котором работает программист (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Для именования пакетов используется нотация lowerCamelCase, то есть слова, из которых состоит имя, записываются слитно, первое слово записывается в нижнем регистре, а каждое следующее записывается с заглавной буквы

Данное правило позволяет отличить классы от пакетов (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

3. Если имя пакета включает аббривиатуру, ее составные символы всегда записываются в верхнем регистре

Данное правило следует из типографического стандарта записи аббривиатур.

4. Имена пакетов записываются латинскими буквами

Данное правило исходит из физической структуры проекта и обеспечивает переносимость на различные платформы.

5. Имя пакета не следует предварять символом обратного слеша (\)

Данное правило облегчает набор имени пакета (принцип *простоты*). Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
namespace PPHP\tools\patterns;
```

6. В качестве корневого пакета должен выступать пакет, имя которого соответствует имени системы

Данное правило позволяет исключить конфликты имен при использовании различных библиотек и систем в проекте.

Отнесение классов к пакетам

1. Все классы системы должны быть отнесены к конкретному пакету по их функциональному назначению

Данное правило позволяет исключить конфликты имен в глобальной области видимости.

Физическая структура

1. Имя класса, интерфейса и trait (без учета имени пакета) должно соответствовать имени файла, в котором он находится с типом файла php

Данное правило позволяет использовать механизм автозагрузки классов.

```
class A{ // класс должен располагаться в файле A.php
}
```

2. Полное имя класса (с учетом имени пакета) должно соответствовать полному адресу файла (его содержащего) в файловой системе от корня системы (сайта)

Данное правило позволяет использовать механизм автозагрузки классов.

```
namespace PPHP\tools\patterns
// полный адрес файла, в котором располагается данный класс, должен иметь вид $SITE$/PPHP/tools/
patterns/A.php, где $SITE$ - это адрес корня системы (сайта)
class A{
}
```

3. Каждый класс, интерфейс и trait должен располагаться в отдельном пакете (файле)

Данное правило позволяет исключить повторную загрузку уже загруженных классов.

Оператор use и использование классов из различных пакетов

1. Операторы use должны следовать друг за другом с новых строк и отбиваться от оператора namespace одной пустой строкой

Данное правило позволяет сгруппировать список используемых данным классом пакетов, классов, интерфейсов и traits (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Если класс, относящийся к одному пакету, использует классы, относящиеся к другим пакетам, должна быть использована операция включения пакета use для сокращения имени используемых классов в коде и создания списка используемых классов, интерфейсов и traits

Данное правило позволяет сформировать список зависимостей класса (принцип читабельности). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

3. Если класс, относящийся к одному пакету, использует классы, относящиеся к другим пакетам, входящим в первый пакет, он может не использовать операцию включения пакета use

Данное правило призвано упростить процесс использования классов в рамках одного пакета (принцип *простоты*).

```
namespace PPHP\tools\patterns\database;

use PPHP\tools\patterns\metadata\reflection as reflection;
use PPHP\tools\patterns\memento as memento;

// Операция включения пакета для identification\OID не используется в связи с близким расположением
используемого класса
abstract class LongObject implements reflection\Reflect, identification\OID, memento\Originator{
  use reflection\TReflect;
  use memento\TOriginator;
  use identification\TOID;
}
```

Переменные и свойства класса

Имена

1. Для именования переменных и свойств класса используется нотация lowerCamelCase, то есть слова, из которых состоит имя, записываются слитно, первое слово записывается в нижнем регистре, а каждое следующее записывается с заглавной буквы

Данное правило позволяет призвано выделить природу свойства класса как прилагательное (принцип

читабельности). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Имя свойства или переменной представляет собой краткое (не более трех слов) описание хранимых в ней данных и их предназначения

Данное правило призвано повысить читабельность кода за счет связывания имен свойств и переменных с хранимыми в них значениями. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Если имя переменной или свойства класса включает аббривиатуру, ее составные символы всегда записываются в верхнем регистре

Данное правило следует из типографического стандарта записи аббривиатур.

4. Если переменная или свойство класса является константой, ее имя записывается в верхнем регистре, а слова разделяются знаком подчеркивания (_)

Данное правило призвано выделить константы в программе (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

5. Имена переменных и свойств класса записываются латинскими буквами

Данное правило обеспечивает переносимость на различные платформы.

```
$testVar = 1;
class A{
  const CONSTANT_VAR;

  public $testPropery;
}
$ISO = 'Standard';
```

Объявление и инициализация

1. Все переменные должны быть инициализированы в конструкторе (если присутствуют данные для инициализации)

Данное правило исходит из природы конструктора и определяет стандартное место инициализации переменных (принцип *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
class A{
  public $varA;
  public $varB;

  public function __construct(){
    $this->varB = 1;
  }
}
```

Модификаторы

1. Для не константных свойств класса всегда указываются модификаторы видимости public, protected и private

Данное правило призвано сделать код более строгим и исключить ошибки инкапсуляции (принцип *эффективности*). Читабельность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *эффективности*.

2. Модификатор видимости указывается перым модификатором, модификатор static вторым

Данное правило призвано повысить читабельность кода за счет быстрого поиска модификаторов по группам. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
class A{
  const $A = 1;

  private $a;

  public static $b;
}
```

Методы класса

Имена

1. Для именования методов используется нотация lowerCamelCase, то есть слова, из которых состоит имя, записываются слитно, первое слово записывается в нижнем регистре, а каждое следующее записывается с заглавной буквы

Данное правило позволяет призвано выделить природу метода класса как глагол (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Имя метода представляет собой краткое (не более трех слов) описания действия, являющееся целью метода, в повелительной форме

Данное правило призвано повысить читабельность кода за счет связывания имен методов с решаемыми ими задачами. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Если имя метода включает аббривиатуру, ее составные символы всегда записываются в верхнем регистре

Данное правило следует из типографического стандарта записи аббривиатур.

4. Имена методов записываются латинскими буквами

Данное правило обеспечивает переносимость на различные платформы.

5. Между именем метода и открывающей круглой скобкой списка аргументов пробел не ставится

Данное правило упрощает процесс набора метода (критерий *простоты*). Перенос списка аргументов на новую строку не выполняется с целью сохранения логической структуры метода. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

```
class A{
  public function getISO($n){
  }
}
```

Аргументы

1. Список аргументов метода не отделяется от круглых скобок, их содержащих, пробелами

Отбиение списка аргументов от круглых скобок условия нисколько не повышает читабельность кода, а лишь добавляет сложность.

2. Запятая (,), разделяющая аргументы, не имеет пробельного отступа слева, но имеет пробельный отступ справа

Данное правило позволяет выделить аргументы (критерий *читабельности*). Эффективноть таких выражений не

страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Символ Равно (=), опеделяющий значение по умолчанию для аргумента, записывается с обрамляющими пробелами

Данное правило позволяет отделить аргумент от значения по умолчанию, а так же выделить аргументы, имеющие эти значения (принцип *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
class A{
  public function a($a, $b = null){
  }
}
```

4. Имя аргумента представляет собой краткое (не более трех слов) описание хранимых в ней данных и их предназначения

Данное правило призвано повысить читабельность кода за счет связывания имен аргументов с хранимыми в них значениями. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

Параметры

1. Если метод получает нетипизированный в аргументе параметр, он обязан предварительно проверить валидность значения за исключением случаев, если метод класса имеет модификатор видимости private или protected

Данное правило служит для исключения ошибок, связанных с невалидными параметрами (принцип *эффективности*). Правило игнорирует критерии *простоты* и *читабельности* из за важности данного механизма.

```
public function methodA($arg){
  if(!is_int($arg)){
    throw new InvalidArgumentException();
  }
}
```

Тело

1. Фигурная скобка, открывающая блок тела метода, должна располагаться сразу за закрывающей круглой скобкой списка аргументов этого метода. За этой фигурной скобкой должен следовать символ перевода строки

Данное правило следует из логической структуры метода. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является конструкция function ...(...), которую проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Фигурная скобка, закрывающая блок тела метода, должна идти сразу после последней операции тела метода разделенной от нее символом перевода строки

Данное правило позволяет четко выделить окончание блока кода структуры (критерий *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

3. Тело функции должно отбиваться одним стандартным отступом

Выделение кода в блоки и отбиение призвано повысить скорость поиска тела метода (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
public function a(){
  ...
}
```

Модификаторы

1. Для свойств класса всегда указываются модификаторы видимости public, protected и private

Данное правило призвано сделать код более строгим и исключить ошибки инкапсуляции (принцип *эффективности*). Читабельность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *эффективности*.

2. Модификатор видимости указывается первым модификатором

Данное правило призвано повысить читабельность кода за счет быстрого поиска модификаторов по группам. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

3. Модификатор абстрактности всегда указывается вторым модификатором

Аналогично правилу 2.

4. Модификатор static всегда указывается третим модификатором

Аналогично правилу 2.

5. Модификатор final всегда указывается четвертым модификатором

Аналогично правилу 2.

```
class A{
  protected abstract function methodA(){}

  public static final function methodB(){}
}
```

Классы, интерфейсы и traits

Имена

1. Имена классов и интерфейсов всегда записывается с заглавной буквы. Если имя класса содержит несколько слов, каждое из них записывается без разделителя с заглавной буквы

Данное правило позволяет призвано выделить природу класса как существительное (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Имя класса, интерфейса или trait представляет собой краткое (не более трех слов) описание предназначения сущности в форме существительного в единственном числе

Данное правило призвано повысить читабельность кода за счет связывания имен классов с описываемыми ими сущностями. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Имена traits записываются подобно именам классов и интерфейсов, но в качестве первого символа ставится заглавная, латинская буква T

Данное правило призвано выделить traits в отдельную группу сущностей с целью предотвращения засорения пространства имен пакета (принцип *эффективности*).

4. Имена классов, интерфейсов и traits записываются латинскими буквами

Данное правило обеспечивает переносимость на различные платформы.

5. Между именем класса, интерфейса или traits и открывающей фигурной скобкой тела пробел не ставится

Данное правило упрощает процесс набора класса (критерий *простоты*). Перенос тела на новую строку не выполняется с целью сохранения логической структуры класса. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

```
class MyClass{
}
interface MyInterface{
}
trait TMyTrait{
}
```

Структура

1. Все члены класса (свойства и методы, а так же документация к ним) отбиваются одим стандартным отступом относительно тела класса

Выделение кода в блоки и отбиение призвано повысить скорость поиска свойств и методов класса (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Все члены класса разделяются одной пустой строкой

Данное правило призвано повысить скорость поиска свойств и методов класса (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Первыми перечисляются свойства класса, вторыми методы

Данное свойство призвано сгруппировать свойства и методы в визуальные группы (принцип *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

4. Сначало указываются константы класса

Аналогично правилу 3.

5. Первыми указываются все private свойства класса, вторыми protected, третьими public

Аналогично правилу 3.

6. Внутри групп, обозначенных в п. 5 первыми указываются static свойства, вторыми частные свойства

Аналогично правилу 3.

7. Первыми указываются все private методы класса, вторыми protected, третьими public

Аналогично правилу 3.

8. Внутри групп, обозначенных в п. 7 первыми указываются абстрактные методы, вторыми static, третьими final

Аналогично правилу 3.

9. Фигурная скобка, закрывающая тело класса указывается сразу после последнего члена класса на следующей за ним строке

Данное правило позволяет четко выделить окончание тела структуры (критерий *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
class A{
  const A;

  const B;

  private static $a;

  private $b;

  protected static $c;

  protected $d;

  public static $e;

  public $f;

  private abstract function ma(){
  }

  private static final function mb(){
  }

  private static function mb(){
  }

  private final function mc(){
  }

  // Аналогично для protected и public методов
}
```

Документирование

Общие положения

1. Все классы, интерфейсы, traits, свойства, переменные и методы должны сопровождаться подробной документацией в формате PHPDoc

Данное правило позволяет сопроводить программный код подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Блок документации должен иметь такую же отбивку слева, как и сущность, которую он документирует относительно первого символа обратного слеша (/)

Данное правило позволяет сгруппировать блок документации к сущности, к которой он относится (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Первый ряд символов звездочка (*) должен иметь отбивку, равную одному пробелу от первого символа обратного слеша (/) блока, к которому он относится

Данное правило позволяет выровнять блок документации (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
/**
 * ...
 */
class A{
  /**
   * ...
   * @var ...
   */
  protected $a = 1;
}
```

Документирование классов, интерфейсов и traits

1. Документация класса, интерфейса или trait должна включать подробное описание его предназначения и возможностей в формате пользовательской документации, автора и пакет, в котором он находится

Данное правило позволяет сопроводить класс подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Текст описания должен быть первым блоком текста в документации, начинаться с заглавной буквы и заканчиваться точкой

Данное правило преследует цель упорядочить и стандартизировать компоненты документации (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Метка абстрактности должна быть вторым блоком текста в документации

Аналогично правилу 2.

4. Имя автора должно быть третим блоком текста в документации

Аналогично правилу 2.

5. Имя пакета должно быть четвертым блоком текста в документации. Оно должно соответствовать текущему namespace

Аналогично правилу 2.

```
/**
 * Данный класс служит для ...
 * @author Artur Sh. Mamedbekov
 * @package PPHP\tools\patterns
 */
class A{
}
```

Документирование свойств

1. Документация свойства класса должна включать подробное описание его предназначения и типа в формате технической документации

Данное правило позволяет сопроводить свойство подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*). Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Метка статичности должна быть первым блоком текста в документации

Данное правило преследует цель упорядочить и стандартизировать компоненты документации (критерий *читабельности*). Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Документация типа свойства и текста описания должна быть вторым блоком

текста в документации. Тип свойства должен включать полное имя типизирующего класса. Текст описания должен начинаться с заглавной буквы и заканчиваться точкой

Данное правило преследует цель упорядочить и стандартизировать компоненты документации (критерий *читабельности*). Использование полного имени необходимо для работы генератора документации. Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
/**
 * @var \PPHP\tools\classes\... Свойство хранит...
 */
private $a = new ...;
```

Документирование переменных

1. Следует документировать каждую локальную переменную в методах класса с использованием тега var следующей структуры: <тип> <имя> <описание>. Описание должно начинаться с заглавной буквы и заканчиваться точкой

Данное правило позволяет не только сопроводить локальные переменные подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*), но и обеспечить правильную работу средств автоматического контроля типа. Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
public static function getProxy($OID){
  /**
   * @var \PPHP\tools\patterns\...\OID $proxy Proxy вызываемого объекта.
   */
  $proxy = new static;
  $proxy->setOID($OID);
  return $proxy;
}
```

Документирование методов

1. Документация метода должна включать подробное описание предназначения и условий использования метода, его параметров, метку абстрактности (если метод абстрактный), метку статичности (если метод статичный), выбрасываемых исключений (если таковые имеются) и возвращаемого значения (если таковое имеется) в формате пользовательской документации

Данное правило позволяет сопроводить свойство подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*). Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Текст описания должен быть первым блоком текста в документации. Он должен начинаться с заглавной буквы и заканчиваться точкой

Данное правило преследует цель упорядочить и стандартизировать компоненты документации (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Метки абстрактности и статичности должны быть вторым блоком текста в документации

Аналогично правилу 2.

4. Документация аргументов метода должна быть третим блоком текста в документации

Аналогично правилу 2.

5. Документация аргументов метода должна иметь следующую структуру: <тип> <имяАргумента> [\[optional\]] <описание>. Тип аргумента должен включать полное имя типизирующего класса. Описание аргумента должно начинаться с заглавной буквы и заканчиваться точкой

Данное правило позволяет не только сопроводить аргументы метода подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*), но и обеспечить правильную работу средств автоматического контроля типа и системы генерации документации. Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

6. Список выбрасываемых исключений и их описание должен быть четвертым блоком текста в документации. Имя класса выбрасываемого исключения должно быть полным. Описание выбрасываемого исключения должно начинаться с заглавной буквы и заканчиваться точкой

Данное правило служит для предоставления подробной информации о возможных альтернативных вариантах работы алгоритма метода (критерий *читабельности*), а так же предоставляет достаточную информацию механизму генерации документации. Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

7. Документация возвращаемого значения метода должна быть пятым блоком текста в документации. Тип возвращаемого значения должен включать полное имя типизирующего класса. Описание возвращаемого значения должно начинаться с заглавно буквы и заканчиваться точкой

Данное правило позволяет не только сопроводить аргументы метода подробным описанием на естественном языке для изучения в будущем или не знакомыми с системой пользователями (критерий *читабельности*), но и обеспечить правильную работу средств автоматического контроля типа и системы генерации документации. Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

8. Если документируемый метод реализует или переопределяет родительский метод и его документация полностью повторяет документацию реализуемого или переопределяемого метода, то следует использовать тег prototype в значении которого необходимо указать полное имя класса, которому принадлежит реализуемый или переопределяемый метод чтобы избежать дублирования документации. В случае, если такого рода метод имеет дополнительное описание, его можно записать в описании метода, при этом оно будет добавлено в конец описания

Данное правило позволяет сократить документацию (критерий *простоты*), а так же исключить необходимость контролировать программистом изменения в дублирующей документации.

```
/**
 * Метод служит для...
 * @return integer ...
 */
public function get(){
}
```

```
/**
 * ...
 * @abstract
 * @static
 * @param integer $var Аргумент определяет ...
 * @throws \PPHP\tools\classes\standard\baseType\exceptions\InvalidArgumentException Выбрасывается в
случае...
 */
public abstract static function set($var);
```

```
abstract class A{
  /**
   * Документация метода...
   */
  public abstract function method();
}

class B extends A{
  /**
   * @prototype \A
   */
  public function method(){
  }
}

class C extends B{
  /**
   * Дополнительные сведения о методе, относящиеся именно к данной реализации...
   * @prototype \A
   */
  public function method(){
    parent::metod();
  }
}
```

Документирование кода

1. В случае, если исходный код класса содержит неочевидное решение задачи, он должен включать комментарий, описывающий принятое решение

Данное правило требует от программиста предоставлять информацию о неочевидном решении задачи (критерий *читабельности*). Эффективность таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

2. Комментарий в коде должен начинаться с заглавной буквы и оканчиваться точкой

Данное правило следует из особенностей реализации генератора документации.

3. Комментарий в коде должен описывать цель кода, а не то, как он работает

Данное правило следует из утверждения, что принцип работы кода должен быть понятен из самого кода.

4. Многострочный комментарий в коде должен заключаться в /* */ конструкцию, при чем не следует записывать открывающий и закрывающий элемент коментария в отдельной строке, достаточно отбить его от содержимого одним пробелом

Данное правило позволяет выделить крупные блоки комментария (критерий *читабельности*), при этом сократив затраты времени на форматирование блока (критерий *простоты*).

```
/* Создание нового счета.
Комментарий вида "Если создается новый счет." уже будет здесь избыточным. */
if($accountType == AccountType.NEW_ACCOUNT){
  ...
}
```

Исключения

Выброс исключений

1. Класс выбрасываемого исключения должен быть логически связан с типом исключительной ситуации
Данное правило повышает читабельность исключений за счет связи исключений с природой их появления.

2. Сообщение в исключении должно коротко объяснять причину появления исключительной ситуации с точки зрения кода, выбросившего это исключение, а так же включать информацию о данных, из за которых произошло исключение
Данное правило позволяет сопроводить исключение подробным описанием на естественном языке (критерий *читабельности*), а так же сократить время поиска ошибок в коде (критерий *эффективности*).

3. Следует следить за уровнем абстракции класса и исключения, которое он выбрасывает. Если класс отличается от уровня абстракции исключения, следует использовать исключение более высокого уровня абстракции
Данное правило позволяет сократить число выбрасываемых исключений до текущего уровня абстракции, объединив по природе.

4. Если в документации к абстрактному методу указаны возможные исключения, они должны быть указаны и в реализующих его методах, за исключением случаев, когда в реализации метода данное исключение не выбрасывается
Данное правило переносит исключения родителя в класс потомка, что упрощает поиск исключений в документации.

```
public function set($val){
  if(!is_integer($val)){
    throw new \InvalidArgumentException('Недопустимый тип параметра, ожидается integer вместо '.(typeof
$val));
  }
}
```

```
interface A{
  /**
   * @throws ...
   */
  public function x();
}

class B implements A{
  /**
   * @throws ...
   */
  public function x(){
    ...
    throw new ...
  }
}
```

Примечания:
• Документирование выбрасываемых методами исключений возможно даже на уровне абстрактных методов. Это связано с тем, что полиморфная природа многих классов требует обработку исключений при использовании интерфейсов.

Перехват исключений

1. Если в коде вызывается метод, который может выбросить исключение, вызов должен быть обрамлен блоком перехвата всех возможных исключений этого метода. Данный блок должен либо обрабатывать эти исключения, либо выбрасывать их выше по иерархии выполнения кода. Данное правило может не выполняться если выброс исключения не возможен в связи с особенностями окружения или реализации вызова

Данное правило требует от программиста контролировать выбрасываемые исключения.

2. При перехвате исключений следует повышать уровень их абстракции до уровня абстракции класса, их обрабатывающего

Данное правило позволяет сократить число выбрасываемых исключений до текущего уровня абстракции, объединив по природе.

```
/**
 * @throws IOException ...
 */
public function getWriter($fileName){
}

public function writeData($fileName, $data){
  try{
    $this->getWriter($fileName)->write($data);
  }
  catch(IOException $e){ // выброс полученного исключения
    throw new NotAccessException('Приемник данных не доступен', null, $e); // Повышение абстракции
исключения
  }
}

/**
 * @param integer $val ...
 * @throws \InvalidArgumentException ...
 */
public function set($val){
}

public function a(){
  $this->set(1); // выброс исключения не возможен физически
}
```