УПРАВЛЕНИЕ ОБРАЗОВАНИЯ ГОРОДА ПЕНЗЫ

Муниципальное казенное учреждение «ЦКОиМОУО» г. Пензы

Муниципальное автономное общеобразовательное учреждение многопрофильная гимназия № 13 г. Пензы

Муниципальное бюджетное образовательное учреждение средняя общеобразовательная школа № 30 г. Пензы

XXII научно–практическая конференция школьников города Пензы
«Я исследую мир»

**Создание генератора кода для реализации доступа к закрытым данным класса**

**Автор:**

Абрамов Максим Сергеевич, учащийся 9Б класса МАОУ Многопрофильная гимназия № 13 г. Пензы

**Научные руководители:**

С. С. Адамский, учитель информатики МБОУ СОШ № 30 г. Пензы

В. А. Зайцев, учитель информатики и физики высшей категории МБОУ СОШ № 30 г. Пензы

Пенза

2018

**Оглавление**

[Введение 3](#_Toc503190046)

[Теоретическая часть 4](#_Toc503190047)

[Концептуальные основы ООП 4](#_Toc503190048)

[Инкапсуляция и необходимость ее применения в ООП 4](#_Toc503190049)

[Проблема дублирования кода 4](#_Toc503190050)

[Способ решения выявленной проблемы 6](#_Toc503190051)

[Практическая часть 7](#_Toc503190052)

[Требования к проекту 7](#_Toc503190053)

[Задачи библиотеки 7](#_Toc503190054)

[Реализация 9](#_Toc503190055)

[Ключевые слова для обозначения типа генерируемого метода 9](#_Toc503190056)

[Модификаторы доступа 10](#_Toc503190057)

[Объявления типов 10](#_Toc503190058)

[Условные операторы. 11](#_Toc503190059)

[Обработчики данных и функции обратного вызова. 12](#_Toc503190060)

[*Псевдонимы полей класса.* 12](#_Toc503190061)

[*Реализация интерфейсов.* 13](#_Toc503190062)

[Источники 16](#_Toc503190063)

# Введение

В наше время информационные технологии стали неотъемлемой частью жизни почти каждого человека. При разработке нового программного обеспечения специалистам приходится писать всё более сложный и объёмный код. Для упрощения этой задачи может применяться объектно-ориентированное программирование (ООП) – подход, позволяющий организовать элементы программы более структурированно.

Несмотря на очевидные преимущества ООП, использование данного метода сопровождается рядом таких сложностей, как: уменьшение производительности ПО и дублирование кода, в том числе в написании однотипных подпрограмм. Чаще всего используются функции «construct», «toString», «get» и «set», которые выполняют вспомогательную роль в работе программы и могут дублироваться слишком часто.

Подпрограммы «get» и «set» требуется писать для реализации корректной работы с данными. По мере увеличения объёма обрабатываемой информации в коде с использованием ООП количество вспомогательных методов возрастает, и программист вынужден копировать похожие подпрограммы, что влечёт за собой увеличение однотипных структур. Похожее строение методов позволяет разработать инструмент для их автоматической генерации. Таким образом, целью нашей работы будет создание библиотеки для уменьшения общего объёма вспомогательных элементов программы.

**Цель:** Оптимизация процесса написания вспомогательного PHP-кода в объектно-ориентированном программировании.

**Задачи:**

1. Выявить проблемы использования методов «get» и «set» в объектно-ориентированном программировании.

2. Проанализировать информационные источники по выбранной теме.

3. Выбрать подходящий способ решения выявленных проблем.

4. Спроектировать библиотеку автоматической генерации PHP-кода.

5. Реализовать библиотеку на практике.

 Программа реализации проекта представлена в приложении № 1.

# Теоретическая часть

## Концептуальные основы ООП

В работе используется специальная терминология, подробно описанная в приложении № 2.

## Инкапсуляция и необходимость ее применения в ООП

Инкапсуляция является одним из важнейших принципов современного программирования в объектно-ориентированном стиле. Она позволяет отделить реализацию компонентов программы от ее интерфейса. Такой подход позволяет программисту работать с API кода, написанного ранее, не задумываясь о подробностях реализации той или иной части программы. [2]

К примеру, существует класс, который содержит в себе матрицу, заполненную целочисленными значениями.

**class** Matrix {

 **public** $matrix;

}

В силу особенностей реализации, значения ячеек матрицы смещены на единицу по осям абсцисс и ординат. Для того чтобы получить значение матрицы из координат (x, y) нужно будет написать следующий код:

$matrix->matrix[$x + 1][$y + 1];

Не зная о такой особенности устройства программы, легко можно допустить ошибку и долго искать причину некорректной работы кода. Вместо того чтобы сделать поле с матрицей общедоступным, изменим модификатор доступа на “private” и создадим аксессоры для переменной.

**class** Matrix {

 **private** $matrix;

 **public function** getCell(int $x, int $y): int { … }

 **public function** setCell(int $x, int $y, int $val): void { … }

}

Теперь получение значения из матрицы будет выглядеть так:

$matrix->getCell($x, $y);

Код, использующий геттеры и сеттеры, работает более прозрачно и позволяет быстрее понять логику работы программы.

## Проблема дублирования кода

Чем больше полей есть в классе, тем больше методов доступа к ним требуется создавать. Если на одно поле приходится до двух аксессоров, то пять полей потребуют создания до десяти похожих методов. Переписывание похожего кода раз за разом занимает достаточно много времени, и конечный вариант класса со всеми требуемыми аксессорами будет состоять из десятков строк похожего кода.

Стандартно существует несколько способов решения проблемы написания аксессоров:

1. оставлять поля класса общедоступными;
2. генерировать геттеры и сеттеры с помощью IDE;
3. использовать встроенный в язык программирования механизм работы со свойствами класса;
4. подключать к программе сторонние библиотеки, позволяющие сократить синтаксис описания свойств класса или же аксессоров для полей.

Первый способ является достаточно распространенным, хотя его нельзя назвать наилучшим из возможных. В поля класса, доступные из любой области кода, могут быть записаны любые данные, что, вероятно, сделает работу программы непредсказуемой и некорректной. Например, запись нуля в поле, хранящее условный делитель, повлечет за собой арифметическую ошибку. Кроме того, программисту, который работает с кодом, созданным ранее, придется изучать не интерфейс класса, а подробности его внутренней реализации, что отнимает много времени и сил.

Генерация геттеров и сеттеров с помощью среды разработки также распространена. Такой способ не отнимает много времени и позволяет создать аксессоры для нескольких полей класса сразу. Однако этот подход не сокращает количество однообразного кода - класс по-прежнему будет содержать в себе десятки строк описания аксессоров.

Использование свойств значительно более эффективно, чем предыдущие два способа. Описание свойства класса в C# выглядит так:

**public** Sometype something {get; set;}

Такой код позволяет сгенерировать и геттер, и сеттер за одну строку кода. Впоследствии аксессоры могут быть изменены и дополнены. Особенностью использования свойств является то, что пользовательский код выполняет доступ к данным напрямую, а не через специальные методы: **obj.something.**

Для автоматического обеспечения доступа к полям класса также могут использоваться сторонние библиотеки. К примеру, для Java есть Project Lombok [6], а для PHP - Properties [7]. Project Lombok генерирует простые аксессоры и некоторые другие методы, используя информацию из аннотаций Java. Properties является реализацией свойств для PHP. Эти библиотеки помогают убрать из кода только самые простые вариации геттеров и сеттеров - те методы, которые выполняют базовые функции: перезапись значения или возврат данных.

Таким образом, ни один из данных способов не предоставляет короткого синтаксиса для описания аксессоров любой сложности.

В каких целях могут быть использованы аксессоры?

Для того чтобы понять, когда происходит дублирование кода, рассмотрим структуру аксессоров.

Стандартная задача геттеров и сеттеров - выполнять операции по чтению и перезаписи поля класса. Самые простые геттер и сеттер выглядят так:

**public function** getSomething(): Sometype {

 **return** $this->something;

}

**public function** setSomething(Sometype $val): void {

 $this->something = $val;

}

При добавлении в программу нового функционала аксессоры могут быть расширены условиями, обработчиками данных и функциями обратного вызова. Например, есть класс, хранящий адрес электронной почты и производящий с ним некоторые манипуляции. Требуется вести подсчет количества изменений email адреса, чтобы знать сколько раз пользователь менял почтовый ящик. Сеттер для поля, хранящего email адрес, будет выглядеть так:

**public function** setEmail(string $val): void {

 **if** (preg\_match('/[a-z\d]+@[a-z]\.[a-z]/i', $val)) {

 $this->email = strtolower($val);

 ++$this->emailChanged;

 } **else** {

 **throw new** Exception('it is impossible to write value given');

 }

 }

В данном случае сеттер выполняет сразу несколько действий:

1. проверяет, возможно ли записать в поле данное значение;
2. приводит строку с адресом в нижний регистр;
3. записывает данное значение в поле с адресом;
4. выполняет инкремент переменной, хранящей количество изменений адреса.

 Значит, аксессоры выполняют следующие функции:

1. валидация данных;
2. обработка данных;
3. выполнение функций обратного вызова;
4. выполнение операций доступа к данным.

Мы можем представить самый сложный вариант сеттера в таком виде:

***public/protected/private*** **function** set***Field***($val): void {

 **if** (***conditions***) {

 ***handlers***

 ***callbacks***

 $this->***field*** = $val;

 } **else** {

 **throw new** Exception(***description***);

 }

}

Обобщенный геттер будет выглядеть так же, за исключением того, что вместо присвоения значения будет выполняться его возврат.

Похожая структура геттеров и сеттеров позволяет создать короткий синтаксис для описания этих методов, что поможет избежать дублирования кода.

## Способ решения выявленной проблемы

Мы предлагаем перенести описание аксессоров в специализированные однострочные комментарии и создать короткий синтаксис для описания методов доступа. К примеру, комментарий, объявляющий публичные геттер и сеттер для одного целочисленного поля будет выглядеть так: **private $something; #: +axs int**.

Соответственно, такой комментарий будет заменять следующий код:

**public function** getSomething(): int {

 **return** $this->something;

}

**public function** setSomething(int $val): void {

 $this->something = $val;

}

Чем более сложны аксессоры, тем более выигрышным становится использование комментариев с коротким синтаксисом описания методов доступа.

# Практическая часть

## Требования к проекту

Один из наиболее важных этапов разработки проекта - определение границ применения библиотеки и её задач. На этом этапе к проекту были выдвинуты следующие требования:

1. Библиотека должна быть полностью совместимой с уже существующим кодом на чистом PHP.
2. Библиотека должна быть доступной и простой для установки и использования.
3. Библиотека должна существенно уменьшать количество однотипного кода и обеспечивать простой и удобный синтаксис.
4. Весь исходный код должен соответствовать стандартам оформления PSR [17] и быть простым для расширения и дополнения.

Для того чтобы код, созданный с использованием библиотеки, был обратно совместим с чистым PHP-кодом, требуется определить способ сокращения однообразных методов. В практике существует два стандартных способа решения проблемы написания аксессоров с использованием сторонних библиотек:

1. создание реализации свойств класса для целевого языка программирования;
2. создание генератора методов доступа для целевого языка программирования.

Особенностью использования свойств класса является то, что клиентский код обращается к свойствам напрямую, без использования вспомогательных методов. Такой подход, в отличие от генерации геттеров и сеттеров, не может обеспечить полную совместимость с существующим проектом и потребует значительные переработки кода, поэтому было принято решение проектировать библиотеку, которая производит обработку методов доступа.

Для обеспечения доступности проекта было решено использовать систему контроля версий “GitHub” [8] и сервис “Packagist” [9], который позволяет публиковать PHP-репозитории, доступные из менеджера зависимостей “Composer” [10]. Такой набор сервисов позволяет быстро опубликовать библиотеку и легко её распространять.

Для существенного уменьшения количества однообразного кода было принято решение разрабатывать синтаксис описания аксессоров, нацеленный на лаконичность и простоту понимания.

## Задачи библиотеки

Основной задачей библиотеки является сокращение количества тем или иным образом продублированного кода. Такая функциональность должна достигаться путем использования специального короткого синтаксиса для объявления методов доступа к полям классов. Это означает, что можно поставить проекту следующие задачи:

1. Генерация аксессоров любой сложности.
2. Возможность использовать сгенерированные методы точно так же, как и обычные.
3. Возможность создавать дополнительные сопутствующие методы, которые имеют схожую функциональность с геттерами и сеттерами.

Чтобы успешно выполнить первую задачу, требуется разработать такой синтаксис описания аксессоров, который сможет описать геттеры и сеттеры любой сложности. Исходя из стандартной структуры методов доступа, которая была приведена в теоретическом разделе работы, можно выделить лексемы, которые обозначают:

1. Тип метода, который требуется сгенерировать. Так как библиотека производит создание аксессоров, можно выделить три вида генерируемых методов для поля класса:
	1. геттер;
	2. сеттер;
	3. оба метода доступа.
2. Модификатор доступа для сгенерированного метода. В PHP существует три модификатора доступа, каждый из которых может быть применен к аксессору в той или иной ситуации:
	1. public;
	2. protected;
	3. private.
3. Тип данных, хранимых полем класса.
4. Условия, которые должны быть выполнены для успешной записи или чтения поля класса. Для большей гибкости библиотеки можно разделить условия на две группы:
	1. условия, встроенные в библиотеку;
	2. условия, представляющие из себя короткие участки кода, вводимые пользователем.
5. Функции обратного вызова (callbacks), которые могут выполнять обработку входных или выходных данных, а также производить любые другие сопутствующие действия. Аналогично условиям, есть причины разделить функции обратного вызова на две группы:
	1. стандартные функции обратного вызова, встроенные в библиотеку;
	2. функции обратного вызова, представляющие из себя короткие участки кода, вводимые пользователем.
6. Псевдонимы имен полей класса, которые могут быть использованы для сокращения названий методов доступа и разрешения возможных конфликтов между автоматически сгенерированными методами.

Для того чтобы код с аксессорами, сгенерированными автоматически, имел такую же пользовательскую часть, как и код на чистом PHP, нужно определить вид сигнатуры геттера и сеттера, то есть названия методов и принимаемые ими аргументы. Если рассматривать аксессоры для целочисленного поля “$field”, можно выделить три возможных варианта сигнатур методов доступа [11]:

1. getField() и setField(int $val);
2. get\_field() и set\_field(int $val);
3. field() и setField().

Для объектно-ориентированного PHP-кода наиболее характерны названия методов первого типа. Такой стиль именования является общепринятым и определён в стандарте PSR-1 [12]. Именование функций с использованием нижнего подчеркивания чаще используется при программировании в процедурном стиле, а третий вид сигнатур более характерен для C-подобных языков программирования и в PHP почти не встречается.

Исходя из этого, было принято решение генерировать методы с сигнатурами первого типа.

PHP поддерживает интерфейсы и абстрактные классы. Это значит, что геттеры и сеттеры могут быть запрошены из разных структур. Чтобы автоматически сгенерированные методы могли включаться в реализацию интерфейсов, можно комментировать заголовки запрошенных методов специальным образом, характерным только для библиотеки. Впоследствии закомментированные сигнатуры методов могут быть разобраны и сохранены, чтобы библиотека могла проверять реализацию абстрактных методов и методов интерфейсов.

Таким образом, сгенерированные методы могут быть использованы точно так же, как и обычные.

Генерация сопутствующих методов доступа будет производиться с использованием информации о типе поля класса. Для каждого из стандартных типов данных будут определены такие методы доступа, которые встречаются в клиентском коде наиболее часто. К примеру, тип “boolean” будет иметь метод “inverse”, который инвертирует логическое значение поля, а для массивов будут определены специальные индексные аксессоры.

## Реализация

Наиболее простым способом реализовать описанную функциональность является перемещение объявлений геттеров и сеттеров в специализированные комментарии, которые будут разобраны библиотекой на определённые ранее лексемы и обработаны соответствующим образом. Такой подход поможет избежать использования API интерпретатора PHP и сосредоточиться на реализации библиотеки.

PHP поддерживает три вида различных комментариев:

1. Многострочные комментарии в стиле C. Такие комментарии заключены между “/\*” и “\*/”.
2. Однострочные комментарии в стиле C, которые расположены после “//”.
3. Однострочные комментарии в стиле UNIX, расположенные после “#”.

Использование многострочных комментариев противоречит концепции библиотеки. Одним из требований к проекту является максимально короткий синтаксис, а значит описания методов доступа должны занимать не более одной строки.

Комментарии UNIX распространены значительно меньше, чем комментарии второго типа. Соответственно они являются более уникальными и практически не используются другими библиотеками для PHP. Это делает комментарии третьего типа наиболее оптимальными для использования.

Чтобы исключить возможность конфликтов с комментариями, оставленными в коде ранее, было принято решение добавить двоеточие к стандартному комментарию UNIX. Таким образом, описание методов доступа для поля класса будет начинаться с “#:”. Комментирование сигнатур методов в абстрактных классах и интерфейсах распространено значительно меньше, поэтому добавлением двоеточия к комментарию с описанием запрошенного метода можно пренебречь. Значит, объявление требуемых к генерации аксессоров внутри абстрактных классов и интерфейсов будет начинаться с “#”.

## Ключевые слова для обозначения типа генерируемого метода

Возможные вариации генерируемых методов были определены в предыдущем пункте. Чтобы обозначить, какие методы должны автоматически создаваться, было решено использовать следующие ключевые слова:

1. “readable” - для описания геттера;
2. “writable” - для описания сеттера;
3. “accessible” - для запроса обоих методов доступа.

Чтобы сократить синтаксис, также были созданы короткие эквиваленты:

1. “rdb”;
2. “wrt”;
3. “axs”.

Подобный набор лексем позволяет значительно сократить комментарии библиотеки.

## Модификаторы доступа

Для обозначения модификаторов доступа можно использовать стандартные ключевые слова PHP:

1. “public” - обозначает общедоступный метод;
2. “protected” - обозначает метод, который может быть вызван из класса-владельца и его потомков;
3. “private” - оставляет метод доступным только внутри конкретного класса.

Такие ключевые слова привычны и легко читаемы, однако они достаточно длинные и будут занимать ценное место в описании методов. Более того, стандартные модификаторы доступа, скорее всего, будут увеличивать когнитивную комплексность кода класса:

**private** $field; #: public readable

В примере выше в одной и той же строке кода имеется сразу два модификатора доступа: для поля класса и для генерируемых методов. Такой синтаксис может запутывать программиста и усложнять чтение кода.

Чтобы избежать подобных отрицательных эффектов, было принято решение заменить стандартные ключевые слова на соответствующие им символы: “+”, “~” и “-”, которые обозначают общедоступный, защищённый и закрытый методы.

## Объявления типов

В PHP версии 5 и выше существуют инструменты контроля типов [13], которые часто применяются в геттерах и сеттерах. К примеру, такой метод:

**public function** setField(int $val): void { … }

Будет принимать в качестве аргумента только целочисленное значение.

Объявления типов являются одной из наиболее важных функций библиотеки. Это значит, что та часть специализированного комментария, в которой содержится описание типа значения поля, должна покрывать как можно больше возможных различных ситуаций, в которых может быть использован контроль типов.

Описание типа поля класса должно следовать после ключевого слова, обозначающего генерируемый метод. Изначально, библиотека будет поддерживать стандартные типы PHP: integer (int), float (double), boolean (bool), string, array, object, resouorce, callable. NULL исключен, так как будет бесполезен.

Объявление сеттера для целочисленного поля класса должно выглядеть так:

**private** $field; #: +wrt int

Чтобы указать сразу несколько возможных типов, требуется разделять их специальным символом. Было принято решение использовать знак модуля “|”, который также применяется при описании типа в комментариях PHPdoc [14]. Таким образом, поле хранящее строку или целочисленное значение будет объявляться так:

**private** $field; #: +axs int|string

Чтобы покрыть все возможные ситуации использования контроля типов, было принято решение убрать ограничение на количество типов в описании. К примеру, объявление типа сможет выглядеть так: “int|bool|string”. Использование подобных объявлений встречается редко, однако может быть полезно, например, при хранении булевого значения: “no”, false, 0.

## Условные операторы.

Стандартными для аксессоров условиями являются математические выражения, представляющие из себя сравнение различных данных с целыми или дробными числами. К примеру, может потребоваться проверить длину строки перед записью в базу данных:

**class** User {

 **private** $name;

 **public** function setName(string $val): void {

 **if** (strlen($val) <= 255) {

 $this->name = $name;

 } **else** {

 **throw new** Exception(...);

 }

 }

}

В данном примере строка приводится к целочисленному значению путем получения её длины. Затем, результат выполнения “strlen()” сравнивается с верхней границей возможного размера имени пользователя. Для реализации проверки математических выражений было решено сделать некоторые стандартные типы PHP “исчисляемыми”:

1. array обрабатывается, как count($var);
2. string обрабатывается, как strlen($var);
3. integer и float остаются такими же;
4. в том случае, если в программу попадает такой тип данных, который не может быть приведен к целочисленному значению, создается исключение “TypeError”.

В качестве операторов сравнения было решено использовать следующие лексемы:

1. диапазон (`1..10`);
2. “больше” и “меньше” (`>`, `<`);
3. нестрогие операторы неравенства (`>=`, `<=`);
4. оператор равенства (`==`).

Условия библиотеки должны быть записаны после объявления типа. Таким образом, пример, приведенный выше, будет выглядеть так с использованием библиотеки:

**class** User {

 **private** $name; #: +wrt string <= 255

}

Также было принято решение добавить возможность задания пользовательских условий, в которых может быть вызван любой код, возвращающий булево значение. Для записи таких выражений будут использоваться обратные кавычки, ограничивающие исполняемый участок кода. К примеру, требуется проверить, соответствует ли определённая строка регулярному выражению. Требуемый код с использованием библиотеки будет выглядеть так:

**class** User {

 **private** $name; #: +wrt string `preg\_match(‘/[a-z]+/i’, $var)`

}

В данном примере выполняется проверка того, что имя пользователя содержит только буквы латинского алфавита. Для обозначения входных или выходных данных в пользовательских условиях используется псевдопеременная “$var”.

Кроме описания единичных условий, также достаточно часто требуется группировать выражения при помощи логических операторов. Чаще всего при группировке условий используется логическое “и”, “или” и “исключающее или”. Для PHP наиболее распространены первые два оператора, третий же почти не встречается. Чтобы обозначать “и” и “или” было решено использовать стандартные лексемы PHP: || (или), && (и).

При совмещении нескольких подусловий в одном выражении часто используются скобки, однако для условных выражений геттеров и сеттеров они будут уже лишними.

Чтобы вычислить значение общего условия будет применяться следующий алгоритм:

1. разделить всё выражение по лексемам “или”;
2. разделить каждое полученное выражение по лексемам “и”;
3. выполнить проверку каждого из полученных условий;
4. произвести вычисление общего результата выражения согласно семантическим значениям логических операторов.

Таким образом, комплексное условие **condition1 && condition2 || condition3** будет приведено к **(condition1 && condition2) || condition3**

Подобные условные выражения позволят выполнять проверки для любых данных.

## Обработчики данных и функции обратного вызова.

При получении и перезаписи значения поля класса бывает необходимо обработать входные или выходные данные. Примером подобной обработки может являться получение отформатированной строки с датой из метки времени:

**class** User {

 **private** $signUpDate;

 **public function** getSignUpDate(): string {

 **return** date($this->signUpDate, ‘d.m.Y’);

}

}

Библиотека также поддерживает короткие обработчики данных:

**private** $signUpDate; #: +rdb string -> `$var = date($var, ‘d.m.Y’)`

# *Псевдонимы полей класса.*

Достаточно часто требуется изменять имя какого-либо поля, которое используется при вызове некоторых методов. К примеру, существует класс с закрытой переменной, имеющей достаточно длинное название.

**class** WithLongFieldName {

 **private** $fieldWithReallyLongName;

}

Чтобы сделать пользовательскую часть кода более понятной и удобной для чтения, будет логичным сократить название геттера для этого поля с “getFieldWithReallyLongName()” до “getField()”. Для реализации схожей функциональности, было принято решение добавить в специализированный комментарий объявление псевдонима для поля класса. Он должен быть указан после обработчиков и знака =>. Таким образом, предыдущий пример будет выглядеть с использованием библиотеки так:

**class** WithLongFieldName {

 **use** Axessors;

 **private** $fieldWithReallyLongName; #: +axs int => field

}

$test = **new** WithLongFieldName();

$test->getField();

# *Реализация интерфейсов.*

В интерфейсах или абстрактных классах могут быть запрошены методы доступа к полям:

**interface** Accessible {

 **public function** getField(): int;

 **public function** setField(int $val): void;

}

Стандартно библиотека не может добавить в класс сгенерированные методы так, чтобы они реализовывали интерфейсы. Для этого требуется изменять PHP на более глубоком уровне путем создания расширения языка, написанного на C, или же использования уже готовых решений, позволяющих изменять поведение программы. К примеру, uopz [15] или runkit [16] могут обеспечивать схожую функциональность. Такой подход поможет сделать программу более гибкой, однако усложнит использование библиотеки, требуя установки дополнительных компонентов. Исходя из этого, было принято решение сделать сигнатуры аксессоров в абстрактных классах и интерфейсах комментируемыми и обрабатываемыми стандартными средствами библиотеки. Для комментирования методов, так же, как и в случае с полями классов, будет использоваться комментарий в стиле UNIX. Таким образом, пример, приведенный выше, будет выглядеть с использованием библиотеки так:

**interface** Accessible {

 # public function getField(): int;

 # public function setField(int $val): void;

}

**class** Implementer **implements** Accessible {

 **use** Axessors;

 **private** $field; #: +axs int

}

В случае если запрошенные методы не были реализованы, библиотека должна выбрасывать необрабатываемое исключение.

***Разбор содержания специализированных комментариев***

При обработке специализированных комментариев библиотека использует:

1. Reflection API [18] - для получения информации о файлах, в которых были объявлены классы, использующие функциональность библиотеки.
2. Регулярные выражения PCRE [19] - для получения отдельных лексем комментария.

Сначала библиотека получает список всех объявленных в программе классов с помощью функции “get\_declared\_classes()” и выбирает те из них, которые содержат в себе трейт “Axessors” или “Axs”. По очереди открывается каждый требуемый файл, и комментарии построчно разбираются с помощью регулярных выражений, которые описывают каждую лексему. Наиболее важные из них представлены ниже:

1. Начало комментария '#:';
2. Модификатор доступа '(\+|~|-)';
3. Ключевое слово '((accessi|(writ|read)a)ble|axs|wrt|rdb)';
4. Псевдоним поля '[a-zA-Z\_][a-zA-Z0-9\_]\*'.

Парсинг комментария осуществляется с помощью класса “Lexer” и его метода “parse()”:

**protected function** parse(string $code, array $expectations, array $requiredItems): array{

 $this->currentSym = 0; // Начало отсчета символов комментария.

 $code = trim($code);

 $result = [];

 **foreach** ($expectations **as** $index => $pattern) { // Цикл для каждой лексемы.

 $this->skipWhitespace($code); // Игнорирование пробельных символов.

 preg\_match($pattern, substr($code, $this->currentSym), $token); // Получение лексемы.

 **if** (empty($token)) { // Проверка, является ли лексема необходимой.

 **if** (in\_array($index, $requiredItems)) {

 **throw new** ParseError("token with pattern \"$pattern\" not found while parsing $this->reflection->getFileName()}:{$this->lineNumber}");

 }

 } **else** {

 $result[$index] = $token[0]; // Добавление лексемы в общий список.

 $this->currentSym += strlen($token[0]); // Перемещение по комментарию.

 }

 } **return** $result;

}

Такой способ разбора комментария позволяет добавлять неограниченное число пробельных символов между отдельными лексемами, что позволяет логически выравнивать блоки кода.

 В приложении № 3 приводятся дополнительные возможности разработанной библиотеки.

***Логика работы библиотеки***

После того как библиотека получает всю информацию о методах доступа, описанных в специализированных комментариях, она сверяет списки сгенерированных методов со списками методов, которые были запрошены в интерфейсах и абстрактных классах. В случае если некоторые из методов не были сгенерированы, выбрасывается исключение “OopError”.

При вызове методов доступа библиотека моделирует работу реальных геттеров и сеттеров в специальных классах “MethodRunner”, “ConditionsRunner” и “HandlersRunner”.

***Апробация библиотеки на практике***

В целях тестирования функциональности библиотека была внедрена в тестовый проект “OOP in UML” [20], представляющий из себя объектно-ориентированную модель офиса. Общий объём файлов, в которых была использована библиотека, сократился с 6,7 кБайт до 4,6 кБайта. Полный код тестового проекта без использования библиотеки и оптимизированного с помощью библиотекb можно найти в репозитории на GitHub.com по ссылке <https://goo.gl/9zsoaM>.

**Заключение**

В процессе работы над проектом были решены все задачи и достигнута основная цель. Процесс написния вспомогательного PHP-кода был значительно оптимизирован пуём создания альтернативного синтаксиса описания аксессоров. По итогам работы можно сделать следующий вывод:

1. В процессе анализа существующих способов уменьшения общего объёма однотипного кода был найден наиболее приемлемый способ решения проблемы.
2. Были определены необходимые требования к проекту и средства его реализации.
3. Был разработан наиболее короткий синтаксис описания методов доступа.
4. Библиотека была спроектирована и реализована на практике.

В перспективе работы над библиотекой планируется:

1. создание более простого синтаксиса специализированных комментариев;
2. реализация более производительной версии библиотеки с использованием расширения uopz [15];
3. перенос библиотеки на другие языки программирования и платформы.

Представленный синтаксис описания методов доступа к данным позволяет значительно сократить общий объём исходного кода и упростить разработку программного обеспечения с использованием объектно-ориентированного программирования.

# Источники

1. Объектно-ориентированное программирование // Википедия. URL: [https://ru.wikipedia.org/wiki/Объектно-ориентированное\_программирование](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (дата обращения: 22.12.2017)
2. Вайсфельд, М. Объектно-ориентированное мышление / М. Вайсфельд - СПб.: Питер, 2014. - 304 с. // (дата обращения: 22.12.2017)
3. PHP // Википедия. URL: <https://ru.wikipedia.org/wiki/PHP> (дата обращения: 22.12.2017)
4. Веб-сервер // Википедия. URL: [https://ru.wikipedia.org/wiki/Веб-сервер](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80) (дата обращения: 24.12.2017)
5. Jeffrey, E.F. Mastering Regular Expressions / E.F. Jeffrey - O’Reilly Media, Inc, 2006. – 515 с. // (дата обращения: 25.12.2017)
6. Project Lombok // URL: <https://projectlombok.org> (дата обращения: 21.12.2017)
7. Properties // GitHub. URL: <https://github.com/SerafimArts/Properties> (дата обращения: 22.12.2017)
8. GitHub // URL: <https://github.com> (дата обращения: 23.12.2017)
9. Packagist. // The PHP Package Repository. URL: <https://packagist.org> (дата обращения: 23.12.2017)
10. Composer // PHP dependency Manager. URL: <https://getcomposer.org> (дата обращения: 24.12.2017)
11. Именование геттеров и сеттеров. // URL: <http://demin.ws/blog/russian/2010/11/09/how-to-name-getters-and-setters/> (дата обращения: 24.12.2017)
12. PSR-1 // PHP Standards Recommendations.URL: <http://www.php-fig.org/psr/psr-1/> (дата обращения: 23.12.2017)
13. Новые функции PHP 5.0 // PHP: Hypertext Preprocessor. URL: <http://php.net/manual/en/migration70.new-features.php> (дата обращения: 23.12.2017)
14. phpDocumentor // URL: <https://phpdoc.org> (дата обращения: 23.12.2017)
15. Uopz // PHP: Hypertext Preprocessor. URL: <http://php.net/manual/ru/book.uopz.php> (дата обращения: 23.12.2017)
16. Runkit // PHP: Hypertext Preprocessor. URL: <http://php.net/manual/ru/book.runkit.php> (дата обращения: 23.12.2017)
17. PSR // PHP Standards Recommendations. URL: <http://www.php-fig.org/psr/> (дата обращения: 23.12.2017)
18. Reflection API // PHP: Hypertext Preprocessor. URL: <http://php.net/manual/ru/book.reflection.php> (дата обращения: 23.12.2017)
19. [Perl-Compatible Regular Expressions // PHP: Hypertext Preprocessor. URL: http://php.net/manual/ru/book.pcre.php](http://php.net/manual/ru/book.pcre.php) (дата обращения: 22.12.2017)
20. [“OOP in UML”](https://github.com/NoOne4rever/Axessors/tree/master/examples/OOP%20in%20UML) // GitHub. URL: <https://github.com/NoOne4rever/Axessors/tree/master/examples/OOP%20in%20UML> (дата обращения: 22.12.2017)
21. [PHP: Препроцессор гипертекста // URL: https://php.net](https://php.net/) (дата обращения: 20.12.2017)
22. Zandstra, M. PHP Objects, Patterns and Practice / M. Zandstra – Apress Media LLC, 2016. – 583 с. // (дата обращения: 20.12.2017)
23. Axessors: generator of getters and setters for PHP // GitHub. URL: <https://github.com/NoOne4rever/Axessors> (дата обращения: 20.12.2017)
24. API // Википедия. URL: <https://ru.wikipedia.org/wiki/API> (дата обращения: 21.12.2017)

*Приложение 1.* ***Программа реализаии проекта.***

|  |  |  |
| --- | --- | --- |
| Номер этапа | Этап реализации | Сроки |
| 1 | Постановка цели и задач, изучение проблематики вопроса | Февраль 2017 |
| 2 | Изучение аналогов (в т.ч. Properties и Project Lombok) | Март 2017 |
| 3 | Реализация библиотеки на практике | Апрель – август 2017 |
| 3.1. | Проектирование библиотеки | Апрель 2017 |
| 3.2. | Разработка синтаксиса описания аксессоров  | Май – Июнь 2017 |
| 3.3. | Разработка библиотеки | Июль – Август 2017 |
| 4 | Документирование возможностей библиотеки; оформление репозитория на GitHub | Сентябрь – Октябрь 2017 |
| 5 | Написание модульных тестов, улучшение внутренней реализации библиотеки | Ноябрь 2017 |
| 6 | Расширение возможностей библиотеки | Декабрь 2017 |
| 7 | Распространение информации о библиотеке в сети Интернет | С января 2018 |
| 8 | Поиск путей внедрения библиотеки в проектную деятельность Web-разработчиков | С февраля 2018 |

*Приложение № 2.* ***Терминология, используемая в работе.***

ООП - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. [1]

Класс - это элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию. Наряду с понятием «объекта» класс является ключевым понятием в ООП. [2]

Объект - экземпляр класса. [2]

Поле - переменная, связанная с классом или объектом. Все данные объекта хранятся в его полях. [2]

Метод - функция или процедура, принадлежащая какому-либо классу или объекту. [2]

Свойство - способ доступа к внутреннему состоянию объекта, имитирующий переменную некоторого типа. [2]

Инкапсуляция - упаковка данных и функций в один компонент. [2]

Сокрытие - принцип проектирования, заключающийся в разграничении доступа различных частей программы к внутренним компонентам друг друга. [2]

Геттер - специальный метод, позволяющий получить данные, доступ к которым напрямую ограничен. [2]

Сеттер - метод, используемый в объектно-ориентированном программировании для того, чтобы присвоить какое-либо значение инкапсулированному полю. [2]

Аксессоры - методы доступа, то есть геттеры и сеттеры. [1]

Интерфейс - программная структура, определяющая отношение между объектами, которые разделяют определённое поведенческое множество и не связаны никак иначе. [2]

PHP - скриптовый язык общего назначения, интенсивно применяемый для разработки веб-приложений. В настоящее время поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов. [3]

Веб-сервер - сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными. [4]

Регулярные выражения - формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов. [5]

API (англ. application programming interface) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением для использования во внешних программных продуктах. [24]

*Приложение 3.* ***Функции, добавленные в библиотеку в версии 1.1.***

***Объявление аксессоров для нескольких полей одновременно***

PHP позволяет объявлять несколько полей класса в одной строке кода. В примере ниже координаты геометрической фигуры записаны с использованием этого синтаксиса.

**class** Shape {

**private** $x, $y;

}

Координаты фигуры имеют методы доступа одинакового вида. Вместо того чтобы дублировать специальный комментарий для каждого поля, есть смысл создать один комментарий для обоих полей.

**private** $x, $y; #: +axs int

Таким образом будет сгенерировано сразу четыре метода доступа:

1. getX(): int;
2. setX(int $val): void;
3. getY(): int;
4. setY(int $val): void.

Единственным ограничением, которое накладывается на специализированный комментарий, принадлежащий сразу нескольким полям, является запрет на объявление псевдонима поля.

**private** $x, $y; #: +axs int => smth

Такой код логически некорректен, и вызовет ошибку работы программы, поэтому псевдоним, объявленный в комментарии к нескольким полям, не обрабатывается.

***Короткий синтаксис “$this->” внутри обратных ударений***

В пользовательских условиях или функциях обратного вызова достаточно часто встречается специальная переменная “$this”, которая используется для доступа к полям класса. Для того чтобы сократить общую длину условного комментария, библиотека поддерживает короткое обозначение “$this”. При этом подстроки комментария соответствующие “$this->” заменяются на “$.”.

Таким образом, пример поля из тестового проекта “OOP in UML”:

**private** $position; #: +axs string -> `$this->addPastPosition(new :PastPosition($this->position, $this->departament))`

Будет сокращен до:

**private** $position; #: +axs string -> `$.addPastPosition(new :PastPosition($.position, $.departament))`

***Исполняемые блоки кода внутри обратных ударений***

По умолчанию библиотека ожидает получить в пользовательском условии или обработчике одно выражение.

**private** $field; #: +axs int `2+2\*2 == 6 `

Однако, может возникнуть необходимость записать внутри обратных ударений несколько команд. Для этого можно использовать анонимную функцию:

private $field; #: +axs int `(function() { $x = rand(1, 10); return $x != 1 && $x != 3; })()`

Это полностью корректная, но достаточно длинная запись. Для того чтобы сократить комментарий, библиотека позволяет уменьшить такое объявление до нескольких выражений, заключенных между фигурными скобками.

private $field; #: +axs int `{$x = rand(1, 10); return $x != 1 && $x != 3;}`

***Поле класса по умолчанию***

В том случае если класс хранит в себе одно основное значение поля, методы доступа могут не иметь постфикса с его названием.

**class** Email {

**private** $email;

**public** **function** get(): string {

 **return** $this->email;

}

}

Библиотека также поддерживает «поля по умолчанию». Для того чтобы объявить аксессоры без постфикса, нужно использовать «default» в качестве псевдонима поля.

**class** Email {

**private** $email; #: +rdb string => default

}

$email->get();

***Сгруппированные условия***

Начиная с версии 1.1, библиотека поддерживает сгруппированные скобками логические условия.

**private** $exactInteger; #: +axs int (!= 4 && (1..10 || > 100))

При этом условия, обозначенные в комментарии, транслируются в чистый PHP-код и исполняются точно так же, как и обработчики внутри обратных ударений. Это способствует увеличению производительности библиотеки и позволяет точнее проверять значения полей класса.