php-annotations Documentation

Release 1.0.0

Rasmus Schultz

Nov 24, 2022

Contents

This library provides support for source-code annotations in PHP.

This library references practices and features established by other languages and platforms with native support for annotations, mainly C#/.NET and Java, drawing on the strengths (while observing the limitations) of the PHP language.

The main areas of this Wiki are listed below:

- **Roadmap, upgrading and release notes*
- **Understanding and applying annotations to PHP source code*
- **Consuming source-code annotations at run-time*
- **Writing your own types of annotations*
- **See a fully documented, step-by-step example of declarative meta-programming at work*
- **Review the standard library of annotations*
- **Learn more about the design considerations behind this library*

This library was created by Rasmus Schultz - a lot of careful planning and design went into this project, which was in development for almost a year before it's initial release.

CHAPTER 1

Getting Started

Below you'll find all needed information to find your way across the library.

1.1 Installation

Library can be installed using Composer like so:

```
$ composer require mindplay/annotations
```

CHAPTER 2

Roadmap, upgrading and release notes

This project implements support for source-code annotations in PHP (5.3+).

Referencing established practices and proven features of annotation-support from other languages and platforms with native support for annotations (mainly C#/.NET and Java), this library implements a complete, "*industrial strength*" annotation engine for PHP, drawing on the strengths (while observing the limitations) of the language.

2.1 Status

The current status of the individual components is as follows:

- The core annotation framework (and API) is stable and complete.
- Documentation is *being written*.
- Some standard annotations are still stubs or only partially done.
- A fully documented *demonstration script* is available.

2.2 Roadmap

The current release consists of the following components:

- Core annotation framework. (adds support for annotations)
- Self-contained unit test suite.
- Documentation.
- Example.

A library of useful standard (PHP-DOC and other) annotations has been started, but is incomplete.

2.3 Upgrading

Version 1.1.x introduces some incompatibilities with previous versions:

- The cache-abstraction has been removed - refer to this note explaining why. If you wrote your own cache-provider, you should remove it.
- If you derived custom annotation-types from the `Annotation\Annotation` base-class, you must rename the `_map()` method to `map()` - the underscore suggested a private method, but this method is actually protected.

CHAPTER 3

Understanding annotations

3.1 What are annotations?

Annotations are meta-data that can be embedded in source code.

You may already be familiar with the PHP-DOC flavor of source code annotations, which are seen in most modern PHP codebases - they look like this:

```
class Foo
{
  /**
   * @var integer
   */
  public $bar;
}
```

When you see `@var integer` in the source code, immediately preceding the `public $bar` declaration, this little piece of meta-data tells us that the `$bar` property is expected to contain a value of type `integer`.

This information is useful to programmers - an IDE can use this information to display popup hints when you're writing code that works with an instance of `Foo`, documentation generators can display this information in reference material, etc.

Now imagine you had to build an HTML form that allows someone to edit a `Foo` object. When this information comes back from a form-post, you will need to validate that the input is in fact an integer - using the information from the `@var` annotation, you could abstract and automate this process, rather than having to code in by hand every time.

The same information about the type of value required for this property could have many other uses besides validation - for example, you could use it to decide what type of input to render on a form, or how to persist the value to a database column.

Using other types of annotations besides `@var`, you could provide more information about the `$bar` property - for example, you might use an annotation to specify the minimum and maximum allowed integer values of a property for validation, or define a label to be displayed next to the input on forms or in the column-header of a list of `Foo` objects:

```
class Foo
{
  /**
   * @var integer
   * @range(0, 100)
   * @label('Number of Bars')
   */
  public $bar;
}
```

We now have the information about the type of value, the allowed range, and the label, all associated with the `Foo::$bar` property member. You may have noticed a subtle difference between the `@var` annotation and the other two annotations in this example: the extra parentheses - we'll get to that below.

It's important to understand that this meta-data does not have a single predefined purpose - it is general information, which when put to use in creative ways, can be used to simplify or eliminate repetitive work, and enables you to write more elegant and reusable code.

3.2 What does this library do?

This library allows you to implement annotation-types as classes, and apply them as objects.

Annotations are translated into objects using a simple rule: `@name` is essentially equivalent to `new NameAnnotation()` - in other words, the annotation name is capitalized and an "Annotation" suffix is added to the class-name; this prevents the class-names of annotation-types from colliding with the names of other classes.

Using this library, annotations applied to classes, methods and properties may be inspected at run-time. Combined with the Reflection API, this enables you to apply reflective meta-programming in your PHP projects.

3.3 Annotation Syntax

This library provides support for two types of annotation syntax. While the difference between the two syntaxes is subtle, the difference in terms of how they function is very different.

The first type of syntax is based on PHP-DOC syntax:

```
/**
 * @var integer
 */
public $bar;
```

PHP-DOC annotations do not have a fixed syntax - that is, everything after `@name` is interpreted differently for each type of annotation. For example, the syntax for `@var` is `@var {type} {description}`, while the syntax for `@param` is `@param {type} {$name} {description}`.

In other words, PHP-DOC style annotations have to go through an extra step at compile-time. (note that there is no performance penalty for this extra step, since the compiled annotation properties are cached.)

For simple annotations (like those defined by the PHP-DOC specification), this syntax is usually preferable.

For custom annotations (perhaps requiring more complex properties), a second syntax using parentheses is supported:

```
/**
 * @range(0, 100)
```

(continues on next page)

(continued from previous page)

```
 */
public $bar;
```

When this syntax is used, the run-time equivalent for this example is something along the lines of:

```
$annotation = new RangeAnnotation();
$annotation->initAnnotation(array(0, 100));
```

In other words, everything between the parentheses is standard PHP array syntax - as you're probably already comfortable with this syntax, there is no additional syntax to learn.

While an annotation-type can optionally implement a custom (PHP-DOC style) syntax annotation, the array-style syntax is supported by every annotation. To achieve compatibility with IDEs and documentation generators, you should use the PHP-DOC style syntax for annotations defined by the PHP-DOC standard.

Both syntaxes have unique advantages:

- PHP-DOC style offers shorter syntax for commonly used annotation-types, and compatibility with IDEs and documentation generators.
- Array-style syntax offers direct access to PHP language features, such as access to class-constants, static method-calls, nested arrays, etc.

Note that both syntaxes cause annotations to initialize in the same way, at run-time - via a call to the `IAnnotation::initAnnotation()` interface, passing an array of property values. The PHP-DOC style syntax simply adds an extra step where the annotation values are parsed, and the initialization code for those properties is generated (and cached).

3.4 So what can I do with this?

See a real, working example of declarative meta-programming with this library in *this commented, step-by-step example* - the same script is available in the demo folder in the project, and can be run from your local web-server.

CHAPTER 4

Working with the Annotation Manager

Note: Some programmers learn best by seeing a practical example - if you belong to those who learn best by seeing things applied, you should start by taking a look at the *demo script*, which provides a minimal, yet practical, real-world example of applying and consuming source code annotations.

The annotation framework lives in the `mindplay\annotations` namespace, and the library of *standard annotations* lives in the mindplay\annotations\standard namespace.

The heart of the annotation framework is the AnnotationManager class, which provides the following functionality:

- Inspecting (and filtering) annotations
- Annotation registry and name-resolution
- Caching annotations in the local filesystem (underneath the hood)

Behind the scenes, the `AnnotationManager` relies on the AnnotationParser to perform the parsing and compilation of annotations into cacheable scripts.

For convenience, a static (singleton) wrapper-class for the annotation manager is also available. This class is named Annotations - we will use it in the following examples.

4.1 Loading and Importing

Going into details about autoloading and importing the annotation classes is beyond the scope of this article.

I will assume you are familiar with these language features, and in the following examples, it is implied that the static wrapper-class has been imported, e.g.:

```
use mindplay\annotations\Annotations;
```

4.1.1 Configuring the Annotation Manager

For convenience, the static `Annotations` class provides a public `$config` array - the keys in this array are applied the singleton `AnnotationManager` instance on first use, for example:

```
Annotations::$config = array(
    'cachePath' => sys_get_temp_dir()
);
```

In this example, when the `AnnotationManager` is initialized, the public `$cachePath` property is set to point to the local temp-dir on your system.

Other configurable properties include:

Property	Type	Description
`$fileMode`	int	...
`$autoload`	bool	...
`$cachePath`	string	...
`$cacheSeed`	string	...
`$suffix`	string	...
`$namespace`	string	...
`$registry`	array	...

4.1.2 The Annotation Registry

...

4.2 Inspecting Annotations

...

4.2.1 Annotation Name Resolution

...

4.2.2 Filtering Annotations

...

CHAPTER 5

Writing your own types of annotations

5.1 What is an Annotation?

An annotation is just a class - it is merely the way it gets initialized (and instantiated) that makes it an annotation.

In order for a class to work as an annotation, it must:

- have a constructor with no arguments - e.g.: `function __construct()` (or no constructor)
- implement the `IAnnotation` interface - e.g.: `function initAnnotation($properties)`
- be annotated with an `@usage` annotation - see below for details.

Beyond the quantitative requirements, you should make some qualitative considerations. Here are some things to consider:

- Annotations are specifications - they can provide default values for various components, or define additional behaviors or metadata. But your components should not *depend* on a specific annotation - if you find you're trying to define an annotation that is *required* for your components to operate, there's a good chance you'd be better off defining that behavior as an interface.
- Try to design your annotation types for general purposes, rather than for a specific purpose - there is a good chance you may be able to use the same metadata in new ways at a later time. Choose broad terms for class-names (and property-names) so as not to imply any specific meaning - just describe the information, not it's purpose.
- Do you need a new annotation type, or can one of the existing types be used to define what you're trying to specify? Be careful not to duplicate your specifications, as this leads to situations where you'll be forced to write the same metadata in two different formats - the point of annotations is to help eliminate this kind of redundancy and overhead.

5.2 UsageAnnotation

The UsageAnnotation class defines the constraints and behavior of an annotation.

An instance of the built-in `@usage` annotation must be applied to every annotation class, or to it's ancestor - the `@usage` annotation itself is inheritable, and can be overridden by a child class.

The standard `@length` annotation, for example, defines it's use as follows:

```
/**
 * Specifies validation of a string, requiring a minimum and/or maximum length.
 *
 * @usage('property'=>true, 'inherited'=>true)
 */
class LengthAnnotation extends ValidationAnnotationBase
{
  ...
}
```

This specification indicates that the annotation may be applied to properties, and that the annotation can be inherited by classes which extend a class to which the annotation was applied.

The `@usage` annotation is permissive; that is, all of it's properties are `false` by default - you have to turn on any of the permissions/features that apply to your annotation class, by setting each property to `true`.

Let's review the available properties.

- The `$class`, `$property` and `$method` flags simply specify to which type(s) of source-code elements an annotation is applicable.
- The `$multiple` flag specifies whether more than one annotation of this type may be applied to the same source-code element
- The `$inherited` flag specifies whether the annotation(s) will be inherited by a class extending the class to which the annotations were applied.

Different combinations of the `$multiple` and `$inherited` flags result in the following behavior:

	`$multiple=true`	`$multiple=false`
`$inherited=true`	Multiples allowed and inherited	Only one allowed, inherited with override
`$inherited=false`	Multiples allowed, not inherited	Only one allowed, not inherited

Note that annotations with `$multiple=false` and `$inherited=true` are a special case, in which only one annotation is allowed on the same code-element, and is inherited - but can be overridden by a child-class which would otherwise inherit the annotation.

When overriding an inherited annotation, it's important to understand that the individual properties of an annotation are *not* inherited - the *entire* annotation is replaced by the overriding annotation.

CHAPTER 6

Fully documented, step-by-step example of declarative meta-programming

File: **demo/index.php**

```
namespace mindplay\demo;

use Composer\Autoload\ClassLoader;
use mindplay\annotations\AnnotationCache;
use mindplay\annotations\Annotations;
use mindplay\demo\annotations\Package;
```

Configure a simple auto-loader

```
$vendor_path = dirname(__DIR__) . '/vendor';

if (!is_dir($vendor_path)) {
    echo 'Install dependencies first' . PHP_EOL;
    exit(1);
}

require_once($vendor_path . '/autoload.php');

$auto_loader = new ClassLoader();
$auto_loader->addPsr4("mindplay\\demo\\", __DIR__);
$auto_loader->register();
```

Configure the cache-path. The static `Annotations` class will configure any public properties of `AnnotationManager` when it creates it. The `AnnotationManager::$cachePath` property is a path to a writable folder, where the `AnnotationManager` caches parsed annotations from individual source code files.

```
Annotations::$config['cache'] = new AnnotationCache(__DIR__ . '/runtime');
```

Register demo annotations.

```
Package::register(Annotations::getManager());
```

For this example, we're going to generate a simple form that allows us to edit a `Person` object. We'll define a few public properties and annotate them with some useful metadata, which will enable us to make decisions (at run-time) about how to display each field, how to parse the values posted back from the form, and how to validate the input.

Note the use of standard PHP-DOC annotations, such as `@var string` - this metadata is traditionally useful both as documentation to developers, and as hints for an IDE. In this example, we're going to use that same information as advice to our components, at run-time, to help them establish defaults and make sensible decisions about how to handle the value of each property.

```
class Person
{
    /**
     * @var string
     * @required
     * @length(50)
     * @text('label' => 'Full Name')
     */
    public $name;

    /**
     * @var string
     * @length(50)
     * @text('label' => 'Street Address')
     */
    public $address;

    /**
     * @var int
     * @range(0, 100)
     */
    public $age;
}
```

To build a simple form abstraction that can manage the state of an object being edited, we start with a simple, abstract base class for input widgets.

```
abstract class Widget
{
    protected $object;
    protected $property;

    public $value;
```

Each widget will maintain a list of error messages.

```
    public $errors = array();
```

A widget needs to know which property of what object is being edited.

```
    public function __construct($object, $property)
    {
        $this->object = $object;
        $this->property = $property;
        $this->value = $object->$property;
    }
```

Widget classes will use this method to add an error-message.

```
    public function addError($message)
    {
        $this->errors[] = $message;
    }
```

This helper function provides a shortcut to get a named property from a particular type of annotation - if no annotation is found, the `$default` value is returned instead.

```
    protected function getMetadata($type, $name, $default = null)
    {
        $a = Annotations::ofProperty($this->object, $this->property, $type);

        if (!count($a)) {
            return $default;
        }

        return $a[0]->$name;
    }
```

Each type of widget will need to implement this interface, which takes a raw POST value from the form, and attempts to bind it to the object's property.

```
    abstract public function update($input);
```

After a widget successfully updates a property, we may need to perform additional validation - this method will perform some basic validations, and if errors are found, it will add them to the `$errors` collection.

```
    public function validate()
    {
        if (empty($this->value)) {
            if ($this->isRequired()) {
                $this->addError("Please complete this field");
            } else {
                return;
            }
        }

        if (is_string($this->value)) {
            $min = $this->getMetadata('@length', 'min');
            $max = $this->getMetadata('@length', 'max');

            if ($min !== null && strlen($this->value) < $min) {
                $this->addError("Minimum length is {$min} characters");
            } else {
                if ($max !== null && strlen($this->value) > $max) {
                    $this->addError("Maximum length is {$max} characters");
                }
            }
        }

        if (is_int($this->value)) {
            $min = $this->getMetadata('@range', 'min');
            $max = $this->getMetadata('@range', 'max');

            if (($min !== null && $this->value < $min) || ($max !== null && $this->
↪value > $max)) {
```

(continues on next page)

(continued from previous page)

```
                $this->addError("Please enter a value in the range {$min} to {$max}");
            }
        }
    }
```

Each type of widget will need to implement this interface, which renders an HTML input representing the widget's current value.

```
    abstract public function display();
```

This helper function returns a descriptive label for the input.

```
    public function getLabel()
    {
        return $this->getMetadata('@text', 'label', ucfirst($this->property));
    }
```

Finally, this little helper function will tell us if the field is required - if a property is annotated with `@required`, the field must be filled in.

```
    public function isRequired()
    {
        return count(Annotations::ofProperty($this->object, $this->property,
↪'@required')) > 0;
    }
}
```

The first and most basic kind of widget, is this simple string widget.

```
class StringWidget extends Widget
{
```

On update, take into account the min/max string length, and provide error messages if the constraints are violated.

```
    public function update($input)
    {
        $this->value = $input;

        $this->validate();
    }
```

On display, render out a simple `<input type="text"/>` field, taking into account the maximum string-length.

```
    public function display()
    {
        $length = $this->getMetadata('@length', 'max', 255);

        echo '<input type="text" name="' . get_class($this->object) . '[' . $this->
↪property . ']"'
            . ' maxlength="' . $length . '" value="' . htmlspecialchars($this->value)␣
↪. '"/>';
    }
}
```

For the age input, we'll need a specialized `StringWidget` that also checks the input type.

```
class IntWidget extends StringWidget
{
```

On update, take into account the min/max numerical range, and provide error messages if the constraints are violated.

```
    public function update($input)
    {
        if (strval(intval($input)) === $input) {
            $this->value = intval($input);
            $this->validate();
        } else {
            $this->value = $input;

            if (!empty($input)) {
                $this->addError("Please enter a whole number value");
            }
        }
    }
}
```

Next, we can build a simple form abstraction - this will hold and object and manage the widgets required to edit the object.

```
class Form
{
    private $object;

    /**
     * Widget list.
     *
     * @var Widget[]
     */
    private $widgets = array();
```

The constructor just needs to know which object we're editing.

Using reflection, we enumerate the properties of the object's type, and using the `@var` annotation, we decide which type of widget we're going to use.

```
    public function __construct($object)
    {
        $this->object = $object;

        $class = new \ReflectionClass($this->object);

        foreach ($class->getProperties() as $property) {
            $type = $this->getMetadata($property->name, '@var', 'type', 'string');

            $wtype = 'mindplay\\demo\\' . ucfirst($type) . 'Widget';

            $this->widgets[$property->name] = new $wtype($this->object, $property->
↪name);
        }
    }
```

This helper-method is similar to the one we defined for the widget base class, but fetches annotations for the specified property.

```
    private function getMetadata($property, $type, $name, $default = null)
    {
        $a = Annotations::ofProperty(get_class($this->object), $property, $type);

        if (!count($a)) {
            return $default;
        }

        return $a[0]->$name;
    }
```

When you post information back to the form, we'll need to update it's state, validate each of the fields, and return a value indicating whether the form update was successful.

```
    public function update($post)
    {
        $data = $post[get_class($this->object)];

        foreach ($this->widgets as $property => $widget) {
            if (array_key_exists($property, $data)) {
                $this->widgets[$property]->update($data[$property]);
            }
        }

        $valid = true;

        foreach ($this->widgets as $widget) {
            $valid = $valid && (count($widget->errors) === 0);
        }

        if ($valid) {
            foreach ($this->widgets as $property => $widget) {
                $this->object->$property = $widget->value;
            }
        }

        return $valid;
    }
```

Finally, this method renders out the form, and each of the widgets inside, with a `<label>` tag surrounding each input.

```
    public function display()
    {
        foreach ($this->widgets as $widget) {
            $star = $widget->isRequired() ? ' <span style="color:red">*</span>' : '';
            echo '<label>' . htmlspecialchars($widget->getLabel()) . $star . '<br/>';
            $widget->display();
            echo '</label><br/>';

            if (count($widget->errors)) {
                echo '<ul>';
                foreach ($widget->errors as $error) {
                    echo '<li>' . htmlspecialchars($error) . '</li>';
                }
                echo '</ul>';
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Now let's put the whole thing to work...

We'll create a `Person` object, create a `Form` for the object, and render it!

Try leaving the name field empty, or try to tell the form you're 120 years old - it won't pass validation.

You can see the state of the object being displayed below the form - as you can see, unless all updates and validations succeed, the state of your object is left untouched.

```
echo <<<HTML
<html>
  <head>
    <title>Metaprogramming With Annotations!</title>
  </head>
  <body>
    <h1>Edit a Person!</h1>
    <h4>Declarative Metaprogramming in action!</h4>
    <form method="post">
HTML;

$person = new Person;

$form = new Form($person);

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if ($form->update($_POST)) {
        echo '<h2 style="color:green">Person Accepted!</h2>';
    } else {
        echo '<h2 style="color:red">Oops! Try again.</h2>';
    }
}

$form->display();

echo <<<HTML
    <br/>
    <input type="submit" value="Go!"/>
    </form>
HTML;

echo "<pre>\n\nHere's what your Person instance currently looks like:\n\n";
var_dump($person);
echo '</pre>';

echo <<<HTML
  </body>
</html>
HTML;
```

CHAPTER 7

Standard annotations library

A library of standard annotations will eventually be (**but is not currently**) part of this package.

Note: Experienced developers (primarily maintainers of libraries that rely on annotations), are encouraged to contribute or suggest changes or additions to the standard library.

The standard library of annotations generally belong to one (or more) of the following categories:

- **Reflective annotations** to describe aspects of code semantics not natively supported by the PHP language.
- **Data annotations** to describe storage/schema aspects of persistent types.
- **Display annotations** to specify input types, labels, display-formatting and other UI-related aspects of your domain- or view-models.
- **Validation annotations** to specify property/object validators for domain- or form-models, etc.
- **PHP-DOC annotations** - a subset of the standard PHP-DOC annotations.

Some of the annotations belong to more than one of these categories - the standard PHP-DOC annotations generally fall into at least one of the other categories.

Most of the standard annotations were referenced from annotations that ship with other languages and frameworks that support annotations natively, mainly .NET and Java. Due to the strict nature of these languages, as compared to the loose nature of PHP, the standard annotations were not merely ported from other languages, but adapted to better fit with good, modern PHP code.

7.1 Available Annotations

Note: The annotation library is not yet available, or still in development.

7.1.1 Category: Reflective, PHP-DOC

Annotation	Scope	Description
MethodAnnotation	Class	Defines a magic/virtual method.
ParamAnnotation	Method	Defines a method-parameter's type.
PropertyAnnotation	Class	Defines a magic/virtual property and it's type.
PropertyReadAnnotation	Class	Defines a magic/virtual read-only property and it's type.
PropertyWriteAnnotation	Class	Defines a magic/virtual write-only property and it's type.
ReturnAnnotation	Method	Defines the return-type of a function or method.
VarAnnotation	Property	Specifies validation of various common property types.
TypeAnnotation	Property	Specifies validation of various common property types.

7.1.2 Category: Display

Annotation	Scope	Description
DisplayAnnotation	Property	Defines various display-related metadata, such as grouping and ordering.
EditableAnnotation	Property	Indicates whether a property should be user-editable or not.
EditorAnnotation	Property	Specifies a view-name (or path, or helper) to use for editing purposes - overrides ViewAnnotation when rendering inputs.
FormatAnnotation	Property	Specifies how to display or format a property value.
TextAnnotation	Property	Defines various text (labels, hints, etc.) to be displayed with the annotated property.
ViewAnnotation	Property/class	Specifies a view-name (or path) to use for display/editing purposes.

7.1.3 Category: Validation

Annotation	Scope	Description
EnumAnnotation	Property	Specifies validation against a fixed enumeration of valid choices.
LengthAnnotation	Property	Specifies validation of a string, requiring a minimum and/or maximum length.
MatchAnnotation	Property	Specifies validation of a string against a regular expression pattern.
RangeAnnotation	Property	Specifies validation against a minimum and/or maximum numeric value.
RequiredAnnotation	Property	Specifies validation requiring a non-empty value.
ValidateAnnotation	Class	Specifies a custom validation callback method.

CHAPTER 8

Design considerations

This page will attempt to explain some of the design considerations behind this library.

8.1 Feature Set

The feature set was mainly referenced from languages with proven annotation support and established use of annotations - the primary inspiration was the .NET platform and Java.

Other existing annotation-libraries for PHP were also referenced, for both good and evil:

- The popular Addendum library brought some good ideas to the table, but adds unnecessary custom syntax and parses annotations at run-time.
- Doctrine's Annotations library achieves a number of good things, but also adds unnecessary custom syntax.
- The Recess framework has good support for annotations and relies on them to solve a number of interesting challenges in highly original ways, making it a great inspiration.
- A proposed native Class MetaData extension to the PHP language: follows no existing standards, (incompatible with existing IDEs, documentation generators, existing practices and codebases); mixes JSON, a data-serialization format, into PHP - I would welcome JSON support in PHP, but not solely for annotations, and it should not replace what can already be achieved with existing PHP language features.

When held against the annotation feature-set of the C#/.NET or Java platforms, these implementations have some weak points:

- These libraries use various custom, data-formats to initialize annotations - neglecting support for common language features, such as class-constants, static method calls and closures.
- Support for inheritance is lacking, limited or incorrect in various ways - inheriting and overriding annotations is an absolute requirement.
- Common constraints are unsupported or too simple - applicable member types, cardinality and inheritance constraints should be easy to specify, and must be consistently enforced.

The absence of these features is what sparked the inception of this library.

8.2 Syntax

The syntax is based on PHP-DOC annotations mixed with PHP standard array syntax.

The decision to use PHP-DOC syntax was made primarily because PHP-DOC source code annotations are already very common in the PHP community, and well-established with good design-time support by many popular IDEs. Many types of useful standard PHP-DOC annotations can be inspected at run-time.

Extending this syntax with standard PHP array syntax is practical for a number of reasons:

- PHP array-syntax is already familiar to PHP developers, and naturally allows you to initialize annotation properties using PHP language constructs, including constants, anonymous functions (closures), static method-calls, etc.
- It reduces the complexity of parsing, since PHP already knows how to parse arrays.
- There is no compelling reason to introduce new syntax (and more complexity) to achieve something that is already supported by the language.

Rather than attempting to reinvent (or having to forego) important aspects of existing language features, this library builds on existing PHP syntax, and existing establish conventions, as much as possible, introducing a minimal amount of new syntax. This makes it feel like a more natural extension to the language itself.

8.3 API

The API has two levels of public interfaces - an annotation-manager, which can be extended, if needed, and a simple static wrapper-class with static methods, mostly for convenience.

Extending the Reflection API with annotation features might seem like a natural approach, since this is where you would find it on other platforms such as .NET. There are a couple of reasons why this is not necessary or practical:

- PHP might very well add native support for annotations to the reflection classes someday - if (or when) that happens, we don't want our API to conflict with (or hide portions of) any eventual extensions to the native reflection API.
- This library minimally relies on reflection itself.
- There is nothing in particular to gain by mixing the annotation APIs with reflection in the first place.

8.3.1 Freedom

This library has no external dependencies on any non-standard PHP modules or third-party libraries.

Annotation-types implement an interface - they do not need to extend a base-class, which enables it to fit into your existing class-hierarchies without requiring you to refactor your existing codebase.

8.4 Performance

From a performance-oriented perspective, a scripting language may not be a good choice for writing any kind of parser. Since some form of parsing is inevitable, the following design choices were made early on to minimize the overhead of using annotations:

- The annotation-parser is only loaded as needed, e.g. after a change (invalidating the cache-file) is made to an inspected script-file.

- The annotation-manager compiles (JIT) and caches annotation data - the annotations from one PHP script file are written to one cache-file. This simple strategy results in one additional script being loaded, when a script is inspected for annotations.
- Since the cache-file itself is a PHP script, the annotation library can take advantage of a bytecode cache for additional performance gains.

In general, as much work as possible (or practical) is done at compile-time, minimizing the run-time overhead - no tokenization or parsing or is performed at run-time, except the first time a script is inspected.