Meta Programming and Reflection in PHP

Gregor Gabrysiak, Stefan Marr, and Falko Menge

Hasso-Plattner Institute, at the Universiy of Potsdam, Germany
{gregor.gabrysiak, stefan.marr, falko.menge}@hpi.uni-potsdam.de

Abstract. A meta program is a program, which has other programs or even itself as its application domain. This technique is commonly used to implement advanced software development tools and frameworks. PHP is one of the most popular programming languages for web applications and provides a comprehensive set of meta programming and reflection facilities. This paper gives an overview of these capabilities and also presents our own enhancements.
The default configuration of PHP includes a set of functions and an object-oriented reflection API. Additional features are provided by extensions to the virtual machine of PHP. On top of the features mentioned above, we implemented two advanced APIs, providing annotation support, type introspection, and object-oriented intercession. A class browser, which is capable of modifying its own source code at runtime, demonstrates the most important reflection features.

1 Introduction

Every computer program has a certain problem domain. This domain is represented in the internal program state and the program reasons about or acts on it. A meta program is a program which has another program as its problem domain, i.e., it analyzes or modifies the internal structures of the other program. This technique is commonly used by tools for development, analysis and visualization of software, e.g., compilers, code generators, or IDEs. Essentially, most software developers use meta programs to create and modify their programs. Putting it a step further, a meta program can also have itself as its problem domain, meaning that it is able to analyze or even modify its own internal structures during it runs. This capability is called reflection and needs to be supported by the runtime environment of the programming language. Introspection, i.e., the capability of a program to inspect its own structure, and intercession, its capability to change its own behavior, have to be supplied. Reflection is useful for the implementation of development tools as well as advanced programming frameworks, e.g., frameworks for aspect-oriented programming (AOP).

The programming language in focus of this paper will be PHP, which is one of the most popular programming languages for web applications. Like many other languages PHP provides a set of meta programming and reflection facilities. This paper gives an overview of these capabilities and discusses stregnths and weaknesses of their implementation. In our earlier work we already developed

own enhancements to the build-in features of PHP and this has been continued throughout the current work.

The remainder of this paper is structured as follows: In the following section, the dynamic scripting language PHP is presented. This includes its history, domains of application, and also its special features in section 2.2. Then, in the context of meta programming, the basic reflection features, such as the historically grown procedural API and its object-oriented reflection API, are presented in section 3.1. Here, the design decisions taken to build the object-oriented part in its own subsystem are emphasized. Afterward, a short overview about the common usage of reflection in the PHP community is given. Section 3.3 introduces our implementation of an annotation mechanism, which is one of the common use cases the reflection API is used for. The main contribution of our current work is presented in section 3.4. It is a prototype of a class browser utilizing the PHP's reflection capabilities to its limits. How intercession is possible in PHP is shown in section 4.1, including the presentation of the Runkit extension and some further extensions for meta programming in general.

2 PHP - A Dynamic Scripting Language

2.1 History and Application Domains of PHP

In 1995, Rasmus Lerdorf improved his personal home page. To simplify this work, he developed a set of Perl scripts, which proved to be very helpful. Therefore, he later published them as "Personal Home Page Tools". Since they were a great success, a C implementation (PHP version 2) followed in 1996. One year later, version 2 was already used by 1% of all Web servers. In the fast growing World Wide Web, the market share of version 3 increased even faster, up to 10% in 1998.

In 2000, PHP4 introduced not only the Zend Engine, a new interpreter and virtual machine for executing PHP scripts, but also a rudimentary object model. This is discussed in more detail in section 2.2. Due to some shortcomings, version 5 introduced a new object model in 2004.

In November 2007, still approx. 75% of all Web servers using PHP run version 4 [1]. That's one of the reasons PHP4 had still been maintained, till its support was discontinued in December 2007. Currently it is available in version 4.4.8 and 5.2.5, while version 6 is already being developed [2].

The traditional domain of PHP is server-side scripting. Whenever a Web server receives a request, the scripts are parsed, interpreted, and executed in the VM, thus, content is created dynamically, mostly in conjunction with some database access. In this domain, scripts usually run only for some milliseconds.

PHP is also used for command-line scripting, were it is partly replacing Perl, Python, awk, and shell scripting. The next area it advances to is desktop applications. This is possible via bindings to GUI libraries, such as GTK+, Qt, and Win32. phpCrow, a desktop application that is explained in detail in section 3.4, uses GTK+ desktop application.

Of course, there are some problems associated with this new domain. The runtime of a desktop application is usually longer than that of a simple PHP script to execute, thus, imposing more severe performance restrictions over time. Since PHP only employs a reference counting algorithm, developers of GUI applications have to take care about memory consumption and cyclic data structures explicitly.

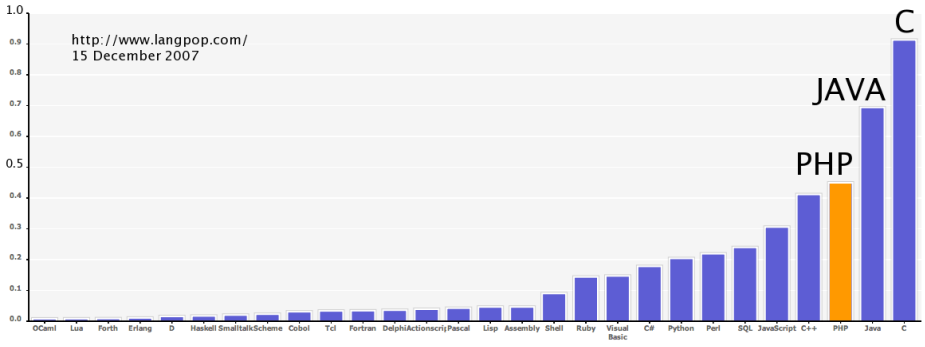

Fig. 1. Popularity of programming languages according to, e.g., the amount of books available at Amazon and the amount of Web sites about them.

2.2 Object Model

When version 4 was released in 2000, an object model was introduced to shift the focus from the purely procedural programming paradigm to object orientation.

The community used these new possibilities to create projects such as PEAR[1] and the PHP Classes Repository[2], however, there were several shortcomings in the introduced object and programming model. Since there were no visibilities available, everything was public and encapsulation could not be realized. The default semantics of passing or assigning objects was by-value instead of using their references. Also, there was no equivalent of `try{}` and `catch()` available, since exceptions were not supported.

In 2004, version 5 was released, which also included a new designed object model. This solved all the aforementioned problems.

While the procedural style of PHP still resembles C, the new object model is similar to Java, adopting the main concepts and supporting the same modifiers for methods and classes (e.g. `public`, `protected`, `private`, `abstract`, `final`). A class can implement multiple interfaces, however, only single inheritance is supported.

[1] PEAR - PHP Extension and Application Repository: http://pear.php.net/
[2] PHP Classes: http://www.phpclasses.org/

As a dynamically typed language, PHP's properties only need a name and a visibility to be used. Such a variable with a default value is declared as follows: `public $name = 'default value';`. If no visibility is supplied, the properties or functions are assumed to be `public`. As opposed to other dynamic languages, PHP version 5.1 introduced type hinting, an optional restriction of a function's parameters. A method's signature, which takes a certain object as a parameter, can be enriched with the object's class name. If any other class' instance is used, the invocation results in a fatal error. This works also for arrays, however, not for primitive types. Nevertheless, it is currently not possible to use type hinting for return values or to specify whether a value is returned at all.

2.3 Magic Methods

Magic methods can be used to customize the behavior for common operations such as object cloning or handling of object properties. Except for `__autoload()`, they are defined on a per class basis.

As in other scripting languages, a class' descriptions can be located in different files. To make use of them, they have to be loaded. Until PHP4, this had to be done explicitly, however, PHP5 introduced `__autoload()` to avoid strong coupling and provide a lazy loading mechanism. Similar to Java's class loaders, it is able to load only the necessary classes, at runtime. To do so, a global `__autoload()` function has to be defined. The user can specify a custom algorithm to define which files are to be loaded. Upon the instantiation of the class (e.g. `$a = new MyClass();`), the interpreter tries to retrieve the class definition from its list of already know classes, if this fails, `__autoload()` will be executed with the name of the class to be loaded as parameter. A simple implementation would try to include a file `'./MyClass.php'` to load the class. If this file does not hold the class' description and eventually the custom `__autoload()` does not load a class with the requested name, the execution will be aborted with an error.

PHP also offers constructors and destructors. On instantiating a class, its `__construct()` method is invoked. If it is not overloaded, the standard constructor is used. The destructor `__destruct()` works in the same manner.

The methods `__get()` and `__set()` are called by the interpreter as soon as a property, which is not part of a class' description, is read from or written to. If these methods are not overloaded by the class, the interpreter will generate a notice upon reading a non-existing property. The default behavior of `__set()` may be surprising at first glance, because if a value is assigned to a non-existing property, no notice is shown, however, the property is dynamically created and added to the instance. Accessing this new property of this instance does not result in a notice anymore, trying to read it from another instance still does.

The correct usage of these magic methods decrease error-proneness for users through additional exceptions, e.g., for non-existent properties and simplifies the implementation of additional features.

In Smalltalk, it is possible to refer invoked messages to other objects if the receiver does not implement them. To enable this behavior, the method

`#doesNotUnderstand:` has to be overloaded to send the original message, including its parameters, to another object. In PHP, the same behavior can be achieved by overloading the `__call()` method.

This behavior is shown in listing 1.1 in appendix A. All messages sent to an object of the class `Poodle`, which `Poodle` does not implement, are delegated to the `$postmanFromTheHood` object. However, in this simple example only the method `getNewspaper()` of the class `Postman` is available.

The correct usage of magic methods can decrease error-proneness through additional exceptions, e.g., for non-existent properties and simplifies the implementation of additional features.

3 Features for Meta Programming and Reflection

In the following section the meta programming and reflection capabilities of PHP will be presented. For the most common use cases it is not possible to use arbitrary extensions to the PHP execution engine. Thus, this section will only consider features available within the standard distribution of PHP.

3.1 The Reflection API

In modern programming languages a typical way to obtain structural information is to leverage the runtime environment of the language and access language elements through a high level reflection API.

Features
As of PHP5, a reflection API has been included in the standard PHP distribution. It provides an API to obtain information about classes, including all of their methods and properties, without the need to parse any source code. Simple functions, runtime objects and PHP extensions can be inspected as well. Introspection is not the only feature of the reflection API. Furthermore, it is also possible to modify properties of objects, invoke arbitrary methods or functions, and instantiate objects, i.e. doing some basic intercession.

An overview of the features available is shown in Fig. 2, which visualizes important functions provided by the procedural API on the left-hand side and the class structure of the object-oriented one on the right-hand side. The design of the object-oriented API part is discussed in sec. 3.1 and a complete model of the reflection API including all methods is shown in Fig. 9 in appendix B.

The procedural part has grown since PHP4 and includes basic functions like `class_exists()` and `get_declared_classes()` to check which classes are available in the current process instance. Also, there are functions to check which script files have been already loaded. `ini_get()` and `ini_set()` can be used to adapt configuration parameters for specific subsystems of the PHP execution environment at runtime. Functions to introspect the current stack or to determine variables defined in the current scope are available, too.

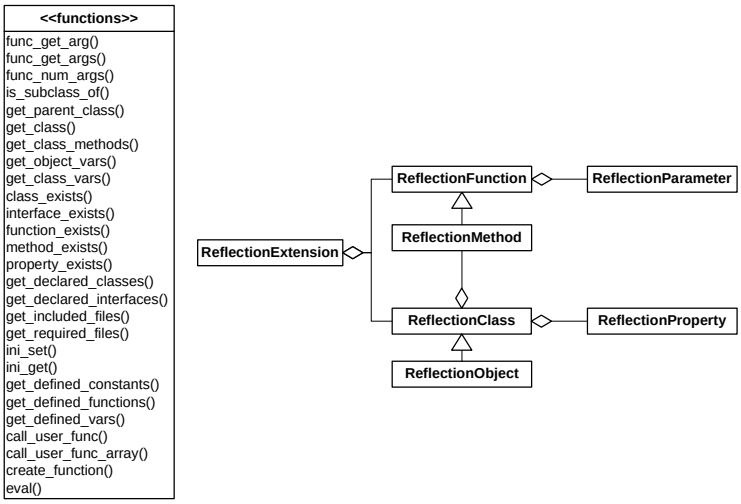

Fig. 2. Overview of PHP5 Reflection API

As already mentioned, PHP provides different mechanisms to invoke arbitrary functions at runtime, e.g., to implement callbacks. The language directly supports a concept similar to *function pointers* using a variable with a function name as its value. The notation looks like `$functionName($a, $b)`, where the value of `$functionName` defines the actual function to be called. More flexible functions are `call_user_func()` and `call_user_func_array()` which enable the user to use object methods as callbacks and arbitrary number of parameters.

Callbacks are not the only way to execute user code. As in many scripting languages, PHP has a function called `eval()` which takes a string and tries to interpret and execute this string as PHP code. There are almost no restrictions on the code to be evaluated. Thus, it is possible to generate code, including function and class definitions at runtime, and execute it using `eval()`. One use case is discussed in section 3.4 in detail.

In some situations, it is nice to save the overhead of writing a distinct function by utilizing the languages capabilities to define anonymous functions right where they are used. This practice is found mainly when working with some sort of arrays or collections. In Smalltalk, *Blocks* are used while C# uses *Delegates* to sort, filter or process all given items.

PHP has a notion of anonymous functions, too. However, in PHP, anonymous functions are not closures and, thus, are not connected to there lexical context. Anonymous functions are build from ordinary strings and the internal implementation uses the same mechanisms as `eval()`. Unlike other languages, `create_function()` is an ordinary function and not a language feature. This might be a benefit if this partial code where it is used in is executed only rarely, because it does not impose additional parse time overhead. However,

`create_function()` implemented as ordinary function has some disadvantages, making it rather difficult to use. Since ordinary strings are used to describe function parameters and the function body, there is no syntax highlighting in common editors and the PHP *string variable parsing* [3] has to be considered. Additionally, there is no caching for anonymous functions and, therefore, the misuse of `create_function()` can result in performance impacts.

Design Characteristics

This part will discuss the object-oriented part of the reflection API and the design decisions taken for its implementation, distinguishing it from common implementations as in Smalltalk or Java. Instead of integrating the object-oriented reflection API tightly into the core system, it is built as a separate subsystem. Fig. 3 illustrates the architecture of the implementation. The storage (rounded rectangle) in the upper part represents the runtime data available within the PHP execution environment, e.g., information about classes, objects, functions, etc. The agent *PHP Core* is able to read some of these information and provides an API to access them, adumbrated by the UML fragment included. The same holds for the agent *Reflection*. Additionally, both are built into different subsystems without a direct connection. The gray arrows, pointing to the storage of the runtime data, imply that the abilities to change data in this storage is restricted and not all data which can be read can be modified.

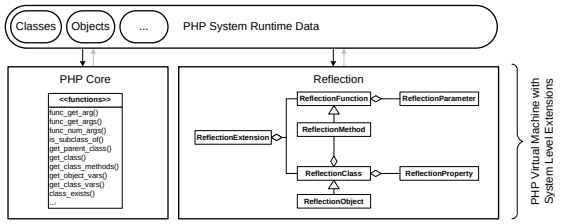

Fig. 3. Reflective subsystems in PHP5 (FMC block diagram [4], agents detailed with UML)

The encapsulation implies that the PHP object model itself has not been enhanced by reflective capabilities. For instance, the `Object` class of Java provides a `getClass()` method to reflect on an object. In PHP, there is no such method in order to ensure a strong separation. Instead, the programmer is forced to use special classes to reflect on the current objects and needs to instantiate reflection objects explicitly.

This design idea has been formulated and its implications illustrated by Bracha and Ungar [5]. One of the side effects mentioned is the fact, that the reflective subsystem can be disabled entirely, which is possible with PHP, too. It can be disabled at compile time to reduce the memory footprint of the whole system.

Encapsulation and stratification are the design principles behind this implementation decision. In contrast to a strict encapsulation and stratification as proposed, PHP does not include all reflection functions into the reflection subsystem. However, the benefits of this design principles are already noticeable. With respect to backward compatibility, the procedural API is still part of the PHP core and has not been moved to the reflection extension, where the object-oriented part is implemented in.

The structural correspondence of the reflection API is only partially, too. As shown in Fig. 2, the reflection API conforms to the meta model of PHP as an object-oriented programming language. That's why, it is possible to reflect nearly all aspects of a given class or object. However, the API provides only rudimentary access to dynamic aspects. Method or function bodies are not accessible directly through this API. It is possible to dynamically invoke methods, but changing code at runtime is not.

Nevertheless, extensions are available for those advanced meta programming approaches, e.g the Runkit extension discussed in sec. 4.1.

3.2 Common Usage of Reflection

In the large and creative PHP community, many Web frameworks exist, which intend to ease the development of Web applications. Each is providing some special features and, sometimes, they are using even reflection, however, in general it is not used widely. In the following, some projects and their usage of reflection are named. Since this is only a small part of the PHP community, it does not reflect the actual usage of those projects in broader scale.

The Symfony Project [6] implements mixin and hook mechanisms using meta programming. Torpedeo [7] and Stubbles [8] are frameworks relying on annotations, implemented using the reflection subsystem.

PicoContainer [9] is an implementation of a container for the *Inversion of Control* pattern, which originates from the Java world. It uses reflection mainly to implement component resolution and instantiation.

S2AOP [10] is a dependency injection container, too. It includes additional usages of reflection for providing basic AOP constructs using the *Proxy* pattern.

For serialization, the reflection API is in broader usage. Phing [11] is using it for XML to PHP mappings and the Zend Framework [12] provides a JSON encoder.

In general, the reflection API is more commonly used to implement developer tools. Such tools include for example the Webservice Helper [13], which generates WSDL files, and our InstantSVC project [14].

PHPUnit [15], as a widely accepted project in the PHP community, makes heavy use of reflection, too. It is not only a testing framework anymore, since it also evolved to a tool for measuring code quality. Its generation of test skeletons, the execution of test cases, its support for annotations, and the computation of code metrics are based on reflection.

3.3 ezcReflection

In our previous work [16] on the InstantSVC project [14] we developed an extended version of PHP's reflection API which introduces annotations and type information to PHP. Meanwhile, this enhancement has been contributed to the eZ Components project [17] and is now available as *ezcReflection*.

For our current work, we enhanced ezcReflection with a delegation mechanism, which allows for integration of other reflection extensions or even alternative implementations, to be able to make use of the isvcRunkit in phpCrow, which is described in 4.2.

Annotations in PHP
Annotations are expressions in the source code associated with certain language elements and following a formal syntax. They are not necessarily evaluated by a language's runtime environment itself. Instead, annotations are typically used by meta programming tools, e.g., code generators.

With PHP 5.1 the capabilities of the reflection API have been enhanced. Now it is possible to extract all associated source code comments using this API. This forms a perfect foundation to build an annotation mechanism for PHP. In order to retrieve annotations through reflection, they have to be part of a docblock comment which must directly precede the language element it describes. An annotation itself starts with an `@` followed by a name and optionally one or more values separated by whitespaces. Listing 1.1 includes some examples how annotations are used to document code.

The ezcReflection component is shown in Fig. 4 which is a combination of an FMC block diagram [4] with a UML class diagram. For all classes of PHP's reflection API new subclasses have been introduced providing additional functionality. In order to access annotations the methods `getAnnotations` and `hasAnnotation` have been added to the classes `ezcReflectionClass`, `ezcReflectionMethod`, `ezcReflectionProperty`, and `ezcReflectionFunction`. Annotations are represented as instances of `ezcReflectionAnnotation` or specialized subclasses. An annotation parser analyzes the source code comments and creates the according annotations objects through a factory. Both, the parser and the factory can be exchanged by alternative implementations.

All extensions made by ezcReflection are implemented entirely in PHP code and the PHP interpreter is not modified at all. A drawback of the current implementation is the performance issue. Using these annotations at runtime in a common web application is not advisable. Instead, it is more appropriate to use this annotation implementation to build tools on top, which will generate something from it. One application would be to generate PHP classes for an object relational mapping and save them to files. This will avoid the overhead for generating this classes at runtime for every single request.

Interestingly, there have been comparable developments in other programming languages, e.g., XDoclet [18] for Java has been a success and led to the inclusion of a annotation mechanism into the language itself.

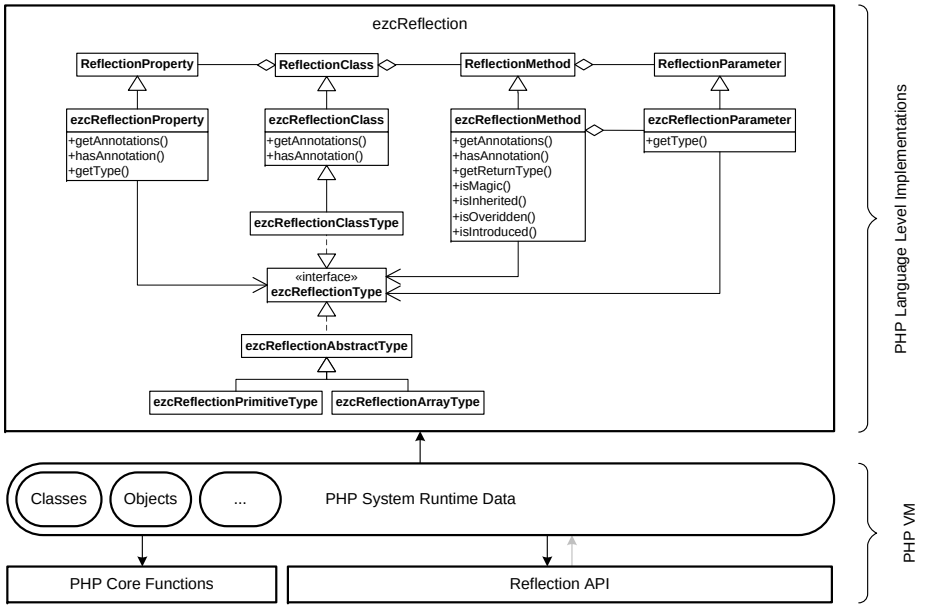

Fig. 4. ezcReflection with annotation accessors and type system.

Type Information

PHP's Reflection API does not provide a complete set of type information due to PHP being a dynamically typed language with only rudimentary type hinting, i.e. in general, types of concrete variables or objects can be determined only at runtime. When looking at common object-oriented PHP applications it stands out that they only rarely use the dynamic typing features of PHP, e.g., to deal with various forms of user input or to produce different types of output. However, large parts are written just as if a statically typed language had been used. The code would work with strict type checks as well. For the code example in the appendix it is known at design-time that the property `Dog::$name` and the return value of the method `Postman::getNewspaper()` will be of type string.

This behavior is also specified in the source code documentation, which is stored in comments of the according language elements. Thus, it appears that in many cases type information already exist statically in the source code. For many years, this information has been used by phpDocumentor [19], a documentation generation tool for PHP code. The annotation syntax specified by phpDocumentor has become the de-facto standard to document PHP source code and is also used by the annotation mechanism in ezcReflection. Hence, it is the logical next step to access the information given in the source code documentation directly at runtime.

The ezcReflection reuses the syntax of phpDocumentor and also some of its annotation tags. To retrieve the data types of properties, parameters, and return values, the annotations `@var`, `@param`, and `@return` are analyzed.

The ezcReflection classes provide methods to retrieve data types of parameters, return values and properties. In order to not only provide type descriptions as strings, the annotations are mapped to an object-oriented type system as shown in Fig. 4. The type system distinguishes three main categories of types: primitives, arrays and class types. This differentiation is made in consideration of the language behavior and the capabilities of WSDL for defining types in a language independent way. Since arrays are treated in a special way, they are additionally classified as simple arrays or maps (dictionaries).

In order to maintain the dynamic nature of the language, ezcReflection does not enforce any constraints at runtime. As a consequence, the component expects a correct source code documentation and it is possible that runtime behavior does not correspond to the behavior expected, due to improper documentation.

The extension of PHP's reflection API allowed a very elegant implementation of a sophisticated WSDL generator. The major advantage of this approach is, that the algorithms for retrieving type information are not tied into the WSDL generator. Instead, the ezcReflection component is a reusable building block which already became a foundation for a whole family of meta programming tools.

Extensibility
Throughout the work for the example shown in section 3.4, we developed a delegation mechanism for the ezcReflection, which allows for integration of other reflection extensions or even alternative implementations. The main idea is to have the constructors of the classes `ezcReflectionClass`, `ezcReflectionMethod`, `ezcReflectionProperty`, and `ezcReflectionFunction` accepting a reflection instance as their argument. This instance will then be used by all methods as their data source, instead of calling the respective parent methods. Thus, the features of ezcReflection can be used on top of alternative implementations of the reflection API. For example, there is some ongoing work on a reflection implementation using static source code analysis instead of PHP's internal data structures. This allows for meta programming tools like phpCallGraph [20], phUML [21], or the WSDL generator of InstantSVC, to leverage static and dynamic analysis through the same API.

Furthermore, the magic method `__call()` has been implemented to redirect calls to the injected reflection instance if it provides own extensions to the API. This way, users who made own enhancements to PHP's reflection API can use them in combination with the features of ezcReflection, without having to maintain multiple reflection objects.

3.4 Reflection used as Development Tool

To demonstrate the limitations of what is possible with PHP, a small tool for developing PHP applications has been built. It uses the concept of Smalltalk's class browser to allow application developers to modify their code and experience the changes at the same time, in the same execution environment. It turned

out that, using the standard PHP distribution, a lot is possible already. Fig. 5 shows a screenshot of the implemented prototype. phpCrow is built using a PHP wrapper for the platform independent GTK2 widget toolkit [22].

The implementation of the class browser as an introspection tool for the current execution environment was quite easy. The reflection API allows to build a full-fledged view on all structural constructs and, therefore, all structural aspects of a running PHP program can be browsed through using this prototype.

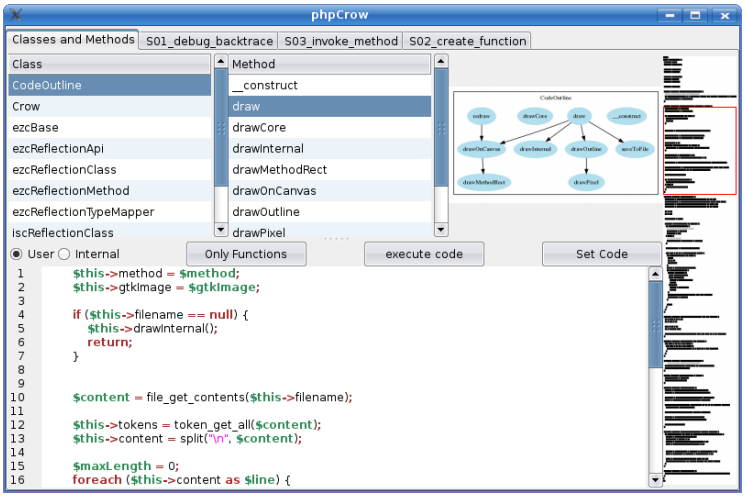

Fig. 5. Classbrowser prototype implemented with PHP and reflection

Classes are listed separately depending on whether they are internal or user defined. Functions are browsable the same way, too. The code of a selected function or method is shown below the class and method lists. It is extracted directly from the source code files, using the information about its start and end line within its file, provided by the reflection API.

The widget next to the method/function list shows a call graph, generated from the currently selected class. For information about the overall class structure, the reflection API is used, however, to extract the relationships between methods the code has to be parsed. Fortunately, PHP provides access to its tokenizer and takes care of the parsing step using the standard lexer of the language. The token stream can then be analyzed to identify the call dependencies among the methods of the selected class. The syntax of PHP is very supportive for tasks like this, since it makes it easier to distinguish dynamic and static method calls inside a class from function calls and calls to methods of other classes.

On the right-hand side, an additional widget shows the outline of the current source code file the selected class is defined in. If a method is selected, it is highlighted with a red border. The outline is created by using the tokenizer. For each non-whitespace token a pixel is drawn. The color of the pixel can be

changed according to the token type, e.g., to be able to distinguish comments from code. The position of the highlighting rectangle is estimated by using the reflection API to retrieve the line numbers of the selected function.

For demonstration purposes, the prototype uses tabs to present different code snippets. In the presentation, these snippets had been evaluated to present the output to the audience. The evaluation has been implemented using `eval()` and some output buffering to redirect the output to a widget, instead of printing it on the console. It is possible to define new classes and functions in these snippets, too. There are no hooks to register a specific callback function on any event like *new class defined* or *new function defined* in the reflection API, but fortunately, the execution engine is fast enough to refill the list widgets of classes and functions after each snippet's evaluation from scratch, without a noticeable delay.

As the prototype illustrates, it is possible to reflect on all structural information in PHP. Additional classes and functions can be defined at runtime, too. However, notifications about changes in these structures are not available. Furthermore, with a standard PHP distribution it is not possible to change existing structures or behavior of functions/methods. Runkit, an extension to PHP, enables developers to change existing structures at runtime. An overview about its features and usage in this prototype class browser is given in the following sections.

4 PHP Extensions for Meta Programming and Reflection

The standard distribution already provides a reasonable set of meta programming and reflection features. However, especially the intercession possibilities are still limited. This section will present several extensions of the PHP virtual machine providing access to its internals. Due to their experimental nature these extensions are not included in the default configuration of PHP. However, they can be easily installed using PEAR [23] or the Windows installer.

4.1 Doing Intercession with Runkit

The Runkit extension [24] shown in Fig. 6 provides a set of functions to do intercession in PHP. At runtime it allows to add, copy, redefine, rename, and remove user-defined function and methods. Constants and class constants can be added, redefined, and removed. Through adopting and emancipating classes, the entire inheritance structure defined by the user can be changed. The import function allows to load or reload class and function definitions from a file, overwriting those existing in the currently running environment. All this is done by modifying PHP's internal data structures, e.g., a method is added by creating an anonymous function and registering it as a method of the desired class.

Another interesting feature is the ability to check inside a method or function whether its return value is going be used by the caller. With this knowledge time-consuming calculations or the creation of complex object structures can

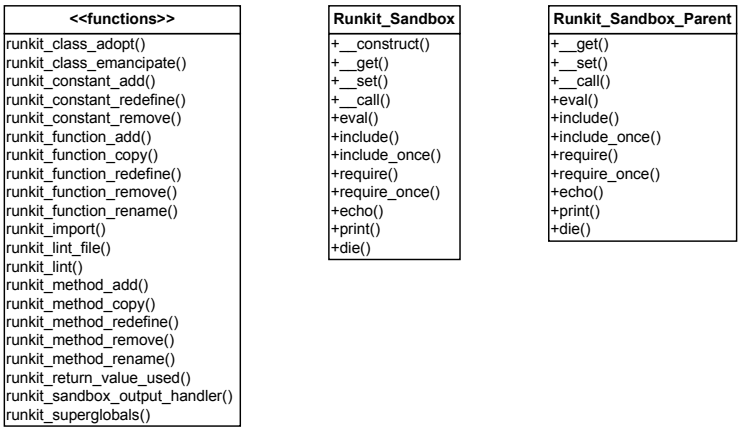

Fig. 6. The Runkit Extension (UML class diagram)

be avoided if the caller is not interested in the result. Hence, this could lead to an increased performance of the application or could be used to detect common programming errors.

Besides the procedural part of Runkit's API it also provides two classes which encapsulate the entire PHP virtual machine into a sandbox within the same process. This allows to execute untrusted code in a controlled environment. The classes also implement communication facilities to exchange data between parent and sandbox and execute functions or methods on the opposite side. Runkit itself uses the sandbox for the syntax checking provided by its lint functions. Furthermore it is useful for meta programming, e.g., software analysis and visualization.

4.2 isvcRunkit and phpCrow

Although the features of Runkit are powerful, the procedural programming model is rather inconvenient when dealing with object-oriented structures. Therefore, we integrated its features into an enhancement to the reflection API called *isvcRunkit*, which now provides an object-oriented API for Runkit. Fig. 7 shows a UML class diagram of isvcRunkit.

We decided against adding this functionality directly to ezcReflection in order to avoid the dependency to an experimental extension. Instead, we introduced the delegation mechanism described in section 3.3 which can be leveraged to use them together with each other. The `getCode` methods we added, is not a feature of Runkit. They obtain the code directly from the source files using the information about filename, start line and end line provided by the reflection API. When the body of a method is changed via `setCode`, the new source code is cached in order to be available for future `getCode` requests.

isvcRunkit is used in our demo application phpCrow to allow the application to modify its own code at runtime. The user of phpCrow can edit the body of

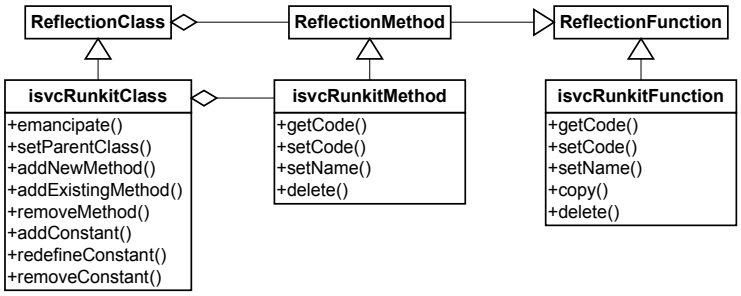

Fig. 7. isvcRunkit: An object-oriented API for Runkit (UML class diagram)

a method selected from the method list and immediately see the reaction in the user interface. For example in our presentation we demonstrated, how the sorting of the class list can be changed at runtime.

4.3 Further Extensions for Meta Programming

Several other PHP extensions may also be of interest for the purpose of meta programming.

The tokenizer as part of the default configuration [25] has already been mentioned in section 3.4 as a foundation of phpCallGraph. It provides among other things the function `token_get_all` which splits a string into an array of parser tokens. It is also used by several other projects, e.g., phpDocumentor, phUML and the upcoming StaticReflection.

The extension Parsekit [26] provides functions to retrieve the opcodes[3] for a given piece of PHP code, generated by the parser.

Another meta programming extension is Parse Tree [27] which generates abstract syntax trees from PHP code. The produced AST is represented in form of an XML document object. This is due to the fact, that this extension is a coproduct of phpAspect, a framework for aspect-oriented programming (AOP) in PHP. The XML representation enables XSLT transformations on source code, which seems to be a powerful concept, given the conciseness and expressiveness of the XSLT sheets used by phpAspect to perform aspect weaving. There are also several other AOP frameworks for PHP, with most of them doing static weaving based on annotations.

[3] Byte codes of the PHP virtual machine

5 Conclusion

PHP provides a reasonable set of meta programming and reflection facilities. The default configuration of PHP includes several functions for introspection and an object-oriented reflection API. Furthermore, basic intercession is possible through so called magic methods and code execution functions. Additional features are provided by extensions to the PHP virtual machine, e.g., the Runkit extension, which introduces a comprehensive set of intercession possibilities.

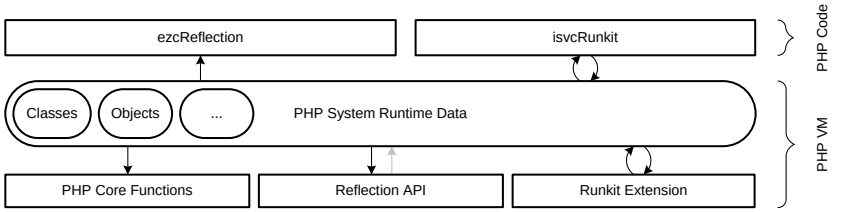

Fig. 8. Complete Overview of PHP's Meta Programming and Reflection Features.

In our earlier work, we developed ezcReflection, an extended version of PHP's reflection API introducing annotations and type introspection. This has now been extended by a delegation mechanism to foster integration of other reflection enhancements or alternative data sources. Such an enhancement is isvcRunkit, an advanced object-oriented API for Runkit, which has also been developed during this work. Our example application phpCrow utilizes isvcRunkit integrated with ezcReflection to demonstrate PHP's instrospection and intercession capabilities. It consists of a GTK-based graphical class browser, which is able to modify its own source code at runtime.

From a design point of view, it is not favorable to have the reflective API divided into procedural and object-oriented parts scattered across multiple VM extensions. That makes it difficult to know all possibilities of the language and especially to find them in the documentation. However, the cleanly separated reflection subsystem containing the object-oriented API has some noticeable advantages and is quite usable.

Since the main use case of PHP is server-side web application development, in which the general execution model is to parse and interpret a PHP script each time a request arrives, extensive intercession is not very practicable when trying to maintain sub-second response times.

Eventually, the reflective capabilities of PHP are not comparable with, e.g., the powerful CLOS Meta Object Protocol. But they are competitive with other popular languages, e.g., Java, and suit the needs for the typical usage scenarios of PHP.

References

1. nexen.net: Evolution de PHP sur Internet (decembre 2007). Documentation (2007) http://www.nexen.net/chiffres_cles/phpversion/17965-evolution_de_php_sur_internet_decembre_2007.php.
2. php.net: Information About the Current Status of PHP. Project site (2008) http://php.net/.
3. PHP Documentation Group: PHP Manual: Strings - Variable parsing. Documentation (2007) http://www.php.net/manual/en/language.types.string.php\#language.types.string.parsing.
4. Knoepfel, A., Groene, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. John Wiley & Sons (2006)
5. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2004) 331–344
6. symfony Project: symfony: Open-Source PHP Web Framework. Project site (2007) http://www.symfony-project.org/.
7. torpedeo Project: torpedeo: Lightweight object relational mapper for PHP. Project site (2007) http://code.google.com/p/torpedeo/.
8. Stubbles Project: Stubbles: A PHP 5 framework. Project site (2007) http://www.stubbles.net/.
9. PicoContainer Project: PicoContainer for PHP. Project site (2006) http://www.picocontainer.org/project.html.
10. Seasar Foundation: S2AOP. Project site (2007) http://www.seasar.org/en/php5/aop.html.
11. Phing Project: Phing: PHing Is Not GNU make. Project site (2007) http://phing.info/.
12. Zend Technologies: Zend Framework. Project site (2007) http://framework.zend.com/.
13. jool.nl: Webservice Helper. Project site (2007) http://www.jool.nl/new/index.php?file_id=1.
14. InstantSVC Group: InstantSVC – The Web Service Builder (2007) http://instantsvc.sf.net/.
15. PHPUnit Project: PHPUnit. Project site (2007) http://phpunit.de/.
16. Marr, S., Menge, F., Gabrysiak, G., Sprengel, M., Perscheid, M., Hartmann, C., Meyer, A., Böttner, S.: Web Service Toolkit für PHP5 (2006) http://instantsvc.sf.net/docs/web-service-toolkit-fuer-php5.pdf.
17. eZ Components Project: eZ Components: an enterprise ready general purpose PHP components library (2007) http://ezcomponents.org/.
18. XDoclet Group: XDoclet: Attribute-Oriented Programming for Java. Documentation (2007) http://xdoclet.sourceforge.net/.
19. phpDocumentor: phpDocumentor: The complete documentation solution for PHP. Open source project (2007) http://phpdoc.org.
20. Menge, F.: phpCallGraph: A call graph generator for PHP. Project site (2008) http://phpcallgraph.sourceforge.net/.
21. Westhoff, J.: phUML: A fully automatic UML class diagramm generator written in PHP. Project site (2007) http://westhoffswelt.de/projects/phuml.html.

22. The PHP Group: PHP-GTK. Documentation (2007)
http://gtk.php.net/.
23. The PEAR Group: PEAR - PHP Extension and Application Repository. Project site (2008)
http://pear.php.net/.
24. Golemon, S.: Runkit. PHP Extension (2008)
http://pecl.php.net/package/runkit.
25. PHP Documentation Group: PHP Manual: Tokenizer Functions. Technical report (2008)
http://www.php.net/manual/en/ref.tokenizer.php.
26. Golemon, S.: Parsekit. PHP Extension (2008)
http://pecl.php.net/package/parsekit.
27. Candillon, W.: Parse Tree. PHP Extension (2008)
http://pecl.php.net/package/Parse_Tree.

A Code Example

```
/**
* Characterizes a common dog.
*/
interface iDog {
    public function setName($name);
    public function sit();

    /**
    * @param Postman $postman Dogs love postmans.
    */
    public function bite(Postman $postman);
}

class Postman {
    /**
    * @return string Some news
    */
    public getNewspaper() {
        return "News of today: ...";
    }
}

abstract class Dog implements iDog {

    /**
    * @var string Holds the dogs name.
    */
    protected $name = "unnamed dog";

    /**
    * Set the name of the dog.
    * Should be used only once.
    * @param string $name
    */
    public function setName($name) {
        $this->name = $name;
    }

    final public function sit() {
        echo "$this->name sat down, barking: ";
        echo $this->bark();
    }

    protected function bark() {
        print "WOOF\n";
    }

    public function bite(Postman $postman) {
        print $this->name." bites a Postman\n";
    }
}

class Poodle extends Dog implements iDog {
    private $postmanFromTheHood;

    public function __construct() {
        $this->postmanFromTheHood = new Postman();
    }

    final protected function bark() {
        print "wiif\n";
    }

    public function __call($method, $params) {
        $class = new ReflectionClass($this->postmanFromTheHood);
```

```
        $method = $class->getMethod($method);
        $method->invoke($this->postmanFromTheHood, $params);
    }
}

$p = new Poodle();            // $p is a new poodle
$p->setName("poodle");        // $p is now named
$p->sit();                    // poodle sat down, barking: wiif
$p->getNewspaper();       // well, and it even knows where to get the latest
    news
```

Listing 1.1. A code example, showing the usage of the object model of PHP.

B Additional Model

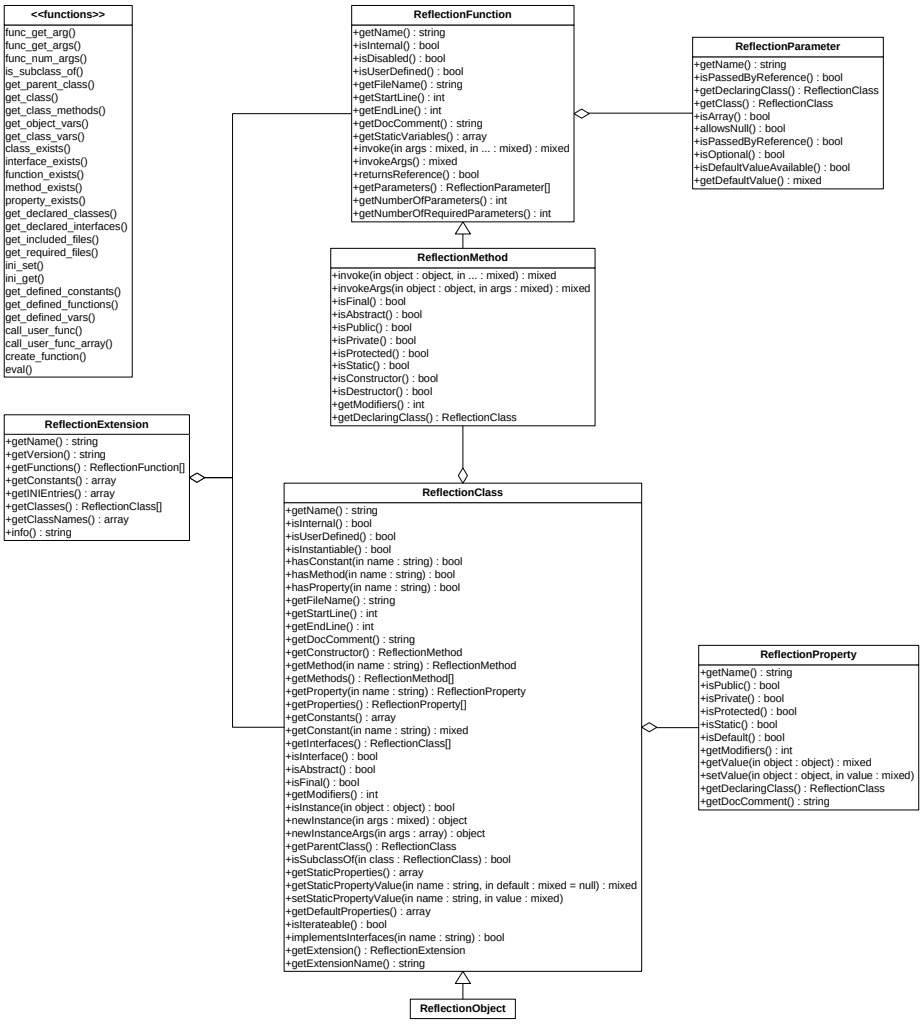

Fig. 9. Complete Overview of PHP's Reflection API