GAP: Generic Aspects for PHP

Sebastian Bergmann
eZ systems AS
Kverndalsgate 8
Postboks 253
N-3701 Skien (Norway)
sb@ez.no

Günter Kniesel
Department of Computer Science III
University of Bonn
Römerstrasse 164
D-53117 Bonn (Germany)
gk@cs.uni-bonn.de

ABSTRACT

In this paper, we explore how aspect-oriented programming can be implemented for the PHP programming language. We start with an overview of existing implementations, identifying their strengths and weaknesses. We then introduce *GAP*, our implementation of aspect-oriented programming for PHP that uses dynamic weaving, supports aspect genericity, and provides a framework to implement custom pointcut languages on top of it. The sum of these features has previously been supported only in experimental research prototypes that have had little impact on commercial software development. In contrast, PHP has a large user community. In the last decade, it has developed from a niche language for adding dynamic functionality to small websites to a powerful tool making strong inroads into large-scale, business-critical Web systems. We expect that GAP will significantly ease development of such systems while promoting a seamless integration of many advanced concepts of aspect-oriented systems: aspect genericity, dynamic weaving, a state-sensitive pointcut language, and extensibility.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Abstract Data Types

General Terms

Design, Languages

Keywords

Generic Aspects, Dynamic Weaving, PHP, Extensible Pointcut Language

1. INTRODUCTION

PHP [17] is a widely-used scripting language. Initially designed for Web programming, it has developed to a general-purpose object-oriented language making strong inroads into large-scale, business-critical Web systems. For instance, financial institutions develop and maintain the BASEL II [18] credit and insurance rating tools using PHP and Yahoo runs all its business on PHP (except the core of the search engine). As of version 5, released July 2004, the PHP language features an object model that is similar to the ones of Java and C# and integrates ideas from other programming languages. The key technical contributor to PHP's success is its simplicity, which translates into shorter development cycles, easier maintenance, and lower training costs. The second one is social – the very large and vibrant community around it, which develops not only PHP itself but also thousands of open source applications that can be used off-the-shelf or as references for new applications.

EWAS 2006 – Third European Workshop on Aspects in Software. August 31, 2006, Enschede, The Netherlands

Given the wide-spread use and impact of PHP on current web-centered software development, the benefits of a well-designed integration of aspect-oriented programming in PHP would be huge. However, existing attempts to support aspect-oriented programming in PHP do not take advantage of the dynamic nature of the language, ignore new aspect-oriented language concepts, such as genericity, or are not compatible with new versions of the language (see Section 2).

In this paper we present an approach that provides dynamicity and compatibility to the official PHP language releases while supporting a powerful aspect language model, including aspect genericity. It is called *GAP*, **G**eneric **A**spects for **P**HP. A predecessor was presented in [1] under the name *AspectPHP*. We have chosen to rename our approach to GAP in order to avoid confusion with the *aspectPHP* project [19] and to emphasize the support for aspect genericity.

2. THE STATE OF AOP FOR PHP

In this section, we give an overview of the different options for implementing an aspect-oriented extension of PHP and review existing implementations.

2.1 Preprocessor

A preprocessor can be used to perform source code transformations and statically weave the aspect code into the base program. The result of this weaving is PHP source code that can then be deployed in a standard PHP environment.

Existing Implementations. *PHPAspect* [20] extends the PHP language with keywords inspired by AspectJ [2]. Figure 1 shows an implementation of the *Singleton* design pattern [3] in PHPAspect. *Aspect-Oriented PHP* [21] is largely similar to *PHPAspect*, the main difference being that the AOPHP compiler is implemented in Java.

PHPAspect provides a compiler, written in PHP, that performs static weaving using source code transformations (see Figure 2). PHPAspect is currently being reimplemented in C, using an XML representation for the abstract syntax trees and using XSLT for the weaving process (see figure 3). With the use of XSLT and XPath the author of *PHPAspect* hopes to achieve independence for the lexical and syntax analysis from the PHP version and more flexibility with regard to the aspect language.

```
<?php aspect Singleton {
  public $instances = array();

  pointcut singleton:new(*(*));

  around singleton {
    $i = $thisJoinPoint->getClassName();

    if (!isset($singleton->instances[$i])) {
      $singleton->instances[$i] = proceed();
    }

    return $singleton->instances[$i];
  }
}
?>
```

Figure 1: Singleton implementation in PHPAspect. PHP identifiers prefixed with a $ sign represent variables. The right arrow -> represents field access or method invocation.

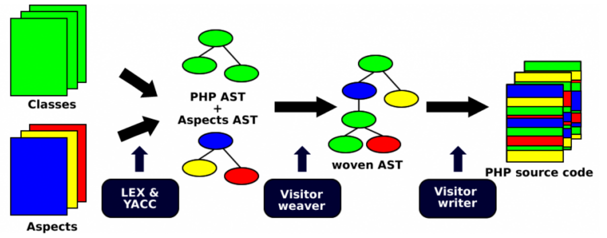

Figure 2: The weaving chain of PHPAspect (from phpaspect.org)

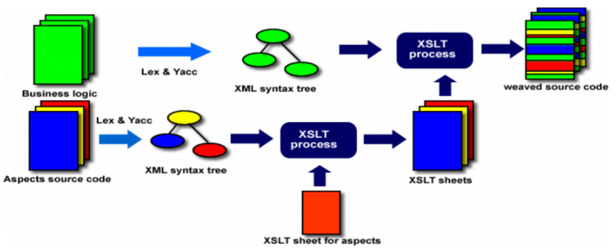

Figure 3: The new weaving chain of PHPAspect (from phpaspect.org)

Evaluation. Since PHP is an interpreted, dynamic language, static weaving comes with both advantages and disadvantages. On the one hand, weaving is performed once before deployment and has no performance impact at run-time. On the other hand, static weaving imposes limits to both join point model and pointcut language with regard to leveraging the dynamic nature of the underlying programming language.

2.2 Aspect-Aware PHP Interpreter

Extensibility is one of the reasons why PHP became the favourite "glue" of the Web. Functionality from existing third-party libraries (database clients or image manipulation toolkits, for instance) can be made available through PHP with the ease of use one expects from a scripting language.

The *Zend Engine*, the compiling and executing core of the PHP Interpreter, can be extended using the C programming language. An extension can be implemented either as a plug-in that can be dynamically loaded or by changing the interpreter's source code.

Evaluation. Source changes enable changing or extending the language syntax by modifying the scanner and parser rules. The resulting language extension can only be distributed in the form of a custom binary or a patch against the original source code of a specific version of PHP. In contrast, a plug-in is developed using the public APIs of the PHP interpreter and can therefore be deployed with different versions of the language. For portability, a plugin-based extension is preferable.

Existing Implementations. The *aspectPHP* prototype [19] is a reimplementation of Aspect-Oriented PHP [21] in C. It is available only as a patch for PHP 4.3.10, tying it closely to this (meanwhile outdated) implementation of the language.

2.3 Meta-Programming

A language extension can be implemented in the PHP programming language itself, using its meta-programming capabilities. These include

- *a Reflection API* [22] for introspecting at runtime classes, methods, etc.
- *interceptor methods* inspired by the `doesNotUnderstand` selector of Smalltalk [4]. In PHP, read and write access to undeclared attributes and calls to undeclared methods of an object are handled by the methods `__get()`, `__set()`, and `__call()`.
- *byte code modification* via the Runkit extension [23].

Existing Implementations. The *AOP Library for PHP* [24] is a PHP library that supports just a very rudimentary join point model and requires extensive manual changes to the base code, failing to support the two main characteristics of aspect-oriented systems: obliviousness and quantification [5].

Figure 4 shows how to declare "aspects" and advice using the AOP Library for PHP such that invocation of `aMethod()` in `AClass()` invokes first the `before` advice code, then the base code of the method, and finally the `after` advice code. The example illustrates that the base code has to be modified extensively for using the aspect. First, the constructor of the base class needs to be changed to accept an `$aspect` object and store it in an instance variable[1]. Then, explicit calls to this object's `_before()` and `_after()` methods have to be inserted to each method of the class. This is not different from any other invocation of library code. Therefore the claim of supporting AOP is hardly justified.

Evaluation. Whereas the only existing attempt to bring aspects to PHP using its metalevel features can hardly be recommended, the metaprogramming approach cannot be dismissed in general. On the contrary, it has the potential to take advantage of PHP's dynamicity and to provide a portable solution, that is compatible with different language versions.

2.4 State of Art Summary

Our short review has shown that from the four attempts to bring aspect-oriented concepts to PHP only three really provide aspect-

[1] `__construct()` is the name of the constructor method in PHP.

```
<?php
require_once 'AClass.php';
require_once 'aop.lib.php';

$aspect   = new Aspect;
$pointCut = $aspect->pointcut('call AClass::aMethod');
$pointCut->_before('... before advice code ...');
$pointCut->_after('... after advice code ...');
$pointCut->destroy();

$object = new AClass($aspect);
?>

<?php class AClass {
    private $aspect;

    public function __construct($aspect) {
        $this->aspect = $aspect;
    }

    public function aMethod() {
        Advice::_before($this->aspect);
        // ... base code ...
        Advice::_after($this->aspect);
    }
} }?>
```

Figure 4: Aspect declaration with the AOP Library for PHP and base class using the aspect. The "aspect" is just a library that is invoked explicitly from the base code.

oriented functionality. Of these, one is hard-coupled to an outdated language version. The other two are based on static weaving, falling short of leveraging on the dynamic language features of PHP in aspects. In addition, they are based on the language model of AspectJ, which does not support some powerful new concepts that are particularly well-suited for the development of large web-based applications: genericity, and a state-aware, extensible pointcut language.

3. GAP: GENERIC ASPECTS FOR PHP

In this section we introduce *GAP*, our implementation of aspect-oriented programming for PHP. GAP supports an extensible pointcut language, aspect genericity, and dynamic weaving. Taking advantage of PHP's meta-programming capabilities, this is achieved without changing the language syntax or interpreter.

3.1 Aspect Basics

In GAP aspects are plain PHP classes that use annotations in comments to declare pointcuts, advice, and inter-type declarations. Advice declarations bind a pointcut expression to an invocation of a plain PHP method that takes a join point object as parameter. The method implements the advice body. It will be executed on all join points matching the pointcut expression.

In GAP, custom pointcuts can be implemented easily based on an open pointcut language framework (see section 4.1). In its standard configuration GAP allows quantifications over three join point types:

- Object initialization
- Field access
- Method call or execution

```
<?php
/* @pointcut allInvocations : method(* *->*(..));
 * @after    allInvocations : Logging->log();
 */
class Logging {
  public function log($joinPoint) {
    printf(
      "%s->%s() called %s->%s()\n",
      $joinPoint->getSource()
                ->getDeclaringClass()
                ->getName(),
      $joinPoint->getSource()
                ->getName(),
      $joinPoint->getTarget()
                ->getDeclaringClass()
                ->getName(),
      $joinPoint->getTarget()
                ->getName()
    );
  }
} ?>
```

Figure 5: GAP aspect that logs all method calls

The currently implemented pointcut syntax is similar to the one of AspectJ (see Figure 1). For instance, the first line of the comment in Figure 5 declares a pointcut that matches all method invocations. The `@pointcut allInvocations` annotation starts the declaration of a pointcut named `allInvocations`. The `method` keyword is the selector for the combined *Method Call* / *Method Execution* join point in GAP's join point model. The pattern `* *->*(..)` matches methods with arbitrary visibility (first star), class name (second star), method name (third star), and parameter list (double dots).

Note that the `allInvocations` pointcut from Figure 5 also matches the execution of the `log()` advice method. However, invoking an advice method for the execution of an advice method is currently disabled in GAP in order to avoid certain sources of aspect interference. Whether this is too restrictive could be a topic of discussion at the workshop.

Join points are represented by `GAP_JoinPoint` instances. For each kind of join point supported by GAP's join point model there is a specific implementation of `GAP_JoinPoint`. For instance, the class `GAP_JoinPoint_MethodCall` implements GAP's combined method call and method execution join point. Its instances include information on the calling object, the calling method, the called object, and the called method. This information is represented by objects from the Reflection API, in this case two instances of the `ReflectionMethod` class, representing the caller and callee, respectively.

Predefined GAP Pointcuts
initialization(*class*(*parameters*))
get(*modifier class*->*attribute*)
set(*modifier class*->*attribute*)
method(*modifier class*->*method*(*parameters*))
source(*modifier class*->*method*(*parameters*))
cflow(*modifier class*->*method*(*parameters*))

Table 1: Implemented pointcuts demonstrating the versatility of GAP's extensible pointcut framework. Italics indicate non-terminals

Through the information provided by the `$joinPoint` an advice method that is invoked for a method execution join point can find out which method called the method associated with the current join point. This information is also used by the additional pointcut expressions supported by GAP's prepackaged pointcut language implementation: source and cflow (see Table 1). The `source()` pointcut expression matches the immediate method that performed an object instantiation, attribute access, or method call. The `cflow()` pointcut expressions matches if the specified method is on the current call stack.

Pointcuts can be associated with the following aspect effects [6]:

- *Advice*: Execution of code before, after or around any of the above-mentioned join points (indicated by the annotations `@before`, `@after`, or `@around`).
- *Declarations*: Addition and change of fields, methods, inheritance relations and interface implementation declarations (indicated by the keyword `@introduce`).
- *Custom Errors*: With the `declare error` or `declare warning` syntax one can customize the response to the occurrence of a join point, as shown in Figure 6.

The second line of the comment in Figure 5 shows a GAP advice. The `@after` annotation binds the method `log` of class `Logging` to the previously defined pointcut `allInvocations`. The overall effect of Figure 5 is the declaration of an aspect named `Logging` that invokes the advice named `log()` after every method call. The join point context is passed to the advice method as an object of the type `GAP_JoinPoint`.

3.2 Aspect Genericity

The implementations of aspect-oriented programming for PHP that we discussed in Section 2 introduce strong dependencies of aspects on base code by requiring aspects to use concrete names of types, classes, methods, and other entities from base programs.

Wildcards, such as `*` and `..` are intended to alleviate this problem but are no real solution since they throw the child out with the bath. Instead of being too specific, they are too general. They match more than intended because it is not possible to express dependencies of the values matched by different wildcards.

As a solution to this problem, generic aspect languages [6] such as LogicAJ [7], Sally [8], Carma [9], and OReA [10] replace wildcards (e.g. `*`) by named logic meta-variables (e.g. `?var`). All occurrences of `?var` in a pointcut expression must match the same value. Further constraints on the legal matches can be expressed by additional predicates that can be used in pointcut definitions. The `source` predicate used in Figure 7 is an example. It does not select join points but only constrains the matches of the `method` predicate.

Figure 7 shows how GAP supports meta-variables in its annotation-based pointcut language. The `localCalls` pointcut captures all method invocations that address methods from the same class. The values of meta-variables that matched during the evaluation of the pointcut expression can be accessed via the `$joinPoint` object's `getMetaVariable` method.

```
<?php
/**
 * @pointcut inFactory : method(Factory->get(..));
 * @pointcut newObject : initialization(Base+(..));
 * @declare error : !inFactory && newObject
 *                : "Factory::get() must be used.";
 */
class Factory {
    public static function get($type)
    {
        return new $type;
    }
}
?>
```

Figure 6: GAP aspect that enforces the use of a Factory method

```
<?php
/* @pointcut localCalls : method(* ?class->*(..))
 *                        && source(* ?class->*(..));
 * @after    localCalls : Logging->log();
 */
class Logging {
  public function log($joinPoint) {
    printf(
      "%s::%s() called %s::%s()\n",
      $joinPoint->getMetaVariable('class'),
      $joinPoint->getSource()->getName(),
      $joinPoint->getMetaVariable('class'),
      $joinPoint->getTarget()->getName()
    );
  }
}
?>
```

Figure 7: GAP aspect using a pointcut with meta-variables to express logging of *local* calls

4. IMPLEMENTATION OF GAP

In this section we show how we implemented GAP in PHP using only its Reflection API, interceptor methods and the Runkit extension.

4.1 Open Architecture

The two main components of the GAP plugin are the `GAP_Weaver` class and the `GAP_Dispatcher` class. They are the core of GAP on top of which the join point model and pointcut language framework are built. The latter provides the building blocks for implementing a pointcut language. Figure 8 shows a subset of these building blocks. Together with the *pointcut registry* they provide a way to capture join points and activate advice code. However, this low-level declaration of join points and advice is not convenient and practical for everyday use. Therefore, these internals of the API are hidden behind the annotation-based pointcut and advice declaration syntax introduced in the previous section. Using the basic classes, other implementors can extend the pointcut language with new built-in pointcut definitions.

4.2 Load-Time Hooks for Dynamic Weaving

The PHP Interpreter uses its *Streams Layer* [25] to load PHP source files. This layer provides a unified approach to the handling of files and sockets. Any stream, once opened, can also have any number of *filters* applied to it, which process data as it is read from or written to the stream.

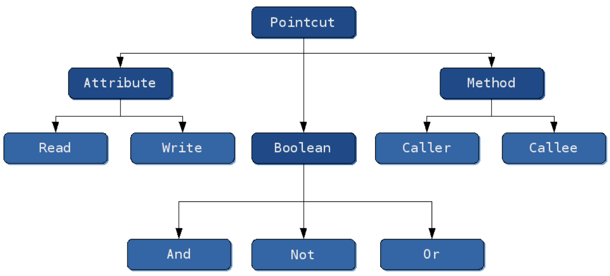

Figure 8: Subset of the GAP Pointcut Framework

Figure 9 illustrates the GAP weaving chain: Using a streams filter written in PHP, the GAP Weaver hooks into the loading of the source code of classes (green) and aspects (blue). The first weaving stage performs source code transformations and passes the modified source code to the compiler integrated in the PHP Interpreter. The second weaving stage operates on the bytecode generated by the compiler and uses the Runkit extension to complete the insertion of generic hooks into the classes. The pointcut expressions of the aspects are evaluated and stored in the *Pointcut Registry* that is used by the *GAP Dispatcher* to capture join point events at run-time and dispatch the appropriate advices. The actual advice execution can be dynamically turned off at runtime.

GAP's annotation-based declaration of pointcuts and advice is implemented as follows:

- When loading the source code for an aspect, the class representing the aspect is searched for annotations, using the Reflection API of PHP. The annotations are parsed and converted into an object representation. For instance, the `@pointcut` annotation, is parsed into a tree of `GAP_Pointcut` objects that is then passed to the Pointcut Registry. Similarly, the associations of pointcuts and advice is parsed and registered.
- When loading the source code for a class, the appropriate inter-type declarations are inserted into the bytecode of the class and the generic hooks for advice execution are inserted into the bytecode of its methods. Byte code manipulation is performed using the Runkit extension of the PHP Interpreter.

In the remainder of this section we explain how the generic hooks for the three different join point types are implemented in detail.

Method Call Hook Each method of the processed class is replaced by a proxy method that calls the `methodCall()` method of the `GAP_Dispatcher` class. This method has access to the original implementation of the proxy method and can execute it between the execution of before- and after-advices. Figure 10 shows the PHP implementation of this scheme.

Attribute Access Hook Each attribute of the class is renamed so that accessing it using the original name triggers a call to the `__get()` (for read access) or `__set()` (for write access) method. Implementations of these interceptor methods are woven into the class, too. They call the `attributeRead()` and `attributeWrite()` methods of the `GAP_Dispatcher` class.

New Object Hook. The weaving of the hook for capturing the *New Object* join point is an exception from the abobe scheme, as it is actually performed on the source code level. It replaces calls to the `new` operator with corresponding calls to the `newObject()` method of the `GAP_Dispatcher` class.

4.3 Run-Time Dispatcher

During program execution, the previously introduced hooks for the join points check whether or not an the pointcut registry contains a pointcut that matches the current join point. If that is the case, the `GAP_Dispatcher` class handles the execution of the corresponding advice and passes the current context in the form of an `GAP_JoinPoint` object to the advice method.

4.4 Evaluation

The flexibility of dynamic weaving comes at the price of a possible performance hit. For every execution, for instance, a request to a PHP-driven website, the aspect code has to be woven into the base program. During the execution of the program there are two additional method calls for every attribute access, method call (see figure 11), and execution of the `new` operator. Run-time measurements and performance comparisons with other implementation schemes are still to be carried out.

5. RELATED WORK

This paper presented GAP, an extension of the PHP programming language with aspect-oriented programming concepts. Compared to other implementations of aspect-oriented programming for PHP (see Section 2) GAP stands out as being the first implementation that supports dynamic weaving, genericity, and an extensible pointcut language.

Compared to non-PHP aspect languages and systems, GAP provides a specific mix of partly well-known concepts and techniques. GAP's design balances the expressive power of generic aspect languages such as LogicAJ [7], Sally [8], Carma [9] and OREA [10] with the simplicity of the design of Classpects [11]. With the formers it shares the concept of logic meta-variables. With the latter it shares the reliance on plain base level methods as the body of advice.

The open architecture of GAP allows the implementation of customized pointcut languages on top of a common kernel that handles the weaving and dispatching of aspect code. In this respect, GAP has similarities with various other systems built on reflection and to open and extensible systems such as Josh [12] and LogicAJ 2 [13].

Implementation-wise, GAP's dispatch mechanism is related to the method wrappers of Brant, Foote, Johnson and Roberts, which are a standard mechanism in Smalltalk implementations and the basis of aspect-oriented extensions of Smalltalk, such as AspectS [14]. Load-time weaving of byte code is based on the filter concept of PHP that provides the generic class file interception functionality that has only recently been integrated into Java 5 (see `java.lang.instrument`) and has previously required specific solutions such as JMangler [15]. Use of load time weaving as a way to implement generic hooks that enable run-time weaving has been pioneered in JAC [16]. The pointcut registry and dispatcher mechanism are recurring themes in various dynamic weaving systems and theoretical models of aspect orientation.

6. CONCLUSIONS AND FUTURE WORK

This paper presented GAP, an extension of the PHP programming language with aspect-oriented programming concepts. Compared

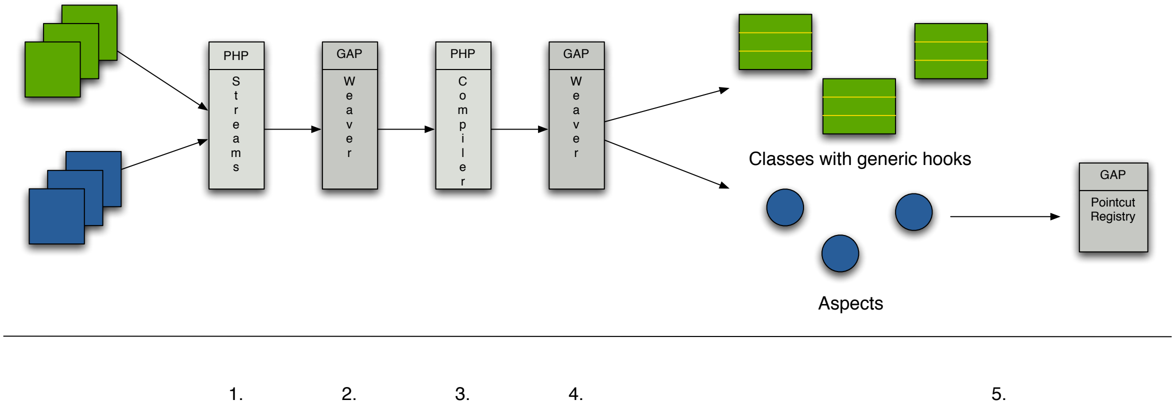

Figure 9: The weaving chain of GAP

```
protected static function weaveMethodJoinPoint(
    ReflectionMethod $method
) {
  runkit_method_rename(
    $method->getDeclaringClass()->getName(),
    $method->getName(),
    '__GAP_' . $method->getName()
  );

  runkit_method_add(
    $method->getDeclaringClass()->getName(),
    $method->getName(),
    self::generateMethodParameters($method),
    'return GAP_Dispatcher::getInstance()->
     methodCall('.
    '  new GAP_JoinPoint_Method'.
    ');'
  );
}
```

Figure 10: The GAP_Weaver::weaveMethodJoinPoint() method

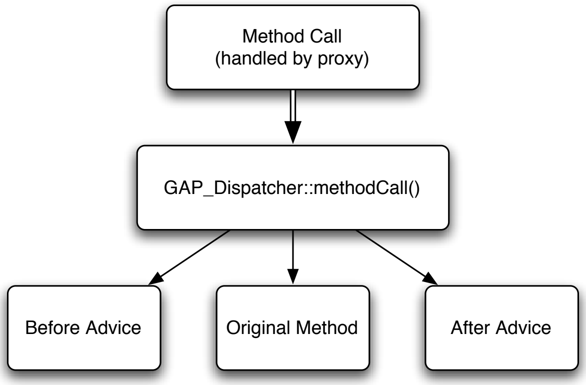

Figure 11: Dispatching Method Call Advices

to other implementations of aspect-oriented programming for PHP GAP stands out as being the first implementation that supports dynamic weaving, genericity, and an extensible pointcut language.

Compared to other aspect languages and systems, GAP provides a specific mix of partly well-known concepts and techniques. Its specific power is the orthogonal integration of all these features into a widely-used programming language. We hope that the use of GAP as a powerful, dynamic extension to PHP will foster the adoption of aspect-oriented technologies in a community that is not using the research prototypes that have first demonstrated some of the more advanced techniques (e.g. generic aspects).

As a long-time contributor to PHP the first author had early access to new Runkit features that are still to be made available publicly. Therefore, we will wait with the first public release of GAP until they are available. We will use this time to further improve GAP's pointcut language and performance.

The dynamic interpretation of the pointcut expressions at runtime allows for state-sensitive pointcut languages. Possible applications of state awareness includes the selective execution of advice depending on information about the user that requests a website document, his domain name or web browser, for instance. We are still thinking about an elegant syntax to integrate support for state sensitivity to our annotation-based pointcut language.

Further benchmarking will have to show whether or not the deployment of GAP in the web-server environment that is usually associated with PHP is practical in spite of the performance hit incurred by the flexibility of dynamic weaving. The `GAP_Dispatcher` class could be implemented in C as an extension to the PHP Interpreter to improve the run-time performance.

Our incentive to develop GAP, however, was not primarily in using it in a web-server environment but rather in the development of tools such as *PHPUnit* [27]. GAP could be used, for instance, to implement Mock Objects through an aspect that implements class posing (following [6][Figure 6]) or to enforce design constraints at development time as illustrated in Figure 6. Another usage scenario for GAP lies within the PHP-GTK [28] environment, which provides an object-oriented interface to GTK+ [29] classes and functions and facilitates the writing of client-side cross-platform GUI applications using PHP.

7. ACKNOWLEDGEMENTS

Sebastian Bergmann, who is a long-time contributor to the PHP project, would like to thank his peers from the PHP community in general and Sara Golemon, the author of the Runkit extension, in particular. He is also thankful for the support and encouragement from Prof. Dr. Armin B. Cremers, head of the Computer Science Department III at the University of Bonn.

8. REFERENCES

[1] Sebastian Bergmann. *AspectPHP: An Aspect-Oriented Programming Extension for the PHP Programming Language*, Poster, In: Student Research Extravaganza. Poster Session of the Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), 2006, Bonn.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. *An Overview of AspectJ*, In: Lecture Notes in Computer Science, volume 2072, pages 327-355, 2001.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[4] Mario Wolczko. *Semantics of Smalltalk-80*, In: Proceedings of the 1987 European Conference on Object-Oriented Programming, J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, Lecture Notes in Computer Science, volume 276, pages 108-120. Springer-Verlag, Paris, June 1987.

[5] Robert E. Filman and Daniel P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*, In: Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.

[6] Günter Kniesel, Tobias Rho. *A Definition, Overview and Taxonomy of Generic Aspect Languages*, L'Objet, vol. 11, 3, pp. , to appear, Hermes Science, London.

[7] Tobias Windeln. *LogicAJ – Eine Erweiterung von AspectJ um logische Meta-Programmierung* (in German), Diploma Thesis, CS Dept. III, University of Bonn, Germany, August 2003.

[8] Stefan Hanenberg, Rainer Unland. *Parametric Introductions*, In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD.03), 2003, Boston, MA.

[9] Kris Gybels, Johan Brichau. *Arranging Language Features for More Robust Pattern-Based Crosscuts*, In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD.03), 2003, Boston, MA.

[10] Maja D'Hondt. *Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality*, Ph.D. Thesis, Vrije Universiteit Brussel, May 2004.

[11] Hridesh Rajan, Kevin J. Sullivan.*Classpects: Unifying Aspect- and Object-Oriented Language Design*, In: Proceedings of the 27th International Conference on Software Engineering, 2005, St. Louis, MO.

[12] Shigeru Chiba, Kiyoshi Nakagawa. *Josh: An Open AspectJ-like Language*, In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD.04), 2004, Lancaster, UK.

[13] Tobias Rho, Günter Kniesel, Malte Appeltauer. *Fine-Grained Generic Aspects*, In: Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), in conjunction with the Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), 2006, Bonn.

[14] Robert Hirschfeld. *AspectS: AOP with Squeak*, In: OOPSLA'01 Workshop on Advanced Separation of Concerns, 2001, Tampa, FL.

[15] Günter Kniesel, Pascal Costanza, Michael Austermann. *JMangler - A Powerful Back-End for Aspect-Oriented Programming*, In: Aspect-Oriented Software Development, Prentice Hall, 2004.

[16] R. Pawlak, L. Duchien, L. Seinturier, G. Florin, F. Legond, L. Martelli. *JAC: A Framework for Separation of Concerns and Distribution*, In: Aspect-Oriented Software Development, Prentice Hall, 2004.

[17] http://www.php.net/

[18] http://en.wikipedia.org/wiki/Basel_ii

[19] http://www.cs.toronto.edu/∼yijun/aspectPHP/

[20] http://www.phpaspect.org/

[21] http://www.aophp.net/

[22] http://www.php.net/manual/en/language.oop5.reflection.php

[23] http://www.php.net/runkit

[24] http://www.phpclasses.org/aopinphp

[25] http://www.php.net/manual/en/streams.php

[26] http://www.php.net/debug_backtrace

[27] http://www.phpunit.de/

[28] http://gtk.php.net/

[29] http://www.gtk.org/