Aspect-Oriented Programming in PHP

M.F.A. Pouw  
Liacs  
University of Leiden  
E-mail: mpouw@liacs.nl

Abstract

Aspect Oriented Programming is a new widely used programming paradigm. This paradigm enables the capability to efficiently implementing crosscutting concerns in new and also existing programs without changing the original code. Unfortunately this is only possible in compiled programming languages. Scripting languages for example don’t have the possibility to weave advices into these concerns before compilation. To enable this paradigm in the scripting language PHP a new approach is taken by weaving these advices into the concerns in runtime. This results in a fast and easy to use solution for Aspect Oriented Programming in PHP.

1 Introduction

After Sequential Programming and Object-Oriented programming a new programming paradigm hype arose between existing software programming views; Aspect-Oriented Programming. As Sequential and Object-Oriented programming languages can be both compiled and interpreted, current implementations of Aspect-Oriented programming are based on Weaving before compilation and thus cannot be applied on interpreted languages. Because Aspect-Oriented Programming offers great advantages with respect to Cross-Cutting concerns, the need for a good solution for Aspect-Oriented Programming in interpreted languages is very welcome.

2 Background

There are thousands of programming languages and new languages are developed every year [6]. In order to distinct all these languages they can
be grouped by several properties. These properties can be *Model of execution* like *Compilation*, *Interpretation* and *Just-in-time Compilation* or a programming languages can have distinct *Programming Paradigms*. The differences between *Java* and *PHP* are clear after discussing these properties, but because the research is focused on *Java* and *PHP*, these languages are also shortly mentioned.

2.1 Execution models

A computer can only execute the machine language that is written with a set of macro instruction, these macro instructions are actually implemented with lower-level microinstructions. Because no modern programming languages is written in this machine language, top-level languages first must be translated to theses lower-level instruction. This cannot be done in a single layer, but this requires an operating system with higher-level primitives than those of the machine languages [3]. The implementation of this translation and therefore the execution of the programming code can be done with several models.

2.1.1 Compiled languages

The first and most used model is used by *Compiled languages*, the programming code is directly translated into the machine language. This translating / compiling is done by a compiler in several steps; first the programming code is analyzed by the *Lexical analyzer* to retrieve only the useful lexical units like identifiers, operators and other special characters from the programming code. Then the *Syntax analyzer* constructs a *parse tree* from these lexical units to represent the syntactic structure of the program. Before translating the code into machine language the program is translated into an *intermediate code* to detect errors and to optimize the program. Finally the *code generator* translates this *intermediate code* into the machine language. All the needed information about the program variables and constants during the compilation are stored in the *symbol table*. Before the machine language can be executed on the hardware it mostly also requires programs from the operating system, the process of collecting these programs and linking them together is done by a program called a *linker*. The great advantage of compilation is that the resulting program can be executed very fast.
2.1.2 Interpreted languages

A totally different model is used by Interpreted languages. In this model the programming code is not translated but interpreted by another program. This interpreter simulates the execution with a set of high-level program instructions like a virtual machine for the interpreted language. The disadvantage of interpretation is that because all the decoding of the higher-level instruction and the availability of the symbol table during interpretation the execution time is much higher than compiled programs and more memory is required. A great advantage is that error messages can refer to source-level units instead of intermediate code units. So many source-level debugging operations are easily implemented and error messages can indicate the source line and even the unit where an error occurred.

2.1.3 JIT languages

A compromise between the mentioned two models is used by Just-in-time languages. These languages are partially compiled like Compiled Languages via the Lexical analyzer and the Syntax analyzer. After that the intermediate code is not translated into machine language, but the intermediate code is interpreted by an interpreter / virtual machine. The advantage of this model is that the compilation part can detect errors and optimizes the programming code. The interpreter simply executes / simulates the intermediate code. Because the intermediate code is defined by an instruction set on a lower level than interpreted instruction set, JIT Languages are executed / simulated faster than Interpreted languages but still slower than Compiled languages.

2.2 Programming paradigms

Different programming languages use different programming paradigms. These programming paradigms, like different ways of software engineering, are grouped in programming methodologies. Examples of paradigms are imperative, functional, logical, object oriented and also Aspect Oriented programming. Not every programming language is limited to a single paradigm, most languages support multiple paradigms and most paradigms can be used in more than one programming language. A programmer can write a the purely procedural paradigm based program in C++ or write a purely object-oriented program in Java. Of course a program can also contain elements of both paradigms.
2.2.1 Imperative Programming

This is the oldest programming paradigm and it is closely related to the machine language of a computer. Different statements are written in a sequence to change the state of the program. This looks much like where in natural languages commands are expressed to take actions. Procedural programming can be seen as an synonym of Imperative programming because only procedure calls are added to this paradigm. These procedures can be routines, subroutines, methods or functions that contain a serie of statements that are executed.

2.2.2 Object-Oriented Programming

One of the most popular programming paradigms today is Object-Oriented programming. Writing sequential lines of code is reduced to defining classes. These classes are compiled into objects that can perform tasks by calling them via messages. Objects are encapsulations of data and procedures. An object addresses the data as its attributes and the procedures are called methods. From a single class, multiple different objects can be created. Classes can be hierarchically structured. Subclasses automatically inherit all attributes and methods of their parent class.

2.3 Programming languages

Different programming languages use different execution models. These languages also support different programming paradigms. Most type of programming paradigms work well with all execution models. A widely used compiled language is Java, this used to be a JIT language, but now only Java Applets use byte code. This byte code is the partially compiled intermediate code that is interpreted by an interpreter / virtual machine. PHP is a scripting language, or interpreted language. Such languages are fully interpreted by an interpreter. Both languages support strict Object Oriented programming, but allow imperative programming on a lower level.

2.3.1 Java

Java is originally based on C++ but is designed to be smaller, simpler and more reliable. When in 1993 the World Wide Web was more widely used, Java was found to be a useful tool for Web programming. [3] All programs in Java are Object Oriented. No functions or methods can be called without encapsulating it in an object or class. Despite Java doesn’t use byte code
anymore, it does require a Java Virtual Machine to run onto. Because of this Java is platform independent, just like PHP.

2.3.2 PHP

PHP (recursive acronym for "PHP: Hypertext Preprocessor" is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. [8] Because PHP is a scripting language, the written code isn’t compiled, but interpreted. From version 5.0 PHP fully supports the Object Oriented Programming paradigm. In this and earlier versions the Imperative programming paradigm is used.

3 Aspect-Oriented Programming

The Object Oriented programming paradigm tries to reduce duplicate code and to make maintenance easier. This is done with separation of concerns by breaking down the program into distinctive parts. These parts have to overlap in functionality as little as possible. This is achieved for single concerns, but unfortunately not for crosscutting concerns. Crosscutting concerns are aspects of a program that overlap multiple concerns. For example when a program has several aspects to interact with, like performance, logging and security.

When each aspect is separately implemented, the resulting code is too complicated, this is called code tangling. Another problem occurs when concerns overlap distinctive parts of the program. This Code scattering results in duplicate code. To solve this the Aspect-Oriented Programming paradigm is developed. This paradigm can be used in combination with other paradigms. Thus combined with the profits of object oriented programming, it is able to handle single concerns as well as crosscutting concerns. The result is that the complexity of the program decreases and maintenance, testability and code reuse increases.

Aspect-Oriented Programming is based on the following three concepts: JoinPoints, Advices and PointCuts. Several aspects that are commonly implemented with Aspect-Oriented programming are error checking and handling, synchronization, monitoring and logging and debugging.
3.1 JoinPoints
First we have to define were the crosscutting concerns occur in the program. This can be any point in the program during execution. This can be
- Method call and execution
- Constructor call and execution
- Initialization of classes and objects

3.2 Advices
An advice is the piece of code that is executed at a certain JoinPoint. This code can be executed before, after or around the JoinPoint.

Around
The before and after advice executes the code before or after the JoinPoint. The around advice is a bit more complex, because code can be executed before the JoinPoint is executed, but the result of the executed code of the JoinPoint determines the effect of the advice that is executed after the JoinPoint.

3.3 PointCuts
A collection of JoinPoints is called a PointCut. This is used to define the set of concerns and what Advice is executed on these JoinPoints.

4 Aspect-Oriented Implementations
With these concepts a particular crosscutting concern is easily defined by a PointCut that encapsulates all JoinPoints that have something to do with the specific concern. After this the necessary code is written to define the concern in an Advice. The concern is than enabled by combining the PointCut with this Advice. Because the JoinPoints, PointCuts and Advices are defined separately of the program, these are easily to maintain. The JoinPoints can be in every part of the program (crosscutting) and can be added very easy. All JoinPoints in a PointCut use the same code that takes care of the concern because they share the same Advice. If the code for a concern is changed, you only change the code of the Advice that belongs to this concern. The original code has not to be modified.
4.1 Weaving

The workhorse of Aspect-Oriented Programming is Weaving. This combines the original code with the separately defined Advices. This weaving is done before the program is compiled. During the weaving the code of the Advices is added to the JoinPoints of the corresponding PointCuts. After this is done, the new combined code is actually compiled. Notice this is only possible when the execution model of the programming language belongs to the set of compiled languages.

4.2 Java implementation

There are a lot of Aspect-Oriented Programming implementations for Java. Like JBoss-AOP and Glassbox, but all are based on, or already merged with AspectJ. The most complete and widely used implementation for Java. Currently there is also a fully supported AspectJ development tool for the Eclipse environment.

4.3 Other implementations

Because of the popularity of this programming paradigm a lot of other implementation for several programming languages are developed. Like .net, C and C++. All of these languages are compiled languages. This makes it possible to weave the advices into the code before compiling the program. This isn’t possible for interpreted languages because the code isn’t compiled but directly interpreted and executed.

4.4 PHP implementations

To solve this problem of weaving before executing the program, some implementations, of the Aspect-Oriented Programming paradigm in PHP, use another script to read the file that is interpreted first, then all Advices are added on the designated locations in the code. After this, the newly created file is interpreted. This requires another script, time to read the original file, a lot of find and replace operations and finally interpreting a non-optimized new file. This can really slow down the execution time of the PHP code, especially with large programs with a lot of Aspects. Another common used solution is to add some libraries to the PHP interpreter. These libraries then take care of the weaving before executing the file. This solution doesn’t remarkably slow down the interpretation and execution process, but needs some extra skills to create the library. Furthermore the PHP installation on
the server is modified to enable these libraries. That can result in less stable services.

4.5 MFAOPHP implementation

A different approach to implement Aspect-Oriented Programming within PHP is not to weave the Aspects before interpretation, but to weave the Aspects by changing the interpreted code by reflection during interpretation. Because you don’t need to modify the service and there is no extra overhead for reading and changing the programming code in a source file, this method probably is more stable and less time and memory consuming than the existing methods. Unfortunately there are no scientific measurements about the performance of MFAOPHP and other solutions.

4.5.1 Problems

How is this runtime weaving done? From PHP5 the PHP language supports all aspects of Object-Oriented Programming including reflection. With this addition it is possible to get information from any self defined object. You can, for example, list the number of attributes of this object, or the methods with their parameters and the properties of these parameters. If these methods are known and we can develop some object that contains one of these methods as a JoinPoint and we develop an object to store these JoinPoints as a set into a PointCut. Then we only need an Advice and add this to the PointCut. The problem is to add the extra code from the Advice to the methods of the object defined as JoinPoints in the PointCut.

4.5.2 Solutions

A lot of packages are written for PHP. Packages are small addition for PHP that can be added without changing or re-installing PHP services. A package can simply be installed by adding the package in the PHP configuration file. The PHP Extension Community Library collects a lot of these packages and makes sure that the library only contains stable and well designed extensions. One of these extensions is the Classkit package. This package allows a running script to add, remove, rename, and redefine class methods without reloading [9]. The first versions of MFAOPHP used Classkit, but since 2004 this extension isn’t developed further. The current version of MFAOPHP therefore uses Runkit, this package is fully backwards compatible with Classkit and contains additional functionality like replace, rename,
and remove user defined functions and classes. And you can also define
customized superglobal variables for general purpose use [10].

4.5.3 Results

Combining the theory of Aspect-Oriented Programming, self reflection and
the Runkit extension results into MFAOPHP. Which exists of three simple
classes to enable Aspect-Oriented Programming in PHP.

The JoinPoint Class

First we have the JoinPoint class. This class simple has a constructor to
create a new JoinPoint that contains the class and the method of the Join-
Point. Only methods defined by the user can be uses as a JoinPoint. This
is checked in the constructor.

The PointCut Class

To create a set of the different JoinPoints, objects of the JoinPoint type
can be added to a PointCut object. The addJoinPoint method can not only
add single JoinPoints, but also selections of JoinPoints. For example:

- All methods of the given Class
- All private methods of the given Class
- All public methods of the given Class
- All protected methods of the given Class

When a selection is added, all methods are checked if it belongs to this se-
lection and if it is user defined.

The Aspect Class

When all JoinPoints are collected into a PointCut the Aspect can be cre-
ated with the Aspect Class. The constructor requires a PointCut, and Ad-
vise and the code that should be added. The Advice can be Before, After,
Around and Throws. This last Advice is not implemented yet, but when the
JoinPoint throws an exception, the code of this Aspect should be executed
(interpreted).

4.5.4 Examples

Aspect Oriented Programming can easily be used for securing or logging
objects in your program. Simply add Advices to important methods and
you can apply a certain Aspect in a uniform way without changing your program. As an example I created a simple class in PHP. In the example I use $MethodName, this variable is automatically added to each method that is changed by an Aspect. $MethodName can be used to see which method is responsible for executing an Advice.

class Example
{
    function Foo()
    {
        echo "Inside foo\n";
    }
    function Bar()
    {
        echo "Inside bar\n";
        return "Return value of bar";
    }
}

// Create a new Example object
$example = new Example();

If we run this without the use of Aspect Oriented Programming the next result can be achieved.

// Show objects without aspects
echo $example->Foo();
echo $example->Bar();

Output:

Inside foo
Inside bar
Return value of bar

Now we add Advices to the different methods of our example class.

// Create a new PointCut
$pointCut = new PointCut();

// Add JoinPoints to the PointCut
$pointCut->addJoinPoint('Example', 'Foo');
$pointCut->addJoinPoint('Example', 'Bar');
// Shorter way:
// $pointCut->addJoinPoint('Example', AllMethods);
// Also AllPrivate, AllPublic or AllProtected can be used.

// Add different types of aspects
$test1 = new Aspect($pointCut, before, 'echo "Before $MethodName";');
$test2 = new Aspect($pointCut, after, 'echo "After $MethodName";');
$test3 = new Aspect($pointCut, around,
  ' $Return = "New return value of $MethodName";');

The next output shows the power of Aspect Oriented Programming in PHP.
We added code into several methods of our own class without changing these
classes themselves!

// Show objects with aspects
echo $example->Foo();
echo $example->Bar();

Output:

Before foo
Inside foo
After foo
New return value of foo
Before bar
Inside bar
After bar
New return value of bar

4.5.5 Discussion
Via the forum on the MFAOPHP website some comments where post [11]:

Constructors can’t return anything
Well, they can but it has no effect. In the JoinPoint constructor the return
true should be removed and the return false replaced with an exception.

Not using defined constants
You’ve declared constants at the top of Aspect.php (should be upper case)
but later in the class you’ve used the numeric literals rather than the con-
stants.

**Formatting and comments**
Not a huge issue but the formatting and commenting isn’t in line with the PEAR guidelines. I only mention this because this is something which could viably be added to the repository. Also, those un-indent, one line com-
ments peppered throughout the code are quite distracting IMHO.

5 **Future Development**
Take into respect the discussion points and add the Throws advice.

6 **Special thanks**
Special thanks to Gregor Kiczales who helped me with the correct definitions and implementing MFAOPHP.

7 **Nice to know**
- MFAOPHP is downloaded over 200 times last year.
- MFAOPHP is translated into English, Chinese and Russian
- MFAOPHP is mentioned in a Software Engineering course of the Uni-
versity of Mannheim (Germany)
- MFAOPHP is mentioned on
  - http://www.aosd.net
  - http://www.phpbuilder.com
  - http://www.cmsdevelopment.com
References

[1] Gertjan Laan (1999), En dan is er... Java, Academic Service


[6] Wikipedia (December 28, 2006), Comparison of programming languages,


[9] The Classkit package
   http://pecl.php.net/package/classkit (January 8, 2008),

[10] The Runkit package
    http://pecl.php.net/package/runkit (January 8, 2008),

    http://www.mfaop.com (January 8, 2008),
A JoinPoint.php

// JoinPoint Class
class JoinPoint {

// Private variables
// Class of this JoinPoint
private $joinClass = '';
// Method of this JoinPoint
private $joinMethod = '';

// JoinPoint Constructor
function __construct ($joinClass, $joinMethod) {
    // Verify the joinClass and joinMethod do exist
    if ($this->verifyJoinPoint($joinClass, $joinMethod)) {
        // If so, set the private variables
        $this->joinClass = $joinClass;
        $this->joinMethod = $joinMethod;
        return true;
    }
    // If not, return false
    else {
        return false;
    }
}

// Verify the joinClass and joinMethod do exist
public function verifyJoinPoint ($joinClass, $joinMethod) {
    // Verify the joinClass with this joinMethod do exist
    if (is_callable(array($joinClass, $joinMethod))) {
        $reflectionClass = new ReflectionClass($joinClass);
        $reflectionMethod = new ReflectionMethod($joinClass, $joinMethod);
        // Verify the joinClass and JoinMethod are user defined
        if ($reflectionClass->isUserDefined() &&
            $reflectionMethod->isUserDefined()) {
            // If so, return true
            return true;
        }
    }
}
else
{
    // If not, return false
    return false;
}

// Set private joinClass method
public function setJoinClass ($joinClass)
{
    // Verify the joinClass with this joinMethod do exist
    if ($this->verifyJoinPoint($joinClass, $this->joinMethod))
    {
        // If so, set the private joinClass variable
        $this->joinClass = $joinClass;
        return true;
    }
    else
    {
        // If not, return false
        return false;
    }
}

// Set private joinMethod method
public function setJoinMethod ($joinMethod)
{
    // Verify the joinMethod in this joinClass do exist
    if ($this->verifyJoinPoint($this->joinClass, $joinMethod))
    {
        // If so, set the private joinMethod variable
        $this->joinMethod = $joinMethod;
        return true;
    }
    else
    {
        // If not, return false
        return false;
    }
}
// Get private joinClass method
public function getJoinClass ()
{
    // Return private joinClass variable
    return $this->joinClass;
}

// Get private joinMethod method
public function getJoinMethod ()
{
    // Return private joinMethod variable
    return $this->joinMethod;
}

B PointCut.php

// Define standard methods to add as JoinPoint
define("userdefinedmethod", 0, true);
define("allmethods", 1, true);
define("allprivate", 2, true);
define("allpublic", 3, true);
define("allprotected", 4, true);

// PointCut Class
class PointCut {

    // Private variables
    // JoinPoints in this PointCut
    private $joinPoints = array();
    // Number of JoinPoints
    private $numberOfJoinPoints = 0;
    // Selected JoinPoint
    private $selectedJoinPoint = 0;

    // Get first JoinPoint method
    public function getFirstJoinPoint ()
    {
        // Verify there is at least one JoinPoint
        if ($this->numberOfJoinPoints > 0)
        {
            // If so, select the first JoinPoint

16
$this->selectedJoinPoint = 0;
// Return the first JoinPoint
    return $this->joinPoints[$this->selectedJoinPoint];
else
{
// If not, return false
    return false;
}

// Get last JoinPoint method
public function getLastJoinPoint ()
{
// Verify there is at least one JoinPoint
    if ($this->numberOfJoinPoints > 0)
    {
// If so, select the last JoinPoint
        $this->selectedJoinPoint = $this->numberOfJoinPoints - 1;
// Return the last JoinPoint
        return $this->joinPoints[$this->selectedJoinPoint];
    } else
    {
// If not, return false
        return false;
    }

// Get selected JoinPoint method
public function getJoinPoint ()
{
// Verify the selected JoinPoint is a JoinPoint
    if ($this->selectedJoinPoint < $this->numberOfJoinPoints &&
        $this->selectedJoinPoint > -1)
    {
// If so, return the selected JoinPoint
        return $this->joinPoints[$this->selectedJoinPoint];
    } else
    {
// If not, return false
        return false;
    }
// Get next JoinPoint method
public function getNextJoinPoint ()
{
    // Verify there is a next JoinPoint
    if ($this->selectedJoinPoint < $this->numberOfJoinPoints - 1)
    {
        // If so, select the next JoinPoint
        $this->selectedJoinPoint += 1;
        // Return the next JoinPoint
        return $this->joinPoints[$this->selectedJoinPoint];
    } else
    // If not, return false
    {
        return false;
    }
}

// Get Previous JointPoint method
public function getPreviousJoinPoint ()
{
    // Verify there is a previous JoinPoint
    if ($this->selectedJoinPoint > 0)
    {
        // If so, select the previous JoinPoint
        $this->selectedJoinPoint -= 1;
        // Return the previous JoinPoint
        return $this->joinPoints[$this->selectedJoinPoint];
    } else
    { // If not, return false
        return false;
    }
}

// Add a new JoinPoint to this PointCut
public function addJoinPoint ($joinClass, $joinMethod)
{
    // Select if JoinMethod is UserDefinedMethod (Default),
    // AllMethods, AllPrivate, AllPublic or AllProtected
    switch ($joinMethod)
    {
        // AllMethods: Add all methods of this Class

case 1:
// Get all methods by reflection
$reflectionClass = new ReflectionClass($joinClass);
$classMethods = $reflectionClass->getMethods();
foreach ($classMethods as $classMethod) {
    // Add this class with each method as a new JoinPoint
    $this->joinPoints[$this->numberOfJoinPoints] =
        new JoinPoint ($joinClass, $classMethod->getName());
    // Increase the number of JoinPoints by one
    $this->numberOfJoinPoints += 1;
}
break;
// AllPrivate: Add all private methods of this Class
case 2:
// Get all methods by reflection
$reflectionClass = new ReflectionClass($joinClass);
$classMethods = $reflectionClass->getMethods();
foreach ($classMethods as $classMethod) {
    // Verify each method is private
    if ($classMethod->isPrivate()) {
        // If so, add this class with this method as a new JoinPoint
        $this->joinPoints[$this->numberOfJoinPoints] =
            new JoinPoint ($joinClass, $classMethod->getName());
        // Increase the number of JoinPoints by one
        $this->numberOfJoinPoints += 1;
    }
}
break;
// AllPublic: Add all public methods of this Class
case 3:
// Get all methods by reflection
$reflectionClass = new ReflectionClass($joinClass);
$classMethods = $reflectionClass->getMethods();
foreach ($classMethods as $classMethod) {
    // Verify each method is public
    if ($classMethod->isPublic()) {
        // If so, add this class with this method as a new JoinPoint
        $this->joinPoints[$this->numberOfJoinPoints] =
            new JoinPoint ($joinClass, $classMethod->getName());
        // Increase the number of JoinPoints by one
        $this->numberOfJoinPoints += 1;
    }
}
$this->numberOfJoinPoints += 1;
}
}
break;
// AllProtected: Add all protected methods of this Class
 case 4:
// Get all methods by reflection
 $reflectionClass = new ReflectionClass($joinClass);
 $classMethods = $reflectionClass->getMethods();
 foreach ($classMethods as $classMethod)
 {
 // Verify each method is protected
 if ($classMethod->isProtected())
 {
 // If so, add this class with this method as a new JoinPoint
   $this->joinPoints[$this->numberOfJoinPoints] =
     new JoinPoint ($joinClass, $classMethod->getName());
 // Increase the number of JoinPoints by one
   $this->numberOfJoinPoints += 1;
 }
 }
break;
// UserDefinedMethods Add the user defined method of a Class
 default:
// Verify the class and the method exist
 if (JoinPoint::verifyJoinPoint($joinClass, $joinMethod))
 {
 // If so, add this class with this method as a new JoinPoint
   $this->joinPoints[$this->numberOfJoinPoints] =
     new JoinPoint ($joinClass, $joinMethod);
 // Increase the number of JoinPoints by one
   $this->numberOfJoinPoints += 1;
   return true;
 } else
 {
 // If not, return false
   return false;
 }
}
// Get number of JoinPoints Method
 public function getNumberOfJoinPoints ()
{
// Return the number of JoinPoints
return $this->numberOfJoinPoints;
}

// Print all JoinPoints in the current window
public function printJoinPoints ()
{
// Remember the selected JoinPoint
$tempSelectedJoinPoint = $this->selectedJoinPoint;
// Set the selected JoinPoint to -1
$this->selectedJoinPoint = -1;
// Loop through the JoinPoints
while ($tempJoinPoint = $this->getNextJoinPoint())
{
// Print the JoinPoint class and the JoinPoint method
echo "JoinPoint $this->selectedJoinPoint.";<B><BR>
    echo "- Class <I>".$tempJoinPoint->getJoinClass()."</I><BR>
    echo "- Method <I>".$tempJoinPoint->getJoinMethod()."</I><BR>
}
// Reset the selected JoinPoint
$this->selectedJoinPoint = $tempSelectedJoinPoint;
// Return true
return true;
}

// Set selected JoinPoint method
public function setSelectedJoinPoint ($joinPoint)
{
// Verify the JoinPoint exists
if ($joinPoint < $this->numberOfJoinPoints && $joinPoint > -1)
{
// If so, set the selected JoinPoint
$this->selectedJoinPoint = $joinPoint;
    return true;
} else
{
// If not, return false
    return false;
}
}

// Get the selected JoinPoint
public function getSelectedJoinPoint ()
{
// Return the selected JoinPoint
    return $this->selectedJoinPoint;

} }

C Aspect.php

// Define standard aspects
define("before", 0, true);
define("after", 1, true);
define("around", 2, true);
define("throws", 3, true);

// Aspect class
class Aspect {

    // Aspect Constructor
    function __construct($pointCut, $advice, $code) {
        // Variable for looping through the JoinPoints of the PointCut
        $joinPointCrawler = 0;
        // Loop through the JoinPoints of the PointCut
        while ($joinPointCrawler < $pointCut->getNumberOfJoinPoints()) {
            // Set selected JoinPoint
            $pointCut->setSelectedJoinPoint($joinPointCrawler);
            // Copy the selected JoinPoint in a temporary JoinPoint
            $tempJoinPoint = $pointCut->getJoinPoint();
            // Get the class and the method
            $tempClass = $tempJoinPoint->getJoinClass();
            $tempMethod = $tempJoinPoint->getJoinMethod();
            // Find a non-existing temporary method
            while (!is_callable(array($tempClass, $tempMethod))) {
                // Add "AOP_" to create a possible non-existing temporary method
                $tempMethod = "AOP_".$tempMethod;
            }
            // Rename the method to the temporary method
            classkit_method_rename(
                $tempClass,
                $tempJoinPoint->getJoinMethod(),
                $tempMethod
            );
        }
    }
// Retrieve the methodType by reflection
$reflectionMethod = new ReflectionMethod($tempClass, $tempMethod);

// Clear the methodType
$methodType = '';

// If the methodType is private, set the methodType private
if ($reflectionMethod->isPrivate())
{
    $methodType = CLASSKIT_ACC_PRIVATE;
}

// Elseif the methodType is protected, set the methodType protected
else if ($reflectionMethod->isProtected())
{
    $methodType = CLASSKIT_ACC_PROTECTED;
}

// else the methodType is public, set the methodType public
else
{
    $methodType = CLASSKIT_ACC_PUBLIC;
}

// Clear the methodArguments
$methodArguments = '';

// Retrieve the method arguments by reflection
foreach ($reflectionMethod->getParameters() as $i => $methodParameters)
{
// Seperate the arguments with ',', '
    if ($i != 0)
    {
        $methodArguments .= ', ';
    }

// Add & if the argument is passed by reference
    if ($methodParameters->isPassedByReference())
    {
        $methodArguments .= '&';
    }

// Finally add the methodname
    $methodArguments .= '$'.$methodParameters->getName();
}

// Select the type of advice of this JoinPoint
switch ($advice)
{
}
// Before: The code is placed before the original method
    case 0:
        $methodCode = $code.'return self::'.$tempMethod.'('.$methodArguments.');';
        break;
    // After: The code is placed after the original method
    // Therefore the return value is lost in the new method
    case 1:
        $methodCode = 'self::'.$tempMethod.'('.$methodArguments.');'.$code.'return $return;';
        break;
    // Around: The code is placed between the original method
    // and the return value of this method
    case 2:
        $methodCode = '$return = self::'.$tempMethod.'('.$methodArguments.');'.$code.'return $return;';
        break;
    // Throws: When throwing or catching an exception, not implemented yet
    case 3:
        $methodCode = 'return self::'.$tempMethod.'('.$methodArguments.');';
        break;
    // Default: No code, just the original method
    default:
        $methodCode = 'return self::'.$tempMethod.'('.$methodArguments.');';
    }

    // The MethodName is added as a variable to the new Method
    $methodName = '.$tempJoinPoint->getJoinMethod().';
    $methodCode = $methodName.$methodCode;

    // Create a new method with the original name, arguments and type
    classkit_method_add
    ( $tempClass,
          $tempJoinPoint->getJoinMethod(),
          $methodArguments,
          $methodCode,
          $methodType
    );
    // Increase the joinPointCrawler by one
    $joinPointCrawler++;
    }