Aspect Oriented Software Development and PHP

This article provides an introduction into the popular paradigm of aspect-oriented software development (AOSD). It includes a multitude of practical examples, provides a view of how to objectify an abstract approach like AOSD, and helps the reader easily grasp its essence and advantages. The article is primarily intended for programmers working with PHP. Its aim is to demonstrate a way of applying AOSD in PHP-based projects that already exist.

by **DMITRY SHEIKO**

PHP: *4.xx*

TO DISCUSS THIS ARTICLE VISIT:
http://forum.phparch.com/297

The object oriented approach to programming has been popular for a number of years. While its advantages are not often obvious for short term projects, major development simply cannot do without it. Object-oriented programming languages provide the tools necessary to present business logic in a demonstrable form. Today, UML Class diagrams (*http://en.wikipedia.org/wiki/Unified_Modeling_Language*) can even be used to develop system logic.

Demonstrable business logic makes it easier for new participants to join in, and helps to save time for developers that come back into the project at later stages. It also reduces the number of mistakes, considerably. Is implementing an object-oriented approach, alone, enough to develop the demonstrable business logic? Obviously not. Creating a smart, object-oriented program architecture is not an easy task—unless you are able to successfully implement methods described in Martin Fowler's book, *Refactoring: Improving the Design of Existing Code*.

Yet, even now, one can not find encapsulated functionality (crosscutting concerns) in a number of various classes (logging, caching, synchronizing, tracing, monitoring, debugging, security checking, starting a transaction, opening a database connection,

etc.). AOSD (aspect-oriented software development, http://en.wikipedia.org/wiki/Aspect-oriented_programming) is capable of organizing this kind of program logic.

What is AOSD?

Aspect-oriented software development is a relatively new approach in the area of developing business applications. This approach is based on the notion of *Aspects*. Each aspect is a point of consideration for some other notion, process or perspective. To be able to quickly discern the underlying idea of this approach, let's try to consider various aspects of a Web site.

Information architecture defines a site's structure. Usability refers to the degree of site's friendliness towards the user. Graphic design determines the visual perception of a site. The functional model describes its business logic. All of these are various components of the Web site development process, and each component requires specific resources, approaches and solutions. For a project to be successful, all of the aforementioned components must be developed and implemented efficiently.

Aspect-oriented software development is a relatively new approach in the area of developing business applications.

If this example seems too complicated, let's consider a simpler and more universal scheme. When an apartment building is being designed, the architect had to first conceive framework, then an electrical scheme, water supply plan, and so on. Each of the stages in this process is a different aspect, a different perspective of the project. A similar approach can be implemented in software development for identifying various aspects of business logic within applications.

More than 20 years ago, Bjarne Stroustrup (http://en.wikipedia.org/wiki/Bjarne_Stroustrup) arranged the program code within C++ into sets of logical entities, whose behaviour and interactions could be defined in a number of ways (inheritance, encapsulation, abstraction, polymorphism). This is a reliable and time-tested paradigm of software development, known as object-oriented programming. Yet, this approach has certain limitations as to the decomposition of application business logic aspects. Over the years that have passed since the time this paradigm emerged, a number of approaches aimed at overcoming the said limitations have emerged. Among them are adaptive programming, composition filters, aspect-oriented programming, hyperspaces, role-modeling, subject-oriented programming, and so on. Lately, the development of such approaches has moved to the area of AOSD. As we have already pointed out, the code that covers cross-cutting is going to be distributed among the various modules, which will inevitably impair the quality of the software with regard to the transparency of business logic, its adaptability and upgradeability. The aim of AOSD is to isolate the cross-cutting concerns and place them outside applications' business logic.

Let us consider an integrated Web application, a content management system for example, and try to imagine the points and ways that we are likely to encounter the above-mentioned difficulties. Suppose a certain number of system functions require converting input data from XML to arrays. Due to the fact that, in this case, the XML parsing aspect involves only a limited number of functions, it doesn't imply considerable extension. As it is clear, in such a case there's no need for extensive decomposition and therefore no need for AOSD to be implemented. The simplest and most logical solution would be to include this procedure into the root class method (or, otherwise, into the external class, depending on the specific circumstances) and activate it when it's needed.

Yet, no manipulation of objects is going to help if we have to monitor the productivity, and our task is to take readings of all functions performed at their entry and exit points. An attentive reader would suggest that we should resort to the previous example and use a trigger for activation or deactivation of data capture. Well, we can only warn him or her that, should a need for logging system transactions in specified cases arise, it will become necessary to recall all of the details of the program architecture and to reorganize them.

It would be much more practical to task the system with handling specific events in certain objects, within a given aspect. Suppose, after some time, new security requirements have to be introduced into a large and

sophisticated project. We then create a procedure for additional security checks and task the system with carrying out this procedure at the activation of specified methods within the security aspect. This is implied by the paradigm of AOSD. We set up a new abstraction level outside the existing program architecture, and then in it, we declare functionality that in some way is applicable to the whole of the system.

or "JoinPoints" (method activation, class construction, access to a class field, etc.). Languages that support aspect-oriented programming (AOP) more commonly employ functions for a set of points, or a "Pointcut." The functionality at those points is determined by the program code which is politely referred to as *Advice (*in *AspectJ)*. Thus, aspects describe crosscutting concerns for a specific set of system components. The components

Over the years that have passed since the time that OOP emerged, a number of approaches aimed at overcoming its limitations have also emerged.

As you can see from Figure 3, it is possible to define program procedures that attend the system—for example, within the security aspect—and then place them outside of the main classes. We can treat the input data validation aspect procedures in the same way, thus making the business logic more evident and demonstrable. As a matter of fact, what we get as a result is clearly-defined business logic in the system's main classes and precisely distributed outside the essential model cross-cutting concerns.

To sum this up:

- The aspect-oriented approach is based on the principle of identifying common program code within certain aspects, and placing the common procedures outside the main business logic.
- The process of aspect orientation and software development may include modeling, design, programming, reverse-engineering and re-engineering.
- The domain of AOSD includes applications, components and databases.
- Interaction with, and integration into other paradigms is carried out with the help of frameworks, generators, program languages and architecture-description languages (ADL).

Basics of Aspect-Oriented Approach

Aspect-oriented programming allows developers to organize cross-cutting concerns into individual declarations—aspects. It provides the possibility to define functionality for specific program execution points, themselves include only business logic that they are supposed to implement. During compilation of the program, the components are associated with aspects (this is called the *Weave*).

To better grasp the basics of aspect-oriented programming, let us once again consider the example, which involves defining the productivity monitoring aspect (see Figure 2). Suppose we want to take the timer readings at each entry and exit point of all of the methods within such classes as `Model`, `Document`, `Record` and `Dispatcher`. So, we have to introduce into the Logging aspect a Pointcut with the listing of all of the required functions. To cover all of the methods of a certain class, most AOP-supporting languages employ a special mask. Now it is possible to define the needed Pointcut. We set up Advice at the entry (Before) and exit (After) points to the methods listed in the Pointcut. Usually, Advice in AOP-supporting languages is available for such events as Before, After and Around, though sometimes other events are also present.

Thus, the following basic AOP declarations may be singled out.

- *Aspect*–a definition of a certain set of cross-cutting concerns performing a specific task
- *Pointcut*–the code of an Aspect's applicability: defines when and where the functionality of a given aspect may be applied (see Figure 3)
- *Advice*–the code of an Aspect's functionality: this is what is going to be executed for the objects listed in the Pointcut.

Still too complicated? Well, I think everything will become clear once we introduce some practical

examples. Let's begin with the simplest of them: http://www.phpclasses.org/browse/package/2633.html. I wrote this small library with the plan of demonstrating both the advantages and availability of AOSD. Additionally, you don't need to know every detail of PHP or custom software to be able to use this library. It is sufficient just to enable `aop.lib.php` in your scripts in PHP 4 (or later) in its standard configuration.

It is possible to define a certain aspect for crosscutting concerns (let's say, for keeping a transaction log), through initiating the *Aspect* class:

```
$aspect1 = new Aspect();
```

Then we set up a *Pointcut* and specify the methods it affects:

```
$pc1 = $aspect1->pointcut(
  "call Sample::Sample  or call Sample::Sample2");
```

The only thing remaining is to specify the program code for entry and exit points for the methods of the current pointcut:

```
$pc1->_before("print 'PreProcess<br />';");
$pc1->_after("print 'PostProcess<br />';");
```

In the same way, we can define an additional aspect, for example:

```
$aspect2 = new Aspect();
$pc2 = $aspect2->pointcut("call Sample::Sample2");
$pc2->_before("print 'Aspect2 preprocessor<br />';");
$pc2->_after("print 'Aspect2 postprocessor<br />';");
```

In order to enable one or several aspects, just use the `Aspect:::apply()` function:

```
Aspect::apply($aspect1);
Aspect::apply($aspect2);
```

As you probably know, before PHP 5 emerged, handling method and class events was somewhat of a problem. Whereas, for global events—PHP errors, for example—one can develop a specific handler, to handle events at entry and exit points of methods it is necessary to "manually" insert some kind of "notifiers." In our case we'd need to insert special `Advice::_before();` and `Advice::_after();` functions:

```
class Sample {
  function Sample() {
    Advice::_before();
    print 'Class initilization<br />';
    Advice::_after();
    return $this;
  }
  function Sample2() {
    Advice::_before();
    print 'Business logic of Sample2<br />';
    Advice::_after();
    return true;
  }
}
```

As it is clear from the above example, we inserted "notifiers" for these events before and after the method's business logic code. When the PHP processor finds such a "notifier," it checks for the active aspects. If there is one, PHP checks for the current function specified in the Pointcut. If the function is there, it is activated for the given event (for example, for `Advice::_before()`). As you see, it is quite simple, but what is the practical value of this approach?

Suppose, we have inserted "notifiers" into all of the class methods of our scripts and enabled the `aop.lib.php` library. Then, one day a need arises to obtain a detailed report on the distribution of workload among the functions of our project. We set up an aspect and define its Pointcut, which includes all functions within the project:

```
$Monitoring = new Aspect();
$pc3 = $Monitoring->pointcut("call *::*");
```

As it is clear from the example, it's possible to use the `*::*` mask. Then, in *Advice*, we can employ a conventional function for the calculation of exact time in milliseconds:

```
function getMicrotime() {
  list($usec, $sec) = explode(" ",microtime());
  return ((float)$usec + (float)$sec);
}
```

With this function's help, we can take time readings at entry points of each of the project's functions and compare them with the readings at the functions' exit points. The obtained time of the business logic operations within the body of the function is stored in a global report variable. The only thing remaining is to display the report at the end of the program cycle. An example of employing a productivity monitoring aspect is presented in the `sample_trace.php` script found in the archive of the

distribution kit.

So you don't get the impression that AOSD can only be used for certain specific tasks, let's consider another example.

PHP is quite tolerant towards various types of variables. On one hand, this is a very positive feature because it means that there's no need to constantly check for compliance with the specified types and waste time/resources with a declaration. On the other, this could lead to errors.

(WAP), WSDL (SOAP), RSS/ATOM or SVG mark-up code. Obviously, in these cases it is unacceptable to display HTML-code in the error message. The "notifier" in the PHP error processor will make the system display the message in XML, or use a non-visual means (for example, send the notification via e-mail).

Anyone who has participated in the development of commercial software knows how difficult it is to solve the issue of updating the final product. Certainly, we are all aware of the existence of version control

Anyone who has participated in the development of commercial software knows how difficult it is to solve the issue of updating the final product.

When a project contains a huge number of functions, it is quite difficult to remember the parameters to each one. Yet, misplacing even one single argument can cause unpredictable changes in a function's behaviour. Can AOSD be of help in such a situation? Definitely! Let's recall the diagram in Figure 3. As you see, the `Document` and `Record` classes contain similar methods: `add()`, `update()`, `delete()`, `copy()`, `get()`. A well-organized program architecture predefines similar syntax for these methods: `add($id, $data)`, `update($id, $data)`, `delete($id)`, `copy($id1,$id2)`, `get($id)`.

AOSD can help us organize the program architecture. We can set up an input data validation aspect and define the Pointcut for the `Document()` and `Record()` class methods. The `add()`, `update()`, `delete()`, `copy()` and `get()` entry event functions can check for the type of the first argument. If it is not integer, an error has surely occurred. It is also possible to set up the second Pointcut for the `add()` and `update()` methods. In this case, we need to check the type of the second argument, which obviously must be an array.

In this way, we can place transaction logging outside of the project business logic. This makes it possible to define functions which require additional checks for security etc., at any time.

Of particular interest is the fact that, with the help of AOSD, we can trigger a specific system error message to be displayed for a specified set of functions. Suppose a number of our functions contain logic to set up WML

software—such as CVS (Concurrent Versions System http://en.wikipedia.org/wiki/Concurrent_Versions_System). The problem is that every new product, based on the previous one, requires certain customization, and more often than not, it is not at all easy to find out whether the update is going to affect areas customized for a specific project. Some of you have surely encountered cases when, after an update, you had to restore the whole project from back-up copies. Now, try to imagine a case when the disappearance of customized features is noticed only a long time after the update! "Well, where does AOSD come in?" you may ask. It can enable us to address this issue: the whole customization code can be placed outside the project's business logic as crosscutting concerns. You only have to define the *Aspect* of interest, specify the area of its application (*Pointcut*) and elaborate the corresponding functionality code. It is worthwhile trying to imagine exactly how this is going to work.

Let's recall my favourite example, content management software. Such a product is sure to contain a function for displaying record sets (`Record::getList()`), and developing code for these sets (`View::applyList()`). `Record::getList()` receives a `recordset` indicator and data selection parameters. This function produces an array with data on the results of the selection. The `View::applyList()` function receives this array at the input point and generates the formatting code—for example, HTML code for a table. Suppose that our product displays a goods catalogue as such `recordset`s. This is a universal

FIGURE 1

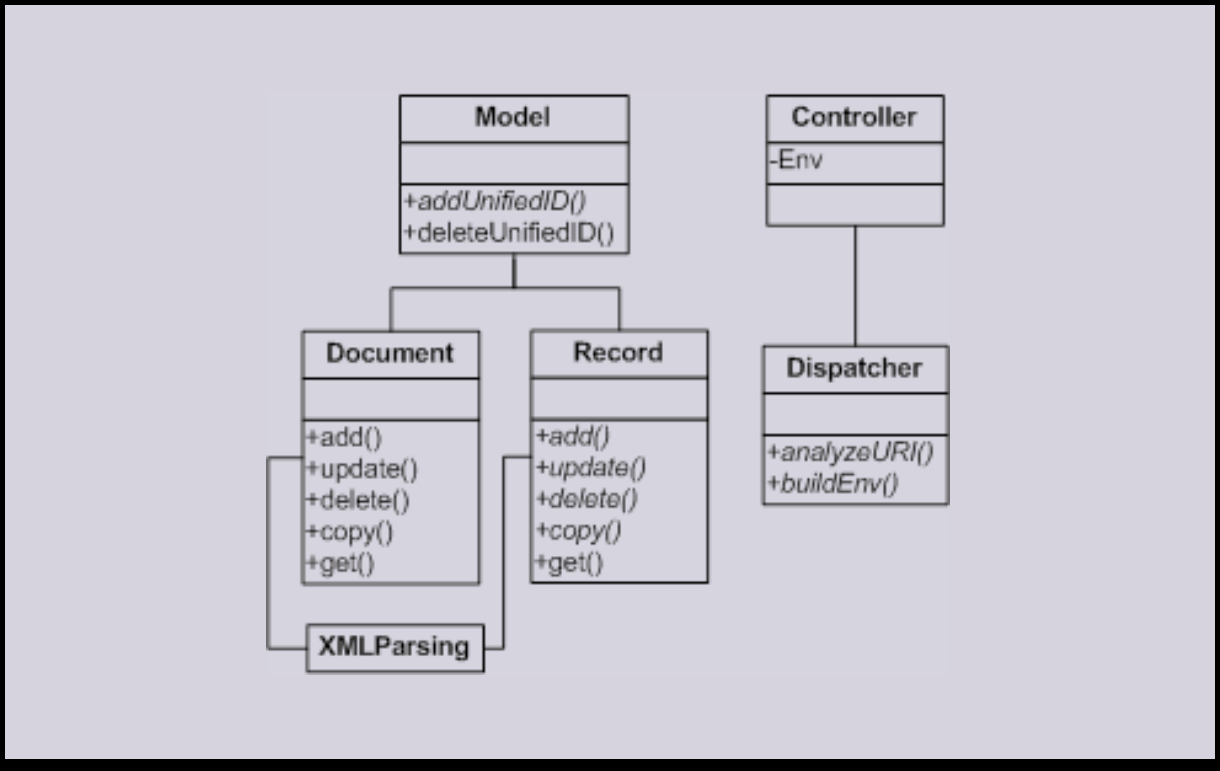


FIGURE 2

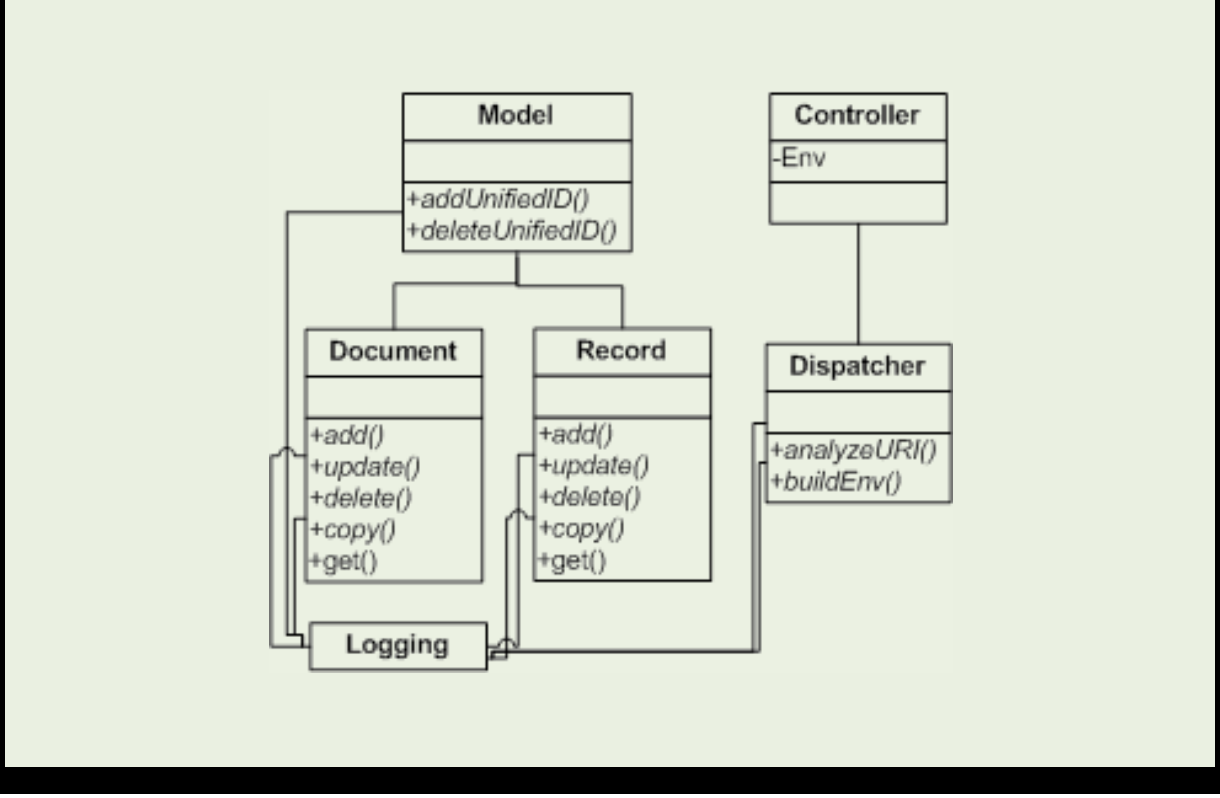
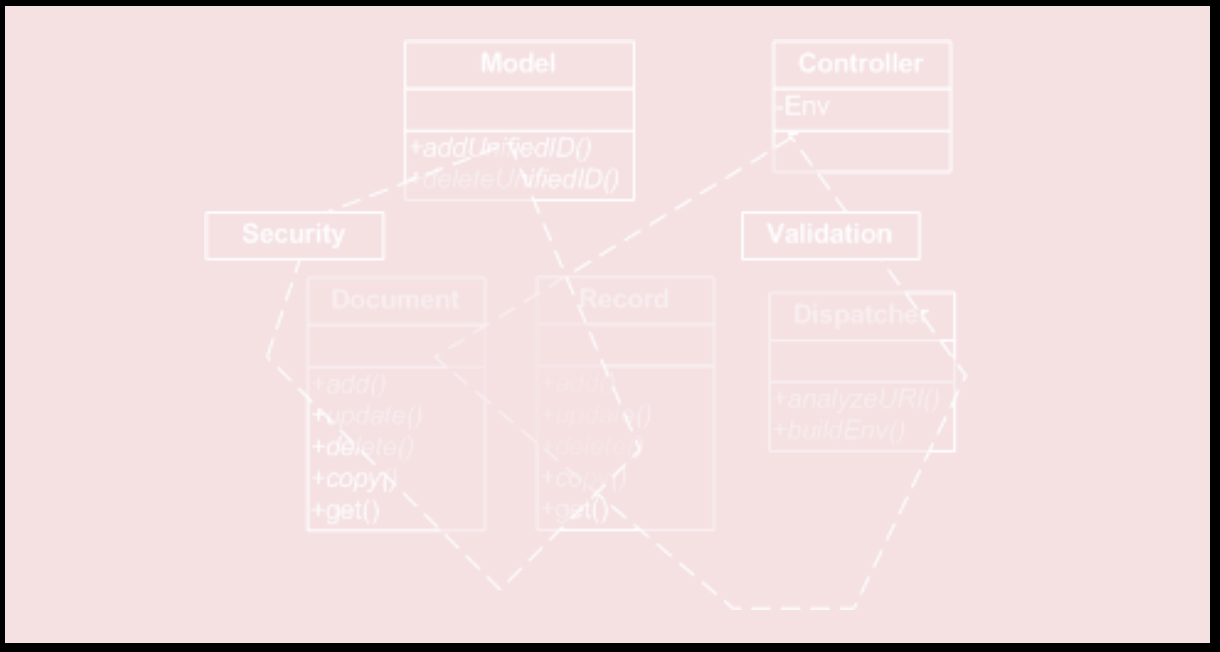


FIGURE 3



solution for a commercial product, yet for every specific project based on this code, it is necessary to introduce an additional column into the sets. For instance, tables in the original product have `Stock Number` and `Item` fields, while we need to add one more, `Customer Rating`, to the new application. In order to do this, we write Advice for the `Record::getList()` function, which establishes that an additional column is inserted into the returned array at the end of the runtime.

If the `View::applyList()` function is incapable of adjusting itself automatically for the changes in the input array, then we would have to write *Advice* for this function as well.

Let's assume that later on, the client demands that we mark out all of the entries in the sets which include goods not found in store at the current time. In this case we add *Advice* for the `View::applyList()` function, in which we check for the value of the *Available in store* attribute. Notice that we may set up a separate `Plugins` folder for aspect declarations and scripts that include their functionality. Thus, we shall have a complete set of customization features for any sort of a project stored in one specified folder, and there will be no upgrading problems whatsoever. We'll be able to easily upgrade any system scripts, except for those stored in the *Plugins* folder.

Aspect-Oriented Software Development in PHP

At present, there are a number of advanced projects, whose authors have introduced various techniques of AOSD implementations in PHP. The AOPHP project (*http://www.aophp.net*) includes a PHP preprocessor written in Java 1.5. We can write a usual PHP code, but we'll also have to notify the preprocessor of our intention to use AOSD. To do this, we'll use a `<?AOPHP ?>` structure instead of `<?PHP .. ?>`. The crosscutting concerns can be put into separate scripts.

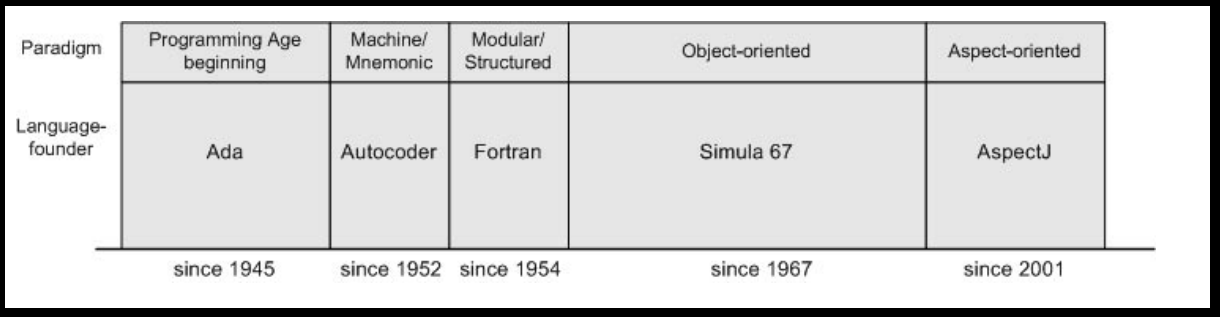
```
before(): execr(add($x,$y)) | execr(sub($x,$y)){
    echo "<font color=red>Im About To Add/Sub $x & $y</font><br>";
}
```

If necessary, these scripts can be enabled by issuing instructions while declaring AOPHP code:

```
<?aophp filename="aotest.aophp,aotest2.aophp" debug="off"
// PHP code
?>
```

The Seasar.PHP project (*www.seasar.org/en/php5/index.html*) employs a different approach. Its authors use XML for structuring aspect declarations, while integration is carried out with the help of PHP, which then executes the resulting code through the `eval()` function.

FIGURE 4



AOP is obviously not going to eliminate all bugs or develop software on its own.

The MFAOP project (www.mfaop.com) is based on an approach slightly resembling the one I mentioned in the above examples. The project's author suggests that first, one should set up a *Pointcut* and then use it according to what is needed:

```
$pointCut = new PointCut();
$pointCut->addJoinPoint('Example', 'Foo');
$pointCut->addJoinPoint('Example', 'Bar');
$test1 = new Aspect($pointCut, before, 'echo "Before
$MethodName";');
$test2 = new Aspect($pointCut, after, 'echo "After $MethodName";');
```

Unlike with the `aop.lib.php` library, when using this solution, there's no need to manually insert "notifiers" for each function. You would, however, have to install an additional PECL Classkit extension on the server.

From my point of view, the most elegant solution has been found by the *PHPAspect* project (www.phpaspect.org). This was possible thanks to the effective use of new capabilities provided by PHP 5—and in particular the possibility to set up abstract classes. *PHPAspect* introduces into the language of PHP a specific structure; which presents the declared aspect in a demonstrable form.

```
aspect TraceOrder{
    pointcut logAddItem:exec(public Order::addItem(2));
    pointcut logTotalAmount:call(Order->addItem(2));
    after logAddItem{
        printf("%d %s added to the cart\n", $quantity, $reference);
    }
    after logTotalAmount{
        printf("Total amount of the cart : %.2f €\n",
          $thisJoinPoint->getObject()->getAmount());
    }
}
```

As it's clear from the example, the interval of the specified aspect is precisely defined. The *Pointcut* and *Advice* are set up in a way that is so simple, that one may think this is "native" PHP syntax. This project provides handling capacities for 7 (!) types of *Join point* events: `call`, execution (`exec`), class construction (`new`), attribute write (`set`), attribute read (`get`), class destruction (`unset`) and block catching (`catch`). There are three possible types of *Advice*: `before`, `after`, `around`. This project developed unusually flexible masks for defining surveillance intervals in Pointcuts. Thus, for instance, it is possible to assign intervals for all classes with the specified prefix in the name:

```
new(*(*));
exec(* Order::addItem(2));
call(DataObject+->update(0));
```

To be able to install PHP *Aspect*, you'll need PHP version 5.0.0 or later, and have the PEAR `Console_Getopt`, `Console_ProgressBar`, and `PHP_Beautifier` libraries installed.

This project was successfully presented last year at

the PHP conference (http://afup.org/pages/forumphp/) in France (this is where the project originates from), and as for as I know, it is making good progress.

Conclusion

Obviously, AOSD is not a universal solution. AOP is not going to eliminate all bugs or develop software on its own. I doubt that every developer using aspect-oriented programming will get a Nobel Prize. Moreover, this approach is hardly going to become the dominant one in the future: the object-oriented approach has been popular for 20 years, yet many developers still confine themselves to procedural code. On the other hand, the advantages of AOSD before object-oriented programming are as obvious as OOP's advantages before procedural programming.

I think I'm on the safe side if I say that developers who use AOSD are one-step ahead of the others. Their code is clearer, easier to perceive, it contains fewer errors and can be easily developed. Thus, AOSD engineers are capable of implementing larger-scale, more reliable and versatile projects.

Importantly, AOSD does not require complete retraining of developers. AOP doesn't alter programming logic drastically, as is the case with transition from procedural to object-oriented programming. AOSD just extends it. When developing a new program architecture, you only extract from the objects the parts that you think don't fit there, and find a suitable place for them. Imagine that for years you've tolerated a piece of sculpture, which totally disagrees with the overall design of your home, but which you treasured as a gift. Then one day you find an inconspicuous niche in the wall where the sculpture seems to fit perfectly. "Well then," you'd say in that case, "this is where it really belongs!"

Apart from all that has already been said, AOSD is easy to get used to—unlike, for example, TDD (http://en.wikipedia.org/wiki/Test_driven_development). At first, one could try and put simple crosscutting concerns, like logging, into aspects, and then gradually extend decomposition of the program architecture, accumulating various domains of the application into aspects.

Perhaps, some are confused by the fact that, at present, PHP does not officially support AOSD. Yet, in this very article I have presented several examples of how you can easily integrate basic AOSD approaches on your own. It is the essence of AOSD that is important, not the particular way in which it is implemented. Whatever approach you chose, if you were able to achieve efficient decomposition in program architecture it is inevitably going to improve the quality of your product.

AOSD is not going to become a new technological revolution, but an inevitable programming evolution.

Today, AOSD is supported mainly by extensions to popular programming languages, yet the leading players in the market are not going to stay out of it, and AOSD is gradually finding its way into the popular programming platforms (http://www.internetnews.com/dev-news/print.php/3106021). AOSD is not going to become a new technological revolution, yet evolution, on the other hand, is inevitable. Whether you follow or take the lead is up to you. ■

DMITRY SHEIKO is a lead We developer at Reg Graphic Systems (www.redgraphic.com), and has been involved in software development since 1987. Since 1998, he has published more than 50 technical articles in various publications, and has been working on architectural solutions and framework development for content management software (CMF, ECM) since 2001. Lately, Dmitry has designed a set of CM business products, including the management platform and Web application framework Site Sapiens (www.sitesapiens.com), and has worked on a specification of language XML Sapiens for developers of CMS-powered Web sites (www.xmlsapiens.org). You can visit his website at www.cmsdevelopment.com.