Aspect-Oriented Programming in PHP

by Edmund P. Zynda III

Aspect-Oriented Programming in PHP

Edmund P. Zynda III

This book is for sale at http://leanpub.com/aopinphp

This version was published on 2016-02-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Edmund P. Zynda III by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Aspect-Oriented Programming in PHP

The suggested hashtag for this book is #aopinphp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#aopinphp

Contents

1. Introduction and Prerequisites **1**
1.1 Introduction 1
1.2 A Super Short History 1
1.3 Prerequisites 1

2. What is Aspect-Oriented Programming? **3**
2.1 Aspect-Oriented Programming in a Nutshell 3
2.2 What Does Aspect-Oriented Programming Look Like? 5
2.3 Using Aspects in PHP 6

1. Introduction and Prerequisites

1.1 Introduction

The idea of aspect-oriented programming has been around for about 20 years already. It has made its way into popular languages like Java and those used in Microsoft's .NET framework. It has helped to make applications written in those languages cleaner and more modular. Aspect-oriented programming has only recently made its way into PHP through various libraries and even an extension to PHP itself.

While aspect-oriented programming does help us solve some unique problems, it isn't a replacement for object oriented programming. Programming with aspects can enhance our object oriented code and as you'll see in this book, it's pretty simple to understand. Let's begin!

1.2 A Super Short History

Aspect-oriented programming was created sometime between 1995 and 1996 by Gregor Kiczales and his colleagues at Xerox's Palo Alto Research Center (PARC). Gregor's group had been working on a Java game, similar to Asteroid, called Space War. Because the game was written in Java, an object oriented language, the team was able to encapsulate the different pieces of the code into separate objects. Asteroid objects had their own logic while a spaceship object has its own logic. This is great because separation of concerns is a principle of good object oriented design.

There was a problem though. To get the different objects to display on the screen, there needs to be a call to some sort of draw or paint method. This method needs to be called every time an object moves on the screen. The logic to do this is pretty much the same though for any object. Creating a draw method for every object would result in a lot of duplicate code.

The team at Xerox, came up with a way to address this by allowing programmers to encapsulate logic that might affect different areas of the application and then apply it only where it was needed. This new paradigm of programming "cross-cutting" concerns was dubbed aspect-oriented programming by Chris Maeda, another member of Kiczales' team. The team went on to create AspectJ, an aspect-oriented language and extension for Java which first appeared in 2001.

1.3 Prerequisites

To run the examples in this book, you will need to have PHP 5.4 or greater installed on your machine. The operating system should not matter but I recommend a POSIX compliant operating system like Ubuntu Linux.

This book will focus on using an aspect-oriented PHP library called “Go! AOP”. This library and others can be installed using Composer (http://getcomposer.org).

It should go without saying, but you should have a basic to intermediate understanding of PHP and object-oriented design. This is not a book for beginners and some advanced concepts will be covered.

2. What is Aspect-Oriented Programming?

2.1 Aspect-Oriented Programming in a Nutshell

Aspect-oriented programming is a paradigm created to address the problem of what is called cross-cutting concerns. A cross-cutting concern is any bit of code that affects multiple concerns or parts of your program. These kinds of concerns are usually difficult to decouple from the rest of the application. Because of this, cross-cutting concerns can lead to scattered, duplicated or tangled code.

What's an example of a cross-cutting concern though? The standard boring example that's always given when talking about aspect-oriented programming is "logging." While it's not exciting, it's a pretty good example because it illustrates exactly the kind of problem aspect-oriented programming was meant to solve.

Let's say you have a pretty sizable PHP application built. All of your classes are organized neatly, following SOLID principles. Everything is working great. Now let's say your boss or client decides that they want logging functionality added to the application. They want to produce some better analytics for their customers and certain areas of the system need to log information for that to happen.

The first thing you might do is create a logging class. All it does is log. That's simple enough. Now you need to figure out what areas of the system need to be logged so you find those classes and decide you can just inject your new logger into those classes and log where needed. But wait! That would violate the 'S' in SOLID. You no longer have separation of concerns because you've tainted your nice clean class with logging logic.

Logger injected through constructor

```
1  <?php
2
3  namespace App\Service;
4
5  use Logging\Logger;
6
7  class Facebook
8  {
9      private $logger;
10
11     public function __construct(Logger $logger)
```

```
12      {
13          $this->logger = $logger;
14      }
15
16      public function doStuff()
17      {
18          // Do Facebook stuff...
19          $this->logger->log('Facebook stuff completed.');
20      }
21  }
```

Next you decide that the better way to go about it is to decorate those classes. If you've coded those classes to an interface all you have to do is create a logging decorator class for each of those classes and just inject the original class into the decorator. From the decorator you just implement the required methods with logging and call the corresponding method from your original class. Then you realize that you need to do this for every...single....class that you want to have logging implemented in. Yikes! Is there a better way?

Service class implementing an interface

```
1   <?php
2
3   namespace App\Service;
4
5   use App\Service\FacebookInterface;
6
7   class Facebook implements FacebookInterface
8   {
9       public function doStuff()
10      {
11          // Do Facebook stuff...
12      }
13  }
```

Logging decorator class

```php
<?php

namespace App\Service;

use App\Service\Facebook;
use App\Service\FacebookInterface;

class LoggingFacebook implements FacebookInterface
{
    private $fb;

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

    public function doStuff()
    {
        $this->fb->doStuff();
        // Log some stuff
    }
}
```

2.2 What Does Aspect-Oriented Programming Look Like?

There is. In aspect-oriented programming we can take a look at our code from the point of view of our logger. In other words, we look it through the logging aspect, hence the name. We want to add logging functionality throughout our code. With aspect-oriented programming we can apply that functionality to those other areas with touching the original code. Sound magical? Here's what a simple aspect-oriented logger might look like in PHP.

An aspect-oriented approach to logging

```php
<?php

namespace App\Aspect;

use Go\Aop\Aspect;
use Go\Aop\Intercept\MethodInvocation;
use Go\Lang\Annotation\After;

class Monitor implements Aspect
{

    /**
     * Log after a method is executed
     *
     * @param MethodInvocation $invocation Invocation
     * @After("execution(public Stark\Service\Facebook->*(*))")
     */
    public function afterMethodExecution(MethodInvocation $invocation)
    {
        $this->log('Executed method: ' . $invocation->getMethod()->getName());
    }

    private function log($msg)
    {
        // Log stuff
    }
}
```

2.3 Using Aspects in PHP

As of this writing, PHP doesn't have any built in functions or libraries for aspect-oriented programming. There is a PECL extension called AOP-PHP that's currently in beta and it provides some simple functions for applying logic to existing code. This book won't cover that extension but you can find the source code and documentation on GitHub[1].

The example above uses a library called Go! AOP. It's a user-land library, meaning it's written in pure PHP and can be installed using a tool like composer. This book and later examples will focus on using this library. The source code and documentation can be found on GitHub[2] as well.

[1] https://github.com/AOP-PHP/AOP

[2] https://github.com/lisachenko/go-aop-php