Mastering Object Oriented PHP

By Brandon Savage

Copyright

Credits

Credits to reviewing parts of Chapters 2 and 3 goes to Anthony Fererra. He also provided the sample code in Chapter 3.

The sample code for Doctrine came from Johnathan Wage, Roman Borschel, Matthew Weier O'Phinney, Kris Wallsmith and Fabien Portencier. The code is licensed under the MIT License and used with permission.

Table of Contents

About The Author

Brandon Savage is a software developer who began his PHP career in 2003, before PHP 5 was even released. He began by developing gaming systems, figuring that having a computer execute complex math was preferable to doing it by hand.

Brandon eventually figured out that he could make money by writing code, and began full time software development in 2008. He quickly outgrew his first few roles, and eventually landed at Mozilla, where he works as a Software Engineer on the Socorro team (Socorro is Portuguese for “help”, and Socorro is the software tool that aggregates Firefox crash data).

Brandon lives in Olney, Maryland with his wife and three cats. He is a private pilot and loves aviation. You might see him flying his red-and-white Piper Cherokee 235 over the skies of the East Coast. He promises not to fly too low, though.

Introduction

Identifying best practices can be hard work, especially for someone just learning the art of programming with PHP. I found this especially difficult when I was first learning how to design object oriented applications. And so, I resolved that I would write down my experiences and highlight what I felt were some of the most important best practices in object oriented PHP.

But I wanted to avoid simply reproducing the PHP manual in book form. I knew this wouldn't be helpful. Developers wouldn't get the most out of simply having the manual repeated over and over again. We needed a different approach.

Enter *Mastering Object Oriented PHP*. This book is a collection of essays, highlighting nine of the best practices in PHP object oriented development. Many of these topics actually arise from broader object oriented development principles. Others are specifically PHP focused. All of them are areas that I felt were underserved but quite important.

As a community, we do a disservice to developers if we assume they come to PHP with knowledge of all computer science best practices. We are diminished if we assume they already know how to use exceptions, or conceptualize abstraction, or know how and when to make use of PHP's magic methods. This book seeks to explain these topics in an easy-to-understand and easy-to-use way.

It is my sincere hope that by the conclusion of these essays, you will have a greater understanding of some of the most important best practices in PHP, be able to apply them to your development practices, and have a methodology for growing your known knowledge of object oriented development.

Chapter 1: An Introduction To Object Oriented Programming

Why Make Code Object Oriented At All?

The object model in PHP was introduced in PHP 4, and refined into a true object model in PHP 5. Almost as soon as the code was released in PHP 4, people wanted to know, "why should I make my code object oriented?" After all, PHP had become popular and useful as a procedural scripting language. A lack of objects was not necessarily a problem for most developers, and since many developers were not programmers in the classical sense, object orientation wasn't high on their list.

There are a number of great reasons why object orientation in PHP makes sense. Here are a few of them.

Object Oriented Code Adheres To Tried-And-True Principles

When developers architect object-oriented code, they almost universally follow a set of core principles that have been developed over time by the software development world. Principles such as abstraction, single responsibility and encapsulation are critical ingredients in a code base that has been well developed.

It's difficult (though not impossible) in procedural code to accomplish the same goals, producing what many developers term "spaghetti code." This type of coding style leads to more bugs, less satisfaction and increasingly frustrated developers and managers alike.

By developing a code base that is object oriented, developers can avoid the spaghetti code phenomenon, provided they are diligent in their architecture and design. Object oriented programming requires the ability to step back and consider the overall design of an application, which leads to better implementation.

It's Easier to Test Object-Oriented Code

Modern unit testing frameworks like PHPUnit[1] are designed to test the object oriented codebase. Unit tests rely upon the ability to remove outside sources of possible failure from the picture, effectively testing the specific unit under consideration. Procedural code doesn't usually allow for mocking of behaviors, and thus makes it difficult to properly conduct unit testing.

Instead, what most developers of procedural codebases do is "functional testing" – testing the system end-to-end. This makes it more difficult to pinpoint API failures or identify when and where something broke.

[1] http://www.phpunit.de/manual/current/en/

Object orientation allows for developers to mock objects, to create stand-in objects and fixtures that can be used in testing. Since most testing frameworks are designed in an object oriented fashion, it's also easy to create encapsulated tests that fully test a given class or set of behaviors.

Object-Oriented Code Doesn't Require Full Domain Knowledge

When first learning Zend Framework, I found that it's object oriented design allowed me to interface with aspects and components of Zend Framework that I did not fully understand. I didn't need to fully understand them, because their API was well defined and because I could generally derive what and why they provided certain information, I didn't need to have domain knowledge over all aspects of Zend Framework.

Procedural programming often doesn't allow this sort of behavior. A fuller (if not complete) domain knowledge is usually required to effectively work with a given codebase.

Object-Oriented Code Is Extensible

Without the extensibility of object oriented code, Zend Framework and most of the other major PHP frameworks would be impossible. Extensibility allows for the design and development of common APIs, along with the implementation of design decisions that developers must abide by when writing their own code. Developers are then free to self-manage implementation details to suit their application.

This enforcement of design decisions makes the code extensible. So long as I follow all the rules and principles of the Zend Framework API for authentication, I can create my own methodology for authenticating users (and I have). I merely have to return the expected data structure, and Zend Framework's other components can understand what I've done, without knowing or caring how.

This would be impossible if I couldn't swap one object in for another. Developers have found ways around this kind of issue with procedural programming (see the extensibility of Wordpress), but it's often difficult to control, understand and work with. Object oriented code makes this much easier.

Terminology In Object Oriented Programming

To understand this book, one must understand the terminology that will be used to describe object oriented programming. Here are some of the basic terms:

Class – The code that defines a particular object's behavior. Class is also the PHP keyword to define a PHP class.
Object – An instance of a particular class. Non-static classes must be instantiated into objects before they can be used.
Property – These are essentially class variables.
Method – These are essentially class functions.

Important Operators In PHP's Object Model

There are some important operators in PHP's object model that developers must understand in order to use the object model.

Public/protected/private or var – One of these must be used to define all properties of the object. For example:

```
<?php

class MyClass {
    public $var1;
    protected $var2;
    private $var3;
    var $var4;
}

?>
```

$this[2] – The $this variable refers to the object itself, as well as it's ancestor classes. It is a special variable that exists in all instantiated objects, and is essentially a reference to the object itself. $this is used to refer to object-level properties and methods. $this cannot be assigned another value, can not be used outside object oriented code.

The Object Operator (->) When referring to a method or property that is a part of an object, the object operator (->) is used. For example, to refer to a method of object $obj, you would refer to it as $obj->myMethod(). The -> operator can be used with $this.

The namespace operator (\)[3] – In PHP 5.3, namespaces were introduced. Namespaces allow for more easily understood naming conventions, since the risk of a namespace collision is lessened. The operator \ is used for namespaces (e.g. \My\Awesome\Namespace).

[2] http://php.net/manual/en/language.oop5.basic.php
[3] http://php.net/manual/en/language.namespaces.rationale.php

Class Constants[4] - In PHP’s object oriented model, constants can be defined within a class:

```
<?php

class MyClass {
    const MY_CONST = ‘some constant value’;
}

?>
```

While there is no requirement to capitalize constants (they can be any legal string), the convention is to capitalize all letters in a constant to help identify them as a constant. Constants are useful for defining values that never change in a class and thus should not be variables.

The Scope Resolution Operator (::)[5] - If you ever see an error called “T_PAAMAYIM_NEKUDOTAYIM”, that error was referring to this symbol. Since PHP was developed by Israelis, they gave this particular symbol the name of Paamayim Nekudotayim, which means “twice colon” in Hebrew. This operator is used to refer to static methods, as well as to refer to scope within the object.

When referring to a static method, the methodology would be MyClass::myMethod(). When referring to a static property, the methodology would be MyClass::$myProperty.

This operator is also used to refer to scope within objects. For example, to refer to a static method defined in the same class, it would be

[4] http://php.net/manual/en/language.oop5.constants.php
[5] http://php.net/manual/en/language.oop5.paamayim-nekudotayim.php

self::myMethod(); a property would be self::$myProperty. If you were in an instantiated class, and wanted to refer to a parent function you had overridden, you would use parent::myOverriddenMethod(). To refer to the parent's parent, you could use MyParentsParent::myOverriddenMethod().

Constants are also accessed with the scope resolution operator; you can access a constant with self::CONSTANT if you are inside the class that defines the constant or MyClassName::CONSTANT if you are outside the class that defines the constant. Constants cannot be protected or private, and they are always referred to with the scope resolution operator (you cannot refer to a constant with $this).

What Is An Object, And How Does It Work?

Wikipedia defines an object (in object oriented programming) as a "data structure with the associated processing routines."[6] Further, it describes an object as being an "instance of a particular class". This defines essentially what objects are in PHP.

Starting in PHP 4, objects were little more than arrays with methods attached. However, the object model in PHP 4 was woefully lacking in functionality and behavior, and so in PHP 5 the object model was effectively rewritten, into what it is today. PHP 5 introduced statics, visibility operators, interfaces, magic methods and a whole host of other features that are common in object oriented applications today.

[6] http://en.wikipedia.org/wiki/Object_%28computer_science%29

In PHP, “object” is a specific type[7], like “string”, “integer” or “array”. As their own type they have a behavior that is unique, and they don’t behave quite like any other data structure in PHP.

The most significant way that objects work differently in PHP from any other data structure is how they are passed around when they are assigned to variables. Some people will say that “objects are copied by reference”, which this isn’t entirely true. References in PHP are assigned like this:

```
<?php

$a = 1;
$b =& $a;
$b = 3;

print $a; // 3
print $b; // 3

$b = "I am a string";

print $a; // I am a string

?>
```

Because $a and $b are referenced to one another, their values are essentially tied together. Whatever $b is set to be (or $a is set to be), the other will follow suit.

Objects behave in a similar (but subtly different) manner. Consider the following code sample:

```
<?php
```

[7] http://php.net/manual/en/language.types.php

```
$obj1 = new stdClass();
$obj1->value1 = 1;

$obj2 = $obj1;

$obj2->value1 = 2;

print $obj1->value1; // 2;

?>
```

The $obj1 and $obj2 variables behaved the same way as the references: the $value1 property changed, and this change was reflected in both objects.

The key here is that $obj1 and $obj2 are part of the **same object.** They are simply two different variables for the same object. This is where the subtle difference between references and objects comes into play: Assigning $obj2 to anything other than $obj1 will **not** automatically reassign $obj1 to the same thing:

```
<?php

$obj1 = new stdClass();
$obj1->value1 = 1;

$obj2 = $obj1;

$obj2->value1 = 2;

print $obj1->value1; // 2;

$obj2 = "string";

var_dump($obj1); // object(stdClass)[1]

?>
```

Notice that reassigning $obj2 to a string value **did not** turn $obj1 into a string value. This is because objects are copied **by assignment**, just like ordinary variables.

But objects represented as variables are copied slightly differently from other types.[8] Rather than copying the data of an object, a referrer to that object in memory is what is actually assigned when an object is assigned to a variable. Just as assigning the same file to two variables does not cause the file to be copied, assigning an object to two different variables still points the variables at the same object.

It's important to understand the distinction and the difference here. Because variables point to specific objects, you can pass objects into functions/methods and expect that once they are transformed, they'll be transformed everywhere. But this means that once they're transformed, they're transformed everywhere: learn to expect this behavior in your applications! Many bugs have been introduced by failing to understand this principle.

Objects can be copied when assigned to variables, and a new object created from an old object. This can be done with the "clone" keyword:

```
<?php

$obj2 = clone $obj1;

?>
```

[8] http://blog.golemon.com/2007/01/youre-being-lied-to.html

When $obj2 is modified, the object associated with $obj1 will not be affected, because $obj2 is a distinct, discrete second object.

Finally, with regards to old code, occasionally you may come across code that assigns an object by reference:

```
<?php

$obj =& new PHP4StyleObject();

?>
```

Do not do this. In PHP 4, it was necessary to assign objects by reference to get the similar behavior I just described, but with PHP 5 this is no longer necessary (and by no means recommended). Replace it in your own code, and reject it in other developer's code. Thankfully, this is becoming less common in libraries and the wild.

Chapter 2: Private Methods Considered (Potentially) Harmful

A few weeks ago, I was tasked with integrating a library that was designed by someone else. This library was intended to access APIs and return the data so that it could be used by my application. This seemed straightforward enough, except that the API I was working with had a few quirks, namely that it interpreted the query string directly, and so it was possible to have a query string similar to this:

```
https://myapi.com/?param1=string1&param1=string2
&param1=string3
```

Obviously, this is “non standard” in the way of API development, but it was the method this API used to access multiple values of the same parameter.

Unfortunately, the developer of the library didn’t consider this use case. They (most of the time correctly) assumed that you would have only one value per parameter, and so they took an array of key-value pairs. If you wanted more than one value for a given parameter, you provided an array as the value for that key, and it interpreted that into a standard format.

"Easy enough," I thought to myself. "I'll just extend the class, override the method, and move on." Sadly, it was not that simple.

The method that needed to be overridden was marked private by the developer. Though there was no obvious reason why the method should be private, there it was, and to make matters worse, since it's an external library, I could not simply change the visibility marker myself.

PHP's Visibility Markers Explained

In PHP, much like in Java, there are visibility markers available to API developers.[9] The most basic (and the default) marker in PHP is "public" – this visibility level is assigned to any method or property that is not given one of the others. Public does not need to be declared, but I recommend declaring it to be explicit.

```
<?php

class TestObject {
    public function testFunc() {
        echo "Hello World!";
    }
}

class ChildObject extends TestObject {
    public function testFunc() {
        echo "Hello Reader!";
    }
}
```

[9] http://php.net/manual/en/language.oop5.visibility.php

```
// Original object
$obj = new TestObject();
$obj->testFunc(); // Hello World!

// Child object
$obj = new ChildObject();
$obj->testFunc(); // Hello Reader!

?>
```

We can also designate methods as “protected”, which means the method can be seen, used and overridden by child classes but cannot be seen or accessed from outside the object. Protected methods are useful for internal methods that relate to the particular object but do not need to be shown outside the object. They also allow you to easily make changes to the internal API without rewriting a lot of external code.

```
<?php

class TestObject {
    protected function testFunc() {
        echo "Hello World!";
    }

    public function getData() {
        return $this->testFunc();
    }
}

class ChildObject extends TestObject {
    protected function testFunc() {
        echo "Hello Reader!";
    }
}

// Original object
$obj = new TestObject();
$obj->testFunc(); // Error

// Original object
```

```
$obj = new TestObject();
$obj->getData(); // Hello World!

// Child object
$obj = new ChildObject();
$obj->testFunc(); // Error

// Child object
$obj = new ChildObject();
$obj->getData(); // Hello Reader!

?>
```

Finally, PHP allows for methods to be marked as a visibility of "private". These methods are internal methods only, and cannot be seen, used or overridden by subclasses.

PHP has three additional designations for methods as well: final[10], static[11] and abstract[12]. A final method is one that has been defined as not being extensible by child classes, but can be used either internally or externally (depending on the visibility setting). Private methods are automatically considered "final". Classes can also be marked final, which prevents the entire class from being extended.

Abstract methods are methods that are not defined in the present class, and are intended to be extended later on by the developer. Marking a method as abstract means that it has no functional content. It also requires that the class be marked abstract; that class cannot be instantiated directly and instead must be extended by a child class. Abstract methods cannot be private or final.

[10] http://php.net/manual/en/language.oop5.final.php
[11] http://php.net/manual/en/language.oop5.static.php
[12] http://php.net/manual/en/language.oop5.abstract.php

The Problem With Private Methods

Many developers believe that the internal methods of their objects should be private. While this thought process is understandable, it’s also potentially problematic. Though you may wish to prevent internal methods from being externally accessible, this only requires that they be marked “protected.”

Take the following example:

```
<?php

class StringPrinter {

    private $_string;

    public function __construct($string = null) {
        if($string) {
            $this->setNewString($string);
        }
    }

    public function printString() {
        print $this->_string;
    }

    public function setNewString($string) {
        $this->_setString($string);
    }

    private function _setString($string) {
        $this->_string = $string;
    }

}

class ExtendedStringPrinter extends StringPrinter {

    public function setUppercaseString($string) {
        if ($string) {
            $this->_setString(strtoupper($string));
            }
        }
}
?>
```

This code is extremely simple and intended to illustrate a specific point. The ExtendedTestObject class cannot override or access the private _setString() method. It does have the ability to access all the public methods, including setNewString(), though a developer may have good reasons for avoiding setNewString() in their particular implementation. However, the developer of ExtendedTestObject does not have the option to avoid using setNewString() if they want to set a string, and anything that setNewString() might do that is unpleasant or undesirable cannot be avoided.

Though the developer could simply override any unpleasant behavior in setNewString() by extending it and redefining it, the developer cannot simply redefine it if they wish to continue using the internal API of TestObject, since all the internal methods are marked private. They would instead be required to completely reimplement the methods that setNewString() calls that are private, and with complex code this can be extremely difficult or time prohibitive.

Finally, no subclass can make direct access of the private property $_string. Even if we were to subclass it and want to override the printString() method, we would be unable to do so because we wouldn't be able to access the private $_string property.

This may well be what the developer intended. When deriving an algorithm for example, this may be the intended behavior and developers may not wish to have the algorithm overridden in child classes. But for the vast majority of other methods, this simply doesn't make sense. If a developer truly wished to prevent extension of the _setString() method, they could have declared it protected final; this would have allowed access in subclasses and also allowed some extension of the class by future developers, but prohibited redefinition of the _setString() method. Everybody wins.

Why Not Make Everything Public?

In other languages like Python, there is no enforced concept of a “protected” method. Instead, every method is public[13], and there are conventions about marking methods with underscores to denote internal methods that shouldn’t be used by other objects.

The problem with marking every method as public is that the public API of an object is automatically presumed to be available for external use unless otherwise agreed upon by convention. This is especially true when visibility operators exist. While it is still often the convention in PHP code to mark methods with an underscore when they are protected (this started in PHP 4 when there were no visibility operators), marking all methods as public makes it impossible to maintain the API without making major changes.

```
<?php

/* This object is difficult to change between versions,
because anyone calling UnchangableObject::_internalCall()
expects the result to remain the same. */
class UnchangableObject {
    public function externalCall() {
        return $this->_internalCall();
```

[13] http://docs.python.org/2/tutorial/classes.html

```
    }

    public function _internalCall() {
        echo "Hello World!";
    }
}

/* This object is easy to change between versions, because
everyone has to call the public method. */
class ChangableObject {
    public function externalCall() {
        return $this->_internalCall();
    }

    protected function _internalCall() {
        echo "Hello World!";
    }
}

?>
```

Instead, it is better to have methods marked as public have consistent arguments, argument orders, and return values throughout releases. For example, a method called "returnDataSet" should consistently accept the same arguments (and in later versions, new arguments that have sane defaults) and return a data structure that is the same. However, the internal, protected methods can change between versions, and it is expected that these internal API elements may be unstable between minor versions.

Be A Good Citizen

At one job I worked, I had a boss that told me I should write code as though the developer coming after me would be a psychotic murderous maniac who had my address. This humorous tip reminded me to always be considerate and careful in my development. As a result, I learned never to use a private method when a protected method would do the trick: it’s just good citizenship in the world of method and property visibility.

Chapter 3: Inheritance Considered (Potentially) Harmful

There is a school of thought within the object oriented community that says the best method to allow interoperability and extensibility in code is to foster composition within objects, over inheritance by objects. This school of thought believes that you achieve the same outcome by making use of common interfaces, with a far cleaner design, and more reusable and extensible.

Understanding Inheritance

When an object inherits from its parent, it also inherits the methods and properties of its parent. An object that inherits a database class, for example, is a database object forever; it should not then be transformed into an object that interfaces with MongoDB or a cache, for example (though it may contain an object that does interface with these protocols).

Inheritance is useful for things that are like one another. For example, a soda can is an object (Soda) which extends from a broader type (Can). Most cans share the same properties, whether they are soda cans, bean cans or cat food cans; they are made of a kind of metal, have a lid, and contain something inside them (which would be an object all its own). Therefore, it makes sense that the same shared methods and properties be available in the can, even if the way the can looks at the end is very different.

Inheritance Only Goes So Far

Unfortunately, inheritance only goes so far in development. For example, it makes great sense that a generic database object be extended to implement specific database types, but it is a terrible methodology for other more complex operations.

Considering the example we had in the last chapter for StringPrinter, inheritance was impossible due to the private methods defined within the class. However, with composition, it would be possible to achieve a similar outcome that is far more flexible. Consider the following code sample (thanks to Anthony Ferrara for the sample[14]):

```
<?php

interface Printer {
    public function printString();
}
class StringPrinter implements Printer {

    private $_string;

    public function __construct($string = null) {
        if($string) {
            $this->setNewString($string);
        }
```

[14] http://bit.ly/UzNtyW

```
    }

    public function printString() {
        print $this->_string;
    }

    public function setNewString($string) {
        $this->_setString($string);
    }

    private function _setString($string) {
        $this->_string = $string;
    }

}
class PrePrinter implements Printer {
    private $printer;
    public function __construct(Printer $printer) {
        $this->printer = $printer;
    }
    public function printString() {
        print '<pre>';
        $this->printer->printString();
        print '</pre>';
    }
}
class StripTagsPrinter implements Printer {
    private $printer;
    public function __construct(Printer $printer) {
        $this->printer = $printer;
    }
    public function printString() {
        ob_start();
        $this->printer->printString();
        $string = ob_get_clean();
        print strip_tags($string);
    }
}

?>
```

There is an obvious advantage here. To start, it is possible to use the StringPrinter interface and objects that implement it in a number of ways:

```php
<?php

$printer = new StringPrinter('<br>foo');
$printer->printString(); // "<br>foo"

$pre = new PrePrinter($printer);
$pre->printString(); // "<pre><br>foo</pre>"

$strip = new StripTagsPrinter($printer);
$strip->printString(); // "foo"

$preStrip = new PrePrinter(new StripTagsPrinter($printer));
$preStrip->printString(); // "<pre>foo</pre>"

?>
```

In fact, this interface is almost infinitely configurable to meet the needs of future implementations. This offers a tremendous advantage, because implementing the interface allows future developers to devise their own methods of displaying strings, without being bound to the string manipulations performed by other developers, but still allows those manipulations to be utilized.

So Which Methodology Is Correct?

It might seem unusual to have essentially two conflicting viewpoints on display in the same book. Unfortunately, software development is as much art as it is science, if not more so. It’s up to individual developers to decide what approach is best; just be sure to remain as consistent as possible.

The approach outlined here is just as valid as the approach outlined in the previous chapter; it ultimately comes down to use case. There’s an argument to be made that private methods are good and that inheritance is harmful; there’s an argument to be made the other way. As a developer, you are responsible for determining which tool in your tool kit makes the most sense, and implementing it.

Composition Over Inheritance

Composition is almost always preferred over inheritance for the simple fact that it’s far more extensible in a wider range of ways. Though it will not make sense each and every time, it will make sense a lot of the time and it is the methodology I encourage for developing your own libraries.

When using other people’s libraries, inheritance will probably be your only option in many cases. Understanding how to do both and the benefits of each will serve you well as a developer.

Chapter 4: Using Traits In PHP 5.4

Beginning in PHP 5.4, developers have the opportunity to use traits[15]. Traits allow for code to be grouped based on behavior and integrated into objects. The trait reduces the complexities associated with multiple inheritance chains.

Traits offer some of the benefits of multiple inheritance without actually implementing multiple inheritance.

Traits are **expressed** in classes.

Using Traits In Objects

Traits can be incorporated into objects, and multiple traits can be incorporated into a single object. This allows developers to create various traits and insert them into objects when necessary, especially in situations where a single base class wouldn't be appropriate.

[15] http://php.net/manual/en/language.oop5.traits.php

However, traits are not typehintable[16]; we therefore cannot use the traits to typehint for objects that contain those types of traits. Proper use of interfaces is therefore required, which provides a convenient method for typehinting objects.

Writing a trait

Traits are defined with the “trait” keyword and are defined essentially like classes. For example:

```
<?php

trait MyTrait {

    function myFunction () {
        // do something here
    }

    abstract public function otherFunction();

}

?>
```

Note that we have included an abstract function in the definition of the trait. Because traits cannot be instantiated directly, we do not need to specifically declare the trait as abstract. The abstract function definition is probably redundant, because an interface could also enforce the function definition; however, by defining it we ensure that any object implementing the trait also implements the otherFunction() method, even if it doesn’t implement a specific interface.

[16] http://php.net/manual/en/language.oop5.typehinting.php

Using traits to define properties

Traits can be used to define properties. They can be used to define properties of any visibility, including private properties; therefore, we can abstract the definition of the property to the trait:

```
<?php

trait MyTrait {
    public $myProperty = true;

    function myFunction () {
        // do something here
    }

    abstract public function otherFunction();

}

?>
```

When defining properties in a trait, be careful not to redefine the same property in the class with a different visibility; if you do, you'll receive a fatal error:

```
Fatal error: MyClass and MyTrait define the same property
($myProperty) in the composition of MyClass. However, the
definition differs and is considered incompatible.
```

If you redefine a property that is in a trait and use the same visibility settings, you'll receive an E_STRICT warning.

Traits and inheritance

Traits can be used in classes that inherit from other classes. Traits are designed to override methods in parent classes; however, if a method is redefined in a class that exhibits the trait, the exhibiting class method overrides the trait.

Trait conflict resolution

The PHP interpreter does not make assumptions about how to resolve conflicts between traits. Therefore, if a method is defined in multiple traits, and those traits are all included in a class, the PHP interpreter will exhibit a fatal error.

```php
<?php

trait A {

    function testFunction() { echo 'hello'; }

}

trait B {
    function testFunction() { echo 'world'; }
}

class TestClass {
    use A, B;
}

$o = new TestClass;
$o->testFunction();
//Fatal error: Trait method testFunction has not been
applied,
// because there are collisions with other trait methods on
// TestClass

?>
```

However, PHP does allow you to define exactly how to resolve these conflicts. You can define which trait to use over another trait; you can also rename specific methods that you wish to use.

```
<?php

trait A {

    function testFunction() { echo 'hello'; }

}

trait B {
    function testFunction() { echo 'world'; }
}

class TestClass {
    use A, B {
        A::testFunction insteadof B;
        B::testFunction as world;
    }
}

$o = new TestClass;
$o->testFunction();
echo ' ';
$o->world();

?>
```

It’s important to know that you must specify the resolution to the conflict for testFunction(); you cannot simply redefine the function names for both A and B without resolving the conflict.

Don’t confuse yourself with traits

Traits offer a tremendous amount of flexibility and power. With that flexibility and power comes the possibility of terribly confusing yourself and other developers with obfuscated code and complex code paths.

For example, traits can be composed of other traits. Though this might seem to be useful, it can also create terribly complex code paths. This is a feature that should be used sparingly.

You can also redefine visibilities with traits. The visibility defined in a trait can be redefined when the trait is expressed in a class. This offers a tremendous amount of flexibility, but could confuse other developers who expect certain visibility rules based on a reading of the code (or expressions of the trait in other areas of the code).

Chapter 5: When Magic Methods Strike Back

Quick quiz question: what methods automatically exist when you instantiate the following object?

```
<?php
class MyObject {
}
$obj = new MyObject();
?>
```

Easy quiz, right? There are no defined methods in that object, so the object contains no methods. But that answer is wrong! There are actually a lot of methods in that object.

These automatic methods, or “magic methods” as PHP refers to them, come into being automatically. Most developers have used at least one magic method: __construct(). But there are, in fact, fourteen magic methods that automatically attach themselves to every object we create.

If we were to implement these methods in an empty object, our object would look something like this:

```
<?php

class onlyMagicMethods {

    public function __construct() {
        // Returns the instantiated object
    }

    public function __destruct() {
        // Run right before the object is destroyed
    }

    public function __call($name, array $arguments =
                            array()) {
        // Called when accessing an inaccessible method in
        // an object
    }

    public static function __callStatic($name, array
                                   $arguments = array()) {
        // Called when accessing an inaccessible static
        // method in an object
    }

    public function __get($name) {
        // Governs the return of inaccessible properties
        // in the object
    }

    public function __set($name, $value) {
        // Governs the setting of inaccessible properties
        // in the object
    }

    public function __isset($name) {
        // Returns true if the property is set; false if
        // not.
    }

    public function __unset($name) {
        // Invoked when unset() is used on a property.
    }

    public function __sleep() {
        // When an object is serialized, this code is
        // executed.
    }

    public function __wakeup() {
        // Used when an object is unserialized
    }

    public function __toString() {
        // Used when an object is cast to a string, by
        // print, echo or (string)
    }

    public function __invoke() {
        // Used when an object is used like a function.
        // Used mainly in anonymous functions.
    }

    public static function __set_state(array $properties =
                                      array()) {
        // Called by var_export(), and can be used to set
        // state on an object
    }

    public function __clone() {
        // Called when an object is cloned with
        // $obj2 = clone    $obj1
    }
}

?>
```

The Challenge In Using Magic Methods

Magic methods are called invisibly by PHP; many of these methods are not explicitly defined in our objects, but serve an important purpose. As a result, we often ignore that they exist, unless they are needed.

The danger in this is that magic methods can be abused by well-intentioned developers. Because magic methods are, by definition, “magical”, they can be implemented but their behavior is invisible to other developers utilizing that code.

The perfect example of this is the construct method that contains more than simple object setup, but performs actual work in the object. Utilizing the construct method in this way may seem to be innocuous, but for an application developer who wishes to have certain behavior elsewhere in their code, this sort of behavior in the construct method can be disastrous (not to mention impossible to test).

Let’s talk about each of the methods, how they are used and how they should not be used.

The __construct() Method

The __construct() method is the most well-known of all the methods. This method is used to define items in the newly created object, or to run code inside the object at instantiation time.

This method should not be used, however, for any actual work of the object. It should only be used for object configuration.

This method cannot have a return value; the object itself is returned when this method is executed.

The __destruct() Method

When the object is destroyed by going out of scope or at the conclusion of run time, this method is called. This method is useful for a number of things, and I have used it most often for caching. For example, if I wish to cache whatever data was in the object before destroying it, this is possible using the __destruct() method.

The method is also useful for closing database or other connections, which can free up resources. However, it is imperative that this method be used intentionally and with care. Developers expect a destroyed object will not continue to interact with other aspects of the application. Since __destruct() provides an opportunity to execute code that could interact with the application, it is important to exercise care.

This method does not have a return value.

The __call() and __callStatic() methods

When accessing a protected or private method that cannot be accessed from the present level, these methods are called. By default this method is undefined. When an inaccessible method is called, PHP emits an error. From time to time, these methods may be useful for allowing access to certain methods outside the rules.

The best use of __call() and __callStatic() is to access dynamic methods that may or may not exist in defined code.

The __set() and __get() Methods

Many developers opt to protect their class properties.These methods can be used to govern access to these methods. These methods can also be used to validate certain properties. For example:

```
<?php

class specialClass {
    protected $string;
    protected $integer;

    public function __set($name, $value) {
       if($name == 'string' && is_string($value)) {
           $this->string = $value;
           return;
       }

       if($name == 'integer' && is_integer($value)) {
           $this->integer = $value;
           return;
       }

       throw new Exception($value . ' is of type ' .
                    gettype($value) . ' and is unsitable
                    for ' . $name);
    }
}

$obj = new specialClass();
$obj->string = 1; // Exception: 1 is of type integer
                  // and is unsitable for string

?>
```

The __isset() and __unset() Methods

The __isset() method is automatically called when isset() is used on a property. I have never personally had a need to extend this method, though it is possible to do so.

The __unset() method is called when the unset() function is called on a property. It is possible to override this and prevent or manage the behavior of unsetting a property; for example, you may wish to also unset all the related properties of a database connection if the main connection is also unset.

The sleep() and wakeup() Methods

These methods are crucial for managing the serialization of objects, and I have used them extensively.

When serialize() is called on an object, it checks to see if __sleep() has been defined. This is the developer's opportunity to clean up any resources, connections and outside items. For example, you may wish to unset the properties that relate to external objects (which would also be serialized), as well as disconnect any database connections.

Then, when the object is unserialized with unserialize(), the __wakeup() method is executed. Most developers can use this to reverse the process executed in __sleep() and reestablish connections, resources and external object resources.

```php
<?php

class makeMeSleepAndWake {
    protected $pdo;

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

    protected function _connectPDO() {
        $pdo = new PDO();
        $this->pdo = $pdo;
    }

    public function __sleep() {
        $this->pdo = null;
    }

    public function __wakeup() {
        $this->connectPDO();
    }
}

?>
```

The Standard PHP Library (SPL) has largely replaced these magic methods with a standard interface called Serializable. The value of the Serializable interface is that it can be typehinted, which allows your objects to know if they can serialize other objects. The Serializable interface is implemented like this[17]:

```
<?php

interface Serializable {

public function serialize();

public function unserialize($serialized_string);

}

?>
```

There is more information on interfaces and typehinting in Chapter 6.

The __toString() Method

This method has been used in many templating systems and frameworks. When an object is cast to a string normally, the following error is produced:

[17] http://php.net/manual/en/class.serializable.php

```
Catchable fatal error: Object of class stdClass could not
be converted to string
```

This obviously makes it impossible to simply output an object as a template in a framework or templating language. However, the __toString() method allows the developer to define a methodology for the object to create a string that is returned, and can be printed.

```
<?php

class StringObject {
    protected $str = 'default string';

    public function setString($string) {
        $this->str = $string;
    }

    public function __toString() {
        return $this->str;
    }
}

$obj = new StringObject();
$obj->setString('Hello world!');
print($obj); // Hello world!

?>
```

The __invoke() Method

Introduced in PHP 5.3, this method exists to make it possible to access objects as though they were functions.

```
<?php

class functionObject {

    public function __invoke($string) {
        echo $string;
    }
}

$obj = new functionObject();
$obj('hello world'); // hello world

?>
```

This function is extremely useful, and we discuss it in greater detail in Chapter 7.

The __setState() Method

It is possible to export objects in PHP, and what you get when you use var_export() looks like this:

```
<?php

class myClass {
    protected $param1 = 'abc';
    protected $param2 = 'def';
    protected $param3 = 'hgi';
}

$obj = new myClass();
var_export($obj);

?>

myClass::__set_state(array( 'param1' => 'abc', 'param2' =>
'def', 'param3' => 'hgi', ))
```

Were you to take the output of var_export() and run it through eval(), you would end up with a fully configured object called myClass() that had the property definitions that are defined in the var_export() string.

This is an alternative to serialization, though it cannot export resources and it also cannot handle circular relationships within objects or arrays.[18]

The __clone() Method

From time to time, developers need to make an exact copy of the object they are working with. I have done this often with DateTime objects, when I needed to execute multiple operations on the same date but didn't want to change the object itself.

When an object is cloned, this method is invoked. Most often, it is used to prohibit cloning an object such as in cases of the Singleton pattern, but it can also be used to great effect to reset database connections or clear resources immediately before cloning.

Magic Methods Are Very Powerful

By providing hooks in the object model, the developers of the PHP core gave application developers a tremendous amount of power in how their objects respond to certain events. This power comes with a tremendous amount of responsibility, since objects behaving in "non-standard" ways holds the potential to confuse other developers, not to mention yourself six months from now. Still, magic methods provide an opportunity to implement some of the best features of PHP applications.

When using magic methods, their use and presence should be clearly documented. Their behavior should be well understood and obvious to developers interacting with the code. The easiest way to do this is with a

[18] http://php.net/manual/en/function.var-export.php

document block which describes the behavior and function of the overridden magic method.

```php
<?php

/**
 * Allows an object to go to sleep after disconnecting the
 * database connections and closing the file resources
 * completely
 */
?>
```

This documentation block makes clear what the __sleep() method is expected to do, and allows future developers to quickly understand how to interact with this object.

Chapter 6: Typehinting Your Way To Success

Since the beginning, PHP has always been a “loosely typed” language. What this means that most any type could fairly easily be converted or treated like any other type. This makes perfect sense when dealing with a web language, where all POST and GET data is treated as a string; as a result, PHP has always lacked the sort of typehinting that Java and other languages offered to specifically hint for a particular data type at runtime.

But when PHP introduced it’s new object model in PHP 5, there was the option to typehint on objects.[19] Since objects specifically possess a certain type, specific to their class and parent classes, it’s possible to definitely enforce which type is being passed around. This offers the object oriented developer an advantage over functional developers.

[19] http://php.net/manual/en/language.oop5.typehinting.php

What Typehinting Is And How It Works

Typehinting is essentially a method of identifying the type of object that is being passed into a method. Most of us already do this to some extent manually when evaluating data that's been passed in from a webform: we examine it to make sure that it contains all numeric characters for a zip code or we ensure that it matches the format for a phone number. But typehinting happens at the PHP parser level, and it is data validation for developers.

Typehints precede variable declarations in the method signature or function signature in objects or functional programs. For example:

```
<?php

class MyClass {
    public function __construct(MyOtherClass $moc) {
        // Do something here
    }
}

?>
```

Notice that in the __construct() method, I have placed the word “MyOtherClass” immediately preceding the $moc variable declaration. When PHP instantiates this object and executes the constructor, if $moc is defined to be any data type or object other than MyOtherClass, a fatal error will be produced.

This is useful because knowing in advance what object type will be passed in allows us to ensure that certain APIs are available to us. For example, if you know that MyOtherClass has a method called printThisString(), you can reliably count on MyOtherClass::printThisString() being available when you typehint for MyOtherClass. If typehinting did not exist and the method accepted an object of any type, developers could run into a scenario where the expected API was not present, and a fatal error could result.

The Role of Inheritance On Object Types

It's important to understand the role that inheritance plays on object types. Objects that inherit from other objects have multiple types, as many types as they have ancestors. For example:

```
<?php

class MyClass {

}

class MySecondClass extends MyClass {

}

class MyThirdClass extends MySecondClass {

}

$obj = new MyThirdClass();

var_dump(($obj instanceof MyThirdClass));  // true
var_dump(($obj instanceof MySecondClass)); // true
var_dump(($obj instanceof MyClass));       // true
var_dump(($obj instanceof stdClass));      // false

?>
```

Because $obj was an instance of MyThirdClass, which inherited from two parents, it automatically became an instance of those other classes. So, if we were typehinting for MySecondClass, the object would pass:

```php
<?php

$obj = new MyThirdClass();

function testFunction(MySecondClass $obj) {
    print get_class($obj);
}

testFunction($obj); // MyThirdClass

?>
```

Even though the object reports being of class MyThirdClass, the typehinting engine permits it to pass as a MySecondClass object, because of its inheritance.

Traits in PHP 5.4 are not considered class types, and typehinting will not work on a trait (though you can test for traits with a class_uses() function).

Using Typehints To Enforce API Constraints

Typehinting is most useful when enforcing a particular API design onto a particular object. For example, you may want to enforce a particular set of database methods onto a data object, and you can use a typehint to do this:

```php
<?php

class MyDataObject {

    public function __construct(DatabaseObject $dbo) {

    }

}

?>
```

The enforcement of the particular object type allows you to expect certain methods and rely upon their definition, as long as they remain part of the defined object.

When typehinting for objects that extend other objects, it is important not to typehint too far up the inheritance tree, but instead only typehint for the API required. For example, many databases contain similar behaviors; if your application supports both PostgreSQL and MySQL, you want to typehint on the lowest common denominator, or BaseDatabaseObject rather than on MysqlDatabaseObject or PostgresqlDatabaseObject. Typehinting on BaseDatabaseObject allows you to enforce the API defined in the base object, but use either MySQL or Postgres.

Using Interfaces In Typehinting

PHP allows the definition of interfaces. These interfaces are not classes, and they cannot be instantiated directly. Interfaces contain only a definition of a public method's signature and the public API for a class that implements the interface. They contain no implementation details, and cannot define protected or private methods.

Interfaces are created differently from classes; they must be designated as interfaces, rather than as classes. For example:

```
<?php

interface DatabaseInterface {

    public function query($sql);

    public function cleanData($data);
    public function get($sql, $args);

    public function set ($sql, $args);

    public function delete ($sql, $args);

}

?>
```

Interfaces are implemented in classes, and unlike extension, which only permits a single parent class, multiple interfaces may be implemented in a given class.

```
<?php

class BaseDatabaseObject implements DatabaseInterface,
   SecondInterface {
    // All the methods required defined here
}
```

Interfaces require that all defined methods be defined and fleshed out in objects that can be instantiated. A fatal error is produced if an object does not define a method required by the interface.

Interfaces can be typehinted by PHP. This is the true power of defining an interface: any and all objects that implement that particular interface will be types of that interface. This allows you to define common APIs for objects that may implement a given interface, and then typehint for the interface alone and ensure that methods are available.

Because interfaces are essentially the lowest level that can be defined and typehinted against, they are effective ways to ensure that the objects passed into a function or method are consistent in their API, even if they are varied in their specific implementation details.

The Benefits of Typehinting

Typehinting allows a developer to enforce constraints on their code, which overall improves the interoperability of the code. By enforcing constraints, especially in situations involving dependency injection, developers can know in advance the API that will be included, and utilize that API, even when the underlying code may not yet be developed (especially useful in cases where other developers implement specific APIs, such as in plugins or data object cases).

Chapter 7: Abstraction Is Your Friend

One Object, One Job (No More, No Less)

We've all seen it (or developed it): The Class To End All Classes. It's got database functions, caching functions, display functions, string conversion functions, it does it all and it's awesome! Or is it?

The most common mistake that introductory object oriented developers make is that they assume they need to package their object oriented code in a similar fashion to the way they wrote their functional code. They look at the object as being responsible for displaying an HTML table from two database tables, and they automatically assume that the object must be capable of making the query, handling the string wrangling and displaying the HTML, all in the same object.

This is wrong.

The purpose of object oriented programming is to provide a methodology for objects to interact with one another in an encapsulated manner.

In object oriented development, there is a principle known as the **single responsibility principle.**[20] Boiled down to its most basic, it essentially says that an object should have one and only one responsibility, and that any additional responsibilities should be handed out to other objects. Put another way, the Class To End All Classes shouldn't query the database and do the string wrangling and handle the display logic. In fact, it probably shouldn't even exist, it's responsibilities properly split out between other objects.

Some developers note (correctly) that this often results in more complicated architectures and a larger number of objects and/or code. Yet the argument that there are more objects assumes that more objects is a bad thing; more objects is not necessarily a bad thing from a development standpoint. When developing objects in object oriented programming, what separates basic developers from great developers is understanding just how many objects to develop, and precisely how to use them for maximum effect. Great developers aren't afraid of more objects when they're necessary; only when they're unnecessary.

The Ties That (Don't) Bind

When developers start using more than one object to accomplish a single task, it is inevitable that those objects eventually must interact with each

[20] http://en.wikipedia.org/wiki/Single_responsibility_principle

other in some way. And so, it’s not uncommon to see classes that look like this:

```
<?php

class MyDataObject {

    protected $_dbo;

    public function __construct(DatabaseDriver $dbo) {
        $this->_dbo = $dbo;
    }

    public function addRow(array $args = array()) {
        $this->_dbo->insertRow($args);
    }
}

class DatabaseDriver {

    public function insertRow(array $args = array()) {
        // functional behavior here
    }
}

?>
```

This would seem to honor the single responsibility principle by breaking out the actual insertion component from the valiadation and management component. Still, we can improve this design.

If you were to try and write unit tests for the MyDataObject class, you could not do *without* including the DatabaseDriver object (and, for that matter, the database). Wherever MyDataObject goes, so does DatabaseDriver; they are *tightly coupled*.[21]

[21] http://en.wikipedia.org/wiki/Loose_coupling

This is where interfaces really shine. Interfaces allow for the decoupling of objects by permitting the interface to be reused in objects that behave totally different ways from one another. Imagine if we had an interface for DatabaseDriver, called DatabaseDriverInterface:

```
<?php

interface DatabaseDriverInterface {

    public function insertRow(array $args = array());

}

?>
```

Now we have the ability to create multiple versions of DatabaseDriverInterface, which implement the interface and behave in functionally different ways:

```
<?php

class DatabaseDriver implements DatabaseDriverInterface {

    public function insertRow(array $args = array()) {
        // functional behavior here
    }
}

class DatabaseDriverTest implements DatabaseDriverInterface
{
    public function insertRow(array $args = array()) {
        // return something that PRETENDS we did an insert
        // but doesn't
    }
}

?>
```

The first class is an actual functional class to interact with our database while the second one is a functionally different class designed to allow testing. This loosely couples our MyDataObject to the DatabaseDriver object, and allows interchangeability.

Applied in real life, this might look something like this for a caching class:[22]

```
<?php

interface BugzillaCacheInterface
{

    public function set($key, $value, $ttl = 300);

    public function get($key);

    public function expire($key);

}

/**
 * Dummy cache for those times you want to test the
functionality
 * WITHOUT the cache
 */
class BugzillaCacheDummy implements BugzillaCacheInterface
{

    public function set($key, $value, $ttl = 300)
    {
        return true;
    }

    public function get($key)
    {
        return;
    }

    public function expire($key)
```

[22] https://github.com/mozilla/mediawiki-bugzilla/tree/master/cache

```
    {
        return true;
    }

}

/**
 * APC cache implementing the cache interface
 */
class BugzillaCacheApc implements BugzillaCacheInterface
{

    public function set($key, $value, $ttl = 300) {
        return apc_store($key, $value, $ttl);
    }

    public function get($key) {
        return apc_fetch($key);
    }

    public function expire($key) {
        return apc_delete($key);
    }

}

?>
```

In this cache, the interface defines how the cache is interacted with by the classes that use it; however for testing purposes (both unit tests and functional tests), the cache can be disabled using the BugzillaDummyCache class; this permits the use of a cache object that does not actually utilize a cache.

Dependency Injection Considered Helpful

When multiple objects are involved, there has to be a way to pass them around so that they can work together with other objects. And so, it would seem to make sense to simply instantiate the objects that we need when we need them:

```php
<?php

class MyClass {

    protected $_database;
    protected $_dataObject;

    public function __construct() {
        $this->_database = new DatabaseObject();
        $this->_dataObject = new DataObject();
    }
}

?>
```

This certainly gives us the objects we need. However, it takes the work we did in creating interfaces and throws it away by tightly coupling our objects together inside the constructor function of MyClass. This is suboptimal.

A better solution is to *inject* the objects into the constructor function as they are needed. This is known as "dependency injection[23]" and is perhaps one of the most hotly debated topics in all the object oriented world. Here's a basic rundown of how dependency injection works.

Rather than instantiating the objects inside the constructor function (or inside the methods that need them), we inject them as arguments. This is where typehinting comes into its own: typehinting is what makes dependency injection so effective.

[23] http://en.wikipedia.org/wiki/Dependency_injection

```
<?php

class MyClass {

    protected $_database;
    protected $_dataObject;

    public function __construct(DatabaseObject $dbo,
                                DataObject $do) {
        $this->_database = $dbo;
        $this->_dataObject = $do;
    }
}

?>
```

This is better, since it injects the objects, but it's still not perfect. To fully take advantage of loose coupling and typehinting, we need to typehint on the lowest common denominator. We built interfaces for these objects, so let's use them:

```
<?php

class MyClass {

    protected $_database;
    protected $_dataObject;

    public function __construct(DatabaseI $dbo,
                                DataI $do) {
        $this->_database = $dbo;
        $this->_dataObject = $do;
    }
}

?>
```

This configuration offers the maximum flexibility with the minimum amount of drag. It fits the test of single responsibility (the dependencies handle additional responsibilities). It also fulfills the requirements of loose coupling, and by using dependency injection we've made it easy to test with unit tests. Of course, it goes without saying that these objects must be instantiated at some point in order to be injected, and that when they are instantiated, the code that creates them will be effectively untestable from a unit testing perspective. This is a reality of object oriented PHP. The goal is not to eliminate the instantiation of objects altogether; it's to maximize the flexibility, testability and purity of the object oriented code that follows.

Models, Views and Controllers, Oh My!

Let's take a closer look at a common "pattern" in the PHP development world. "Pattern" is in quotation marks, because in truth model-view-controller (or MVC) is not truly a pattern, at least in the traditional sense of patterns; it is instead an architecture.[24]

Most frameworks in PHP follow some sort of MVC structure. The theory is simple: the user interfaces with the view, which passes information to a controller. The controller then passes that information to a model, and the model passes information back to the controller. The controller effectively stands between the view and the model.

Simple? Hardly.

Entire books have been written[25] (and will be written in the future) focusing on the MVC architecture. It would be hubris to assume we could cover every

[24] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[25] http://www.phpro.org/tutorials/Model-View-Controller-MVC.html

aspect of this architecture in the scope of this book. Instead, let's focus on a few salient points about the MVC architecture that can be beneficial for our purposes.

Fat Model, Skinny Controller

Since the purpose of the controller is to manage the information that comes from the view and the model, passing information back and forth between the two, it would stand to reason that the controller should contain a majority of the business logic for the application.

Truthfully, this is not the case.

Controllers have a unique position of being the intermediary, but their role is effectively to pass information between the two *with a minimum of interference*. The purpose of the controller is little more than to take data from the view and manipulate it into a format understandable by the model, and then to take model data and transform it into data understandable by the view. The controller's mission is limited completely by the amount of data manipulation that is required.

The purpose of the controller is not to validate the data from the view or to parse the data from the model. The purpose of the controller is *not* to synthesize new data from disparate datasets. The purpose of the controller is not to handle display logic, format data so it can easily be displayed, or to handle templating.

There's a very good reason for this. In the discussion about dependency injection, I admitted that at some point you must instantiate the objects that will ultimately be utilized in the application. More often than not, this takes place in the controller. This makes the controller inherently untestable in many implementations (Zend Framework 1.x was a perfect example). Since the controller was difficult if not impossible to properly test, placing business logic in the controller was a good way to ensure it would never be tested.

Similarly, business logic does not belong in the views (more accurately described as "templates").

Instead, business logic properly belongs in the model, where data is not mapped directly to tables per se, but is synthesized into data objects that represent the overall picture of a particular piece of data. Since the business logic most often belongs here, in the model, the models tend to contain the majority of the code, from validation to manipulation to execution. Hence the expression, "fat model, skinny controller."[26]

A Closer Look At The Model

The majority of the business logic in classes belongs within the model. Since the model contains most of the business logic, the methodology for constructing the model must be a primary focus when architecting a new application.

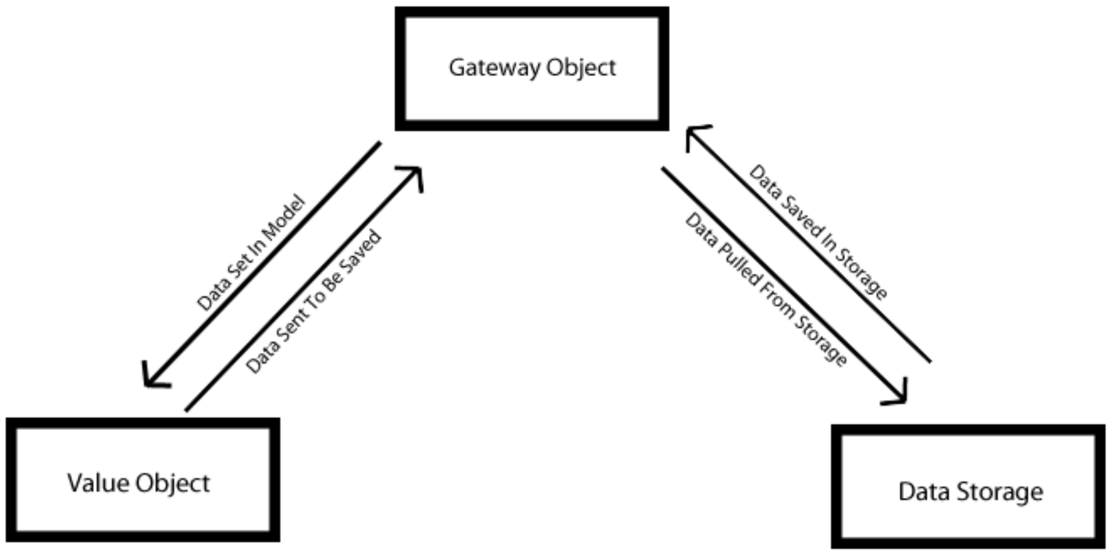
Many model implementations focus on mapping one model to one table in a database. I find these model constructions to be short-sighted and wrong, for a few reasons. First, models are no longer solely built against databases. Second, it ties the flexibility and functionality of a model to the flexibility and

[26] http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model

functionality of its data store. Third, it makes it difficult to disconnect the model from the database should a new data store be introduced later on.

Instead, I prefer to figure out "what is it that I'm modeling here?" This is often referred to as the **domain model,** and it's also the preferred model type of Matthew Weier O'Phinney, primary author of Zend Framework.[27]

Wikipedia defines the domain model thusly: "[The] domain model can be thought of as a conceptual model of a system which describes the various entities involved in that system and their relationships."[28] The domain model looks similar to this:



The primary question that needs to be asked when constructing a domain model is, "what am I modeling?" This is a distinctly different question from "what do my database tables look like?" For example, if you are creating an application for contacts, you are modeling a contact. Meanwhile, your

[27] http://mwop.net/blog/202-Model-Infrastructure.html
[28] http://en.wikipedia.org/wiki/Domain_model

database probably allows for multiple phone numbers, multiple email addresses and multiple notes for each contact. But the structure of the data storage is not a concern when considering the structure of the model itself.

For our contact, there are a few components that need to be included with a contact. First and foremost, a contact has a unique name. They also have some collection of email addresses, phone numbers, and notes. We need to have a way to create and update phone numbers, email addresses and notes. We need to be able to create a new contact, and delete an old contact.

This has nothing to do with the way our database is going to look. Instead, this is only focused on what a contact looks like.

We will probably want to have a way to list contacts, and so this is a different model. A list of contacts should be iterable, and should contain critical data about each contact.

A “Gateway” To Interact

Matthew Weier O’Phinney writes that there should be a “gateway” to interact with what you’re modeling. For our contacts, our gateway might look something like this:

```
<?php

interface ContactGatewayI {

    public createNew(array $arguments = array());

    public updateExisting($uniqueKey, array $arguments =
                            array());
    public deleteExisting($uniqueKey);

    public getList(array $criteria = array());

}

?>
```

The job of the gateway is to essentially allow us to easily access, update and retrieve what we're modeling in an easy and effective way. The gateway stands between the data storage layer and the value object, providing a convenient interface for working with both.

Constructing The Modeled Data

Once we have a gateway for interfacing with our value object, we need to actually construct the value object interface.

```
<?php

class Single_Contact {

    protected $_gateway;
    protected $_data;

    public function __construct($data, $gateway) {
        $this->setGateway($_gateway);

        /** do something here **/
    }

    public function setGateway(ContactGatewayI $gateway) {
        $this->_gateway = $gateway;
    }

    public function getGateway() {
        return $this->_gateway;
    }
    public function save() {
        $gateway = $this->getGateway();
        $gateway->save($this);
    }

    public function getPhoneNumbers() {
        return $this->_data['phone_nunmbers'];
    }

    public function getName() {
        return $this->_data['first_name'] . ' ' .
                    $this->_data['last_name']
    }

}

?>
```

Our data model above has a number of key benefits. First and foremost, it's not tied to any particular data storage engine. The data could be stored in MySQL, Postgres, MongoDB or CouchDB; the value object doesn't care. Second, it's not tied to the gateway by anything but an interface, making it effectively a loosely coupled, stand alone object.

There are a number of ORM[29] (object resource management) tools on the market that effectively do what I just described. The biggest challenge in using such tools is that they often tie your model to a particular data source. As more and more applications make use of data coming from multiple data sources (databases, local NoSQL data stores, APIs, etc.) these ORMs become less practical. Additionally, for large applications it's crucial to understand exactly how the ORM applications will affect scalability and performance.

[29] http://docs.doctrine-project.org/en/latest/reference/architecture.html

Data Validation In The Model

Because the controller is not responsible for data validation, the model must take this role and execute it. So who is responsible for data validation?

The value object is responsible for validating that the data set within it is correct. Attempting to set invalid data should produce an immediate error that is caught and handled by the gateway and passed back to the controller and the view for the user to see.

This is a perfect opportunity to use the __set() magic method. Because we should at a minimum setting the internal data of the value object to protected or private, the __set() magic method is encountered each time we attempt to set a value.

Validation should not be performed in the gateway, since modifying the value object's structure will result in changing the gateway as well: a sure sign of tight coupling and a lack of proper abstraction.

Avoid Making Lasagna

In Chapter 1, I warned against the possibility of producing "spaghetti code" through overuse of procedural programming. We discussed how object oriented programming offers benefits, including the fact that it helps prevent the spaghetti code phenomenon. And in this chapter, we focused on abstraction.

It’s worth noting, then, that it is possible to over-abstract code. This is called producing “lasagna code”, so named for the myriad of layers that are produced by the over-abstraction.

A lasagna is produced when layers of cheese, vegetables and meat are separated by layers of lasagna noodles. Just as a lasagna that had hundreds of layers of pasta noodles would not be delicious, code that has hundreds of small layers is not desirable. This is because the more abstract the code gets, the harder it is to maintain it (the same problem as with spaghetti code). It also requires a fuller domain knowledge than properly abstracted code, reducing the ability of developers to fix bugs without understanding the whole system. Essentially, over-abstraction produces the same results as functional programming: a messy, incomprehensible code base.

Avoiding lasagna code requires intentional effort by a developer. When abstracting code, it’s crucial to determine whether or not the abstraction benefits the code or simply makes it “purer”. Abstracting code for the sake of abstraction is the easiest way to produce lasagna; avoid this when possible.

A Career-Long Process

The core principles here are not principles that ever are fully learned. They are constantly relearned, reconsidered and reevaluated as each developer progresses in their career. The skills of abstraction that a developer uses today may change tomorrow as they approach new problems and have new experiences.

Whole books have been written on these principles alone, and there will probably be new ones written in the future. As a developer it is up to each of us to continue our study of these principles and advance our own knowledge and understanding.

Chapter 8: The Care And Feeding Of Anonymous Functions

Starting with PHP 5.3, developers were able to make use of anonymous functions, or functions defined at runtime within the scope of the application. Definition is easy and fairly standard: the user simply defines the function, assigns it to a variable, and then uses the variable as though it were a function.

```
<?php

$myFunc = function($string) {
    echo $string;
}; // not the closing semicolon

$myFunc('hello world'); // hello world

?>
```

This seems easy enough. PHP also makes it possible to include in-scope variables inside these functions automatically, through the “use” keyword. These are referred to as “closures”.[30]

If we take the anonymous function we defined and we examine it with var_dump(), we see something interesting: our anonymous function isn’t actually a function at all, but an instance of class Closure, a built-in PHP object.

The Closure object makes use of the __invoke() magic method, and effectively turns the code that you provide into the internals of that function within the Closure object. This makes sense: since the function is assigned to a variable, and objects can be assigned to variables, the implementation details were made more simple in this way.

But a user need not extend the Closure class to make use of the __invoke() magic method. Instead, this magic method is available to be overridden in all objects that are created in PHP, and this leads to some very interesting use cases.

One use case that I’ve had is advanced processing of a regular expression, or set of regular expressions, on a given set of text. Take the following example:

```
<?php
class MyClosureExample {
    public function __invoke($matchElements, $subject) {
```

[30] http://php.net/manual/en/functions.anonymous.php

```
        return $this->_complexProcess($matchElements,
$subject);
    }

    protected function complexProcess($matchElements,
$subject) {
        // execute complex process
    }

}

// In another file

preg_replace_callback($pattern, new MyClosureExample(),
$subject);

?>
```

Most experienced developers will say, “wait a minute, we’ve been able to use objects as callbacks for years.” And they would be right: prior to PHP 5.3, you could in fact use an object as a callback. The process was clunky, however, requiring an array containing the object and a string of which method to use:

```
<?php

call_user_func(array($obj, ‘methodString’));

?>
```

In my opinion, this is far less clean that the methodology of simply passing an object that implements the __invoke() method.

Users can also define an anonymous function immediately prior to implementing the preg_replace_callback() function; this too is a valid option. The drawback to this is that it’s far messier to have a complex closure implemented in the midst of complex code. The more complex the code, the more complex the closure is almost certain to be; this only makes the code messy and difficult to understand, not to mention difficult to test independently later on.

My approach suffers from a significant deficiency: the ability to incorporate in-scope variables into the closure directly. This is certainly a drawback and one to consider if you intend to utilize this particular methodology.

This Seems Strange; How Can You Call This A Best Practice?

I'll admit: this is probably one of the more unusual quirks about the way the __invoke() magic method and the closure system in PHP works. Implementing it in this fashion is not without it's drawbacks or limitations. The biggest limitation/drawback is that this is a "clever" solution to the problem at hand.

What outweighs the particular "cleverness" of this solution is the cleanliness of breaking the code out in this fashion, and allowing complex code to live in a file all its own, defined in an object of its own purpose. Whenever a developer comes back to revisit this section (as I did several times when I wrote code that was similar to this), having the code broken out in this way makes maintainability far easier. And ease of maintainability is most certainly a best practice.

Additionally, closures are **not testable** when implemented in line in code. For critical functionality, the lack of testability becomes problematic; when breaking the closures out into their own objects, the ability to test them is much greater.

Whether or not you implement this solution is entirely up to you. For simple functions, I recommend that you continue to define the anonymous function locally.

Chapter 9: The Exception To The Rule

In PHP 5, the exception model was introduced. Before exceptions, the only way of raising an error or halting processing was to use PHP errors; with exceptions, we have a way to alter the flow of a program based on certain conditions.

Exceptions, as their name implies, are for exceptional conditions. These conditions might be missing data, misconfiguration, outright errors or connection problems. Exceptions aren't designed for flow control beyond managing these exceptional conditions (use if-else and switch for this).

The exception is perhaps one of the most misunderstood components in the whole of the PHP language. Let's delve into this area of PHP and explore what exceptions are meant for, how they should be used, and how developers can use them to make their code more effective.

Proper Use of Exceptions

Using exceptions properly is perhaps one of the most difficult aspects of understanding exceptions. Exceptions are for **exceptional situations**. That is, an exception is never expected in normal operations. An exception is a type of error, and it should be used like one.

Exceptions are unique among errors, because they can be caught. For example:

```
<?php

class MyClass {

    public $var;

    public function __construct($var = null) {
        try {
            if(empty($var)) {
                throw new Exception("No var value!");
            }
        } catch (Exception $e) {
            $var = '123';
        }

        $this->var = $var;
    }

}

$obj = new MyClass();
echo $obj->var; // 123

?>
```

The try-catch block here captures the exception of type Exception, and executes a block of code upon encountering that exception. If we left the exception uncaught, we would end up with an error message and the program would halt at the point of the error:

```
<?php

class MyClass {

    public $var;

    public function __construct($var = null) {
        if(empty($var)) {
            throw new Exception("No var value!");
        }

        $this->var = $var;
    }

}

$obj = new MyClass();

?>
```

```
Fatal error: Uncaught exception 'Exception' with message
'No var value!'
```

Exceptions are useful for identifying and alerting the developer to an unexpected condition in an application. This is one of their primary functions.

An Exception For Every Occasion

Observant developers will see that exceptions are PHP objects. In fact, exceptions are a built in type of that are special because they are "throwable" by the PHP interpreter (no other object type can be thrown).

The problem with the base Exception type is that it doesn't provide us very much information. Certainly we could examine the message for types of data, or even the error code (which is perhaps the most underused component of exceptions). But the best way to use exceptions is to extend them.

```
<?php

class CustomErrorException extends Exception {

}

?>
```

By extending an exception, we are able to better understand and utilize the particular kind of exception that is raised (thrown). Exceptions, being merely objects, can be extended to infinity. The only requirements are that the base class of all exceptions extends the Exception built-in.

Because you can extend and have multiple exception types, it is possible to catch multiple exception types. Simply create catch blocks for each type you wish to catch:

```
<?php

class MyClass {

    public function myFunc($a) {
        try {
            if(is_int($a)) {
                throw IsIntException();
            } else if (is_string($b)) {
                throw IsStringException();
            } else {
                // do something here
            }
        } catch (IsIntException $e) {
            // do something here
        } catch (IsStringExceptionn $e) {
            // do something else here
        }
    }
}

?>
```

Abusing Exceptions Is Bad Juju

Because of the ability to define and catch multiple exception types, it becomes obvious that exceptions could be used as a method of controlling the flow of an application. For example:

```
<?php

class MyClass {

    public function BadJuju($a) {
        try {
            // A will either be int or string
            if(is_int($a)) {
                throw IsIntException();
            } else if (is_string($b)) {
                throw IsStringException();
            }
        } catch (IsIntException $e) {
            // Process in some way
        } catch (IsStringExceptionn $e) {
            //  Process in some other way
        }
    }
}

?>
```

Do not do this.

Though this looks similar to the previous code sample, there's a subtle difference: the two exceptions here are expected, while the exceptions in the previous example occur when the function receives something unexpected. They are then used to define how the system is processing the various bits of data. Exceptions are exceptional; they should never be expected.

Honor Abstraction With Exceptions

From time to time exceptions bubble up to the surface of an application. This should be avoided. Exceptions should be caught by the layer that produced them, so that the exception is controlled and managed.

For example, if your database layer throws an exception, the layer that called the database should handle that exception. This does not prevent that layer from raising it's own exception, and so on, all the way up to the controller. However, before the exception reaches the user, it should be handled, and if the exception is fatal, the user should be informed in a polite way, not with an ugly error message (you do run PHP with display_errors = off in production, right?).

Preserving Exceptions For Future Examination

Whenever an exception is raised, it should be logged for further study. It's up to individual developers to determine precisely how they wish to log exceptions, whether they want to log individual exceptions or log exceptions that rise to a certain level. But failing to log exceptions means that there's no way to know what happened.

Exception logging is relatively easy. Exceptions that rise to the top level are automatically logged by PHP's error logger. For exceptions that are handled, this does not occur. It is up to you to log them.

Logging exceptions is made easy by the fact that exceptions all implement the __toString() magic method. Thus, casting an exception to a string is simple and produces a useful error message:

```
<?php

error_log($exception);

?>
```

Whatever you do, do not silence exceptions. Exceptions should never be trapped and ignored like this:

```
<?php

try {
    throw new Execption();
} catch (Exception $e) {

}

?>
```

Exceptions should not be thrown when there is no intention of resolving it. Failing to resolve or handle the exception defeats the purpose of exceptions altogether. Never silence exceptions by trapping them and disposing of them without first processing them in some way.

Exceptions Are Objects For A Reason

The power of exceptions makes them attractive for many types of development, but exceptions should really only be used in object oriented development. It's easy to make the assumption that exceptions should be used anywhere, but this is a mistake. Exceptions are really designed for object oriented programs, and should not be used in functional programming.

Because exceptions are objects, they can be instatiated and treated just like objects. It is therefore possible to execute code similar to this:

```
<?php

$e = new Exception('my message here');
throw $e;

?>
```

Nested Exceptions in PHP

It is possible to nest exceptions in PHP. A nested exception is an exception that is thrown within an exception loop. For example:

```
<?php

class ExceptionA extends Exception {}
class ExceptionB extends Exception {}

try {
    try {
        throw new ExceptionA('message 1');
    } catch (ExceptionA $e) {
        throw new ExceptionB($e->getMessage());
    }
} catch (ExceptionB $e) {
    var_dump($e->getMessage());
}

?>
```

Note the nested try-catch blocks that we have. This provides the ability to raise an exception while handling an exception.

Since PHP 5.3 developers can also pass a prior exception as an argument of the new exception.

```
<?php

class ExceptionA extends Exception {}
class ExceptionB extends Exception {}

try {
    try {
        throw new ExceptionA('message 1');
    } catch (ExceptionA $e) {
        throw new ExceptionB('New Exception', 0, $e);
    }
} catch (ExceptionB $e) {
    var_dump($e->getPrevious()->getMessage());
}

?>
```

This method preserves the stack trace and allows you to iterate on execptions throughout your application. Passing exceptions in this fashion does not (in my opinion) violate the principle of exception layer abstraction.

Exceptions Are Exceptional

When used properly, exceptions are a powerful component of object oriented development. They show when a problem has been encountered, and they help developers have a tremendous amount of control over how errors are handled. Exceptions can be fatal when required, and non-fatal when the problem can be handled. Use exceptions correctly, and they will serve you well.

Chapter 10: Autoloading For Fun And Profit

Anybody who uses PHP for any length of time quickly learns that autoloading is the only real method to avoid insanity with includes and ensure that all the files are available when they are needed (but that excess files are not included when they are unnecessary).

The autoloading functionality in PHP was introduced as far back as PHP 5.1.2, when spl_autoload_register() was implemented.

Using spl_autoload_register()

The spl_autoload_register() function is very easy to use, requiring only one argument.

The signature of spl_autoload_register() is as follows:

```
bool spl_autoload_register ([ callable $autoload_function
[, bool $throw = true [, bool $prepend = false ]]] )
```

The $autoload_function argument is the required argument, and should be a callable function that spl_autoload_register() can use to search for classes that are not presently in scope. Additionally, the $throw argument (default of true) will cause an execption to be raised if the function cannot be loaded. We will discuss the third argument, $prepend, next.

Autoload Order

The PHP documentation describes there being an autoload stack[31], along with describing the behavior of multiple autoloaders as being similar to a queue. This is a bit confusing, especially if you're not familiar with stacks and queues.

When you queue up at the store (get in line), the first person to get in line is helped first. And so it is with queues in programming: the first thing into the queue gets handled first (First In, First Out or FIFO).[32] A stack is similar to stacking a bunch of papers on your desk. When you sort them, the last one to end up on your desk will almost always be the first one you pick up to sort. And so it is with stacks in software development (First In, Last Out or FILO).[33]

It turns out that registering multiple autoloaders works exactly as you'd expect: the first one is executed first, the second one is executed second, and so on, like so:

```
<?php
```

[31] http://php.net/manual/en/function.spl-autoload-register.php
[32] http://en.wikipedia.org/wiki/FIFO_%28computing%29
[33] http://en.wikipedia.org/wiki/LIFO_%28computing%29

```
function autoload1($class) {
    echo 'I was registered first';
}


function autoload2($class) {
    echo 'I was registered second';
}


function autoload3($class) {
    echo 'I was registered third';
}

spl_autoload_register('autoload1');
spl_autoload_register('autoload2');
spl_autoload_register('autoload3');

$obj = new DoesNotExist();

// I was registered first
// I was registered second
// I was registered third
// Fatal error: Class 'DoesNotExist' not found

?>
```

It is possible to force spl_autoload_register() to place a newly registered autoloader at the front, effectively stacking it like the pile of papers on your desk. The third argument in spl_autoload_register() makes this possible.

```
<?php

function autoload1($class) {
    echo 'I was registered first';
}


function autoload2($class) {
    echo 'I was registered second';
}
```

```
function autoload3($class) {
    echo 'I was registered third';
}

spl_autoload_register('autoload1', true, true);
spl_autoload_register('autoload2', true, true);
spl_autoload_register('autoload3', true, true);

$obj = new DoesNotExist();

// I was registered third
// I was registered second
// I was registered first
// Fatal error: Class 'DoesNotExist' not found

?>
```

It is important to carefully consider the order in which you want various autoloaders to execute. In some framework implementations, when an autoloader is unable to load a particular file, they may 404 or otherwise display an error page; in your case, you may wish to autoload items and choose to prepend your autoloader onto the front to ensure that it is utilized first, and an error page is avoided.

Autoloading Strategies

When implementing an autoloader, or series of autoloaders, there must be a relatively standard way to determine where files are located. There are numerous strategies, but let's discuss three that seem to make the most sense.

First, there's the registry autoloading strategy. This strategy has each module register its component classes with a centralized autoloader. The autoloader might be written as such:

```php
<?php

abstract class MyAutoloader {

    protected static $_autoloadList = array();

    public static function register($class, $filePath) {
        if(file_exists($filePath)) {
            self::$_autoloadList[$class] = $filePath;
            return true;
        }

        return false;
    }

    public static function autoload($class) {
        if(isset(static::$_autoloadList[$class])) {
            require_once static::$_autoloadList[$class];
        }
    }
}
?>
```

This methodology allows the registration of individual components and their classes. The benefits of this approach are that by requiring registration, the file structure doesn't need to be specific and it makes it easy to have a pluggable architecture since plugin developers only need to register their classes. Obvious drawbacks include the fact that this method of autoloading requires registration, so should you forget to include a file in the registration process, it will error out. Additionally, it can be difficult to determine where in the file system a particular class lives based solely on its name. Finally, if care is not taken to avoid overwriting a key (there is no key protection in my example), namespace collisions could occur.

Another methodology for autoloading comes from early frameworks including Zend Framework 1.x, and this is to use the name of the class to determine the file path structure. This requires a set file structure, but eliminates the guessing involved and allows developers to quickly figure out where the source code lives for a certain file:

```php
<?php

function autoloader($class) {
    $classPath = str_replace('_', '/', $class) . '.php';
    var_dump($classPath); die;
    if(file_exists($classPath)) {
        require_once $classPath;
    }
}

?>
```

The benefits of this method include the fact that explicit class registration is not required, and that the autoloader can automatically seek and load the class based solely upon the agreed upon file structure. Depending on your rules, you can make this as complex or simple as possible. It’s up to the developer to decide what the rules are, and what the rules of the file system will be, and then to implement them. The major drawback here is that the file system must be organized in a particular fashion as decided by the developer, which can make extensibility of an application more difficult. Additionally, should the file system be rearchitected at some point, there’s a possibility that all old code need to change or that the established rules may no longer apply.

A third method, and one that is gaining popularity now that PHP includes namespacing, is to use the as a way to translate the file system. Namespaces are more complex, especially when mapping namespaces to the filesystem, because namespaces can be aliased; however, it is possible to implement namespaced autoloaders. Here’s an example from Doctrine of how their autoload function is registered (the complete autoloader class would not fit for the book, and the whole class can be seen at https://gist.github.com/221634):

```
<?php

$classLoader = new SplClassLoader('Doctrine\Common',
      '/path/to/doctrine');
classLoader->register();

?>
```

This allows for the definition of a namespace and a path for that namespace, which when registered, permits proper handling of autoloading classes regardless of whether or not the namespace has been aliased.

Chapter 11: Good Object Oriented Citizenship

Through its very nature, object oriented programming is one of the most unique aspects of software development. It's far more different than procedural programming, and it requires a very different mindset from the programming you've likely done in the past.

As part of that thought process, it's important to understand the precise rules that govern good citizenship when developing an object oriented application.

Because of the unique nature of the object oriented world, there are certain rules you should always follow when developing your application and implementing your methods and properties.

These rules are by no means hard and fast, but they are meant to encourage your best behavior in the object oriented programming world.

Avoid static methods and properties

PHP offers developers the ability to implement static methods and properties. These methods are not part of an instantiated object, but instead can be called from any scope and without creating any kind of an object. Static methods essentially serve as namespaced functions, placed into a class and accessed through the class name.

Static methods have often been used for other purposes as well, including the creation of Singleton objects. The Singleton pattern is frowned upon since it's impossible to test effectively and can create problems within an application.

Since static methods and properties cannot be tested or mocked by test objects, they are considered problematic and dangerous to applications that wish to be tested. Additionally, since they are difficult if not impossible to override, they prevent any kind of inheritance to take place.

Always return something

Occasionally I see a developer who simply returns "null" or no value at all in a method once the method has done it's work. For example:

```
<?php

public function mySetterMethod($value) {
    $this->value = $value;
}

?>
```

This method would evaluate to NULL in PHP were you to attempt an evaluation. There's no way for a developer to know that the method has succeeded; no way to know that the method has even run correctly. Even though the code presented above is fairly straightforward and simple, this practice is common even where there is less simple and more complex code.

The practice of not returning a value is dangerous, and it prevents certain behaviors that are to some degree considered beneficial.

The reason that you often return a value is to evaluate what that value was. For example, if the setter succeeded, returning TRUE would allow the developer to evaluate that behavior. A FALSE response would indicate failure (or an exception).

Another option in more simple functions like the one illustrated above is to return the object itself by simply returning $this. By returning $this, you can chain setter calls together:

```
<?php
$obj->setter1($val)->setter2($val2)->setter3($val3);
?>
```

This kind of chaining is useful when using multiple setters to configure or change an object en masse. It's often seen in cache objects, or in object relational mapping packages:

```
<?php
```

```
$db->select()->where('id', DB::LT, '21')->orderBy('date',
DB:DESC)

?>
```

This allows for a SQL-like statement without actually having to write any SQL.

When using method chains (or fluent interfaces), it's important to remember that they create a de facto challenge to code readability. Therefore, they should be avoided in two situations:

- When the return value of the method is another object other than the object initially being acted upon; or
- When the use of the fluent interface would create confusion as to the intent of the developer.

For example, you could theoretically chain objects together with PDO:

```
<?php

$result = $pdo->prepare($sql)->execute($params)-
>fetchAll();

?>
```

In the above example, the PDO object's prepare() method returns a PDOStatement object. This violates the first rule because the PDOStatement object is not the first object. It also violates the second rule by causing a developer unfamiliar with PDO to assume that the $pdo object contains an execute() method and a fetch() method.

Fluent interfaces can be powerful time-savers, but they can also introduce a host of complexity and challenge into your code that is frustrating and difficult to solve. Avoid this, and you will find yourself a much happier developer overall.

Fail quickly, in the constructor if possible

Startups are fond of saying “fail fast.” What they mean is that you should fail quickly so that you can move on to the next big idea, rather than folding entirely when your idea doesn’t pan out at the end. This logic has applications to software development as well.

In complex applications, there are often hundreds of things that take place during the startup or operation of the application. It is therefore important that when things go badly, the failures are fast, obvious and clear as to their origins. This brings us to the point of failing quickly.

In any application, failures will occur. It’s just a fact of life. Databases will be flakey, parameters will be misconfigured, or the application itself could be fed bad data.

When this happens, it’s up to the application to identify and handle these problems, even if that means raising an exception or other means.

The concept of “failing fast” means quickly identifying that the application is unable to meet its intended purpose and terminating quickly. For example, if the user provides bad information that causes the data to be unreadable, there’s no reason the application should continue on, attempting to process the information.

This kind of failure planning often results in failure modes being detectable in the constructor methods of the objects. This is good, because the constructor is executed first and can be the first line of defense against bad or corrupt data.

Developers should think carefully about how they validate their data, and ensure that the data is validated as quickly as possible, that objects are sure they have the resources they need, and the application is able to run. IF these conditions are unmet, the application should fail quickly, and alert the user and the developers to the problem.

Make your application easy to test

An application that is impossible to test will not be tested; it's simply a fact of life. Developers as a group tend not to want to bend over backwards to do boring tasks like write unit tests to begin with; as a result, if they are faced with huge hurdles to clear in order to write unit tests, the tests won't get written.

Developers can help themselves by making the testing process easier. They can do this by improving the overall architecture of their applications.

Smart developers recognize the limitations of their frameworks and do a few things that can help their applications be tested effectively.

Move business and validation logic to the model

Most frameworks utilize a model-view-controller layout. The view and the controller are necesarilly places where lots of objects are either instantiated or used. They are not places that allow for easy dependency injection.

The model is different: it allows for easy injection of objects from the controller. This injection makes the model far easier to test, even if it depends on outside data sources like a database (see Chapter 7 for more on how to construct testable models).

By using the model as a place for validation and business logic, this logic can necessarily be tested. As a result the application becomes far easier to test, even if the controller and the view themselves are not testable.

Create libraries for your application

Many times developers use the same code repeatedly, or are required to integrate their code into an object that has external dependencies that cannot be mocked. When this happens, developers should create libraries that can be integrated into the objects that require the custom code.

A library can be tested more easily than an application. Libraries also have the advantage of reusability. But even in cases when a library will only be used once, by creating a library, the developer has the advantage of a comprehensive and complete code base that can be tested in its component parts, independent from the dependencies of the framework or application as a whole.

Use interfaces to encourage testability

Developers love to make their jobs easier. For many of us (me included), this is why we went into programming: to simplify our problems and automate their solutions. Developers are loathe to write more code than they need to solve any problem, and this often means developers are loathe to use tools like interfaces when perfectly adequate tools such as inheritance are available.

But wise developers also know that avoiding regressions is a crucial component of software development.

Interfaces help testability because they provide a public API for writing tests against. With an interface, it is possible to create mock objects that can be used in the object to be tested. This is more difficult when extending another class.

By using an interface, testability is dramatically improved. The ability to mock objects is critical to testing, and it's critical to the ability of developers to effectively determine whether or not their changes break the application in other ways.

Raise exceptions when asking for something unreasonable

It's important to sanity check the inputs that are given to your objects. When objects receive data that they cannot validate or do not pass the sanity check, it's critical that the object raise an exception right away.

The reason that this point exists is because it is important to fail quickly, but it's also important to alert the rest of the application to the problem. Since exceptions can be caught, they can be handled; data can be corrected, recreated or inserted in a fashion that fulfills the validation and sanity checking requirements.

In contrast, if the application simply permits a return value of null when a request is unreasonable, the requestor becomes responsible for validating the response rather than the requestee validating the request. This is backwards, and wrong.

Raising the exception places the responsibility where it belongs: with the layer that made the request. Even though validation does not occur in this layer, it is the layer that should be responsible for correctly and completely passing along the data and handling the consequences.

Raise exceptions when asking for something reasonable but impossible

Along the same lines of raising validation exceptions, it's also crucial that layers raise exceptions when you ask them to do something that's within their scope, but that they are unable to complete for some reason.

For example, a database layer should raise an exception when a result set is requested but the database is unavailable. It should not simply return a null set, which could indicate that there were no results.

It's critical that these exceptions be descriptive and that they allow for layers to handle them in some meaningful way (or pass them along).

And again it's the responsibility of the layer receiving the request to determine that the request is reasonable and impossible, not the responsibility of the requestor to make this determination.

Log, but log only valuable information

Developers eventually learn to love logs. Log files provide a host of vital information about the state of an application, and they offer tons of valuable information about problems that might arise over time.

There’s a tendency once a developer learns to love logs that they log every little transaction, behavior and method in the application. But this leads to problems over time.

Wise developers are careful to log only the most important information. They also learn how to use logging levels, from debug to error, so that they can filter their logs based on the type of information they’re requesting.

The reason this matters is because information overload can make debugging as difficult as having no information at all. Thus, by logging only the most important information (or limiting the scope of the logs to informational and higher), developers limit the information they collect and have an easier time making sense out of the log data.

Even though it’s compelling to try and log every little detail, you should avoid this and only log the most critical information you need for debugging. It will make your life easier in the long run.

Chapter 12: You, The Master

Over the years, many different people have devised different methods for identifying when mastery takes place. One common rule of thumb holds that you must do something for 10,000 hours before you can be a master. Other rules are less specific, and focus on specific domain knowledge or a given amount of years experience. Whatever the measure, there is one common point between them: mastery takes time, effort and dedication.

Mastery of a subject is not something that takes place instantaneously. Instead, it's something that is slow, and often goes unnoticed by the person who is achieving mastery. Mastery is more than technical ability, but a wisdom that comes through experience and understanding of the subject at hand.

How does anyone achieve mastery, or work towards it? Here are some ways you can work to achieve mastery.

Read code, lots and lots of code.

Software development is primarily about reading existing code and fixing, adding to or rewriting it. Thus, a strong ability to read code contributes to mastery of that code, and mastery of the language.

Reading object oriented code is an art form, and requires several abilities. For example, envisioning the way the code works without compiling it is a crucial skill. Being able to hold variables, properties, methods and objects in the mind and manipulate them as part of the reading of the code takes practice. These are crucial skills for mastery.

Read as much code as possible. Review it, consider how it works, figure out how it works without testing it and then test the hypothesis.

Write even more code.

Software developers get better at writing code by doing it. There is no shortcut. Write lots and lots of code: good code, bad code, whimsical code, silly code, pointless code, code that works, code that doesn't. Writing code allows for the practice of vital skills, and encourages the development of understanding.

I often write code to test a hypothesis, solve a small problem, or just to develop an understanding of how a particular function works. All the code in this book, for example, was written in a text editor and compiled, examined and tested (having syntax errors in the book would be bad!) I tested several functions, examined their behavior and return values, and worked with them to construct the overall content in the book.

So write more code – lots and lots of it. It's the best way to reach mastery.

Have code reviewed by masters

Code reviews offer a great opportunity to learn, especially if the reviewer is already a master in the topic area. Mozilla uses code review extensively to provide an opportunity to find bugs, create learning and help ensure community and staff contributions are of the highest quality. I encourage code reviews for every developer.

Even if a formal code review policy is not possible in a particular company due to internal issues, this doesn't preclude a developer from soliciting code review from someone they respect and trust. Developers can always say to someone else "hey, can you review this?" Most developers are happy to help, and if they are truly masters they will love an opportunity to teach.

Fix bugs in unfamiliar systems

A great way to combine all the skills of mastery is to fix bugs in unfamiliar systems, especially established open source projects with a method for pushing patches upstream.

Fixing bugs in unfamiliar systems combines all the skills by requiring great code reading abilities, fantastic problem solving abilities, the ability to generate new code, and an expectation that the code will be reviewed by another developer in the open source project. Bug fixing is perhaps the most effective way to successfully master a particular subject.

Conclusion

Object oriented programming is often seen as equal parts art and science, and it's clearly obvious as to why. The decision-making process for each developer is different, and each developer brings their unique worldview to the object oriented experience.

Perhaps the flexibility afforded in the object oriented world is a testament to the innovation it allows, by making it possible to embrace several disparate or competing worldviews at the same time, even sometimes in the same application.

It is impossible to nail down a specific set of "must do's" in the object oriented world, but the best practices do not need to be hard and fast rules; the principles and practices described here are agreed upon or thought of as "best" not because they are the only way to accomplish a task, but because they are generally accepted as being the most effective, efficient way to achieve a desired outcome.

Still, as developers, each of us has an obligation to implement the solutions that we think are right for the problems that we are solving. Focusing on ease of testability, ease of maintainability, and ease of scalability are huge components in our thought processes, and are unique for each challenge. The beauty of software development is every problem more often than not has more than one answer. The elegance, grace and beauty of a particular solution is what separates good programmers from great ones.

The best practice in object oriented programming is to always be learning. By subscribing to a philosophy that no code can be perfect, but it can be good enough, and by adhering to a personal rule that says knowledge is valuable, we can improve our own personal programming skills, and hopefully share those skills with others.