PHP 5 Objects, Patterns, and Practice

MATT ZANDSTRA

Apress®

PHP 5 Objects, Patterns, and Practice

Copyright © 2004 by Matt Zandstra

ISBN (pbk): 1-59059-380-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore

Technical Reviewer: Tolan Blundell

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Sofia Marchant

Copy Edit Manager: Nicole LeClerc

Copy Editor: Ami Knox

Production Manager: Kari Brooks-Copony

Production Editor: Janet Vail

Compositor: Susan Glinert Stevens

Proofreader: Sue Boshers

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 8

Some Pattern Principles

Although design patterns simply describe solutions to problems, they tend to emphasize solutions that promote reusability and flexibility. To achieve this, they manifest some key object-oriented design principles. We will encounter some of them in this chapter and in more detail throughout the rest of the book.

This chapter will cover

- *Composition*: How to use object aggregation to achieve greater flexibility than you could with inheritance alone
- *Decoupling*: How to reduce dependency between elements in a system
- *The power of the interface*: Patterns and polymorphism
- *Pattern categories*: The types of pattern that this book will cover

The Pattern Revelation

I first started working in an object-oriented context using the Java language. As you might expect, it took a while before some concepts clicked. When it did happen, though, it happened very fast, almost with the force of revelation. The elegance of inheritance and encapsulation bowled me over. I could sense that this was a different way of defining and building systems. I "got" polymorphism, working with a type and switching implementations at runtime.

All the books on my desk at the time focused on language features and the very many APIs available to the Java programmer. Beyond a brief definition of polymorphism, there was little attempt to examine design strategies.

Language features alone do not engender object-oriented design. Although my projects fulfilled their functional requirements, the kind of design that inheritance, encapsulation, and polymorphism had seemed to offer continued to elude me.

My inheritance hierarchies grew wider and deeper as I attempted to build new classes for every eventuality. The structure of my systems made it hard to convey messages from one tier to another without giving intermediate classes too much awareness of their surroundings, binding them into the application and making them unusable in new contexts.

It wasn't until I discovered *Design Patterns*, otherwise known as the Gang of Four book, that I realized I had missed an entire design dimension. By that time I had already discovered some of the core patterns for myself, but others contributed to a new way of thinking.

I discovered that I had overprivileged inheritance in my designs, trying to build too much functionality into my classes. But where else can functionality go in an object-oriented system?

I found the answer in composition. Software components can be defined at runtime by combining objects in flexible relationships. The Gang of Four boiled this down into a principle: "favor composition over inheritance." The patterns described ways in which objects could be combined at runtime to achieve a level of flexibility impossible in an inheritance tree alone.

Composition and Inheritance

Inheritance is a powerful way of designing for changing circumstances or contexts. It can limit flexibility, however, especially when classes take on multiple responsibilities.

The Problem

As you know, child classes inherit the methods and properties of their parents (as long as they are protected or public elements). We use this fact to design child classes that provide specialized functionality.

Figure 8-1 presents a simple example using the UML.

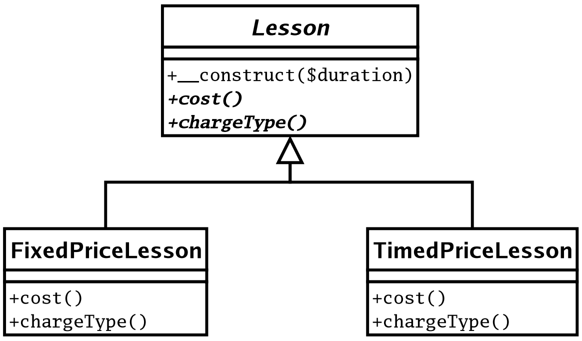

Figure 8-1. *A parent class and two child classes*

The abstract `Lesson` class in Figure 8-1 models a lesson in a college. It defines abstract `cost()` and `chargeType()` methods. The diagram shows two implementing classes, `FixedPriceLesson` and `TimedPriceLesson`, which provide distinct charging mechanisms for lessons.

Using this inheritance scheme, we can switch between lesson implementations. Client code will know only that it is dealing with a `Lesson` object, so the details of costing will be transparent.

What happens, though, if we introduce a new set of specializations? We need to handle lectures and seminars. Because these organize enrollment and lesson notes in different ways, they require separate classes. So now we have two forces that operate upon our design. We need to handle pricing strategies and separate lectures and seminars.

Figure 8-2 shows a brute-force solution.

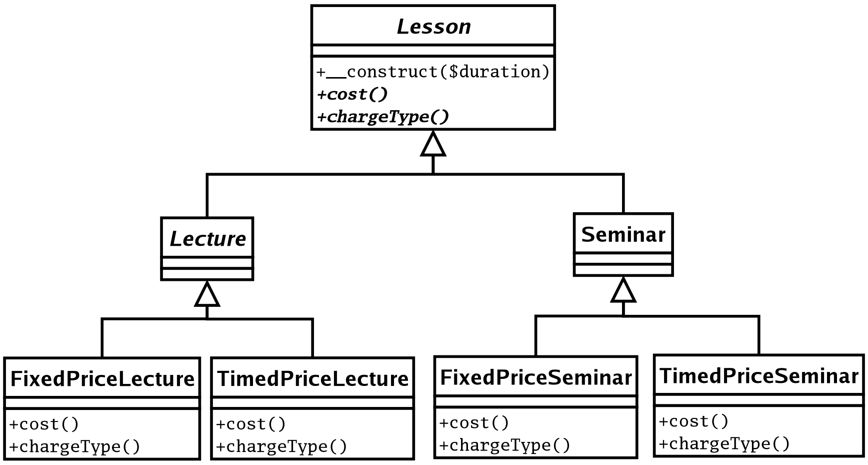

Figure 8-2. *A poor inheritance structure*

Figure 8-2 shows a hierarchy that is clearly faulty. We can no longer use the inheritance tree to manage our pricing mechanisms without duplicating great swathes of functionality. The pricing strategies are mirrored across the `Lecture` and `Seminar` class families.

At this stage, we might consider using conditional statements in the `Lesson` super class, removing those unfortunate duplications. Essentially, we remove the pricing logic from the inheritance tree altogether, moving it up into the super class. This is the reverse of the usual refactoring where we replace a conditional with polymorphism. Here is an amended `Lesson` class:

```
abstract class Lesson {
    protected $duration;
    const     FIXED = 1;
    const     TIMED = 2;
    private   $costtype;

    function __construct( $duration, $costtype=1 ) {
        $this->duration = $duration;
        $this->costtype = $costtype;
    }
    function cost() {
        switch ( $this->costtype ) {
            CASE self::TIMED :
                return (5 * $this->duration);
                break;
            CASE self::FIXED :
                return 30;
                break;
            default:
                $this->costtype = self::FIXED;
                return 30;
        }
    }

    function chargeType() {
        switch ( $this->costtype ) {
            CASE self::TIMED :
                return "hourly rate";
                break;
            CASE self::FIXED :
                return "fixed rate";
                break;
            default:
                $this->costtype = self::FIXED;
                return "fixed rate";
        }
    }

    // more lesson methods...
}

class Lecture extends Lesson {
    // Lecture-specific implementations ...
}

class Seminar extends Lesson {
    // Seminar-specific implementations ...
}
```

You can see the new class diagram in Figure 8-3.

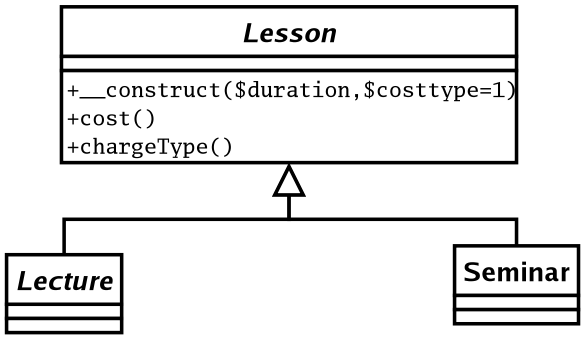

Figure 8-3. *Inheritance hierarchy improved by removing cost calculations from subclasses*

We have made the class structure much more manageable, but at a cost. Using conditionals in this code is a retrograde step. Usually, we would try to replace a conditional statement with polymorphism. Here we have done the opposite. As you can see, this has forced us to duplicate the conditional statement across the `chargeType()` and `cost()` methods.

We seem doomed to duplicate code.

Using Composition

We can use the Strategy pattern to compose our way out of trouble. Strategy is used to move a set of algorithms into a separate type. In moving cost calculations, we can simplify the `Lesson` type. You can see this in Figure 8-4.

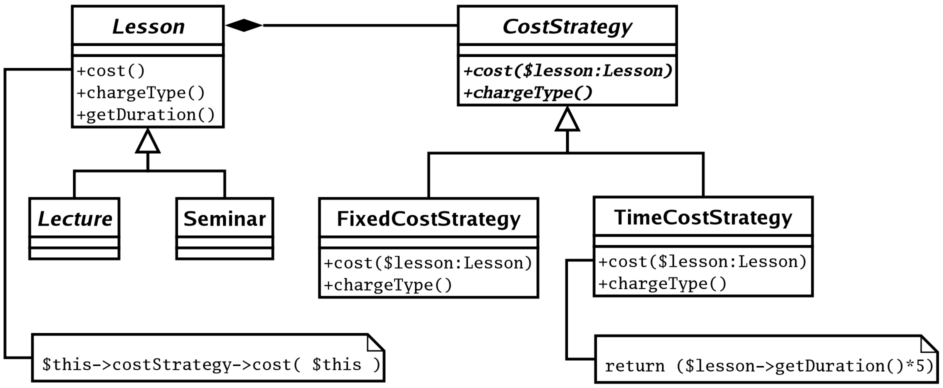

Figure 8-4. *Moving algorithms into a separate type*

We create an abstract class, `CostStrategy`, which defines the abstract methods `cost()` and `chargeType()`. The `cost()` method requires an instance of `Lesson`, which it will use to generate cost data. We provide two implementations for `CostStrategy`. `Lesson` objects work only with the `CostStrategy` type, not a specific implementation, so we can add new cost algorithms at any time by subclassing `CostStrategy`. This would require no changes at all to any `Lesson` classes.

Here's a simplified version of the new `Lesson` class illustrated in Figure 8-4:

```
abstract class Lesson {
    private   $duration;
    private   $costStrategy;

    function __construct( $duration, CostStrategy $strategy ) {
        $this->duration = $duration;
        $this->costStrategy = $strategy;
    }

    function cost() {
        return $this->costStrategy->cost( $this );
    }

    function chargeType() {
        return $this->costStrategy->chargeType( );
    }

    function getDuration() {
        return $this->duration;
    }

    // more lesson methods...
}
```

The `Lesson` class requires a `CostStrategy` object, which it stores as a property. The `Lesson::cost()` method simply invokes `CostStrategy::cost()`. Equally, `Lesson::chargeType()` invokes `CostStrategy::chargeType()`. This explicit invocation of another object's method in order to fulfill a request is known as delegation. In our example, the `CostStrategy` object is the delegate of `Lesson`. The `Lesson` class washes its hands of responsibility for cost calculations and passes on the task to a `CostStrategy` implementation. Here it is caught in the act of delegation:

```
    function cost() {
        return $this->costStrategy->cost( $this );
    }
```

Here is the `CostStrategy` class, together with its implementing children:

```
abstract class CostStrategy {
    abstract function cost( Lesson $lesson );
    abstract function chargeType();
}
```

```
class TimedCostStrategy extends CostStrategy {
    function cost( Lesson $lesson ) {
        return ( $lesson->getDuration() * 5 );
    }
    function chargeType() {
        return "hourly rate";
    }
}

class FixedCostStrategy extends CostStrategy {
    function cost( Lesson $lesson ) {
        return 30;
    }

    function chargeType() {
        return "fixed rate";
    }
}
```

We can change the way that any `Lesson` object calculates cost by passing it a different `CostStrategy` object at runtime. This approach then makes for highly flexible code. Rather than building functionality into our code structures statically, we can combine and recombine objects dynamically.

```
$lessons[] = new Seminar( 4, new TimedCostStrategy() );
$lessons[] = new Lecture( 4, new FixedCostStrategy() );

foreach ( $lessons as $lesson ) {
    print "lesson charge {$lesson->cost()}. ";
    print "Charge type: {$lesson->chargeType()}\n";
}

// output:
// lesson charge 20. Charge type: hourly rate
// lesson charge 30. Charge type: fixed rate
```

As you can see, one effect of this structure is that we have focused the responsibilities of our classes. `CostStrategy` objects are responsible solely for calculating cost, and `Lesson` objects manage lesson data.

So, composition can make your code more flexible because objects can be combined to handle tasks dynamically in many more ways than you can anticipate in an inheritance hierarchy alone. There can be a penalty with regard to readability, though. Because composition tends to result in more types, with relationships that aren't fixed with the same predictability as they are in inheritance relationships, it can be slightly harder to digest the relationships in a system.

Decoupling

We saw in Chapter 6 that it makes sense to build independent components. A system with highly interdependent classes can be hard to maintain. A change in one location can require a cascade of related changes across the system.

The Problem

Reusability is one of the key objectives of object-oriented design, and tight-coupling is its enemy. We diagnose tight coupling when we see that a change to one component of a system necessitates many changes elsewhere. We aspire to create independent components so that we can make changes in safety.

We saw an example of tight coupling in Figure 8-2. Because the costing logic was mirrored across the `Lecture` and `Seminar` types, a change to `TimedPriceLecture` would necessitate a parallel change to the same logic in `TimedPriceSeminar`. By updating one class and not the other, we would break our system, but without any warning from the PHP engine. Our first solution, using a conditional statement, produced a similar dependency between the `cost()` and `chargeType()` methods.

By applying the Strategy pattern, we distilled our costing algorithms into the `CostStrategy` type, locating them behind a common interface, and implementing each only once.

Coupling of another sort can occur when many classes in a system are embedded explicitly into a platform or environment. Let's say that you are building a system that works with a MySQL database, for example. You might use functions such as `mysql_connect()` and `mysql_query()` to speak to the database server.

Should you be required to deploy the system on a server that does not support MySQL, you *could* convert your entire project to use SQLite. You would be forced to make changes throughout your code, though, and face the prospect of maintaining two parallel versions of your application.

The problem here is not the dependency of the system upon an external platform. Such a dependency is inevitable. We need to work with code that speaks to a database. The problem comes when such code is scattered throughout a project. Talking to databases is not the primary responsibility of most classes in a system, so the best strategy is to extract such code, and group it together behind a common interface. In this way you promote the independence of your classes. At the same time, by concentrating your "gateway" code in one place, you make it much easier to switch to a new platform without disturbing your wider system.

Loosening Your Coupling

To handle database code flexibly, we should decouple the application logic from the specifics of the database platform it uses. Fortunately, this is as easy as using a PEAR package: `PEAR::DB`.

Here is some code that uses `PEAR::DB` to work first with MySQL, and then with SQLite:

```
require_once("DB.php");
$dsn_array[] = "mysql://bob:bobs_pass@localhost/bobs_db";
$dsn_array[] = "sqlite://./bobs_db.db";
```

```
foreach ( $dsn_array as $dsn ) {
    print "$dsn\n\n";
    $db = DB::connect($dsn);
    $query_result = $db->query( "SELECT * FROM bobs_table" );
    while ( $row = $query_result->fetchRow( DB_FETCHMODE_ARRAY ) ) {
        printf( "| %-4s| %-4s| %-25s|", $row[0], $row[2], $row[1] );
        print "\n";
    }
    print "\n";
    $query_result->free();
    $db->disconnect();
}
```

Note that we have stripped this example of error handling for the sake of brevity. I covered PEAR errors and the DB package in Chapter 4.

The `DB` class provides a static method called `connect()` that accepts a Data Source Name (DSN) string. According to the makeup of this string, it returns a particular implementation of a class called `DB_Common`. So for the string "mysql://", the `connect()` method returns a `DB_mysql` object, and for a string that starts with "sqlite://", it returns a `DB_sqlite` object. You can see the class structure in Figure 8-5.

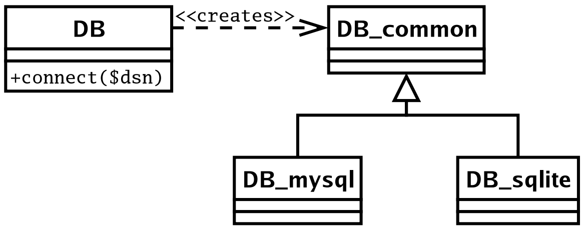

Figure 8-5. *PEAR::DB decouples client code from database objects*

The `PEAR::DB` package, then, enables you to decouple your application code from the specifics of your database platform. As long as you use uncontroversial SQL, you should be able to run a single system with MySQL, SQLite, MSSQL, and many others without changing a line of code (apart from the DSN, of course, which is the single point at which the database context must be configured).

This design bears some resemblance to the Abstract Factory pattern described in the Gang of Four book, and later in this book. Although it is simpler in nature, it has the same motivation, to generate an object that implements an abstract interface without requiring the client to instantiate the object directly.

Of course, by decoupling your system from the specifics of a database platform, the DB package still leaves you with your own work to do. If your (now database-agnostic) SQL code is sprinkled throughout your project, you may find that a single change in one aspect of your project causes a cascade of changes elsewhere. An alteration in the database schema would be the most common example here, where an additional field in a table might necessitate changes to many duplicated database queries. You should consider extracting this code and placing it in a single package, thereby decoupling your application logic from the details of a relational database.

Code to an Interface Not an Implementation

This principle is one of the all-pervading themes of this book. We saw in Chapter 6 (and in the last section) that we can hide different implementations behind the common interface defined in a super class. Client code can then require an object of the super class's type rather than that of an implementing class, unconcerned by the specific implementation it is actually getting.

Parallel conditional statements, like the ones we built into `Lesson::cost()` and `lesson::chargeType()`, are a common signal that polymorphism is needed. They make code hard to maintain because a change in one conditional expression necessitates a change in its twins. Conditional statements are occasionally said to implement a "simulated inheritance."

By placing the cost algorithms in separate classes that implement `CostStrategy`, we remove duplication. We also make it much easier should we need to add new cost strategies in the future.

From the perspective of client code, it is often a good idea to require abstract or general types in your methods' parameter lists. By requiring more specific types, you could limit the flexibility of your code at runtime.

Having said that, of course, the level of generality you choose in your argument hints is a matter of judgment. Make your choice too general, and your method may become less safe. If you require the specific functionality of a subtype, then accepting a differently equipped sibling into a method could be risky.

Still, make your choice of argument hint too restricted, and you lose the benefits of polymorphism. Take a look at this altered extract from the `Lesson` class:

```
function __construct( $duration,
    FixedPriceStrategy $strategy ) {
    $this->duration = $duration;
    $this->costStrategy = $strategy;
}
```

There are two issues arising from the design decision in this example. Firstly, the `Lesson` object is now tied to a specific cost strategy, which closes down our ability to compose dynamic components. Secondly, the explicit reference to the `FixedPriceStrategy` class forces us to maintain that particular implementation.

By requiring a common interface, you can combine a `Lesson` object with any `CostStrategy` implementation.

```
function __construct( $duration, CostStrategy $strategy ) {
    $this->duration = $duration;
    $this->costStrategy = $strategy;
}
```

You have, in other words, decoupled your `Lesson` class from the specifics of cost calculation. All that matters is the interface and the guarantee that the provided object will honor it.

Of course, coding to an interface can often simply defer the question of how to instantiate your objects. When we say that a `Lesson` object can be combined with any `CostStrategy` interface at runtime, we beg the question, "But where does the `CostStrategy` object come from?"

When you create an abstract super class, there is always the issue as to how its children should be instantiated. Which one do you choose in which condition? This subject forms a category of its own in the GoF pattern catalog, and we will examine some of these in the next chapter.

The Concept That Varies

It's easy to interpret a design decision once it has been made, but how do you decide where to start?

The Gang of Four recommend that you "encapsulate the concept that varies." In terms of our lesson example, the "varying concept" is the cost algorithm. Not only is the cost calculation one of two possible strategies in the example, but it is obviously a candidate for expansion: special offers, overseas student rates, introductory discounts, all sorts of possibilities present themselves.

We quickly established that subclassing for this variation was inappropriate and we resorted to a conditional statement. By bringing our variation into the same class, we underlined its suitability for encapsulation.

The Gang of Four recommend that you actively seek varying elements in your classes and assess their suitability for encapsulation in a new type. Each alternative in a suspect conditional may be extracted to form a class extending a common abstract parent. This new type can then be used by the class or classes from which it was extracted. This has the effect of

- Focusing responsibility
- Promoting flexibility through composition
- Making inheritance hierarchies more compact and focused
- Reducing duplication

So how do we spot variation? One sign is the misuse of inheritance. This might include inheritance deployed according to multiple forces at one time (lecture/seminar, fixed/timed cost). It might also include subclassing on an algorithm where the algorithm is incidental to the core responsibility of the type. The other sign of variation suitable for encapsulation is, of course, a conditional expression.

Patternitis

One problem for which there is no pattern is the unnecessary or inappropriate use of patterns. This has earned patterns a bad name in some quarters. Because pattern solutions are neat, it is tempting to apply them wherever you see a fit, whether they truly fulfill a need or not.

The eXtreme Programming methodology offers a couple of principles that might apply here. The first is "You aren't going to need it" (often abbreviated to YAGNI). This is generally applied to application features, but it also makes sense for patterns.

When I build large environments in PHP, I tend to split my application into layers, separating application logic from presentation and persistence layers. I use all sorts of core and enterprise patterns in conjunction with one another.

When I am asked to build a feedback form for a small business Web site, however, I may simply use procedural code in a single page script. I do not need enormous amounts of flexibility, I won't be building upon the initial release. I don't need to use patterns that address problems in larger systems. Instead I apply the second XP principle: "Do the simplest thing that works."

When you work with a pattern catalog, the structure and process of the solution are what stick in the mind, consolidated by the code example. Before applying a pattern, though, pay close attention to the "problem" or "when to use it" section, and read up on the pattern's consequences. In some contexts, the cure may be worse than the disease.

The Patterns

This book is not a pattern catalog. Nevertheless, in the coming chapters, I will introduce a few of the key patterns in use at the moment, providing PHP implementations and discussing them in the broad context of PHP programming.

The patterns described will be drawn from key catalogs including *Design Patterns*, *Patterns of Enterprise Application Architecture*, and *Core J2EE Patterns*. I follow the Gang of Four's categorization, dividing patterns as follows:

Patterns for Generating Objects

These patterns are concerned with the instantiation of objects. This is an important category given the principle "Code to an interface." If we are working with abstract parent classes in our design, then we must develop strategies for instantiating objects from concrete subclasses. It is these objects that will be passed around our system.

Patterns for Organizing Objects and Classes

These patterns help us to organize the compositional relationships of our objects. More simply, these patterns show how we combine objects and classes.

Task-oriented Patterns

These patterns describe the mechanisms by which classes and objects cooperate to achieve objectives.

Enterprise Patterns

We look at some patterns that describe typical Internet programming problems and solutions. Drawn largely from *Patterns of Enterprise Application Architecture* and *Core J2EE Patterns*, the patterns deal with database persistence, presentation, and application logic.

Summary

In this chapter, we looked at some of the principles that underpin many design patterns. We looked at the use of composition to enable object combination and recombination at runtime, resulting in more flexible structures than would be available using inheritance alone. We introduced decoupling, the practice of extracting software components from their context to make them more generally applicable. We reviewed the importance of interface as a means of decoupling clients from the details of implementation.

In the coming chapters, we will examine some design patterns in detail.