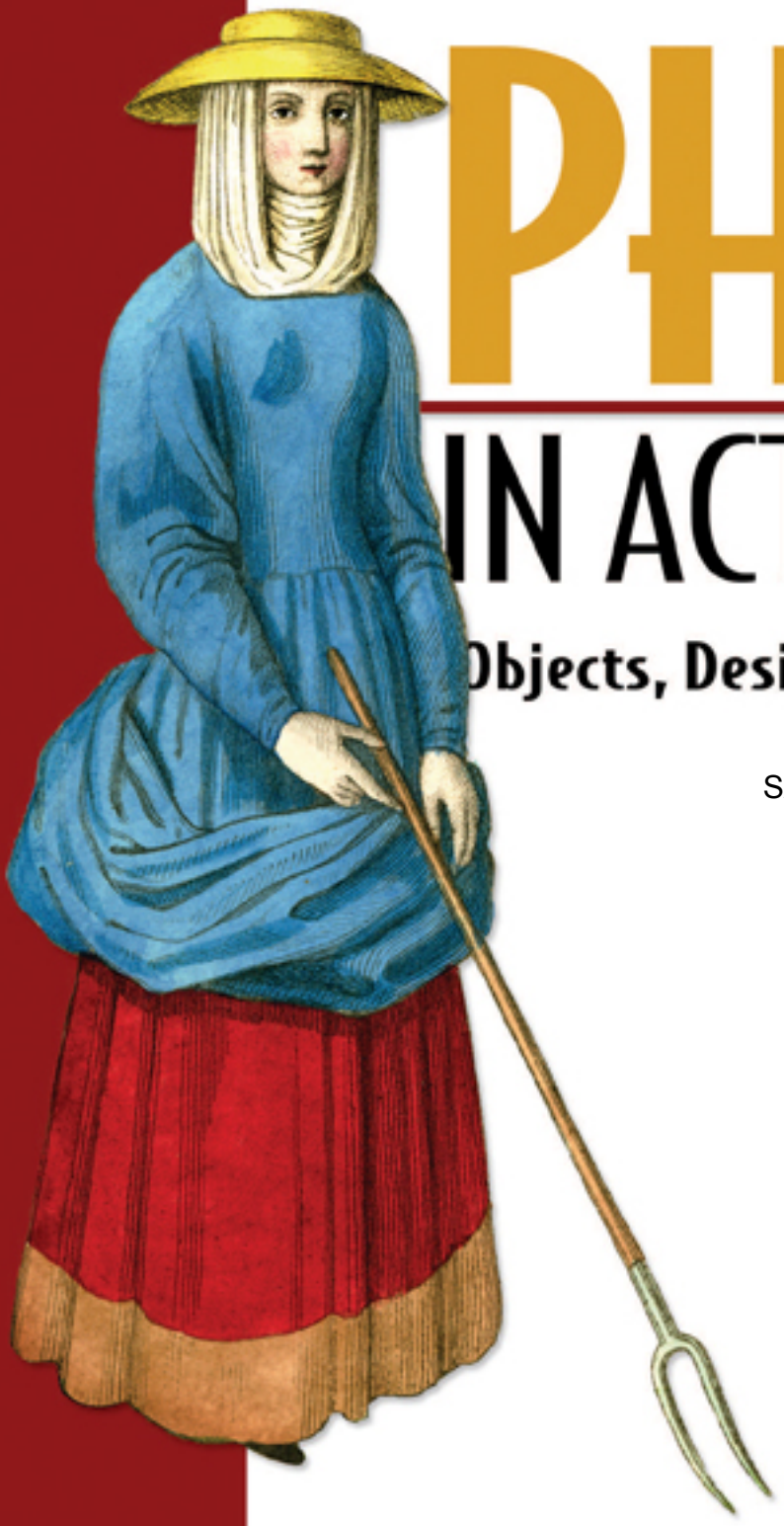

PHP IN ACTION

Objects, Design, Agility

SAMPLE CHAPTER

Dagfinn Reiersøl
Marcus Baker
Chris Shiflett

MANNING

PHP in Action

by Dagfinn Reiersøl
Marcus Baker
Chris Shiflett

Chapter 7

brief contents

CHAPTER 7

Design patterns

Not long ago, I was trying to get my son, age five, to ski. It started well: he saw his older sister skiing down a mild slope, and immediately became eager to show us how easily he could do the same thing.

He couldn't, though. He insisted on putting his skis on in the middle of the slope, and immediately fell flat on his back. I told him it was a nice try, but he disagreed. He had simply lost all interest. I suggested we go lower where it was less challenging. Instead he insisted on going to the top. I humored him and we went up. He had one look down the slope and said, "It's scary."

Of course it was scary. Of course he refused to try it.

I finally persuaded him to do it at the very bottom where the surface was almost flat. Better than nothing, I told myself.

Afterward, I started to ponder the cognitive limitations of a five-year-old. At that age, a child is capable of learning the skill. He's OK with the "how," but the "why" is beyond his ken. The idea that it will be more fun later if he takes the time to practice is meaningless to him. So is the concept that there is some middle ground between scary (the fear of falling) and boring (trudging across a flat field of snow as if wearing snowshoes).

This may seem like an odd introduction to design patterns, but the thing is, "why" is an important question when applying patterns. You can learn how to implement

them, but if you don't know what good they are and in what situations to apply them, you may well do more harm than good by using them.

Much of the literature on software design nowadays focuses on design patterns. Design patterns are an attempt to make the principles of good object-oriented design more explicit. Patterns are defined as "a recurrent solution to a problem." But using them is not as simple as following a cookbook recipe. Applying a pattern can be daunting, since the description in a book is usually somewhat abstract and you have to figure out how exactly to use it in a situation that is different from the example given in the book.

As we've already hinted, an even greater challenge is discovering when you have the problem that the pattern is supposed to solve. Unless your requirement is extremely similar to an example you've seen, it's seldom obvious. And there are lots of situations in which you can use a design pattern but would be better off not doing it because you don't need the extra flexibility that the pattern provides. For instance, the book *Design Patterns* [Gang of Four] describes a pattern called *Command*, which involves creating an object-oriented class for each type of command in your program. So if you have an Edit command, you write a class called `EditCommand` and when you want to run the command, you instantiate the class and run a method that does whatever the command is supposed to do:

```
$editcommand = new EditCommand;
$editcommand->execute();
```

But why? You don't need a separate class just to execute a command. A simple function will do. (Even in strict object-oriented languages such as Java, you don't need a class for each command, just a method.)

Then what's the point? According to the book, the intent of the Command pattern is to encapsulate a request as an object so that you can "parameterize clients with different requests, queue or log requests, and support undoable operations." There are other suggestions as well for when the Command pattern is applicable. But if you don't need to do *any* of those things, creating command objects probably won't do you any good. Unless using the pattern actually results in code that is simpler, has less duplication, or is easier to understand, it may be better to steer clear of it.

Martin Fowler says, "I like to say that patterns are 'half baked,' meaning that you always have to figure out how to apply it to your circumstances. Every time I use a pattern I tweak it a little here and there." The converse is also often the case. If I've developed a design, partly by designing it first and partly by refactoring it, I often find that it can be described by a pattern, or several patterns, without matching any of them exactly.

The problem with many applications of design patterns is that the designers haven't taken the time to compare the design with one without the pattern or with one that uses a different pattern.

In this chapter, we will look at some of the more basic design patterns, primarily from the book *Design Patterns* [Gang of Four]. The selection of patterns is necessarily somewhat arbitrary. Whole books have been written about patterns, so it's impossible to cover them all. The ones we will see in this chapter are Strategy, Adapter, Decorator, Null Object, Iterator, and Composite. Several others will be covered in later chapters.

7.1 STRATEGY

The Strategy pattern is crucial, perhaps the most crucial pattern in modern object-oriented design. It's about creating pluggable, replaceable, reusable components. One example of this is the Template object described in the section on the single-responsibility principle in the previous chapter. If we pass the File and TemplateData objects into the constructor as suggested, we are getting close to a Strategy pattern.

For a more complete, yet still simplistic, example of the Strategy pattern, let's implement a basic example from earlier chapters using this pattern. This is a simplistic example, and the Strategy pattern is overkill in this case. But the example shows how the Strategy pattern is implemented and how it can be an alternative to implementation inheritance. We'll study the basic mechanics using "Hello world." The example is too simple to be meaningful in the real world, so in addition we'll discuss its usefulness in real situations.

The Strategy pattern will also recur in many contexts in later chapters.

7.1.1 "Hello world" using Strategy

Figure 7.1 shows the class diagram for the example shown in chapter 2. The parent class, HtmlDocument, implements the generic features represented by the start and end tags of the HTML document. The HelloWorld child class implements the specific features, represented by the actual content of the document. So to generate something other than a greeting, say an announcement, we can add another child class that generates the content of an announcement.

We can move the `getContents()` method to a Strategy object instead. Instead of using a subclass of HtmlDocument, we can use HtmlDocument configured with a Strategy object instead. In UML, this looks like figure 7.2.

This may look impressive; it's hard to tell from the UML diagram that it represents totally unnecessary complexity. We are just using it to make sure we understand the mechanical aspects of the pattern. HtmlContentStrategy might as well be

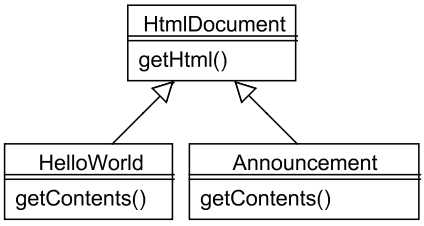

Figure 7.1
Class diagram of the simplistic HelloWorld example with related classes added

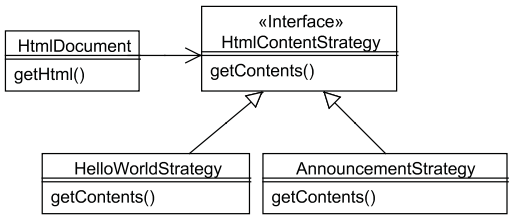

Figure 7.2
HelloWorld as a Strategy class

an abstract class, but I've defined it as an interface to make it clear that it doesn't need to contain any working code. This means that there is no implementation inheritance left in the design.

But what does it look like in code? The HtmlDocument class still generates the start and end of the document. But rather than get the content from a method that's implemented in a subclass, it gets it from the Strategy object.

```
class HtmlDocument {
    private $strategy;

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

    public function getHtml() {
        return "<html><body>".$this->strategy->getContents().
            "</body></html>";

    }
}
```

We want to be able to plug different Strategy objects into the HtmlDocument object. So the HtmlDocument object needs a consistent way to call the Strategy object. In other words, it needs a consistent interface, which is defined by an `interface`.

```
interface HtmlContentStrategy {
    public function __construct($name);
    public function getContents();
}
```

Now any HtmlDocument object will be able to use any Strategy object that implements this interface, since all it requires is the ability to call the `getContents()` method.

But wait a minute. What about the constructor? The interface defines that, too. The Strategy object for generating the "Hello world" message needs the world name as an argument to the constructor. Are we sure that other Strategy objects for generating HTML content will also need the same thing? I'm afraid not; in fact, I fear that they will need all sorts of other information to do their jobs.

What do we do about that? It's simple; we just eliminate the constructor from the interface. Since the HtmlDocument class doesn't instantiate the Strategy class, all

objects that implement the interface can be used even if their constructors differ. So the interface just needs the `getContents()` method:

```
interface HtmlContentStrategy {
    public function getContents();
}
```

Now we can implement the "Hello world" feature as a Strategy class:

```
class HelloWorldStrategy implements HtmlContentStrategy {
    var $world;
    public function __construct($world ) {
        $this->world = $world ;
    }

    public function getContents() {
        return "Hello ".$this->world ."!";
    }
}
```

What this class does is trivial, but the pattern is extremely useful in more complex situations.

7.1.2 How Strategy is useful

Using Strategy in place of implementation inheritance is the way to create pluggable components and is useful in implementing the open-closed principle.

The most important reason for this is the fact that parent and child classes are highly coupled. They depend on each other in ways that are not necessarily obvious. An object that belongs to a class hierarchy can call a method from any class in the hierarchy (unless it is a private method) simply by using `$this`. And `$this` gives no clue as to which of the classes in the hierarchy the method belongs to.

Contrast this with the situation in which an object holds a reference to an object that is not part of an inheritance hierarchy. Let's say we have a User object that contains an Address object. In a method in the user object, we can call a method on `$this` or `$this->address`. In either case, it is clear which class the method belongs to. And unless we give the User object a reference to the Address object, the Address object is unable to call methods belonging to the User object (except by creating a new User object). So we have a one-way dependency; this makes it much more likely that we can reuse the Address class in another context. This means that the classes are much easier to disentangle than a parent and a child class that may use each other's methods freely.

This shows why there is high coupling, but this high coupling can also be convenient, since it's easy to use all those methods.

Strategy can be used in so many different situations that it is almost impossible to narrow its range of application. It can be applied to express almost any difference in behavior.

While Strategy is about pluggable behavior for a class, the next pattern—Adapter—is about changing the interface of an existing class to make it pluggable in a different context than its original one.

7.2 ADAPTER

The Adapter pattern is typically used to retrofit a class with an altered API. You may need a different API to make it compatible with another, existing class. Or perhaps the original API is too cumbersome and hard to use.

An Adapter is extra complexity, so if you can, it might be better to refactor the original class so it gets the API you want in the first place. But there might be good reasons why you can't or don't want to do that. Two of the reasons may be

- The class is already in use by many clients, so changing its interface will require changing all the clients.
- The class is part of some third-party software, so it's not practical to change it. You can, of course, change open-source software, but that means you're in trouble when the next version arrives.

In an ideal world, you might get to design everything for yourself and redesign it when necessary. Then you would rarely need Adapters, if ever. But in the real world, they become necessary because of constraints such as these.

In this section, we'll start with an extremely simple example, moving from there to an example showing how to adapt real template engines. Then we'll see an even more advanced example involving multiple classes. Finally, we'll discuss what to do if we need compatibility between several different interfaces so that a more generic interface is required.

7.2.1 Adapter for beginners

Sometimes all you need to do when creating an Adapter is change the names of methods. This is easy. If we have a template class with the method `assign()` and we want the name `set()` instead, we can use a simple Adapter that just delegates all the work to the template class.

Take our "simplest-possible template engine" example, the Template class from the previous chapter. It has the methods `set()` and `asHtml()`. What if we want to use the names Smarty uses instead: `assign()` and `fetch()`? The example in listing 7.1 shows how this can be done.

Listing 7.1 The Simplest-possible template adapter class

```
class SimpleTemplateAdapter {
    private $template;

    public function __construct($template) {
         $this->template = $template;
    }
```

```
    public function assign($var,$value) {
        $this->template->set($var,$value);
    }

    public function fetch() {
        return $this->template->asHtml();
    }
}
```

To use this class, all we have to do is wrap the template object in the adapter by passing it in the constructor:

```
$template = new SimpleTemplateAdapter(new Template('test.php'));
```

`$template` now uses the Smarty method names, but it does not work quite like a Smarty object, since it's still defined as a template rather than a template engine. In the next section, we will see how to overcome this more challenging, conceptual difference.

7.2.2 Making one template engine look like another

For a more realistic example, let's use two template engines: Smarty and PHPTAL. Smarty is perhaps the most widely-known and popular template engine. PHPTAL is interesting and different. We'll discuss that further in chapter 13; for now, we're just looking at the possibilities of the Adapter pattern, and these two template engines are different enough to make it a challenge.

In particular, the two template engines are conceptually different in their design. PHPTAL uses a template object that is constructed with a specific template file. So you set the template first, add the variables you want inserted into the HTML output, and then execute it:

```
$template = new PHPTAL_Template('message.html');
$template->set('message','Hello world');
echo $template->execute();
```

A Smarty object is a different kind of animal: it's not a template; it's an instance of the template engine. After you've created the Smarty object, you can hand it any template file for processing.

The conceptual difference creates a difference in sequence. With PHPTAL, you specify the template file first and then you set the variables; with Smarty, it's the other way around:

```
$smarty = new Smarty;
$smarty->assign('message','Hello world');
$smarty->display('message.tpl');
```

Imagine that our site is currently based on Smarty, but we want to change it to PHPTAL. In order to avoid having to rewrite all the PHP code that uses the templates, we want the templates to still appear to the PHP code as Smarty templates, so we can leave the code that uses them mostly unchanged. In other words, the Smarty interface

is the one we want to keep using, even though the actual templates are PHPTAL templates. So the Adapter class will give the PHPTAL template engine a Smarty "skin." With one exception, the methods we'll write are the most basic ones needed to display a simple HTML page based on a template. If we need more methods, we can add them later.

We'll start by defining the PHPTAL template interface formally. As always in PHP, declaring the interface is not strictly needed, but it gives us a useful overview of what we're doing.

```
interface SmartyTemplateInterface {
    public function fetch($template);
    public function display($template);
    public function assign($name,$value);
    public function get_template_vars();
}
```

The Adapter reflects the conceptual differences between the two template engines. A Smarty object requires no constructor arguments, so we can skip the constructor in this class. The PHPTAL_Template object has to be constructed, but it demands the template file name in the constructor. Since the Smarty interface does not supply the file name until we generate the output using `fetch()` or `display()`, we have to wait until then before constructing the PHPTAL template object. Listing 7.2 shows the Adapter class.

Listing 7.2 Adapter to make PHPTAL templates conform to the Smarty interface

```
class SmartySkin implements SmartyTemplateInterface {
    private $vars = array();

    public function assign($name,$value) {
        $this->vars[$name] = $value;
    }

    public function fetch($template) {
        $phptal = new PHPTAL_Template(
                str_replace('.tpl','.html',$template));
        $phptal->setAll($this->vars);
        return $phptal->execute();
    }

    public function get_template_vars($name=FALSE) {
       if ($name) return $this->vars[$name];
       return $this->vars;
    }

    public function display($template) {
        echo $this->fetch($template);
    }
}
```

1 **Store variables before PHPTAL object exists**

2 **Create and execute PHPTAL object**

3 **Emulate Smarty's variable getter**

4 **PHPTAL has no display() method**

❶ Since we don't create the PHPTAL object before it's time to generate the output, we have to store the variables in the meantime. This is done using the Smarty-compatible `assign()` method. We keep the variable in the `$vars` array belonging to the Adapter.

❷ It's only when the `fetch()` method is called that we have the template file name available. So now we can create the PHPTAL_Template object. Since the Smarty and the PHPTAL templates normally have different file extensions, we convert from one (`.tpl`) to the other (`.html`). Now we can copy the variables from the Adapter class to the template. PHPTAL has a convenient `setAll()` method to do this. Since we now have both the template filename and the variables set, we can generate the output by using PHPTAL's `execute()` method.

❸ `get_template_vars()` is Smarty's way of retrieving a variable that has been set in the Smarty object. We emulate its behavior by returning a specific variable if its name has been specified, or the whole array of variables if it hasn't.

❹ PHPTAL has no `display()` method, but it's trivial to implement by echoing the output from the `fetch()` method.

7.2.3 Adapters with multiple classes

Sometimes we have to do even more tricks to get an adapter to work. If the API we're emulating uses more than one class, we may have to emulate all of them. One example is the opposite process of the one we just did. If we want to give a Smarty template a PHPTAL skin, we run into a different kind of challenge: The PHPTAL template class has no way of retrieving the variables you've set in it. Instead, you have to get an object called a Context from the template object and get the variables from that object:

```
$context = $template->getContext();
$message = $context->get('message');
```

This might not be a problem in normal use of the template engine, but if we have used the Context object (testing is a likely use for it), we might want it in the adapter interface.

Let's see how we can do that. Here is the PHPTAL interface:

```
interface PhptalTemplateInterface {
    public function set($name,$value);
    public function execute();
    public function getContext();
}
```

Now for the Adapter itself. Listing 7.3 shows how the Adapter uses a Smarty object internally to do the actual work, while appearing from the outside as a PHPTAL template with limited functionality.

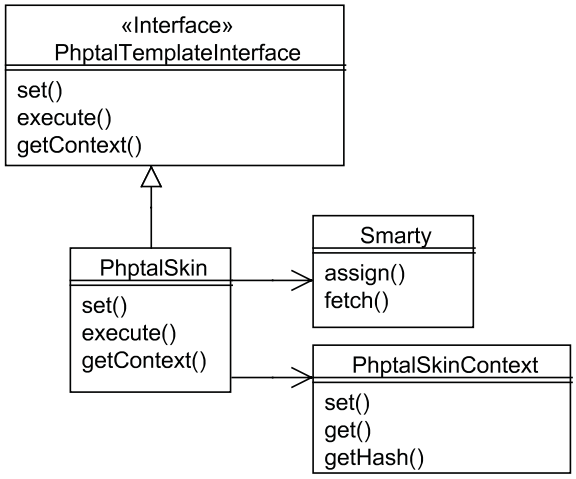

Figure 7.3 Adapting Smarty to make it look like PHPTAL

Figure 7.3 is a class diagram showing the structure of the design. The interface, the PhptalSkin class, and the PhptalContext class all belong to the adapter, but all the real work is done by the humble Smarty class.

In the real world, the Smarty class is not so humble. This example is simplified to utilize only a few basic methods of the PHPTAL interface and the Smarty object. We have shown only two methods of the around 40 methods of the Smarty object. In practice, we would be likely to implement more of them, although in most projects there is no good reason to implement more than we actually need.

Listing 7.3 shows how the PhptalSkin class is implemented.

Listing 7.3 Adapter to make Smarty templates conform to the PHPTAL interface

```
class PhptalSkin implements PhptalTemplateInterface {
    private $smarty;
    private $path;
    private $context;
    public function __construct($path) {
        $this->smarty = new Smarty;
        $this->path = str_replace('.html','.tpl',$path);
        $this->context = new PhptalSkinContext;
    }

    public function execute() {
        return $this->smarty->fetch($this->path);
    }

    public function set($name,$value) {
        $escaped = htmlentities($value,ENT_QUOTES,'UTF-8');
        $this->smarty->assign($name,$escaped);
        $this->context->set($name,$escaped);
    }
```

❶ Create Smarty and Context objects

❷ Execute with template name

❸ Set value in Smarty and Context

```
    public function getContext() {
        return $this->context;
    }
}
```

❹ getContext() as with real PHPTAL

❶ The Smarty object is the Smarty template engine, the object that's going to do the real work. The PHPTAL interface requires that we specify the template file when we construct the template object. Since the Smarty object does not store the name of the template file, we have to keep it in the Adapter. The file name conversion is the inverse of the file name conversion from the previous Adapter.

Since a PHPTAL template returns a PHPTAL_Context object, the adapter needs an object that does a similar job without being an actual PHPTAL_Context object. For this purpose, we use a PhptalSkinContext object. We'll take a look at the class in a moment. It is just a simple variable container, and for now, all we need to know about it is that we can store variables in it with a `set()` method.

❷ The `execute()` method calls Smarty's equivalent, the `fetch()` method. Since the `fetch()` method requires the template name (or more specifically a template resource), we give it the template name that was supplied to the constructor.

While we're at it, let's change the way the `assign()` method works to make it more secure. All output should be escaped to prevent cross-site scripting (XSS) attacks. With Smarty, this means you either have to escape the strings before adding them to the template or explicitly use Smarty's escaping features. The problem is that the escaping in this example is primitive and not applicable to anything beyond simple values. It would have to be made much more sophisticated to allow it to work in existing applications. The subject of template security will be discussed further in chapter 13.

❸ The `set()` method sets the corresponding variable in the Smarty object. It also sets the variable in the context object so that it can be retrieved in the PHPTAL fashion.

An alternative way to implement this would be to store the variables in the Adapter and to copy all of them into the Context or Smarty object when they're needed. The current solution duplicates the data, but there is no reason right now why that should cause problems, so there is probably little practical difference between the alternatives.

❹ We can use the `getContext()` method to return the PhptalSkinContext object, so that we can retrieve the variables in the same way as with a real PHPTAL_Context object.

Listing 7.4 shows the PhptalSkinContext class. This is just a thin wrapper around a PHP array.

Listing 7.4 PhptalSkinContext class—the Adapter's counterpart of the PHPTAL_Context class

```
class PhptalSkinContext {
    private $vars = array();

    public function set($name,$value) {
        $this->vars[$name] = $value;
    }

    public function get($name) {
        return $this->vars[$name];
    }

    public function getHash() {
        return $this->vars;
    }
}
```

The class has a subset of the interface of PHPTAL_Context class. `get()` retrieves a single variable; `getHash()` retrieves all of them.

7.2.4 Adapting to a generic interface

You may ask, why not use inheritance? Why not let the Adapter be a child class of Smarty or PHPTAL? In fact, the Gang of Four book indicates this as an option. The effect of letting the first of our Adapters inherit from the Smarty class will be to allow the use of any Smarty method that's not in the PHPTAL interface. The Adapter's interface then becomes a somewhat messy mixture of Smarty and PHPTAL methods. But if we're switching to Smarty anyway, that might be just as well. Developers could gradually switch to using the Smarty interface.

But there is one more consideration: in Uncle Bob's terminology, we've now taken the first bullet. We were cruising happily along, using PHPTAL templates for all our web pages, and suddenly someone hits us with the requirement to use Smarty instead. We know now that a certain kind of change can happen: switching template engines. And if it happens once, it could happen again. So what we probably want to do is to protect the system from further changes of the same type. The way to go in this case would be to move toward a generic template interface, which would not be identical to either the PHPTAL interface or the Smarty interface. The generic interface should be as easy as possible to adapt to a new template engine. In other words, it should be easy to write an Adapter that has the generic interface and delegates the real work to the new template engine.

So far, we have at least some indication of what's needed for a generic Template-Adapter interface. It will need to have an interface that re-creates the functionality of both the PHPTAL and the Smarty objects. We don't want to have to use fancy tricks such as the Context object. So the interface should have a method to get variables. It should also have a `display()` method. And the need to convert the template name

is a tricky thing that needs to be smoothed over. If we assume that the template only needs a single template file name in some form, the generic interface might just require the file name without the extension and add the extension automatically.

Adapter is a pattern that works by wrapping an object in another. A Decorator also does that, but for a different purpose.

7.3 DECORATOR

Adapters are the tortillas of object-oriented programming. You wrap an object in an Adapter, and it looks completely different but tastes almost the same. Decorator is another kind of wrapper, but the intent is not to change the interface. Instead, a Decorator changes the way an object works—somewhat—but leaves its appearance relatively intact. So it's more like sprinkling salt on the dish: the result tastes slightly different, but looks similar.

But technically, what Adapters and Decorators do is mostly the same: you wrap the decorator around another object. A term that has been used to describe this principle is Handle-Body. There is a "Handle" object that wraps a "Body" object.

For an example, we'll use a so-called Resource Decorator for a database connection. Then we'll discuss how to make sure we can add multiple Decorators to an object.

7.3.1 Resource Decorator

For an example, let's try a Resource Decorator [Nock]. This is typically used to add extra behavior to a database connection. Let's say we're dissatisfied with the way PEAR DB handles errors. We want to use PHP 5 exceptions instead. One way to achieve that is to wrap the PEAR DB connection in a class that generates the exceptions. We'll start with a simple example using only one decorator (see listing 7.5).

Listing 7.5 Decorator that wraps a PEAR DB object and generates exceptions if errors occur

```
class PearExceptionDecorator {
    private $connection;
    public function __construct($connection) {
        $this->connection =  $connection;
        if (DB::isError($this->connection)) {
            throw new Exception($this->connection->getMessage());
        }
    }

    public function query($sql) {
        $result = $this->connection->query($sql);
        if (DB::isError($result)) {
            throw new Exception($result->getMessage()."\n".$sql);
        }
        return $result;
    }
```

Use PEAR DB connection ❶

query() method with error handling ❷

```
    public function nextID($name) {
        return $this->connection->nextID($name);
    }
}
```

❸ One example of simple delegation

❶ The constructor accepts a PEAR DB object as an argument. This means that we can create our decorated connection as follows:

```
$connection = new PearExceptionDecorator(DB::Connect(
       'mysql://user:password@localhost/webdatabase'));
```

Passing the "Body" object in the constructor is typical of decorators, but in this simple case, it would work even if we instantiated the PEAR DB connection inside the constructor.

❷ The `query()` method calls the PEAR DB connection's `query()` method and throws an exception if there is an error (an SQL syntax error, for example).

❸ The `nextID()` method just delegates to the PEAR DB object. This method is really just one example of many methods that are available from the PEAR DB object that we don't need to change. To get the decorated object to work like the original object, we might want to implement a lot of these delegating methods.

In this case, there are at least two benefits to using a Decorator. One is that we can't simply change the PEAR package to add this feature to it. (Strictly speaking, we can change it, since it's open source, but then we have to maintain it afterward, and that's not worth the trouble.) The other is that our way of handling exceptions is more likely to change than the PEAR package. The PEAR package is relatively stable; it has to be, because it has lots of users. The Decorator might change because we need a different kind of error handling. Perhaps we want to use exceptions in a somewhat more sophisticated way, using a more specific exception class, for instance. Perhaps we want compatibility with PHP 4. We could have a similar decorator that would work in PHP 4, using some error handling or logging capability that is not exception based, and just swap the decorators depending on the PHP version.

7.3.2 Decorating and redecorating

The previous example is the simplest form of a decorator. The more advanced thing to do is to decorate and "redecorate." Since the decorated object works in a way that's similar to the original object, you can apply more than one Decorator to add different responsibilities. For example, if we had a Decorator to add logging to the connection, we could do something like this.

```
$connection = new PearLoggingDecorator(
    new PearExceptionDecorator(
    DB::Connect('mysql://user:password@localhost/webdatabase')));
```

But what if we have a lot of delegating methods—such as the `nextID()` method in the Decorator we've just seen? We don't want to duplicate all those in both Decorators. So we'll make a parent class to keep the delegating methods in (see listing 7.6).

Listing 7.6 Decorator parent class to make redecoration easier

```
abstract class PearDecorator {
    protected $connection;

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

    public function query($sql) {
        return $this->connection->query($sql);
    }

    public function nextID($name) {
        return $this->connection->nextID($name);
    }
}
```

As in the previous example, a practical version of the class is likely to contain many more delegating methods.

Any decorator for a PEAR DB object can now be derived from the abstract parent class. We need only override the methods we want to change. Figure 7.4 shows this simple inheritance hierarchy. The parent class is abstract, but its methods are not. Any method that is not implemented in a child Decorator will work like the method in the decorated object. The Logger class is just a helper for the logging Decorator.

Therefore, the PearExceptionDecorator no longer needs the `nextID()` method or any other method it doesn't add anything to. This is shown in listing 7.7.

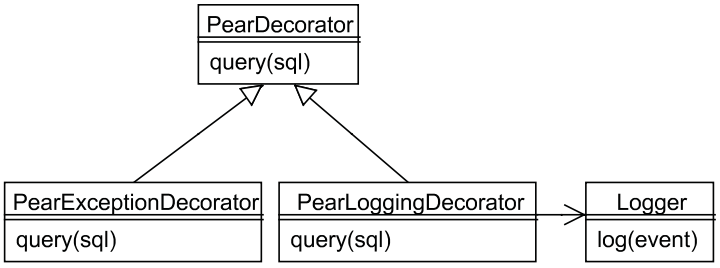

Figure 7.4 Using a parent class for Decorators to provide default method implementations and to make sure the Decorators are compatible

Listing 7.7 Deriving the PearExceptionDecorator class from the Parent class

```
class PearExceptionDecorator extends PearDecorator {
    public function __construct($connection) {
        $this->connection =  $connection;
        if (DB::isError($this->connection)) {
            throw new Exception($this->connection->getMessage());
        }
    }

    public function query($sql) {
        $result = $this->connection->query($sql);
        if (DB::isError($result)) {
            throw new Exception($result->getMessage()."\n".$sql);
        }
        return $result;
    }

}
```

Now we can implement the logging Decorator using the same procedure. What we want to log will depend on the circumstances, but for the example, let's log every query. Perhaps we would want to do that while our application is in the testing stages. When it becomes stable, we can remove the Decorator. A more conventional alternative would be to disable logging; the advantage of the Decorator is that we can get rid of the logging code entirely so it doesn't clutter the application.

Listing 7.8 shows the logging Decorator.

Listing 7.8 PearLoggingDecorator class that can be used in addition to the exception Decorator

```
class PearLoggingDecorator extends PearDecorator {
    private $logger;
    public function __construct($connection) {
        $this->connection =  $connection;
        $this->logger = Log::factory(
                'file', '/tmp/out.log', 'SQL');
    }

    public function query($sql) {
        $this->logger->notice('Query: '.$sql);
        $result = $this->connection->query($sql);
        return $result;
    }
}
```

We are using the PEAR Log package. In the constructor, we store a logger object in an instance variable. When we call the `query()` method on the decorated connection, it logs the SQL statement as a notice before executing the query.

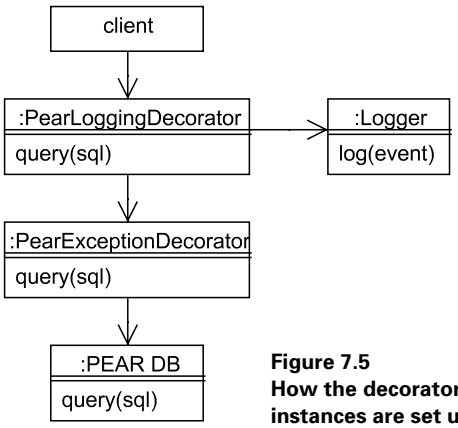

Figure 7.5
How the decorator instances are set up

While figure 7.4 illustrates the relationships between the classes, the configuration of objects at runtime is something else. This is shown in the UML object diagram in figure 7.5. The colons (:PearLoggingDecorator) indicate that we are dealing with objects—instances of the named classes—rather than with the classes as such.

The PearLoggingDecorator uses the PearExceptionDecorator, which uses the PEAR DB object. The `query()` call is passed from the top to the bottom of this chain, and the results are passed back up.

NOTE There is no deeper meaning to the words "top" and "bottom," "up" and "down" in this context. They just refer to the placement of the objects in the diagram. This placement is arbitrary.

The decorators are set up in an order that seems logical, but if we swapped the two decorators, it would still work, and we might not notice the difference.

From a pattern skeptic point of view, we may ask some critical questions when a Decorator is suggested. Is the decorator really needed? Do the component and the Decorator really need to be separate, or can they be merged into one class? You might want to keep them separate because the Decorator's behavior is not always needed, or to comply with the single-responsibility principle: if the decorator's behavior is likely to change for different reasons than the component's. Resource Decorators may be considered an example of this: the software that handles the database might change, but it's probably more stable than what you are adding to it.

Strategy is for changing and replacing behavior. Decorator is a way to add behavior. When we want to stop a behavior from happening, we can either write a plain old conditional statement or use a Null Object.

7.4 NULL OBJECT

"Don't turn on the dark light," my five-year-old son reproaches me when I turn out the lights in his room. The mental model revealed by this statement is an interesting

and striking simplification of the physics involved. Instead of being opposites, he sees turning the light off and on as variations of the same process. There's a bright and a dark light, and you can turn either one on. In object-oriented lingo, both the bright light class and the dark light class have a `turnOn()` operation or method. Like the `dress()` method of the Boy and Girl classes in chapter 4, this is polymorphism, a case of different actions being represented as basically the same.

In this section, we'll see how Null Objects work, and then discover how to use them with the Strategy pattern.

7.4.1 Mixing dark and bright lights

A Null Object is the dark light of our object-oriented world. It looks like an ordinary object, but doesn't do anything real. Its *only* task is to look like an ordinary object so you don't have to write an `if` statement to distinguish between an object and a non-object. Consider the following:

```
$user = UserFinder::findWithName('Zaphod Beeblebrox');
$user->disable();
```

If the UserFinder returns a non-object such as NULL or FALSE, PHP will scold us:

```
Fatal error: Call to a member function disable() on a non-object
in user.php on line 2
```

To avoid this, we need to add a conditional statement:

```
$user = UserFinder::findWithName('Zaphod Beeblebrox');
if (is_object($user))
    $user->disable();
```

But if `$user` is a Null Object that has `disable()` method, there is no need for a conditional test. So if the UserFinder returns a Null Object instead of a non-object, the error won't happen.

A simple NullUser class could be implemented like this:

```
class NullUser implements User {
    public function disable() { }
    public function isNull() { return TRUE; }
}
```

The class is oversimplified, since it implements only one method that might be of real use in the corresponding user object: `disable()`. The idea is that the real user class, or classes, would also implement the interface called User. So, in practice, there would be many more methods.

7.4.2 Null Strategy objects

A slightly more advanced example might be a Null Strategy object. You have one object that's configured with another object that decides much of its behavior, but in some cases the object does not need that behavior at all.

An alternative to using the Logging decorator shown earlier might be to build logging into the connection class itself (assuming we have control over it). The connection class would then contain a logger object to do the logging. The pertinent parts of such a connection class might look something like this:

```
class Connection {
    public function __construct($url,$logger) {
        $this->url = $url;
        $this->logger = $logger;
        // More initialization
        // ...
    }

    public function query($sql) {
        $this->logger->log('Query: '.$sql);

        // Run the query
        // ...
    }
}
```

Since this class accepts a logger object as input when it's created, we can configure it with any logger object we please. And if we want to disable logging, we can pass it a null logger object:

```
$connection = new Connection(
    mysql://user:password@localhost/webdatabase,
    new NullLogger
);
```

A NullLogger class could be as simple as this:

```
class NullLogger implements Logger{
    public function log {}
}
```

Figure 7.6 shows the relationships between these classes. The interface may be represented formally using the `interface` keyword or an abstract class, or it may be implicit using duck typing as described in chapter 4.

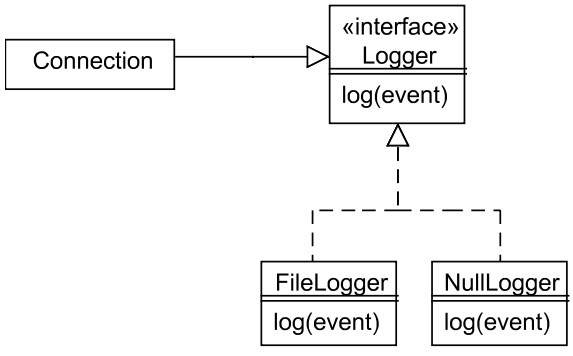

Figure 7.6
Using a NullLogger as a Strategy object

The PEAR Log package has a Null logger class called Logger_null that is somewhat more sophisticated than the one we just saw.

Although a Null Object might do something such as return another Null Object, frequently it's about doing nothing at all. The next pattern, Iterator, is about doing something several times.

7.5 ITERATOR

An *iterator* is an object whose job it is to iterate, usually returning elements one by one from some source. Iterators are popular. One reason may be that it's easy to understand what they do, in a certain limited way, that is. It is relatively easy to see how they work and how to implement one. But it's less obvious how and when they're useful compared to the alternatives, such as stuffing data into a plain PHP array and using a `foreach` loop to iterate.

In this section, we will see how iterators work, look at some good and bad reasons to use them, contrast them with plain arrays, and see how we can improve iterators further by using the Standard PHP Library (SPL).

7.5.1 How iterators work

An iterator is an object that allows you to get and process one element at a time. A while loop using an SPL (Standard PHP Library) iterator has this form:

```
while ($iterator->valid()) {
    $element = $iterator->current();
    // Process $element
    $iterator->next();
}
```

There are various interfaces for iterators, having different methods that do different things. However, there is some overlap. Above all, to be useful at all, every iterator needs some way of getting the next element and some way to signal when to stop. Table 7.1 compares the SPL iterator interface with the standard Java iterator interface and the interface used in the Gang of Four [Gang of Four] book.

Table 7.1 Comparing three different iterator interfaces

	Gang of Four iterator	Java iterator	PHP SPL iterator
Move to next element	Next()	next()	next()
Return the current element	CurrentItem()		current()
Check for end of iteration	IsDone()	hasNext()	valid()
Start over at beginning	First()		rewind()
Return key for current element			key()
Remove current element from collection		remove()	

7.5.2 Good reasons to use iterators

Three are three situations in which an iterator is undeniably useful in PHP:

- When you use a package or library that returns an iterator
- When there is no way to get all the elements of a collection in one call
- When you want to process a potentially vast number of elements

In the first case, you have no choice but to use the iterator you've been given. Problem 3 will happen, for example, when you return data from a database table. A database table can easily contain millions of elements and gigabytes of data, so the alternative—reading all of them into an array—may consume far too much memory. (On the other hand, if you know the table is small, reading it into an array is perfectly feasible.)

Another example would be reading the results from a search engine. In this case, problems 2 and 3 might both be present: you have no way of getting all the results from the search engine without asking repeatedly, and if you did have a way of getting all of them, it would far too much to handle in a simple array.

In addition to the undeniably good reasons to use iterators, there are other reasons that may be questioned, because there are alternatives to using iterators. The most important alternative is using plain arrays. In the previous situations, using plain arrays is not a practical alternative. In other situations, they may be more suitable than iterators.

7.5.3 Iterators versus plain arrays

The general argument in favor of iterators is that they

- Encapsulate iteration
- Provide a uniform interface to it

Encapsulation means that the code that uses an iterator does not have to know the details of the process of iteration. The client code can live happily ignoring those details, whether they involve reading from a database, walking a data structure recursively, or generating random data.

The *uniform interface* means that iterators are pluggable. You can replace an iterator with a different one, and as long as the single elements are the same, the client code will not know the difference.

Both of these are advantages of using iterators. On the other hand, both advantages can be had by using plain arrays instead.

Consider the following example. We'll assume we have a complex data structure such as a tree structure (this is an example that is sometimes used to explain iterators).

```
$structure = new VeryComplexDataStructure;
for($iterator = $structure->getIterator();
    $iterator->valid();
    $iterator->next()) {
    echo $iterator->current() . "\n";
}
```

The simpler way of doing it would be to return an array from the data structure instead of an iterator:

```
$structure = new VeryComplexDataStructure;
$array = $structure->getArray();
foreach ($array as $element) {
    echo $value . "\n";
}
```

It's simpler and more readable; furthermore, the code required to return the array will typically be significantly simpler and leaner than the iterator code, mostly because there is no need to keep track of position as we walk the data structure, collecting elements into an array. As the Gang of Four say, "External iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates." In other words, iterating internally in the structure is easier.

In addition, PHP arrays have another significant advantage over iterators: you can use the large range of powerful array functions available in PHP to sort, filter, search, and otherwise process the elements of the array.

On the other hand, when we create an array from a data structure, we need to make a pass through that structure. In other words, we need to iterate through all the elements. Even though that iteration process is typically simpler than what an iterator does, it takes time. And the `foreach` loop is a second round of iteration, which also takes time. If the iterator is intelligently done, it won't start iterating through the elements until you ask it to iterate. Also, when we extract the elements from the data structure into the array, the array will consume memory (unless the individual elements are references).

But these considerations are not likely to be important unless the number of elements is very large. The guideline, as always, is to avoid premature optimization (optimizing before you know you need to). And when you do need it, work on the things that contribute most to slow performance.

7.5.4 SPL iterators

The Standard PHP Library (SPL) is built into PHP 5. Its primary benefit—from a design point of view—is to allow us to use iterators in a `foreach` loop as if they were arrays. There are also a number of built-in iterator classes. For example, the built-in DirectoryIterator class lets us treat a directory as if it were an array of objects representing files. This code lists the files in the `/usr/local/lib/php` directory.

```
$iter = new DirectoryIterator('/usr/local/lib/php');
foreach($iter as $current) {
    echo $current->getFileName()."\n";
}
```

In chapter 19, we will see how to implement a decorator for a Mysqli result set to make it work as an SPL iterator.

7.5.5 How SPL helps us solve the iterator/array conflict

If you choose to use plain arrays to iterate, you might come across a case in which the volume of data increases to the point where you need to use an iterator instead. This might tempt you to use a complex iterator implementation over simple arrays when this is not really needed. With SPL, you have the choice of using plain arrays in most cases and changing them to iterators when and if that turns out to be necessary, since you can make your own iterator that will work with a `foreach` loop just like the ready-made iterator classes. In the VeryComplexDataStructure example, we can do something like this:

```
$structure = new VeryComplexDataStructure;
$iterator = $structure->getIterator();
foreach($iterator as $element) {
    echo $element . "\n";
}
```

As you can see, the `foreach` loop is exactly like the `foreach` loop that iterates over an array. The array has simply been replaced with an iterator. So if you start off by returning a plain array from the VeryComplexDataStructure, you can replace it with an iterator later without changing the `foreach` loop. There are two things to watch out for, though: you would need a variable name that's adequate for both the array and the iterator, and you have to avoid processing the array with array functions, since these functions won't work with the iterator.

The previous example has a hypothetical VeryComplexDataStructure class. The most common complex data structure in web programming is a tree structure. There is a pattern for tree structures as well; it's called Composite.

7.6 COMPOSITE

Composite is one of the more obvious and useful design patterns. A Composite is typically an object-oriented way of representing a tree structure such as a hierarchical menu or a threaded discussion forum with replies to replies.

Still, sometimes the usefulness of a composite structure is not so obvious. The Composite pattern allows us to have any number of levels in a hierarchy. But sometimes the number of levels is fixed at two or three. Do we still want to make it a Composite, or do we make it less abstract? The question might be whether the Composite simplifies the code or makes it more complex. We obviously don't want a Composite if a simple array is adequate. On the other hand, with three levels, a Composite is likely to be much more flexible than an array of arrays and simpler than an alternative object-oriented structure.

In this section, we'll work with a hierarchical menu example. First, we'll see how the tree structure can be represented as a Composite in UML diagrams. Then we'll implement the most essential feature of a Composite structure: the ability to add child nodes to any node that's not a leaf. (In this case, that means you can add submenus

or menu options to any menu.) We'll also implement a so-called fluent interface to make the Composite easier to use in programming. We'll round off the implementation by using recursion to mark the path to a menu option. Finally, we'll discuss the fact that the implementation could be more efficient.

7.6.1 Implementing a menu as a Composite

Let's try an example: a menu for navigation on a web page such as the example in figure 7.4. Even if we have only one set of menu headings, there are still implicitly three levels of menus, since the structure as a whole is a menu. This makes it a strong candidate for a Composite structure.

Figure 7.7 A simple navigation menu

The menu has only what little functionality is needed to illustrate the Composite. We want the structure itself and the ability to mark the current menu option and the path to it. If we've chosen Events and then Movies, both Events and Movies will be shown with a style that distinguishes them from the rest of the menu, as shown in figure 7.7.

First, let's sketch the objects for the first two submenus of this menu. Figure 7.8 shows how it can be represented. Each menu has a set of menu or menu option objects stored in instance variables, or more likely, in one instance variable which is an array of objects. To represent the fact that some of the menus and menu options are marked, we have a simple Boolean (TRUE/FALSE flag). In the HTML code, we will want to represent this as a CSS class, but we're keeping the HTML representation out of this for now to keep it simple. Furthermore, each menu or menu option has a string for the label. And there is a menu object to represent the menu as a whole. Its label will not be shown on the web page, but it's practical when we want to handle the menu.

A class diagram for the Composite class structure to represent menus and menu options is shown in figure 7.9 It is quite a bit more abstract, but should be easier to grasp based on the previous illustration. Figure 7.8 is a snapshot of a particular set of object instances at a particular time; figure 7.9 represents the class structure and the operations needed to generate the objects.

There are three different bits of functionality in this design:

- Each menu and each menu option has a *label*, the text that is displayed on the web page.
- The `add()` method of the Menu class is the one method that is absolutely required for generating a Composite tree structure.
- The rest of the methods and attributes are necessary to make it possible to mark the current menu and menu option.

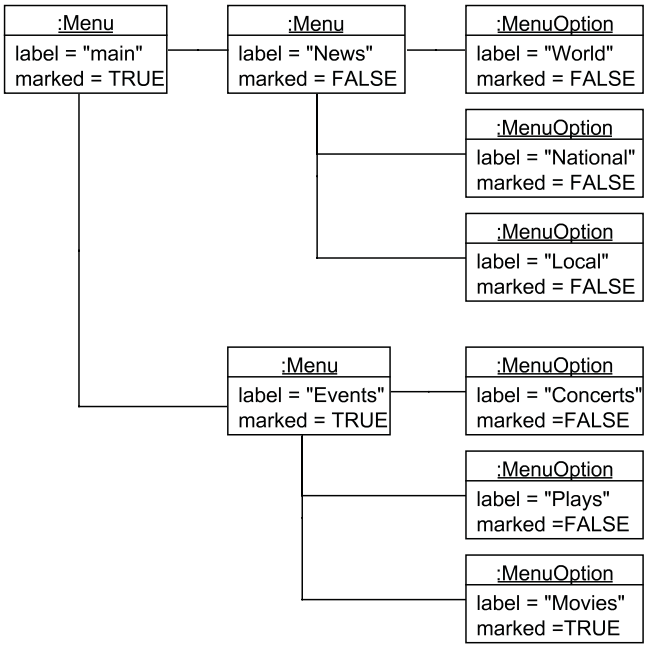

Figure 7.8 An object structure for the first two submenus

The two methods `hasMenuOptionWithId()` and `markPathToMenuOption()` are abstract in the MenuComponent class. This implies that they must exist in the Menu and MenuOption classes, even though they are not shown in these classes in the diagram.

The leftmost connection from Menu to MenuComponent implies the fact—which is clear in figure 7.8 as well—that a Menu object can have any number of menu components (Menu or MenuOption objects).

Methods to get and set the attributes are not included in the illustration.

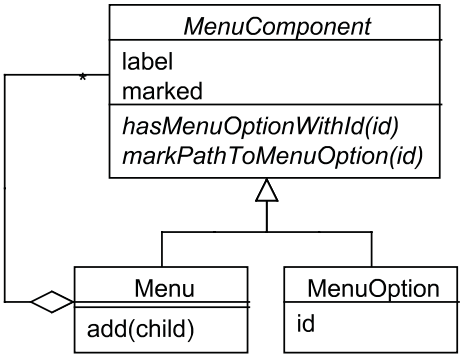

Figure 7.9
A Composite used to represent a menu with menu options in which the current menu option can be marked

7.6.2 The basics

Moving on to the code, we will start with the MenuComponent class. This class expresses what's similar between menus and menu options (listing 7.9). Both menus and menu options need a label and the ability to be marked as current.

Listing 7.9 Abstract class to express similarities between menus and menu options

```
abstract class MenuComponent {
    protected $marked = FALSE;
    protected $label;

    public function mark() { $this->marked = TRUE; }
    public function isMarked() { return $this->marked; }

    public function getLabel() { return $this->label; }
    public function setLabel($label) { $this->label = $label; }

    abstract public function hasMenuOptionWithId($id);

    abstract public function markPathToMenuOption($id);
}
```


❶ `mark()` and `isMarked()` let us set and retrieve the state of being marked as current.

❷ We have simple accessors for the label. We will also set the label in the constructor, but we're leaving that part of it to the child classes.

❸ `markPathToMenuOption()` will be the method for marking the path; both the menu object and the menu option object have to implement it. `hasMenuOptionWithId()` exists to support the marking operation.

To implement the most basic Composite structure, all we need is an `add()` method to add a child to a node (a menu or menu option in this case).

```
class Menu extends MenuComponent {
    protected $marked = FALSE;
    protected $label;
    private $children = array();

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

    public function add($child) {
        $this->children[] = $child;
    }
}
```

`add()` does not know or care whether the object being added is a menu or a menu option. We can build an arbitrarily complex structure with this alone:

```
$menu = new Menu('News');
$submenu = new Menu('Events');
$menu->add($submenu);
$submenu = new Menu('Concerts');
$menu->add($submenu);
```

7.6.3 A fluent interface

This reuse of temporary variables is rather ugly. Fortunately, it's easy to achieve what's known as a *fluent interface*:

```
$menu->add(new Menu('Events'))->add(new Menu('Concerts'));
```

All we have to do is return the child after adding it:

```
public function add($child) {
    $this->children[] = $child;
    return $child;
}
```

Or even simpler:

```
public function add($child) {
    return $this->children[] = $child;
}
```

A mentioned, this is all we need to build arbitrarily complex structures. In fact, if the menu option is able to store a link URL, we already have something that could possibly be useful in a real application.

7.6.4 Recursive processing

But we haven't finished our study of the Composite pattern until we've tried using it for recursion. Our original requirement was to be able to mark the path to the currently selected menu option. To achieve that, we need to identify the menu option. Let's assume that the menu option has an ID, and that the HTTP request contains this ID. So we have the menu option ID and want to mark the path to the menu option with that ID. Unfortunately, the top node of our composite menu structure cannot tell us where the menu option with that ID is located.

We'll do what might be the Simplest Thing That Could Possibly Work: search for it. The first step is to give any node in the structure the ability to tell us whether it contains that particular menu option. The Menu object can do that by iterating over its children and asking all of them whether they have the menu option. If one of them does, it returns `TRUE`, if none of them do, it returns `FALSE`:

```
class Menu extends MenuComponent...
    public function hasMenuOptionWithId($id) {
        foreach ($this->children as $child) {
            if ($child->hasMenuOptionWithId($id)) return TRUE;
```

```
        }
        return FALSE;
    }
}
```

The recursion has to end somewhere. Therefore, we need the equivalent method in the MenuOption class to do something different. It simply checks whether its ID is the one we are looking for, and returns `TRUE` if it is:

```
class MenuOption extends MenuComponent {
    protected $marked = FALSE;
    protected $label;
    private $id;

    public function __construct($label,$id) {
        $this->label = $label;
        $this->id = $id;
    }
    public function hasMenuOptionWithId($id) {
        return $id == $this->id;
    }
}
```

Now we're ready to mark the path.

```
class Menu extends MenuComponent...
    public function markPathToMenuOption($id) {
        if (!$this->hasMenuOptionWithId($id)) return FALSE;
        $this->mark();
        foreach ($this->children as $child) {
            $child->markPathToMenuOption($id);
        }
    }
}
```

If this menu contains the menu option with the given ID, it marks itself and passes the task on to its children. Only the one child that contains the desired menu option will be marked.

The MenuOption class also has to implement the `markPathToMenuOption()` method. It's quite simple:

```
class MenuOption extends MenuComponent...
    public function markPathToMenuOption($id) {
        if ($this->hasMenuOptionWithId($id)) $this->mark();
    }
}
```

But our traversal algorithm is not the most efficient one. We're traversing parts of the tree repeatedly. Do we need to change that?

7.6.5 Is this inefficient?

We have deliberately sacrificed efficiency in favor of readability, since the data structure will never be very large. The implementation uses one method (`hasMenuOptionWithId`) to answer a question and another (`markPathToMenuOption`) to

make a change. This is a good idea, which is why there is a refactoring to achieve this separation, called *Separate Query from Modifier*.

To make it slightly faster, we could have let the first method return the child that contains the menu option we're searching for. That would have enabled us to avoid the second round of recursion. But it would also have made the intent of the `hasMenuOptionWithId()` method more complex and therefore harder to understand. It would have been premature optimization.

And this premature optimization would have involved a premature, low-quality decision. If we did want to optimize the algorithm, approaching optimization as a task in itself, we should be looking at more alternatives. For example, we could do the search, have it return a path to the menu option as a sequence of array indexes, and then follow the path. Or we could do it with no recursion at all if we kept a list of all menu options indexed by ID and added references back to the parents in the composite structure. Starting with the menu option, we could traverse the path up to the root node, marking the nodes along the way.

One thing the Composite pattern does is to hide the difference between one and many. The Composite, containing many elements, can have the same methods as a single element. Frequently, the client need not know the difference. In chapter 17, we will see how this works in the context of input validation. A validator object may have a `validate()` method that works the same way whether it is a simple validator or a complex one that applies several different criteria.

The Composite View pattern (which is the main subject of chapter 14) is related, though not as closely as you might think.

7.7 SUMMARY

While design principles are approximate guidelines, design patterns are more like specific recipes or blueprints; they cannot be used mindlessly. To apply them, we need to understand where, how, and why they're useful. We need to look at context, consider alternatives, tweak the specifics, and use the object-oriented principles in our decision-making.

We have seen a small selection of design patterns. All of them are concerned with creating pluggable components. Strategy is the way to configure an object's behavior by adding a pluggable component. Adapter takes a component that is not pluggable and makes it pluggable. Decorator adds features without impairing pluggability. Null Object is a component that does nothing, but can be substituted for another to prevent a behavior from happening without interfering with the smooth running of the system. Iterator is a pluggable repetition engine that can even be a replacement for an array. Composite is a way to plug more than one component into a socket that's designed for just one.

In the next chapter, we will use date and time handling as a vehicle for making the context and the alternatives for design principles and patterns clearer.

PHP/PROGRAMMING

PHP IN ACTION

Dagfinn Reiersøl • Marcus Baker • Chris Shiflett

PHP is known for quick web scripts. But it has also been used for some of the most demanding, high-performance applications on the Web. The techniques taught in this book are a must for anyone who wants to build robust PHP web applications, whether large or small. This book will help you master professional design and development concepts including unit testing, refactoring, and design patterns. You will learn good web-programming styles, adopt agile methods, and learn the practical benefits of object-orientation.

Written by experienced developers in a clean and self-assured manner, **PHP in Action** offers a unique set of qualities—it is pragmatic and readable, and it makes object oriented philosophy come to life with concrete examples.

PHP in Action introduces you to advanced development topics in an approachable and immediately useful way. You will learn not only how to do OO PHP but also why. The book is refreshingly undogmatic: it gives you a straightforward understanding of OOP's strengths and weeknesses, as well as why for many tasks you're better off staying with procedural code.

"Well written and focused—these authors are experts."

—Kiernan Mathieson
Associate Professor of Management Information Systems, Oakland University

"A textbook for beginners and the quintessential reference for the rest of us. They have outdone themselves with this masterpiece."

—Andrew Grothe
COO, Eliptic Webwise

What's Inside

- Design patterns and how they help
- Unit testing and test-driven development
- In-depth server-side and client-side input validation
- How to handle database connections in an OO application
- How to write database-independent code
- Design patterns for web interfaces and OO data access
- Security in PHP

Since 1997, **Dagfinn Reiersøl** has designed and developed web applications, web content mining software, web programming tools, and text analysis programs, mostly in PHP. He and coauthors **Marcus Baker** and **Chris Shifflet** are active members of the PHP community.

For more information, code samples, and to purchase an ebook visit www.manning.com/PHPinAction

ISBN-10: 1-932394-75-3
ISBN-13: 978-1-932394-75-7

$39.99 / Can $51.99