CHAPTER 11

Performing and Representing Tasks

In this chapter, we get active. I look at patterns that help you to get things done, whether interpreting a minilanguage or encapsulating an algorithm.

This chapter will cover

- *The Interpreter pattern*: Building a minilanguage interpreter that can be used to create scriptable applications
- *The Strategy pattern*: Identifying algorithms in a system and encapsulating them into their own types
- *The Observer pattern*: Creating hooks for alerting disparate objects about system events
- *The Visitor pattern*: Applying an operation to all the nodes in a tree of objects
- *The Command pattern*: Creating command objects that can be saved and passed around

The Interpreter Pattern

Languages are written in other languages (at least at first). PHP itself, for example, is written in C. By the same token, odd as it may sound, you can define and run your own languages using PHP. Of course, any language you might create will be slow and somewhat limited. Nonetheless, minilanguages can be very useful, as you will see in this chapter.

The Problem

When you create web (or command line) interfaces in PHP, you give the user access to functionality. The trade-off in interface design is between power and ease of use. As a rule, the more power you give your user, the more cluttered and confusing your interface becomes. Good interface design can help a lot here, of course, but if 90 percent of users are using the same 30 percent of your features, the costs of piling on the functionality may outweigh the benefits. You may wish to consider simplifying your system for most users. But what of the power users, that 10 percent who use your system's advanced features? Perhaps you can accommodate them in a different way. By offering such users a domain language (often called a DSL—Domain Specific Language), you might actually extend the power of your application.

Of course, you have a programming language at hand right away. It's called PHP. Here's how you could allow your users to script your system:

```
$form_input = $_REQUEST['form_input'];
// contains: "print file_get_contents('/etc/passwd');"
eval( $form_input );
```

This approach to making an application scriptable is clearly insane. Just in case the reasons are not blatantly obvious, they boil down to two issues: security and complexity. The security issue is well addressed in the example. By allowing users to execute PHP via your script, you are effectively giving them access to the server the script runs on. The complexity issue is just as big a drawback. No matter how clear your code is, the average user is unlikely to extend it easily and certainly not from the browser window.

A minilanguage, though, can address both these problems. You can design flexibility into the language, reduce the possibility that the user can do damage, and keep things focused.

Imagine an application for authoring quizzes. Producers design questions and establish rules for marking the answers submitted by contestants. It is a requirement that quizzes must be marked without human intervention, even though some answers can be typed into a text field by users.

Here's a question:

```
How many members in the Design Patterns gang?
```

You can accept "four" or "4" as correct answers. You might create a web interface that allows a producer to use regular expression for marking responses:

```
^4|four$
```

Most producers are not hired for their knowledge of regular expressions, however. To make everyone's life easier, you might implement a more user-friendly mechanism for marking responses:

```
$input equals "4" or $input equals "four"
```

You propose a language that supports variables, an operator called `equals` and Boolean logic (`or` and `and`). Programmers love naming things, so let's call it MarkLogic. It should be easy to extend, as you envisage lots of requests for richer features. Let's leave aside the issue of parsing input for now and concentrate on a mechanism for plugging these elements together at runtime to produce an answer. This, as you might expect, is where the Interpreter pattern comes in.

Implementation

A language is made up of expressions (that is, things that resolve to a value). As you can see in Table 11–1, even a tiny language like MarkLogic needs to keep track of a lot of elements.

Table 11–1. *Elements of the MarkLogic Grammar*

Description	EBNF Name	Class Name	Example
Variable	`variable`	`VariableExpression`	`$input`
String literal	`<stringLiteral>`	`LiteralExpression`	`"four"`
Boolean and	`indexer`	`BooleanAndExpression`	`-$input equals '4' and $other equals '6'`
Boolean or	`orExpr`	`BooleanOrExpression`	`-$input equals '4' or $other equals '6'`
Equality test	`equalsExpr`	`EqualsExpression`	`$input equals '4'`

Table 11–1 lists EBNF names. So what is EBNF all about? It's a notation that you can use to describe a language grammar. EBNF stands for Extended Backus-Naur Form. It consists of a series of lines (called productions), each one consisting of a name and a description that takes the form of references to other productions and to terminals (that is, elements that are not themselves made up of references to other productions). Here is one way of describing my grammar using EBNF:

```
expr     ::= operand (orExpr | andExpr )*
operand  ::= ( '(' expr ')' | <stringLiteral> | variable ) ( eqExpr )*
orExpr   ::= 'or' operand
andExpr  ::= 'and' operand
eqExpr   ::= 'equals' operand
variable ::= '$' <word>
```

Some symbols have special meanings (that should be familiar from regular expression notation): * means zero or more, for example, and | means or. You can group elements using brackets. So in the example, an expression (`expr`) consists of an `operand` followed by zero or more of either `orExpr` or `andExpr`. An `operand` can be a bracketed expression, a quoted string (I have omitted the production for this), or a `variable` followed by zero or more instances of `eqExpr`. Once you get the hang of referring from one production to another, EBNF becomes quite easy to read.

In Figure 11–1, I represent the elements of my grammar as classes.

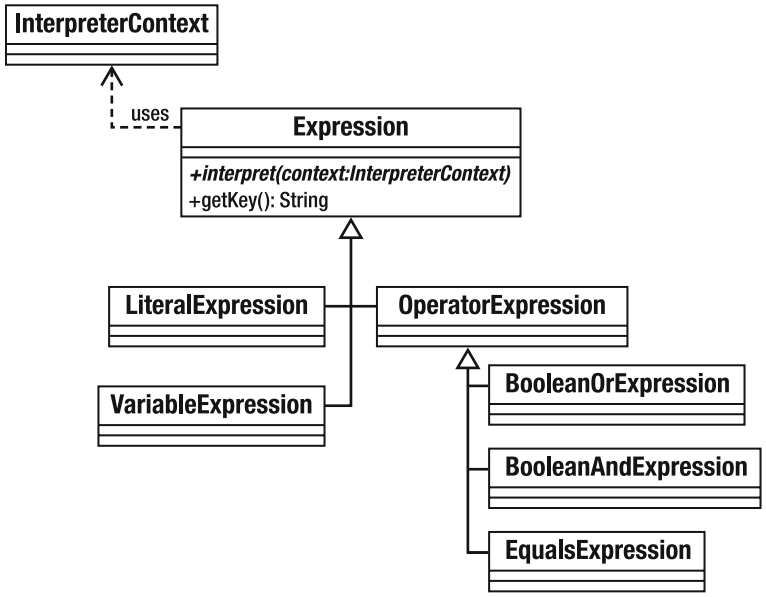

Figure 11–1. *The Interpreter classes that make up the MarkLogic language*

As you can see, `BooleanAndExpression` and its siblings inherit from `OperatorExpression`. This is because these classes all perform their operations upon other `Expression` objects. `VariableExpression` and `LiteralExpression` work directly with values.

All `Expression` objects implement an `interpret()` method that is defined in the abstract base class, `Expression`. The `interpret()` method expects an `InterpreterContext` object that is used as a shared data store. Each `Expression` object can store data in the `InterpreterContext` object. The `InterpreterContext`

will then be passed along to other Expression objects. So that data can be retrieved easily from the InterpreterContext, the Expression base class implements a getKey() method that returns a unique handle. Let's see how this works in practice with an implementation of Expression:

```
abstract class Expression {
    private static $keycount=0;
    private $key;
    abstract function interpret( InterpreterContext $context );

    function getKey() {
        if ( ! asset( $this->key ) ) {
            self::$keycount++;
            $this->key=self::$keycount;
        }
        return $this->key;
    }
}

class LiteralExpression extends Expression {
    private $value;

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

    function interpret( InterpreterContext $context ) {
        $context->replace( $this, $this->value );
    }
}

class InterpreterContext {
    private $expressionstore = array();

    function replace( Expression $exp, $value ) {
        $this->expressionstore[$exp->getKey()] = $value;
    }

    function lookup( Expression $exp ) {
        return $this->expressionstore[$exp->getKey()];
    }
}

$context = new InterpreterContext();
$literal = new LiteralExpression( 'four');
$literal->interpret( $context );
print $context->lookup( $literal ) . "\n";
```

Here's the output:

```
four
```

I'll begin with the InterpreterContext class. As you can see, it is really only a front end for an associative array, $expressionstore, which I use to hold data. The replace() method accepts an Expression object as key and a value of any type, and adds the pair to $expressionstore. It also provides a lookup() method for retrieving data.

The Expression class defines the abstract interpret() method and a concrete getKey() method that uses a static counter value to generate, store, and return an identifier.

This method is used by InterpreterContext::lookup() and InterpreterContext::replace() to index data.

The LiteralExpression class defines a constructor that accepts a value argument. The interpret() method requires a InterpreterContext object. I simply call replace(), using getKey() to define the key for retrieval and the $value property. This will become a familiar pattern as you examine the other expression classes. The interpret() method always inscribes its results upon the InterpreterContext object.

I include some client code as well, instantiating both an InterpreterContext object and a LiteralExpression object (with a value of "four"). I pass the InterpreterContext object to LiteralExpression::interpret(). The interpret() method stores the key/value pair in InterpreterContext, from where I retrieve the value by calling lookup().

Here's the remaining terminal class. VariableExpression is a little more complicated:

```
class VariableExpression extends Expression {
    private $name;
    private $val;

    function __construct( $name, $val=null ) {
        $this->name = $name;
        $this->val = $val;
    }

    function interpret( InterpreterContext $context ) {
        if ( ! is_null( $this->val ) ) {
            $context->replace( $this, $this->val );
            $this->val = null;
        }
    }

    function setValue( $value ) {
        $this->val = $value;
    }

    function getKey() {
        return $this->name;
    }
}
$context = new InterpreterContext();
$myvar = new VariableExpression( 'input', 'four');
$myvar->interpret( $context );
print $context->lookup( $myvar ). "\n";
// output: four

$newvar = new VariableExpression( 'input' );
$newvar->interpret( $context );
print $context->lookup( $newvar ). "\n";
// output: four
```

```
$myvar->setValue("five");
$myvar->interpret( $context );
print $context->lookup( $myvar ). "\n";
// output: five
print $context->lookup( $newvar ) . "\n";
// output: five
```

The VariableExpression class accepts both name and value arguments for storage in property variables. I provide the setValue() method so that client code can change the value at any time.

The interpret() method checks whether or not the $val property has a nonnull value. If the $val property has a value, it sets it on the InterpreterContext. I then set the $val property to null. This is in case interpret() is called again after another identically named instance of VariableExpression has changed the value in the InterpreterContext object. This is quite a limited variable, accepting only string values as it does. If you were going to extend your language, you should consider having it work with other Expression objects, so that it could contain the results of tests and operations. For now, though, VariableExpression will do the work I need of it. Notice that I have overridden the getKey() method so that variable values are linked to the variable name and not to an arbitrary static ID.

Operator expressions in the language all work with two other Expression objects in order to get their job done. It makes sense, therefore, to have them extend a common superclass. Here is the OperatorExpression class:

```
abstract class OperatorExpression extends Expression {
    protected $l_op;
    protected $r_op;

    function __construct( Expression $l_op, Expression $r_op ) {
        $this->l_op = $l_op;
        $this->r_op = $r_op;
    }

    function interpret( InterpreterContext $context ) {
        $this->l_op->interpret( $context );
        $this->r_op->interpret( $context );
        $result_l = $context->lookup( $this->l_op );
        $result_r = $context->lookup( $this->r_op  );
        $this->doInterpret( $context, $result_l, $result_r );
    }

    protected abstract function doInterpret( InterpreterContext $context,
                                             $result_l,
                                             $result_r );
}
```

OperatorExpression is an abstract class. It implements interpret(), but it also defines the abstract doInterpret() method.

The constructor demands two Expression objects, $l_op and $r_op, which it stores in properties.

The interpret() method begins by invoking interpret() on both its operand properties (if you have read the previous chapter, you might notice that I am creating an instance of the Composite pattern here). Once the operands have been run, interpret() still needs to acquire the values that this yields. It does this by calling InterpreterContext::lookup() for each property. It then calls doInterpret(), leaving it up to child classes to decide what to do with the results of these operations.

■ **Note** `doInterpret()` is an instance of the Template Method pattern. In this pattern, a parent class both defines and calls an abstract method, leaving it up to child classes to provide an implementation. This can streamline the development of concrete classes, as shared functionality is handled by the superclass, leaving the children to concentrate on clean, narrow objectives.

Here's the `EqualsExpression` class, which tests two `Expression` objects for equality:

```
class EqualsExpression extends OperatorExpression {
    protected function doInterpret( InterpreterContext $context,
                                    $result_l, $result_r ) {
            $context->replace( $this, $result_l == $result_r );
    }
}
```

`EqualsExpression` only implements the `doInterpret()` method, which tests the equality of the operand results it has been passed by the `interpret()` method, placing the result in the `InterpreterContext` object.

To wrap up the `Expression` classes, here are `BooleanOrExpression` and `BooleanAndExpression`:

```
class BooleanOrExpression extends OperatorExpression {
    protected function doInterpret( InterpreterContext $context,
                                    $result_l, $result_r ) {
        $context->replace( $this, $result_l || $result_r );
    }
}

class BooleanAndExpression extends OperatorExpression {
    protected function doInterpret( InterpreterContext $context,
                                    $result_l, $result_r ) {
        $context->replace( $this, $result_l && $result_r );
    }
}
```

Instead of testing for equality, the `BooleanOrExpression` class applies a logical `or` operation and stores the result of that via the `InterpreterContext::replace()` method. `BooleanAndExpression`, of course, applies a logical `and` operation.

I now have enough code to execute the minilanguage fragment I quoted earlier. Here it is again:

```
$input equals "4" or $input equals "four"
```

Here's how I can build this statement up with my `Expression` classes:

```
$context = new InterpreterContext();
$input = new VariableExpression( 'input' );
$statement = new BooleanOrExpression(
    new EqualsExpression( $input, new LiteralExpression( 'four' ) ),
    new EqualsExpression( $input, new LiteralExpression( '4' ) )
);
```

I instantiate a variable called `'input'` but hold off on providing a value for it. I then create a `BooleanOrExpression` object that will compare the results from two `EqualsExpression` objects. The first of

these objects compares the VariableExpression object stored in $input with a LiteralExpression containing the string "four"; the second compares $input with a LiteralExpression object containing the string "4".

Now, with my statement prepared, I am ready to provide a value for the input variable, and run the code:

```
foreach ( array( "four", "4", "52" ) as $val ) {
    $input->setValue( $val );
    print "$val:\n";
    $statement->interpret( $context );
    if ( $context->lookup( $statement ) ) {
        print "top marks\n\n";
    } else {
        print "dunce hat on\n\n";
    }
}
```

In fact, I run the code three times, with three different values. The first time through, I set the temporary variable $val to "four", assigning it to the input VariableExpression object using its setValue() method. I then call interpret() on the topmost Expression object (the BooleanOrExpression object that contains references to all other expressions in the statement). Here are the internals of this invocation step by step:

- $statement calls interpret() on its $l_op property (the first EqualsExpression object).
- The first EqualsExpression object calls interpret() on *its* $l_op property (a reference to the input VariableExpression object which is currently set to "four").
- The input VariableExpression object writes its current value to the provided InterpreterContext object by calling InterpreterContext::replace().
- The first EqualsExpression object calls interpret() on its $r_op property (a LiteralExpression object charged with the value "four").
- The LiteralExpression object registers its key and its value with InterpreterContext.
- The first EqualsExpression object retrieves the values for $l_op ("four") and $r_op ("four") from the InterpreterContext object.
- The first EqualsExpression object compares these two values for equality and registers the result (true) together with its key with the InterpreterContext object.
- Back at the top of the tree the $statement object (BooleanOrExpression) calls interpret() on its $r_op property. This resolves to a value (false, in this case) in the same way as the $l_op property did.
- The $statement object retrieves values for each of its operands from the InterpreterContext object and compares them using ||. It is comparing true and false, so the result is true. This final result is stored in the InterpreterContext object.

And all that is only for the first iteration through the loop. Here is the final output:

```
four:
top marks
```

```
4:
top marks

52:
dunce hat on
```

You may need to read through this section a few times before the process clicks. The old issue of object versus class trees might confuse you here. `Expression` classes are arranged in an inheritance hierarchy just as `Expression` objects are composed into a tree at runtime. As you read back through the code, keep this distinction in mind.

Figure 11–2 shows the complete class diagram for the example.

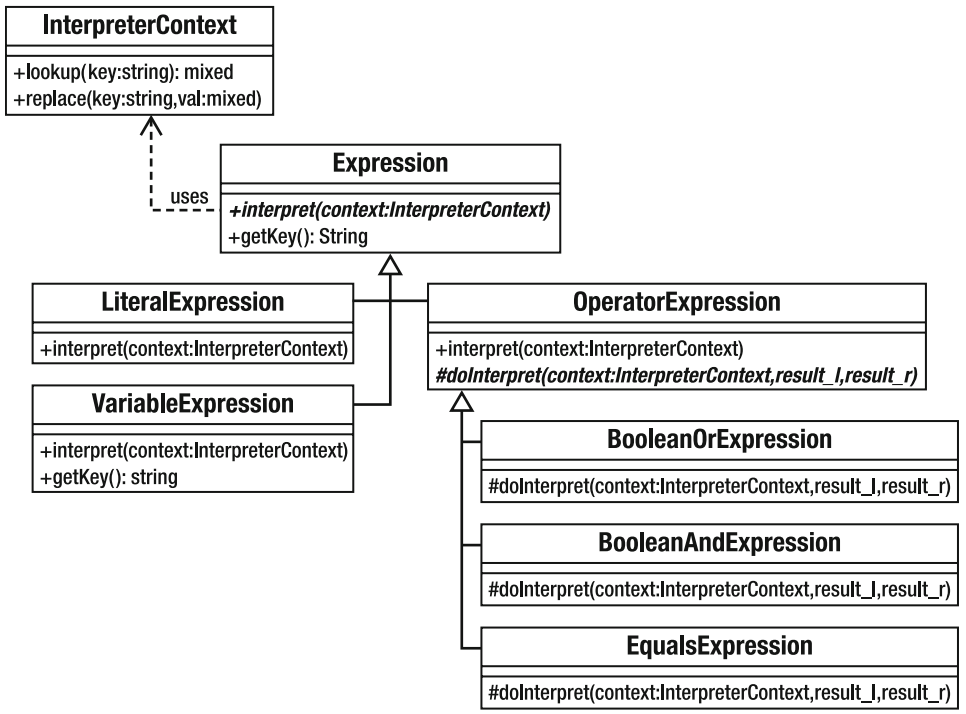

Figure 11–2. *The Interpreter pattern deployed*

Interpreter Issues

Once you have set up the core classes for an Interpreter pattern implementation, it becomes easy to extend. The price you pay is in the sheer number of classes you could end up creating. For this reason, Interpreter is best applied to relatively small languages. If you have a need for a full programming language, you would do better to look for a third-party tool to use.

Because Interpreter classes often perform very similar tasks, it is worth keeping an eye on the classes you create with a view to factoring out duplication.

Many people approaching the Interpreter pattern for the first time are disappointed, after some initial excitement, to discover that it does not address parsing. This means that you are not yet in a

position to offer your users a nice, friendly language. Appendix B contains some rough code to illustrate one strategy for parsing a minilanguage.

The Strategy Pattern

Classes often try to do too much. It's understandable: you create a class that performs a few related actions. As you code, some of these actions need to be varied according to circumstances. At the same time, your class needs to be split into subclasses. Before you know it, your design is being pulled apart by competing forces.

The Problem

Since I have recently built a marking language, I'm sticking with the quiz example. Quizzes need questions, so you build a `Question` class, giving it a `mark()` method. All is well until you need to support different marking mechanisms.

Imagine you are asked to support the simple MarkLogic language, marking by straight match and marking by regular expression. Your first thought might be to subclass for these differences, as in Figure 11–3.

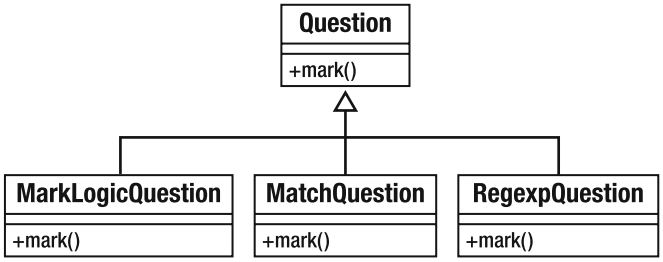

Figure 11–3. *Defining subclasses according to marking strategies*

This would serve you well as long as marking remains the only aspect of the class that varies. Imagine, though, that you are called on to support different kinds of questions: those that are text based and those that support rich media. This presents you with a problem when it comes to incorporating these forces in one inheritance tree as you can see in Figure 11–4.

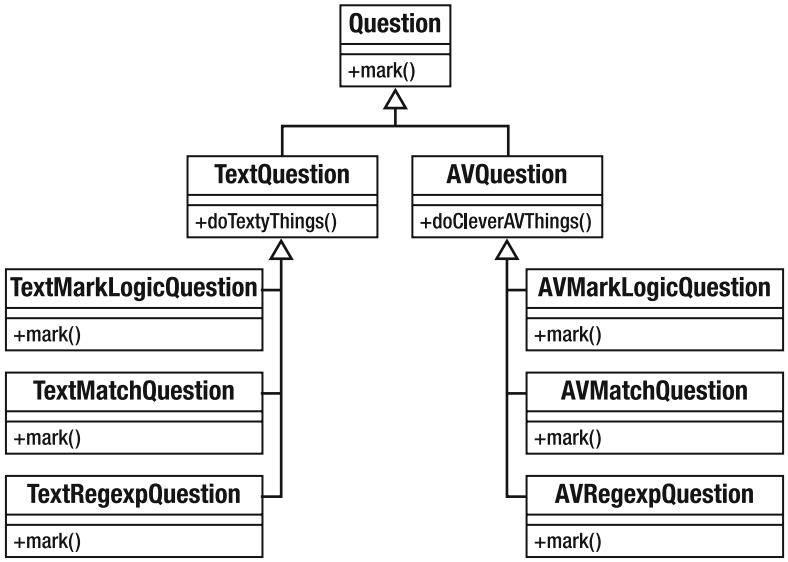

Figure 11–4. *Defining subclasses according to two forces*

Not only have the number of classes in the hierarchy ballooned, but you also necessarily introduce repetition. Your marking logic is reproduced across each branch of the inheritance hierarchy.

Whenever you find yourself repeating an algorithm across siblings in an inheritance tree (whether through subclassing or repeated conditional statements), consider abstracting these behaviors into their own type.

Implementation

As with all the best patterns, Strategy is simple and powerful. When classes must support multiple implementations of an interface (multiple marking mechanisms, for example), the best approach is often to extract these implementations and place them in their own type, rather than to extend the original class to handle them.

So, in the example, your approach to marking might be placed in a `Marker` type. Figure 11–5 shows the new structure.

Remember the Gang of Four principle "favor composition over inheritance"? This is an excellent example. By defining and encapsulating the marking algorithms, you reduce subclassing and increase flexibility. You can add new marking strategies at any time without the need to change the `Question` classes at all. All `Question` classes know is that they have an instance of a `Marker` at their disposal, and that it is guaranteed by its interface to support a `mark()` method. The details of implementation are entirely somebody else's problem.

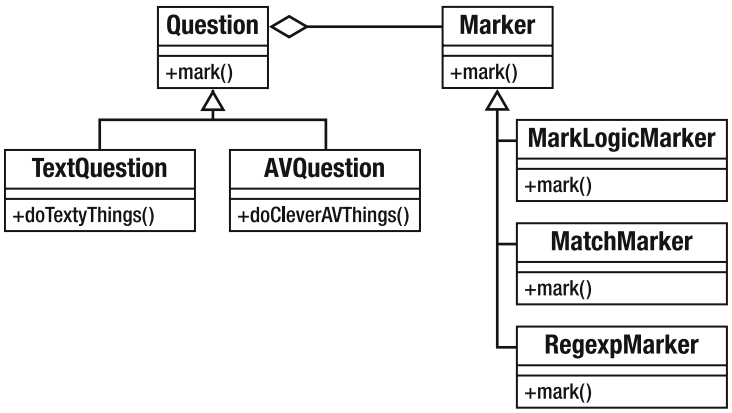

Figure 11–5. *Extracting algorithms into their own type*

Here are the `Question` classes rendered as code:

```
abstract class Question {
    protected $prompt;
    protected $marker;

    function __construct( $prompt, Marker $marker ) {
        $this->marker=$marker;
        $this->prompt = $prompt;
    }

    function mark( $response ) {
        return $this->marker->mark( $response );
    }
}

class TextQuestion extends Question {
    // do text question specific things
}

class AVQuestion extends Question {
    // do audiovisual question specific things
}
```

As you can see, I have left the exact nature of the difference between `TextQuestion` and `AVQuestion` to the imagination. The `Question` base class provides all the real functionality, storing a prompt property and a `Marker` object. When `Question::mark()` is called with a response from the end user, the method simply delegates the problem solving to its `Marker` object.

Now to define some simple `Marker` objects:

```
abstract class Marker {
    protected $test;

    function __construct( $test ) {
```

```
        $this->test = $test;
    }

    abstract function mark( $response );
}

class MarkLogicMarker extends Marker {
    private $engine;
    function __construct( $test ) {
        parent::__construct( $test );
        // $this->engine = new MarkParse( $test );
    }

    function mark( $response ) {
        // return $this->engine->evaluate( $response );
        // dummy return value
        return true;
    }
}

class MatchMarker extends Marker {
    function mark( $response ) {
        return ( $this->test == $response );
    }
}

class RegexpMarker extends Marker {
    function mark( $response ) {
        return ( preg_match( $this->test, $response ) );
    }
}
```

There should be little if anything that is particularly surprising about the `Marker` classes themselves. Note that the `MarkParse` object is designed to work with the simple parser developed in Appendix B. This isn't necessary for the sake of this example though, so I simply return a dummy value of `true` from `MarkLogicMarker::mark()`. The key here is in the structure that I have defined, rather than in the detail of the strategies themselves. I can swap `RegexpMarker` for `MatchMarker`, with no impact on the `Question` class.

Of course, you must still decide what method to use to choose between concrete `Marker` objects. I have seen two real-world approaches to this problem. In the first, producers use radio buttons to select the marking strategy they prefer. In the second, the structure of the marking condition is itself used: a match statement was left plain:

```
five
```

A MarkLogic statement was preceded by a colon:

```
:$input equals 'five'
```

and a regular expression used forward slashes:

```
/f.ve/
```

Here is some code to run the classes through their paces:

```
$markers = array(   new RegexpMarker( "/f.ve/" ),
```

```
                    new MatchMarker( "five" ),
                    new MarkLogicMarker( '$input equals "five"' )
        );

foreach ( $markers as $marker ) {
    print get_class( $marker )."\n";
    $question = new TextQuestion( "how many beans make five", $marker );
    foreach ( array( "five", "four" ) as $response ) {
        print "\tresponse: $response: ";
        if ( $question->mark( $response ) ) {
            print "well done\n";
        } else {
            print "never mind\n";
        }
    }
}
```

I construct three strategy objects, using each in turn to help construct a `TextQuestion` object. The `TextQuestion` object is then tried against two sample responses.

The `MarkLogicMarker` class shown here is a placeholder at present, and its `mark()` method always returns `true`. The commented out code does work, however, with the parser example shown in Appendix B, or could be made to work with a third-party parser.

Here is the output:

```
RegexpMarker
        response: five: well done
        response: four: never mind
MatchMarker
        response: five: well done
        response: four: never mind
MarkLogicMarker
        response: five: well done
        response: four: well done
```

Remember that the `MarkLogicMarker` in the example is a dummy which always returns `true`, so it marked both responses correct.

In the example, I passed specific data (the `$response` variable) from the client to the strategy object via the `mark()` method. Sometimes, you may encounter circumstances in which you don't always know in advance how much information the strategy object will require when its operation is invoked. You can delegate the decision as to what data to acquire by passing the strategy an instance of the client itself. The strategy can then query the client in order to build the data it needs.

The Observer Pattern

Orthogonality is a virtue I have described before. One of objectives as programmers should be to build components that can be altered or moved with minimal impact on other components. If every change you make to one component necessitates a ripple of changes elsewhere in the codebase, the task of development can quickly become a spiral of bug creation and elimination.

Of course, orthogonality is often just a dream. Elements in a system must have embedded references to other elements. You can, however, deploy various strategies to minimize this. You have

seen various examples of polymorphism in which the client understands a component's interface but the actual component may vary at runtime.

In some circumstances, you may wish to drive an even greater wedge between components than this. Consider a class responsible for handling a user's access to a system:

```
class Login {
    const LOGIN_USER_UNKNOWN = 1;
    const LOGIN_WRONG_PASS = 2;
    const LOGIN_ACCESS = 3;
    private $status = array();

    function handleLogin( $user, $pass, $ip ) {
        switch ( rand(1,3) ) {
            case 1:
                $this->setStatus( self::LOGIN_ACCESS, $user, $ip );
                $ret = true;
                break;
            case 2:
                $this->setStatus( self::LOGIN_WRONG_PASS, $user, $ip );
                $ret = false;
                break;
            case 3:
                $this->setStatus( self::LOGIN_USER_UNKNOWN, $user, $ip );
                $ret = false;
                break;
        }
        return $ret;
    }

    private function setStatus( $status, $user, $ip ) {
        $this->status = array( $status, $user, $ip );
    }

    function getStatus() {
        return $this->status;
    }

}
```

In a real-world example, of course, the `handleLogin()` method would validate the user against a storage mechanism. As it is, this class fakes the login process using the `rand()` function. There are three potential outcomes of a call to `handleLogin()`. The status flag may be set to `LOGIN_ACCESS`, `LOGIN_WRONG_PASS`, or `LOGIN_USER_UNKNOWN`.

Because the `Login` class is a gateway guarding the treasures of your business team, it may excite much interest during development and in the months beyond. Marketing might call you up and ask that you keep a log of IP addresses. You can add a call to your system's `Logger` class:

```
    function handleLogin( $user, $pass, $ip ) {
        switch ( rand(1,3) ) {
            case 1:
                $this->setStatus( self::LOGIN_ACCESS, $user, $ip );
                $ret = true;
                break;
            case 2:
```

```
                $this->setStatus( self::LOGIN_WRONG_PASS, $user, $ip );
                $ret = false;
                break;
            case 3:
                $this->setStatus( self::LOGIN_USER_UNKNOWN, $user, $ip );
                $ret = false;
                break;
        }
        Logger::logIP( $user, $ip, $this->getStatus() );
        return $ret;
    }
```

Worried about security, the system administrators might ask for notification of failed logins. Once again, you can return to the login method and add a new call:

```
        if ( ! $ret ) {
            Notifier::mailWarning( $user, $ip,
                                   $this->getStatus()  );
        }
```

The business development team might announce a tie-in with a particular ISP and ask that a cookie be set when particular users log in, and so on, and on.

These are all easy enough requests to fulfill but at a cost to your design. The `Login` class soon becomes very tightly embedded into this particular system. You cannot pull it out and drop it into another product without going through the code line by line and removing everything that is specific to the old system. This isn't too hard, of course, but then you are off down the road of cut-and-paste coding. Now that you have two similar but distinct `Login` classes in your systems, you find that an improvement to one will necessitate the same changes in the other, until inevitably and gracelessly they fall out of alignment with one another.

So what can you do to save the `Login` class? The Observer pattern is a powerful fit here.

Implementation

At the core of the Observer pattern is the unhooking of client elements (the observers) from a central class (the subject). Observers need to be informed when events occur that the subject knows about. At the same time, you do not want the subject to have a hard-coded relationship with its observer classes.

To achieve this, you can allow observers to register themselves with the subject. You give the `Login` class three new methods, `attach()`, `detach()`, and `notify()`, and enforce this using an interface called `Observable`:

```
interface Observable {
    function attach( Observer $observer );
    function detach( Observer $observer );
    function notify();
}

// ... Login class
class Login implements Observable {

    private $observers;
    //...

    function __construct() {
```

```
        $this->observers = array();

    }


    function attach( Observer $observer ) {
        $this->observers[] = $observer;
    }

    function detach( Observer $observer ) {
        $newobservers = array();
        foreach ( $this->observers as $obs ) {
            if (  ($obs !== $observer) ) {
                $newobservers[]=$obs;
            }
        }
        $this->observers = $newobservers;
    }

    function notify() {
        foreach ( $this->observers as $obs ) {
            $obs->update( $this );
      }
  }
//...
```

So the `Login` class manages a list of observer objects. These can be added by a third party using the `attach()` method and removed via `detach()`. The `notify()` method is called to tell the observers that something of interest has happened. The method simply loops through the list of observers, calling `update()` on each one.

The `Login` class itself calls `notify()` from its `handleLogin()` method.

```
function handleLogin( $user, $pass, $ip ) {
        switch ( rand(1,3) ) {
            case 1:
                $this->setStatus( self::LOGIN_ACCESS, $user, $ip );
                $ret = true; break;
            case 2:
                $this->setStatus( self::LOGIN_WRONG_PASS, $user, $ip );
                $ret = false; break;
            case 3:
                $this->setStatus( self::LOGIN_USER_UNKNOWN, $user, $ip );
                $ret = false; break;
        }
        $this->notify();
        return $ret;
    }
```

Here's the interface for the `Observer` class:

```
interface Observer {
    function update( Observable $observable );
}
```

Any object that uses this interface can be added to the Login class via the attach() method. Here's create a concrete instance:

```
class SecurityMonitor extends Observer {
    function update( Observable $observable ) {
        $status = $observable->getStatus();
        if ( $status[0] == Login::LOGIN_WRONG_PASS ) {
            // send mail to sysadmin
            print __CLASS__.":\tsending mail to sysadmin\n";
        }
    }
}
$login = new Login();
$login->attach( new SecurityMonitor() );
```

Notice how the observer object uses the instance of Observable to get more information about the event. It is up to the subject class to provide methods that observers can query to learn about state. In this case, I have defined a method called getStatus() that observers can call to get a snapshot of the current state of play.

This addition also highlights a problem, though. By calling Login::getStatus(), the SecurityMonitor class assumes more knowledge than it safely can. It is making this call on an Observable object, but there's no guarantee that this will also be a Login object. I have a couple of options here. I could extend the Observable interface to include a getStatus() declaration and perhaps rename it to something like ObservableLogin to signal that it is specific to the Login type.

Alternatively, I can keep the Observable interface generic and make the Observer classes responsible for ensuring that their subjects are of the correct type. They could even handle the chore of attaching themselves to their subject. Since there will be more than one type of Observer, and since I'm planning to perform some housekeeping that is common to all of them, here's an abstract superclass to handle the donkey work:

```
abstract class LoginObserver implements Observer {
    private $login;
    function __construct( Login $login ) {
        $this->login = $login;
        $login->attach( $this );
    }

    function update( Observable $observable ) {
        if ( $observable === $this->login ) {
            $this->doUpdate( $observable );
        }
    }

    abstract function doUpdate( Login $login );
}
```

The LoginObserver class requires a Login object in its constructor. It stores a reference and calls Login::attach(). When update() is called, it checks that the provided Observable object is the correct reference. It then calls a Template Method: doUpdate(). I can now create a suite of LoginObserver objects all of whom can be secure they are working with a Login object and not just any old Observable:

```
class SecurityMonitor extends LoginObserver {
    function doUpdate( Login $login ) {
        $status = $login->getStatus();
        if ( $status[0] == Login::LOGIN_WRONG_PASS ) {
```

```
            // send mail to sysadmin
            print __CLASS__.":\tsending mail to sysadmin\n";
        }
    }
}

class GeneralLogger  extends LoginObserver {
    function doUpdate( Login $login ) {
        $status = $login->getStatus();
        // add login data to log
        print __CLASS__.":\tadd login data to log\n";
    }
}

class PartnershipTool extends LoginObserver {
    function doUpdate( Login $login ) {
        $status = $login->getStatus();
        // check IP address
        // set cookie if it matches a list
        print __CLASS__.":\tset cookie if IP matches a list\n";
    }
}
```

Creating and attaching `LoginObserver` classes is now achieved in one go at the time of instantiation:

```
$login = new Login();
new SecurityMonitor( $login );
new GeneralLogger( $login );
new PartnershipTool( $login );
```

So now I have created a flexible association between the subject classes and the observers. You can see the class diagram for the example in Figure 11–6.

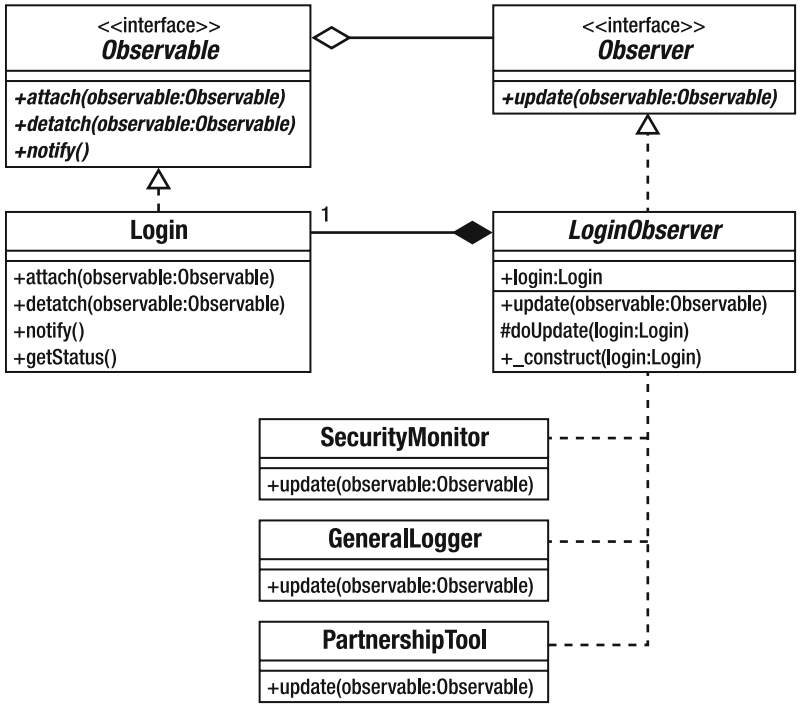

Figure 11–6. *The Observer pattern*

PHP provides built-in support for the Observer pattern through the bundled SPL (Standard PHP Library) extension. The SPL is a set of tools that help with common largely object-oriented problems. The Observer aspect of this OO Swiss Army knife consists of three elements: `SplObserver`, `SplSubject`, and `SplObjectStorage`. `SplObserver` and `SplSubject` are interfaces and exactly parallel the `Observer` and `Observable` interfaces shown in this section's example. `SplObjectStorage` is a utility class designed to provide improved storage and removal of objects. Here's an edited version of the Observer implementation:

```
class Login implements SplSubject {
    private $storage;
    //...

    function __construct() {
        $this->storage = new SplObjectStorage();
    }
    function attach( SplObserver $observer ) {
        $this->storage->attach( $observer );
    }

    function detach( SplObserver $observer ) {
        $this->storage->detach( $observer );
    }
```

```
    function notify() {
        foreach ( $this->storage as $obs ) {
            $obs->update( $this );
        }
    }
    //...
}

abstract class LoginObserver implements SplObserver {
    private $login;
    function __construct( Login $login ) {
        $this->login = $login;
        $login->attach( $this );
    }

    function update( SplSubject $subject ) {
        if ( $subject === $this->login ) {
            $this->doUpdate( $subject );
        }
    }

    abstract function doUpdate( Login $login );
}
```

There are no real differences as far as `SplObserver` (which was `Observer`) and `SplSubject` (which was `Observable`) are concerned, except, of course, I no longer need to declare the interfaces, and I must alter my type hinting according to the new names. `SplObjectStorage` provides you with a really useful service however. You may have noticed that in my initial example my implementation of `Login::detach()` applied `array_udiff` (together with an anoymous function) to the `$observable` array, in order to find and remove the argument object. The `SplObjectStorage` class does this work for you under the hood. It implements `attach()` and `detach()` methods and can be passed to `foreach` and iterated like an array.

■ **Note** You can read more about SPL in the PHP documentation at `http://www.php.net/spl`. In particular, you will find many iterator tools there. I cover PHP's built-in Iterator interface in Chapter 13, "Database Patterns."

Another approach to the problem of communicating between an `Observable` class and its `Observer` could be to pass specific state information via the `update()` method, rather than an instance of the subject class. For a quick-and-dirty solution, this is often the approach I would take initially. So in the example, `update()` would expect a status flag, the username, and IP address (probably in an array for portability), rather than an instance of `Login`. This saves you from having to write a state method in the `Login` class. On the other hand, where the subject class stores a lot of state, passing an instance of it to `update()` allows observers much more flexibility.

You could also lock down type completely, by making the `Login` class refuse to work with anything other than a specific type of observer class (`LoginObserver` perhaps). If you want to do that, then you may consider some kind of runtime check on objects passed to the `attach()` method; otherwise, you may need to reconsider the `Observable` interface altogether.

Once again, I have used composition at runtime to build a flexible and extensible model. The `Login` class can be extracted from the context and dropped into an entirely different project without qualification. There, it might work with a different set of observers.

The Visitor Pattern

As you have seen, many patterns aim to build structures at runtime, following the principle that composition is more flexible than inheritance. The ubiquitous Composite pattern is an excellent example of this. When you work with collections of objects, you may need to apply various operations to the structure that involve working with each individual component. Such operations can be built into the components themselves. After all, components are often best placed to invoke one another.

This approach is not without issues. You do not always know about all the operations you may need to perform on a structure. If you add support for new operations to your classes on a case-by-case basis, you can bloat your interface with responsibilities that don't really fit. As you might guess, the Visitor pattern addresses these issues.

The Problem

Think back to the Composite example from the previous chapter. For a game, I created an army of components such that the whole and its parts can be treated interchangeably. You saw that operations can be built into components. Typically, leaf objects perform an operation and composite objects call on their children to perform the operation.

```
class Army extends CompositeUnit {

    function bombardStrength() {
        $ret = 0;
        foreach( $this->units() as $unit ) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}

class LaserCannonUnit extends Unit {
    function bombardStrength() {
        return 44;
    }
}
```

Where the operation is integral to the responsibility of the composite class, there is no problem. There are more peripheral tasks, however, that may not sit so happily on the interface.

Here's an operation that dumps textual information about leaf nodes. It could be added to the abstract `Unit` class.

```
    // Unit
    function textDump( $num=0 ) {
        $ret = "";
        $pad = 4*$num;
        $ret .= sprintf( "%{$pad}s", "" );
        $ret .= get_class($this).": ";
        $ret .= "bombard: ".$this->bombardStrength()."\n";
```

```
        return $ret;
    }
```

This method can then be overridden in the `CompositeUnit` class:

```
    // CompositeUnit
    function textDump( $num=0 ) {
        $ret = parent::textDump( $num );
        foreach ( $this->units as $unit ) {
            $ret .= $unit->textDump( $num + 1 );
        }
        return $ret;
    }
```

I could go on to create methods for counting the number of units in the tree, for saving components to a database, and for calculating the food units consumed by an army.

Why would I want to include these methods in the composite's interface? There is only one really compelling answer. I include these disparate operations here because this is where an operation can gain easy access to related nodes in the composite structure.

Although it is true that ease of traversal is part of the Composite pattern, it does not follow that every operation that needs to traverse the tree should therefore claim a place in the Composite's interface.

So these are the forces at work. I want to take full advantage of the easy traversal afforded by my object structure, but I want to do this without bloating the interface.

Implementation

I'll begin with the interfaces. In the abstract `Unit` class, I define an `accept()` method:

```
    function accept( ArmyVisitor $visitor ) {
        $method = "visit".get_class( $this );
        $visitor->$method( $this );
    }

    protected function setDepth( $depth ) {
        $this->depth=$depth;
    }

    function getDepth() {
        return $this->depth;
    }
```

As you can see, the `accept()` method expects an `ArmyVisitor` object to be passed to it. PHP allows you dynamically to define the method on the `ArmyVisitor` you wish to call. This saves me from implementing `accept()` on every leaf node in my class hierarchy. While I was in the area, I also added two methods of convenience `getDepth()` and `setDepth()`. These can be used to store and retrieve the depth of a unit in a tree. `setDepth()` is invoked by the unit's parent when it adds it to the tree from `CompositeUnit::addUnit()`.

```
    function addUnit( Unit $unit ) {
        foreach ( $this->units as $thisunit ) {
            if ( $unit === $thisunit ) {
                return;
            }
        }
```

```
        $unit->setDepth($this->depth+1);
        $this->units[] = $unit;
    }
```

The only other `accept()` method I need to define is in the abstract composite class:

```
    function accept( ArmyVisitor $visitor ) {
        $method = "visit".get_class( $this );
        $visitor->$method( $this );
        foreach ( $this->units as $thisunit ) {
            $thisunit->accept( $visitor );
        }
    }
```

This method does the same as `Unit::accept()`, with one addition. It constructs a method name based on the name of the current class and invokes that method on the provided `ArmyVisitor` object. So if the current class is `Army`, then it invokes `ArmyVisitor::visitArmy()`, and if the current class is `TroopCarrier`, it invokes `ArmyVisitor::visitTroopCarrier()`, and so on. Having done this, it then loops through any child objects calling `accept()`. In fact, because `accept()` overrides its parent operation, I could factor out the repetition here:

```
    function accept( ArmyVisitor $visitor ) {
        parent::accept( $visitor );
        foreach ( $this->units as $thisunit ) {
            $thisunit->accept( $visitor );
        }
    }
```

Eliminating repetition in this way can be very satisfying, though in this case I have saved only one line, arguably at some cost to clarity. In either case, the `accept()` method allows me to do two things:

- Invoke the correct visitor method for the current component.
- Pass the visitor object to all the current element children via the `accept()` method (assuming the current component is composite).

I have yet to define the interface for `ArmyVisitor`. The `accept()` methods should give you some clue. The visitor class should define `accept()` methods for each of the concrete classes in the class hierarchy. This allows me to provide different functionality for different objects. In my version of this class, I also define a default `visit()` method that is automatically called if implementing classes choose not to provide specific handling for particular `Unit` classes.

```
abstract class ArmyVisitor  {
    abstract function visit( Unit $node );

    function visitArcher( Archer $node ) {
        $this->visit( $node );
    }

    function visitCavalry( Cavalry $node ) {
        $this->visit( $node );
    }

    function visitLaserCannonUnit( LaserCannonUnit $node ) {
        $this->visit( $node );
    }
```

```
    function visitTroopCarrierUnit( TroopCarrierUnit $node ) {
        $this->visit( $node );
    }

    function visitArmy( Army $node ) {
        $this->visit( $node );
    }
}
```

So now it's just a matter of providing implementations of `ArmyVisitor`, and I am ready to go. Here is the simple text dump code reimplemented as an `ArmyVisitor` object:

```
class TextDumpArmyVisitor extends ArmyVisitor {
    private $text="";

    function visit( Unit $node ) {
        $ret = "";
        $pad = 4*$node->getDepth();
        $ret .= sprintf( "%{$pad}s", "" );
        $ret .= get_class($node).": ";
        $ret .= "bombard: ".$node->bombardStrength()."\n";
        $this->text .= $ret;
    }

    function getText() {
        return $this->text;
    }
}
```

Let's look at some client code and then walk through the whole process:

```
$main_army = new Army();
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );
$main_army->addUnit( new Cavalry() );

$textdump = new TextDumpArmyVisitor();
$main_army->accept( $textdump  );
print $textdump->getText();
```

This code yields the following output:

```
Army: bombard: 50
    Archer: bombard: 4
    LaserCannonUnit: bombard: 44
    Cavalry: bombard: 2
```

I create an `Army` object. Because `Army` is composite, it has an `addUnit()` method that I use to add some more `Unit` objects. I then create the `TextDumpArmyVisitor` object. I pass this to the `Army::accept()`. The `accept()` method constructs a method call and invokes `TextDumpArmyVisitor::visitArmy()`. In this case, I have provided no special handling for `Army` objects, so the call is passed on to the generic `visit()` method. `visit()` has been passed a reference to the `Army` object. It invokes its methods (including the newly added, `getDepth()`, which tells anyone who needs to know how far down the object hierarchy the unit is) in order to generate summary data. The call to `visitArmy()` complete, the `Army::accept()`

operation now calls `accept()` on its children in turn, passing the visitor along. In this way, the `ArmyVisitor` class visits every object in the tree.

With the addition of just a couple of methods, I have created a mechanism by which new functionality can be plugged into my composite classes without compromising their interface and without lots of duplicated traversal code.

On certain squares in the game, armies are subject to tax. The tax collector visits the army and levies a fee for each unit it finds. Different units are taxable at different rates. Here's where I can take advantage of the specialized methods in the visitor class:

```
class TaxCollectionVisitor extends ArmyVisitor {
    private $due=0;
    private $report="";

    function visit( Unit $node ) {
        $this->levy( $node, 1 );
    }

    function visitArcher( Archer $node ) {
        $this->levy( $node, 2 );
    }

    function visitCavalry( Cavalry $node ) {
        $this->levy( $node, 3 );
    }

    function visitTroopCarrierUnit( TroopCarrierUnit $node ) {
        $this->levy( $node, 5 );
    }

    private function levy( Unit $unit, $amount ) {
        $this->report .= "Tax levied for ".get_class( $unit );
        $this->report .= ": $amount\n";
        $this->due += $amount;
    }

    function getReport() {
        return $this->report;
    }

    function getTax() {
        return $this->due;
    }
}
```

In this simple example, I make no direct use of the `Unit` objects passed to the various visit methods. I do, however, use the specialized nature of these methods, levying different fees according to the specific type of the invoking `Unit` object.

Here's some client code:

```
$main_army = new Army();
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );
$main_army->addUnit( new Cavalry() );
```

```
$taxcollector = new TaxCollectionVisitor();
$main_army->accept( $taxcollector );
print "TOTAL: ";
print $taxcollector->getTax()."\n";
```

The `TaxCollectionVisitor` object is passed to the `Army` object's `accept()` method as before. Once again, `Army` passes a reference to itself to the `visitArmy()` method, before calling `accept()` on its children. The components are blissfully unaware of the operations performed by their visitor. They simply collaborate with its public interface, each one passing itself dutifully to the correct method for its type.

In addition to the methods defined in the `ArmyVisitor` class, `TaxCollectionVisitor` provides two summary methods, `getReport()` and `getTax()`. Invoking these provides the data you might expect:

```
Tax levied for Army: 1
Tax levied for Archer: 2
Tax levied for LaserCannonUnit: 1
Tax levied for Cavalry: 3
TOTAL: 7
```

Figure 11–7 shows the participants in this example.

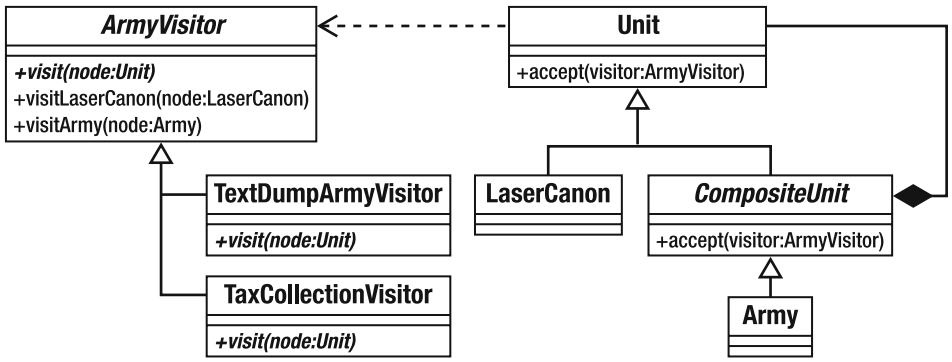

Figure 11–7. *The Visitor pattern*

Visitor Issues

The Visitor pattern, then, is another that combines simplicity and power. There are a few things to bear in mind when deploying this pattern, however.

First, although it is perfectly suited to the Composite pattern, Visitor can, in fact, be used with any collection of objects. So you might use it with a list of objects where each object stores a reference to its siblings, for example.

By externalizing operations, you may risk compromising encapsulation. That is, you may need to expose the guts of your visited objects in order to let visitors do anything useful with them. You saw, for example, that for the first Visitor example, I was forced to provide an additional method in the `Unit` interface in order to provide information for `TextDumpArmyVisitor` objects. You also saw this dilemma previously in the Observer pattern.

Because iteration is separated from the operations that visitor objects perform, you must relinquish a degree of control. For example, you cannot easily create a `visit()` method that does something both before and after child nodes are iterated. One way around this would be to move responsibility for

iteration into the visitor objects. The trouble with this is that you may end up duplicating the traversal code from visitor to visitor.

By default, I prefer to keep traversal internal to the visited classes, but externalizing it provides you with one distinct advantage. You can vary the way that you work through the visited classes on a visitor-by-visitor basis.

The Command Pattern

In recent years, I have rarely completed a web project without deploying this pattern. Originally conceived in the context of graphical user interface design, command objects make for good enterprise application design, encouraging a separation between the controller (request and dispatch handling) and domain model (application logic) tiers. Put more simply, the Command pattern makes for systems that are well organized and easy to extend.

The Problem

All systems must make decisions about what to do in response to a user's request. In PHP, that decision-making process is often handled by a spread of point-of-contact pages. In selecting a page (`feedback.php`), the user clearly signals the functionality and interface she requires. Increasingly, PHP developers are opting for a single-point-of-contact approach (as I will discuss in the next chapter). In either case, however, the receiver of a request must delegate to a tier more concerned with application logic. This delegation is particularly important where the user can make requests to different pages. Without it, duplication inevitably creeps into the project.

So, imagine you have a project with a range of tasks that need performing. In particular, the system must allow some users to log in and others to submit feedback. You could create `login.php` and `feedback.php` pages that handle these tasks, instantiating specialist classes to get the job done. Unfortunately, user interface in a system rarely maps neatly to the tasks that the system is designed to complete. You may require login and feedback capabilities on every page, for example. If pages must handle many different tasks, then perhaps you should think of tasks as things that can be encapsulated. In doing this, you make it easy to add new tasks to your system, and you build a boundary between your system's tiers. This, of course, brings us to the Command pattern.

Implementation

The interface for a command object could not get much simpler. It requires a single method: `execute()`.

In Figure 11–8, I have represented `Command` as an abstract class. At this level of simplicity, it could be defined instead as an interface. I tend to use abstracts for this purpose, because I often find that the base class can also provide useful common functionality for its derived objects.

Figure 11–8. *The Command class*

There are up to three other participants in the Command pattern: the client, which instantiates the command object; the invoker, which deploys the object; and the receiver on which the command operates.

The receiver can be given to the command in its constructor by the client, or it can be acquired from a factory object of some kind. I like the latter approach, keeping the constructor method clear of arguments. All Command objects can then be instantiated in exactly the same way.

Here's a concrete Command class:

```
abstract class Command {
    abstract function execute( CommandContext $context );
}

class LoginCommand extends Command {
    function execute( CommandContext $context ) {
        $manager = Registry::getAccessManager();
        $user = $context->get( 'username' );
        $pass = $context->get( 'pass' );
        $user_obj = $manager->login( $user, $pass );
        if ( is_null( $user_obj ) ) {
            $context->setError( $manager->getError() );
            return false;
        }
        $context->addParam( "user", $user_obj );
        return true;
    }
}
```

The LoginCommand is designed to work with an AccessManager object. AccessManager is an imaginary class whose task is to handle the nuts and bolts of logging users into the system. Notice that the Command::execute() method demands a CommandContext object (known as RequestHelper in *Core J2EE Patterns*). This is a mechanism by which request data can be passed to Command objects, and by which responses can be channeled back to the view layer. Using an object in this way is useful, because I can pass different parameters to commands without breaking the interface. The CommandContext is essentially an object wrapper around an associative array variable, though it is frequently extended to perform additional helpful tasks. Here is a simple CommandContext implementation:

```
class CommandContext {
    private $params = array();
    private $error = "";

    function __construct() {
        $this->params = $_REQUEST;
    }

    function addParam( $key, $val ) {
        $this->params[$key]=$val;
    }

    function get( $key ) {
        return $this->params[$key];
    }

    function setError( $error ) {
        $this->error = $error;
    }

    function getError() {
```

```
        return $this->error;
    }
}
```

So, armed with a `CommandContext` object, the `LoginCommand` can access request data: the submitted username and password. I use `Registry`, a simple class with static methods for generating common objects, to return the `AccessManager` object with which `LoginCommand` needs to work. If `AccessManager` reports an error, the command lodges the error message with the `CommandContext` object for use by the presentation layer, and returns `false`. If all is well, `LoginCommand` simply returns `true`. Note that `Command` objects do not themselves perform much logic. They check input, handle error conditions, and cache data as well as calling on other objects to perform the operations they must report on. If you find that application logic creeps into your command classes, it is often a sign that you should consider refactoring. Such code invites duplication, as it is inevitably copied and pasted between commands. You should at least look at where the functionality belongs. It may be best moved down into your business objects, or possibly into a Facade layer. I am still missing the client, the class that generates command objects, and the invoker, the class that works with the generated command. The easiest way of selecting which command to instantiate in a web project is by using a parameter in the request itself. Here is a simplified client:

```
class CommandNotFoundException extends Exception {}

class CommandFactory {
    private static $dir = 'commands';

    static function getCommand( $action='Default' ) {
        if ( preg_match( '/\W/', $action ) ) {
            throw new Exception("illegal characters in action");
        }
        $class = UCFirst(strtolower($action))."Command";
        $file = self::$dir.DIRECTORY_SEPARATOR."{$class}.php";
        if ( ! file_exists( $file ) ) {
            throw new CommandNotFoundException( "could not find '$file'" );
        }
        require_once( $file );
        if ( ! class_exists( $class ) ) {
            throw new CommandNotFoundException( "no '$class' class located" );
        }
        $cmd = new $class();
        return $cmd;
    }
}
```

The `CommandFactory` class simply looks in a directory called `commands` for a particular class file. The file name is constructed using the `CommandContext` object's `$action` parameter, which in turn should have been passed to the system from the request. If the file is there, and the class exists, then it is returned to the caller. I could add even more error checking here, ensuring that the found class belongs to the `Command` family, and that the constructor is expecting no arguments, but this version will do fine for my purposes. The strength of this approach is that you can drop a new `Command` object into the `commands` directory at any time, and the system will immediately support it.

The invoker is now simplicity itself:

```
class Controller {
    private $context;
    function __construct() {
        $this->context = new CommandContext();
```

```
    }

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

    function process() {
        $cmd = CommandFactory::getCommand( $this->context->get('action') );
        if ( ! $cmd->execute( $this->context ) ) {
            // handle failure
        } else {
            // success
            // dispatch view now..
        }
    }
}

$controller = new Controller();
// fake user request
$context = $controller->getContext();
$context->addParam('action', 'login' );
$context->addParam('username', 'bob' );
$context->addParam('pass', 'tiddles' );
$controller->process();
```

Before I call `Controller::process()`, I fake a web request by setting parameters on the `CommandContext` object instantiated in the controller's constructor. The `process()` method delegates object instantiation to the `CommandFactory` object. It then invokes `execute()` on the returned command. Notice how the controller has no idea about the command's internals. It is this independence from the details of command execution that makes it possible for you to add new `Command` classes with a relatively small impact on this framework.

Here's one more `Command` class:

```
class FeedbackCommand extends Command {

    function execute( CommandContext $context ) {
        $msgSystem = Registry::getMessageSystem();
        $email = $context->get( 'email' );
        $msg   = $context->get( 'msg' );
        $topic = $context->get( 'topic' );
        $result = $msgSystem->send( $email, $msg, $topic );
        if ( ! $result ) {
            $context->setError( $msgSystem->getError() );
            return false;
        }
        return true;
    }
}
```

■ **Note** I will return to the Command pattern in Chapter 12 with a fuller implementation of a Command factory class. The framework for running commands presented here is a simplified version of another pattern that you will encounter: the Front Controller.

As long as this class is contained within a file called `FeedbackCommand.php`, and is saved in the correct `commands` folder, it will be run in response to a "feedback" action string, without the need for any changes in the controller or `CommandFactory` classes.

Figure 11–9 shows the participants of the Command pattern.

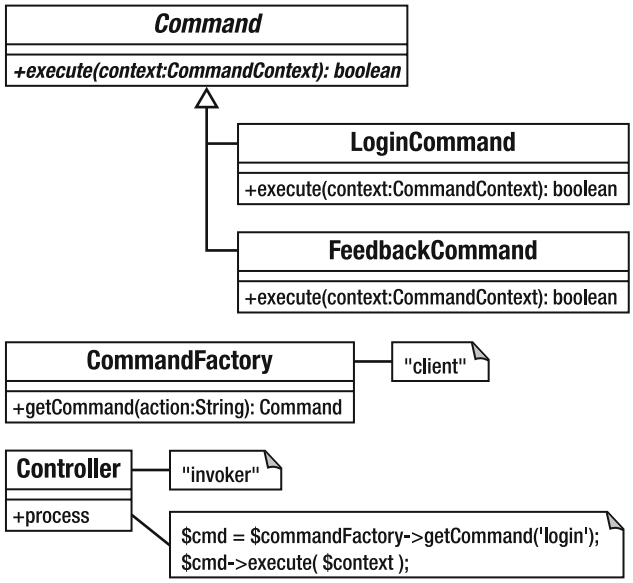

Figure 11–9. *Command pattern participants*

Summary

In this chapter, I wrapped up my examination the Gang of Four patterns. I designed a minilanguage and built its engine with the Interpreter pattern. You encountered in the Strategy pattern another way of using composition to increase flexibility and reduce the need for repetitive subclassing. The Observer pattern solved the problem of notifying disparate and varying components about system events. You revisited the Composite example, and with the Visitor pattern learned how to pay a call on, and apply many operations to, every component in a tree. Finally, you saw how the Command pattern can help you to build an extensible tiered system.

In the next chapter, I will step beyond the Gang of Four to examine some patterns specifically oriented toward enterprise programming.