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 21

brief contents

C H A P T E R 2 1

Data class design

As we've seen, the marriage of a database and an object-oriented program is usually not a match made in heaven. It's more of an arranged marriage, although casual relationships are also possible.

Some ceremony is helpful in getting the liaison started. The patterns presented by Martin Fowler in *Patterns of Enterprise Application Architecture* [P of EAA] are a good rough guide to the alternative ways of creating a systematic and lasting bond between data storage and program code. In this chapter, we will approach these patterns from a slightly different perspective, trying to create some shortcuts along the way.

And, of course, we would like the classes and the database to live happily ever after, but like it or not, we are likely to find ourselves having to work on the relationship. Refactoring is as important here as in other parts of the application.

In this chapter, we'll start with the simplest object-oriented approaches to retrieving data from a database and writing it back. Then we'll see how to build persistence into the objects themselves. We'll also discover the approach of keeping persistence out of the objects to avoid making them dependent on the persistence mechanism. Finally, we'll take a look at these patterns from the point of view of using them rather than implementing them.

21.1 THE SIMPLEST APPROACHES

A database is like a piggy bank: putting something into it and taking something out of it are distinct challenges. They require mostly different procedures and skills. The SqlGenerator object we created previously has methods to generate `INSERT`, `UPDATE`, and `DELETE` statements, but none for `SELECT`s. A `SELECT` statement is much more complex, and other information is needed to generate it. When a database is used for object storage, we need to convert and repackage the data in different ways depending on whether we're reading from or writing to the database.

So why not have one class to find and one to save data? You get more classes, obviously. You may not necessarily want to do that in real life. But it is a clean solution both conceptually and technically. And it is instructive: studying the two separately helps elucidate the differences and similarities between the various approaches and design patterns.

Therefore, we'll start this section by studying how to retrieve data with Finder classes. Then we'll add a way to insert and update data, and discover that this approach has been named the Table Data Gateway pattern.

21.1.1 Retrieving data with Finder classes

To find data in a database, it's tempting to use a query object class. But a query object that handles complex queries and joins can be quite complex. So, for a realistic example that's relatively hard to generalize, let's try one that requires a join. The example is a Finder class for News articles with author data that is stored in a separate User table. We can handle this using a less generalized approach to SQL. Listing 21.1 shows such a class, using the Creole library and prepared statements to get the data.

Listing 21.1 News Finder class using a Creole database connection

```
class NewsFinder {
    public function __construct() {
        $this->connection =
            CreoleConnectionFactory::getConnection();
    }

    public function find($id) {
        $stmt = $this->connection->prepareStatement(
                "SELECT headline,introduction,text, ".
                "concat(Users.firstname,' ',Users.lastname) ".   ❶
                "AS author, ".
                "UNIX_TIMESTAMP(created) AS created,id ".        ❷
                "FROM Documents, Users ".
                "WHERE Documents.author_id = Users.user_id ".
                "AND id = ?");
        $stmt->setInt(1,$id);                                    ❸
        $rs = $stmt->executeQuery();                             ❹
        $rs->first();
        return $rs->getRow();
    }
```

❶ Concatenate the names in SQL

❷ Easy but MySQL-specific

❸ Using placeholders

❹ Return first row as array

```
    public function findWithHeadline($headline) {
        $stmt = $this->connection->prepareStatement(
                "SELECT headline,introduction,text, ".
                "concat(Users.firstname,' ',Users.lastname) ".
                "AS author, ".
                "UNIX_TIMESTAMP(created) AS created,id ".
                "FROM Documents, Users ".
                "WHERE Documents.author_id = Users.user_id ".
                "AND headline = ?");
        $stmt->setString(1,$headline);
        $rs = $stmt->executeQuery();
        $rs->first();
        return $rs->getRow();
    }

    public function findAll() {
        return $this->connection->executeQuery(      ❺ Return the result set iterator
                "SELECT headline,introduction,text, ".
                "concat(Users.firstname,' ',Users.lastname) ".
                "AS author, ".
                "UNIX_TIMESTAMP(created) AS created,id ".
                "FROM Documents, Users ".
                "WHERE Documents.author_id = Users.user_id ".
                "ORDER BY created DESC");
    }
```

❶ We're using SQL to generate the full name from the first and last names. If we wanted to make an object out of the row, it might be better to do this job inside the object instead of in SQL. That would mean storing the names in separate variables in the object and having a method that would generate the full name.

❷ Assuming that `created` is a MySQL `datetime` or `timestamp` column, using `UNIX_TIMESTAMP()` is an easy MySQL-specific way of getting the date and time in a format that is convenient to use in PHP.

❸ In this situation, using placeholders is almost equivalent to interpolating (or concatenating) the ID directly in the SQL statement. The most important difference is that Creole will escape the value if it's a string.

❹ In this and the following method, we use the Creole result set to return an array representing the first row.

❺ In the `findAll()` method, we return the Creole result set iterator. This is interesting, since it will work as an SPL iterator, allowing us to do this:

```
$rs = $this->finder->findAll();
foreach ($rs as $row) {
    print $row['headline']."\n";
}
```

In other words, the iterator acts as if it's an array of associative arrays. On the other hand, it will also work like a JDBC-style iterator:

```
while($rs->next()) {
    $row = $rs->getRow();
    print $row['headline']."\n";
}
```

This is really only useful to those who are used to JDBC. What *is* useful is the ability to get specific data types. For example, if we want to get rid of the MySQL-specific `UNIX_TIMESTAMP()` function, we can remove it from the SQL statement and use the result set's `getTimestamp()` method as follows:

```
foreach ($rs as $row) {
    print $row['headline']."\n"; // or: $rs->getString('headline');
    print $rs->getTimestamp('created','U')."\n";
}
```

This is an odd hybrid approach, using the result set and the `$row` array interchangeably, but it works.

In listing 21.1, there is a lot of duplicated SQL code. As mentioned, that can be a good idea, since it leaves SQL statements relatively intact, and that may make them more readable. The duplication creates a risk of making inconsistent changes. But when the statements are concentrated in one class as they are here, it's possible to avoid that.

On the other hand, the amount of duplication is so large in this case, and the differences between the statements so small, that it's tempting to eliminate it. We can do that by having a separate method to prepare the statement, as in listing 21.2.

Listing 21.2 News Finder class with SQL duplication eliminated

```
class NewsFinder {
    public function find($id) {
        $stmt = $this->prepare("AND id = ?");
        $stmt->setInt(1,$id);
        return $stmt->executeQuery();
    }

    public function findWithHeadline($headline) {
        $stmt = $this->prepare("AND headline = ?");
        $stmt->setString(1,$headline);
        return $stmt->executeQuery();
    }

    public function findAll() {
        $stmt = $this->prepare ("ORDER BY created DESC");
        return $stmt->executeQuery();
        return $result;
    }

    private function prepare($criteria) {
        return $this->connection->prepareStatement(
                sprintf("SELECT headline,introduction,text, ".
                "concat(Users.firstname,' ',Users.lastname) ".
                "AS author, ".
                "UNIX_TIMESTAMP(created) AS created,id ".
                "FROM Documents, Users ".
                "WHERE Documents.author_id = Users.user_id %s",
                $criteria));
    }
}
```

To my eyes, that makes the code more, rather than less, readable, although the `prepare()` method requires a rather odd-looking `SQL` fragment representing an addition to the existing `WHERE` clause.

Figure 21.1 is a simple UML representation of the class. We will use this as a building block for the Table Data Gateway in the next section.

There is more duplication here, though. To get the results in the form of arrays of associative arrays, we're repeating the code to get the data from the Creole result set. This is not much of an issue for this one class. But if we want to write more Finder classes, we need to do something about it.

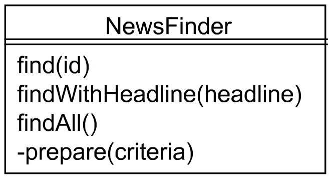

Figure 21.1 NewsFinder class

21.1.2 Mostly procedural: Table Data Gateway

Fowler's Table Data Gateway pattern is a class that—in its simplest form—handles access to a single database table. The principle is to have a class that finds and saves plain data in the database. The data that's retrieved and saved is typically not in the form of objects, or if it is, the objects tend to be rather simple data containers.

The Finder we just built is half a Table Data Gateway. In other words, the methods in a Table Data Gateway that retrieve data from a database are like the Finder methods in the previous examples. The methods that are missing are methods to insert, update, and delete data.

Building a Table Data Gateway

To insert and update data we use `insert()` and `update()` methods and specify all the data for the row as arguments to the method:

```
$gateway->insert($headline,$introduction,$text,$author);
```

A Table Data Gateway is not very object-oriented. It's more like a collection of procedural code in a class. Although with a Table Data Gateway we do create an instance of the class, the resulting object is primarily an engine for executing procedural code.

Instead of building a complete Table Data Gateway, let's do the part that we're missing: update, insert, and delete. We can have a separate class called NewsSaver to do this, as in listing 21.3. In the listing, we're using Creole with prepared statements and a connection factory as discussed in the previous chapter.

Listing 21.3 A NewsSaver class to complement the NewsFinder

```
class NewsSaver {

    private $connection;
    public function __construct() {
        $this->connection
            = CreoleConnectionFactory::getConnection();
    }

    public function delete($id) {
        $sql = "DELETE FROM News where id =".$id;
        $this->connection->executeQuery($sql);
    }

    public function insert($headline,$intro,$text,$author_id) {
        $sql = "INSERT INTO News ".
        "(headline,author_id,introduction,text,created) ".
        "VALUES (?,?,?,?,?)";
        $stmt = $this->connection->prepareStatement($sql);
        $stmt->setString(1,$headline);
        $stmt->setInt(2,$author_id);
        $stmt->setString(3,$intro);
        $stmt->setString(4,$text);
        $stmt->setTimestamp(5,time());
        $stmt->executeUpdate();
        $rs = $this->connection->executeQuery(
                "SELECT LAST_INSERT_ID() AS id");
        $rs->first();
        return $rs->getInt('id');
    }

    public function update($id,$headline,$intro,$text,$author_id)
    {
        $sql = "UPDATE News SET ".
            "headline = ?, ".
            "author_id = ?, ".
            "introduction = ?, ".
            "text = ? ".
            "WHERE id = ?";
        $stmt = $this->connection->prepareStatement($sql);
        $stmt->setString(1,$headline);
        $stmt->setInt(2,$author_id);
        $stmt->setString(3,$intro);
        $stmt->setString(4,$text);
        $stmt->setInt(5,$id);
        $stmt->executeUpdate();
    }
}
```

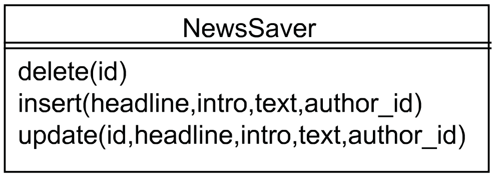
NewsSaver
delete(id)
insert(headline,intro,text,author_id)
update(id,headline,intro,text,author_id)

Figure 21.2
NewsSaver class

The approach to generating and executing SQL in this example is so similar to earlier examples that the details should be self-explanatory. The argument lists are shown in bold, since they are the distinguishing feature of this particular approach. The data to be stored is introduced as single values rather than arrays or objects. Summarizing the class in UML, we get figure 21.2.

If you want a complete Table Data Gateway, you can simply merge the NewsFinder and the NewsSaver classes, as shown in figure 21.3.

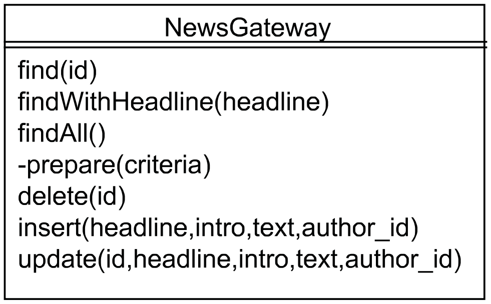
NewsGateway
find(id)
findWithHeadline(headline)
findAll()
-prepare(criteria)
delete(id)
insert(headline,intro,text,author_id)
update(id,headline,intro,text,author_id)

Figure 21.3
Merging the NewsFinder and NewsSaver classes into a Table Data Gate-

Alternatively, you can keep the NewsFinder and NewsSaver classes and use delegation to make NewsGateway a simple facade for those two classes. Listing 21.4 shows just two methods as an example.

Listing 21.4 A partial implementation of a NewsGateway as a Facade for the NewsSaver and NewsFinder classes

```
class NewsGateway {
    private $finder;
    private $saver;
    public function __construct() {
        $this->finder = new NewsFinder;
        $this->saver = new NewsSaver;
    }

    public function find($id) {
        return $this->finder->find($id);
    }

    public function delete($id) {
        $this->saver->delete($id);
    }
}
```

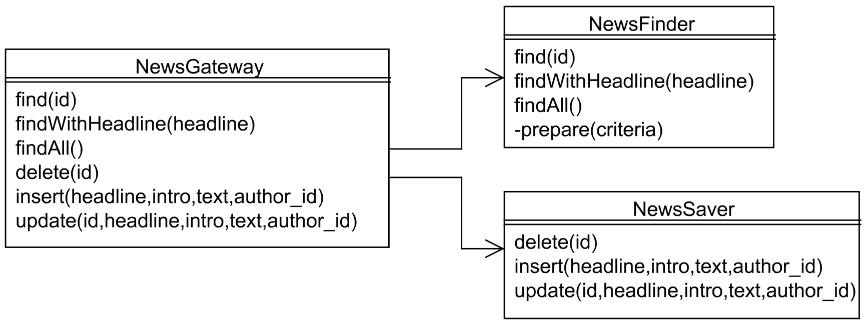
NewsGateway
find(id)
findWithHeadline(headline)
findAll()
delete(id)
insert(headline,intro,text,author_id)
update(id,headline,intro,text,author_id)
NewsFinder
find(id)
findWithHeadline(headline)
findAll()
-prepare(criteria)
NewsSaver
delete(id)
insert(headline,intro,text,author_id)
update(id,headline,intro,text,author_id)

Figure 21.4 Instead of merging the NewsFinder and NewsSaver classes, we can combine them by having the NewsGateway delegate work to the other two

Figure 21.4 is a UML representation of the complete structure. In UML, each class is identical to the ones shown before, except that the NewsGateway class no longer has the private `prepare()` method. In actual implementation, the NewsGateway class is completely different from the merged NewsGateway class, since the methods delegate all the work to the NewsFinder or NewsSaver classes instead of doing it themselves.

This alternative may be considered a cleaner separation of responsibilities; on the other hand, having three classes instead of one may seem unnecessarily complex if you find the merged version easy to understand. The main point is to show how easily you can juggle and combine these classes.

Finding and formatting data using Table Data Gateway

According to Fowler, what to return from Finder methods is tricky in a Table Data Gateway. What should the table row look like when you use it in the application? Should it be an array or an object? What kind of array or object? And what happens when you need to return a number of rows from the database?

The path of least resistance in PHP is to let a row be represented by an associative array and to return an array of these. Listing 21.5 is a dump of two rows in this kind of data structure.

Using an SPL-compatible iterator is a variation on this. From the client code's point of view, it looks very similar.

Listing 21.5 Return data from a Table Data Gateway as arrays

```
Array
(
    [0] => Array
        (
            [user_id] => 1
            [username] => victor
            [firstname] => Victor
            [lastname] => Ploctor
            [password] => 5d009bfda017b80dd1ce08c7e68458be
            [email] => victor@example.com
            [usertype] => regular
        )

    [1] => Array
        (
            [user_id] => 2
            [username] => elietta
            [firstname] => Elietta
            [lastname] => Floon
            [password] => 7db7c42a13bd7e3202fbbc94435fb85a
            [email] => elietta@example.com
            [usertype] => regular
        )
```

Another possibility is to return each row as a simplified object. This would typically not be a full domain object with all capabilities, but rather an object that acts primarily as a data holder. This kind of object is what Fowler calls a Data Transfer Object (sometimes abbreviated as DTO), since it's an object that's easy to serialize and send across a network connection.

PEAR DB has a feature that makes this easy, since you can return objects from the `fetchRow()` method. Listing 21.6 shows a dump of this kind of return data.

Listing 21.6 Return data from a Table Data Gateway as objects

```
Array
(
    [0] => User Object
        (
            [user_id] => 1
            [username] => victor
            [firstname] => Victor
            [lastname] => Ploctor
            [password] => c7e72687ddc69340814e3c1bdbf3e2bc
            [email] => victor@example.com
            [usertype] => regular
        )

    [1] => User Object
        (
            [user_id] => 2
            [username] => elietta
            [firstname] => Elietta
            [lastname] => Floon
            [password] => 053fbca71905178df74c507637966e02
            [email] => elietta@example.com
            [usertype] => regular
        )
```

The example in the PEAR DB manual shows how to get PEAR DB to return DB_row objects, but it's also possible to return objects belonging to other classes if they inherit from DB_row. That means you can add other methods to the objects if you want:

```
class User extends DB_Row {
    function getName() {
        return $this->firstname." ".$this->lastname;
    }
}
```

To make PEAR DB return the objects, do the following:

```
$db = DB::Connect("mysql://$dbuser:$dbpassword@$dbhost/$dbname");
if (DB::isError($db)) die ($db->getMessage());
$db->setFetchMode(DB_FETCHMODE_OBJECT, 'User');
$res =$db->query('SELECT * FROM Users');
while ($row = $res->fetchRow()) {
    $rows[] = $row;
}
print_r($rows);
```

But is there a good reason to prefer objects over arrays to represent database rows? In many programming languages, using objects would help you by making sure there was a clear error message if you tried to use a nonexistent member variable. But in PHP, the basic error checking for arrays and objects is similar. If you use

```
error_reporting(E_ALL)
```

you will get a `Notice` if you try to use an undefined index of an array or an undefined member variable of an object. In either case, there is no danger that mistyping a name will cause a bug, unless you start modifying the array or object after it's been retrieved from the database. If you do want to modify it, you'll be safer by making it an object and modifying it only through method calls. For example, if you want to remember the fact that a user has read news article X during the current session, you might want to add that information to the news article object by having a method in the news article object to register that fact. And if you want to display the age of a news article, you could have a method that calculates and returns the age.

The real advantage of using objects lies in being able to do this kind of processing. If you need to get the date and time in a certain format, you can add the capability to the object without changing the way the data is stored in or retrieved from the database. Another advantage is that we can let the objects themselves handle persistence.

21.2 LETTING OBJECTS PERSIST THEMSELVES

Bringing up children becomes less work when they start to be able to go to bed in the evening without the help of adults: when they can brush their teeth, put on their pajamas, and go to sleep on their own. This is like the idea of letting objects store themselves in a database. Plain, non-object-oriented data is like a baby that has to be carried around, dressed, and put to bed. An object that has the ability to do things on its own is something quite different.

The Table Data Gateway pattern works with “babies”: plain data that has few capabilities of its own. We use specialized objects to do the job of storing and getting data from the database, but the data itself does not have to be—and has mostly not been—in the form of objects.

If we do represent data as objects, we gain the ability to add behaviors to these objects. That means we can make them responsible. It’s a relatively simple and intuitive way to implement object persistence: letting the objects store themselves in the database. The application code creates an object, and then calls an `insert()` method (or alternatively, a `save()` method) on the object to keep it in the database: something like this

```
$topic = new Topic('Trains');
$topic->insert();
```

This approach is used in several of Fowler’s and Nock’s data storage patterns. Fowler has Row Data Gateway and Active Record; Nock has Active Domain Object. They are all similar in principle, but there are some interesting differences, especially in retrieving objects from the database.

This section has the same structure as the previous one: we’ll first look at finding data and then see how objects can insert themselves to the database.

21.2.1 Finders for self-persistent objects

You can retrieve objects from the database in an explicit or implicit way. Implicit retrieval gets the data from the object from the database behind the scenes when the object is constructed. So all you need to do is specify the object’s ID, and the object is retrieved from the database without any need to tell it to do that.

```
$newsArticle = new NewsArticle(31);
$newsArticle->setHeadline('Woman bites dog');
$newsArticle->save();
```

This is the approach taken by Nock. From the application’s point of view, it is the prettiest and most consistent way of implementing self-persistent objects. The code does not mention anything but the persistent object itself and the class it belongs to.

Explicit data retrieval is more like what we’ve done earlier. So to get an object with a specific ID, we can use a finder class as before:

```
$finder = new NewsFinder;
$newsArticle = $finder->find(31);
$newsArticle->setHeadline('Woman bites dog');
$newsArticle->save();
```

This may look less appealing. On the other hand, since it’s more explicit, it could be easier to understand. We want our code to be as readable as possible, and concealing what is actually going on (a database read) may cause misunderstandings. Method names should generally describe the intention of the method. What about constructors, then? Constructors usually just instantiate and initialize the object; reading data from a database is more than we normally expect. Perhaps it's better to have a method such as `find()` that tells us more precisely what is going on.

There is another problem with the implicit approach. The first time we create an object, it's not in the database yet. So obviously, we need to construct the object without reading it from the database. In other words, we need a constructor that just does basic initialization. In Java, this is solved by having different constructors that do different things depending on the list of arguments. Yet I find it confusing that one constructor reads from the database and one doesn't. I have to remember that the one that reads from the database is the one that accepts only the ID as an argument.

In PHP, this problem is even worse, since you can have only one constructor. So we need conditional logic based on the number and/or types of arguments, and that's not pretty. But there is a way around PHP's inability to have more than one constructor: using creation methods. If one of those methods finds an object by ID in the database, we might as well call it `find()`. So we've just taken a detour, and now we're back to the explicit approach.

Yet another factor is the fact that we will likely need to get objects from the database by other criteria than just the ID. In other words, we need other finder methods. If one of the finder methods is implemented as a constructor and the others aren't, that's yet another inconsistency. Nock solves this problem by having a separate collection object: there is a Customer object that's found by using the constructor, and a CustomerList object whose constructor gets all the customers from the database. In addition, you can have other constructors that represent customer lists returned by other queries.

Let's try Fowler's strategy of doing all reading from the database explicitly by finder methods.

How do we implement it? We've already developed finder classes that do everything except actually instantiate the objects. All we need to do is to add that last step.

The easiest way to do this is to add a `load()` method to the finder to do the job of creating the object from the array (or object) representing the row:

```
class NewsFinder {
    public function load($row) {
        if (!$row) return FALSE;
        extract($row);
        $object = new NewsArticle(
            $headline,
            $introduction,
            $text,
            $author_id,
            new DateAndTime($created)
        );
        $object->setID($id);
        return $object;
    }
    public function fetchObjects($stmt) {
        $rs = $stmt->executeQuery();
        $result = array();
        while($rs->next()) {
            $result[] = $this->load($rs->getRow());
        }
        return $result;
    }
}
```

The `load()` function is mostly code that's specific to the NewsFinder. The `fetchObjects()` method, on the other hand, could be used in Finders for other objects as well.

To deal with this, the path of least resistance is to extract `fetchObjects()` into a parent class. The alternative is to decorate the prepared statement class and keep the code there. This may be somewhat more logical, since returning results in a generic way is already part of the statement's responsibility.

But the kind of object to be returned depends on which specific finder we are using. How do we get the right class of object? There is an elegant solution: we can pass the fetch method(s) an object that creates the output objects we want. This object will be a kind of factory object: it takes a database row (or possibly a result object returned from the database abstraction layer) and generates the output object from that. We can call it a loader to distinguish it from other kinds of factories.

Since we want to pass the loader to the statement object, let's have an interface (or an abstract class). That allows us to use type hints.

```
interface Loader {
    public function load($row);
}
```

The loader itself does nothing but instantiate the object, set its data, and return it:

```
class NewsLoader implements Loader {
    public function load($row) {
        if (!$row) return FALSE;
        extract($row);
        $object = new NewsArticle(
            $headline,
            $introduction,
            $text,
            $author_id,
            new DateAndTime($created)
        );
        $object->setID($id);
        return $object;
    }
}
```

We've used the opportunity to convert the timestamp into an object as well. So the NewsArticle object will contain a DateAndTime object rather than a plain timestamp.

But this means the SQL statement must return a UNIX timestamp. As mentioned earlier, this means that we either have to store the timestamp as an integer in the database or use RDBMS-specific functions to convert it. Instead of relying on the database to return the timestamp, we can use a date conversion class:

```
$dbComponentFactory = new MysqlComponentFactory;
extract($row)
$converter = $dbComponentFactory->getDateConverter();
$dateandtime = new DateAndTime($converter->toUnixTimestamp($created));
```

Without going into it deeply, let us just note that this is another example of the Abstract Factory pattern [Gang of Four]. Once we've created the component factory that's appropriate for the current RDBMS, we might use it to create objects for doing other RDBMS-specific tasks such as limiting queries or ID generation.

Now let us get back to our original goal—letting a decorator for the prepared statement class generate domain objects while remaining independent of the specific domain object class. The structure in figure 21.5 is what we need. It may seem complex, but the important roles are played by the NewsLoader and CreoleStatement classes. We've isolated the knowledge about the particular domain object, the NewsArticle, putting it in the NewsLoader class. To reuse this with another kind of domain object, the NewsLoader class is the only one we have to replace.

Let's look at the implementation of the `fetchFirst()` and `fetchAll()` methods that have the ability to generate objects without knowing their type. Listing 21.7 shows how the CreoleStatement class accepts a Loader object and uses it to generate domain objects.

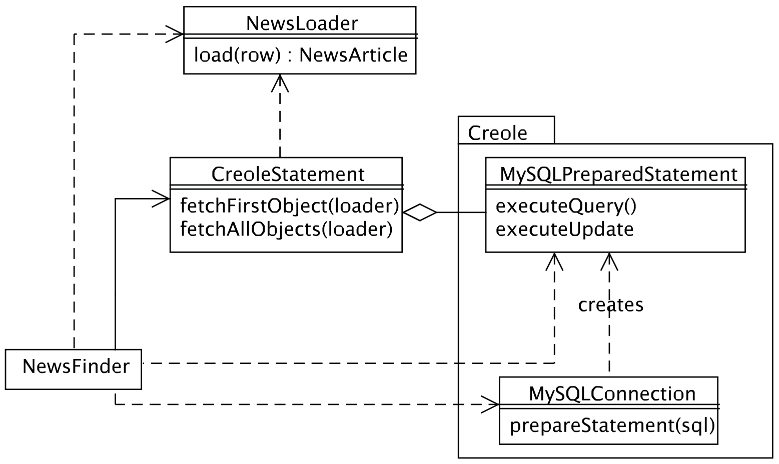

Figure 21.5 Using a NewsLoader object and a statement decorator to generate objects from retrieved data

Listing 21.7 Putting general result handling code into a CreoleStatement decorator

```
class CreoleStatement {
    private $statement;
    public function __construct($statement) {
        $this->statement = $statement;
    }

    public function executeQuery() {
        return $this->statement->executeQuery();
    }

    public function executeUpdate() {
        return $this->statement->executeUpdate();
    }

    public function fetchFirstObject(Loader $loader) {
        $rs = $this->executeQuery();
        $rs->first();
        return $loader->load($rs->getRow());
    }

    public function fetchAllObjects(Loader $loader) {
        $rs = $this->executeQuery();
        $result = array();
        while($rs->next()) {
            $result[] = $loader->load($rs->getRow());
        }
        return $result;
    }
}
```

❶ Decorate the original statement object

❷ Two sample methods

❸ Use the loader to create objects

❶ We create the decorator as usual by passing the object to be decorated into the constructor.

❷ We will need methods from the decorated object. Only two are shown here, but many more may be relevant.

❸ Methods to get objects use the loader to create an object from an array representation of a database row.

Now all we need to do is to use these methods in the NewsFinder, passing a newly created NewsLoader object to the statement object.

```
class NewsFinder {

    public function find($id) {
        $stmt = $this->prepare("AND id = ?");
        $stmt->setInt(1,$id);
        return $stmt->fetchFirstObject(new NewsLoader);
    }

    public function findAll() {
        $stmt = $this->prepare("ORDER BY created DESC");
        return $stmt->fetchAllObjects(new NewsLoader);
    }
}
```

Passing an object this way is standard object-oriented procedure for passing some code for execution, but it *could* be achieved by passing anonymous functions or just a function or method name instead.

21.2.2 Letting objects store themselves

Now we can write the `insert()`, `update()`, and `delete()` methods. These methods are suspiciously similar to their equivalents in the Table Data Gateway—that is, our NewsSaver as shown in listing 21.4. The main difference is that the values to be saved are taken from the object itself. Listing 21.8 shows how this works. This is a Row Data Gateway pattern in Fowler's terminology. If we add domain logic to it, it will be an Active Record. It's tempting to drop the distinction and call this an Active Record.

Listing 21.8 A NewsArticle class for objects that can insert, update, and delete themselves

```
class NewsArticle implements DomainObject {
    public function insert() {
        $sql = "INSERT INTO News ".
        "(headline,author_id,introduction,text,created) ".
        "VALUES (?,?,?,?,?)";
        $stmt = $this->connection->prepareStatement($sql);
        $stmt->setString(1,$this->getHeadline());
        $stmt->setInt(2,$this->getAuthorID());
        $stmt->setString(3,$this->getIntroduction());
        $stmt->setString(4,$this->getText());
        $stmt->setTimestamp(5,time());
        $stmt->executeUpdate();
        $rs = $this->connection->executeQuery(
                "SELECT LAST_INSERT_ID() AS id");
        $rs->first();
        return $rs->getInt('id');
    }

    public function update() {
        $sql = "UPDATE News SET ".
            "headline = ?, ".
            "author_id = ?, ".
            "introduction = ?, ".
            "text = ? ".
            "WHERE id = ?";
        $stmt = $this->connection->prepareStatement($sql);
        $stmt->setString(1,$this->getHeadline());
        $stmt->setInt(2,$this->getAuthorID());
        $stmt->setString(3,$this->getIntroduction());
        $stmt->setString(4,$this->getText());
        $stmt->setInt(5,$this->getID());
        $stmt->executeUpdate();
    }
    public function delete() {
        $sql = "DELETE FROM News where id =".$this->getID();
        $this->connection->executeQuery($sql);
    }
}
```

Figure 21.6 is a simplified UML representation that illustrates the principle. The NewsArticle class has the attributes of a news article (id, author, headline, text) and methods to delete, insert, and update itself. Compared to the NewsSaver in figure 21.2, the main difference is that the class has all the data that needs to be stored in the database, so there is no need for the method arguments.

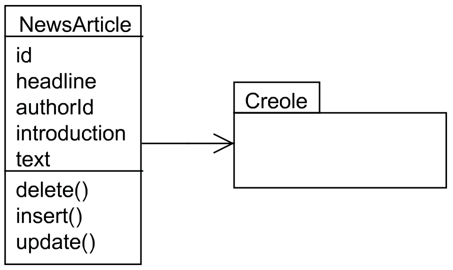

Figure 21.6 NewsArticle class as Row Data Gateway/Active Object

There are several ways to reduce the amount of duplication between the class in listing 21.8 and other persistent objects. One is to use the SqlGenerator from the previous chapter. Another approach, which is the one Fowler takes, is to extract as much generic code as possible into a parent class. The code to generate the ID is the most obvious candidate to be extracted in this way.

We've seen how to delegate the persistence work to the object we're persisting. As ideal as this may seem, we may get rid of some difficult dependencies by doing the exact opposite. This is the point of Fowler's Data Mapper pattern.

21.3 THE DATA MAPPER PATTERN

Having the object itself take care of persistence is convenient, but it's no guarantee of eternal bliss, since it makes the domain objects dependent on the persistence mechanism. The problem shows up if you want to transplant your objects to another application. Suddenly the close tie to the persistence mechanism and the database becomes a liability. There may be no easy way to use them without bringing the entire database along. That's why it may be a good idea to do something completely different: leave persistence to classes that only handle the objects temporarily instead of having a permanent relationship with them.

In this section, we'll first do a slight variation of an earlier example, making it compatible with the idea of a Data Mapper. Then we'll look at the similarities and differences between the data access patterns we've seen so far.

21.3.1 Data Mappers and DAOs

In Fowler's terminology, a Data Mapper is an object that gets objects from and stores objects to a database. The pattern differs from Active Record in using a completely separate mapper object to do the job, rather than the domain object itself.

The J2EE pattern Data Access Object [Alur et al.] is similar in principle, although the description of the pattern is concerned with retrieving and storing so-called Transfer Objects rather than real domain objects. But DAO is frequently used as a more general term encompassing more data access strategies than Data Mapper.

NOTE In the book, Transfer Objects are called Value Objects, but this is inconsistent with the usage by several gurus. In the online description of the pattern, they are called Transfer Objects.

Again, finding and saving are rather different. The part of a Data Mapper that gets objects from a database is identical to a Finder for an Active Record as described earlier in this chapter. Saving the objects is slightly different. Figure 21.7 shows the principle. This is similar to figure 21.2; the difference is that the NewsArticle object has replaced the list of single data values.

Listing 21.9 shows the implementation.

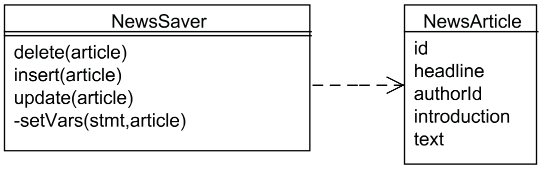

Figure 21.7 NewsSaver class, Data Mapper style

Listing 21.9 A NewsSaver class that is half of a Data Mapper

```
class NewsSaver {
    private $connection;
    public function __construct() {
        $this->connection
            = CreoleConnectionFactory::getConnection();
    }

    public function insert($article) {
        $sql = "INSERT INTO News ".
        "(headline,author_id,introduction,text,created) ".
        "VALUES (?,?,?,?,?)";
        $stmt = $this->connection->prepareStatement($sql);
        $this->setVars($stmt,$article);
        $stmt->setTimestamp(5,time());
        $stmt->executeUpdate();
        $rs = $this->connection->executeQuery(
                "SELECT LAST_INSERT_ID() AS id");
        $rs->first();
        return $rs->getInt('id');
    }
    public function update($article) {
        $sql = "UPDATE News SET ".
            "headline = ?, ".
            "author_id = ?, ".
            "introduction = ?, ".
            "text = ? ".
            "WHERE id = ?";
        $stmt = $this->connection->prepareStatement($sql);
        $this->setVars($stmt,$article);
        $stmt->setInt(5,$article->getId());
        $stmt->executeUpdate();
    }

    private function setVars($stmt,$article) {
        $stmt->setString(1,$article->getHeadline());
        $stmt->setInt(2,$article->getAuthorId());
        $stmt->setString(3,$article->getIntroduction());
        $stmt->setString(4,$article->getText());
    }

    public function delete($article) {
        $sql = "DELETE FROM News where id = ".$article->getId();
        $this->connection->executeQuery($sql);
    }
}
```

The `setVars()` method contains the code that's needed by both the `update()` and the `insert()` methods. In fact, looking back at the previous listing (21.8), we can see that a similar method might be extracted there.

A full Data Mapper can be achieved simply be merging this class with the Finder for the active record.

21.3.2 These patterns are all the same

Fowler's data source patterns look different in UML, but are very similar in implementation.

An Active Record is a Row Data Gateway with domain logic

Although many details and variations differ in Fowler's descriptions of these two patterns, the bottom line is that a Row Data Gateway is transformed into an Active Record if you move domain logic into it.

Fowler says, "If you use Transaction Script with Row Data Gateway, you may notice that you have business logic that's repeated across multiple scripts...Moving that logic will gradually turn your Row Data Gateway into an Active Record."

An Active Record is an object with a built-in Data Mapper

The Active Record pattern just means introducing data access code into a domain object. From the client's perspective, this is equivalent to having a Data Mapper inside the object and using it to do `INSERT`, `UPDATE`, and `DELETE` operations, as in the following fragment of a class:

```
class NewsArticle {
    public function __construct() {
        $this->mapper = new NewsMapper;
    }

    public function insert() {
        $this->mapper->insert($this);
    }
}
```

In fact, we can use the NewsSaver in Data Mapper style to achieve this. Starting out with the previous figure (21.7), adding the methods to the NewsArticle object and changing the places of the two objects, we have figure 21.8 which shows this design.

A Data Mapper is a Table Data Gateway that deconstructs an object

The Table Data Gateway and Data Mapper patterns are also very similar. The Table Data Gateway accepts the data as single values; the Data Mapper accepts it in the form of objects. In a simple Table Data Gateway, the `insert()` method may have this signature:

```
public function insert($subject,$text) { }
```

Then we use `$subject` and `$text` as input to the SQL `INSERT` statement. In the corresponding Data Mapper, it will be like this instead:

```
public function insert($message) { }
```

Now, instead of `$subject` and `$text`, we can use `$message->getSubject()` and `$message->getText()`. There's no obvious reason why there should be more of a difference.

Fowler refers to Data Mappers as complex, but the fact is that the complexity is in the mapping process rather than the Data Mapper pattern itself. Mappings that involve multiple class and table relationships are complex to program.

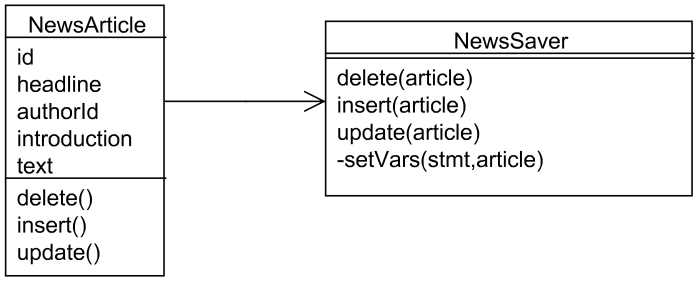

Figure 21.8 Active Record that delegates data storage to a separate Data Mapper (NewsSaver) class

21.3.3 Pattern summary

As mentioned, the principle of self-persistent objects is the theme of Nock's Active Domain Object pattern and of Fowler's Row Data Gateway and Active Record patterns.

The NewsLoader is an instance of Nock's Domain Object Factory pattern.

Reconstituting a DateAndTime object from the database is an example of Fowler's Embedded Value pattern. Entity objects usually need a separate table for storage; value objects such as dates can be represented by one or more columns in a table that stores objects belonging to another class, in this case news articles.

So far we've looked mostly at how we can implement data classes. Now it's time to compare how they work in actual use.

21.4 FACING THE REAL WORLD

We've discussed many of the pros and cons of different patterns, approaches, and techniques. What we haven't considered yet is the way they work in an actual web application. One good reason to do that is the fact that PHP objects don't survive from one HTTP request to the next unless you specifically store them in a session. Using an object-oriented approach can sometimes mean creating objects only to save the data inside them, and that may seem cumbersome.

Starting from that, we'll see how the patterns compare when we try to use them in a typical web application. Then we'll take a look at another challenge that the real world sometimes throws at us: optimizing queries.

21.4.1 How the patterns work in a typical web application

The different data storage patterns have different APIs that affect the programming of the web application. For a simple web application, the interface needs to be simple and comfortable.

Finders are easy to use. Whether they return arrays or objects, there's no problem putting the data out on the web page. Most template engines have ways of getting data from objects by running getter methods.

For saving data, it's a bit more complex. The Table Data Gateway API is practical in web applications. When you want to insert, update, or delete something, you typically have a collection of data from a form or (when you're deleting) an ID from a URL. In either case, you're starting out with request variables, and you can use these directly as method arguments. When the user has entered a new article, you might be doing this:

```
// Table data gateway insert
$gateway = new NewsGateway;
$gateway->insert(
        $request->get('headline'),$request->get('intro'),
        $request->get('text'),$request->get('author_id'));
```

Compared to this, a more object-oriented approach might seem cumbersome. You have to create an object before you can store the data. This is not so bad when you insert a new object:

```
// Row Data Gateway insert
$article = new NewsArticle(
        $request->get('headline'),$request->get('intro'),
        $request->get('text'),$request->get('author_id'));
$article->insert();
```

But when you need to update an existing article, you have to get the object from the database first and set all the data:

```
//Row Data Gateway update
$article = NewsFinder::find($request->get('id'));
$article->setHeadline($request->get('headline'));
$article->setIntro($request->get('intro'));
$article->setText($request->get('text'));
$article->setAuthorId($request->get('author_id'));
$article->update();
```

The Table Data Gateway approach is somewhat simpler:

```
// Table Data Gateway update
$gateway = new NewsGateway;
$gateway->update($request->get('id'),$request->get('headline'),
         $request->get('intro'),$request->get('text'),
         $request->get('author_id'));
```

Also, there's one less database query, since we're just doing the `UPDATE` without doing a `SELECT` to get the row first.

But the more object-oriented approach *is* more flexible. As long as we're only doing one or two things with an object, its virtues are less apparent. But what if we want to make different kinds of changes in different circumstances? For example, what if we had a separate form just for changing the headline? Or, perhaps more likely, what if we have one form for users to change their password, and another to change their preferences? If the information is represented as an object, we just change the data in the object that we need to change and update the object. With the Table Data Gateway approach, we need an extra method in the Gateway to handle a similar situation. We need one method to update the user's preferences and another method to update the user's password. If we don't, if we make one method to update all of them at once, the problem is how to find the data that the user hasn't specified in the form. We will have to either read it from the database or keep it in session variables. Both of these alternatives are neater if we use an object to represent the data.

We've covered most of the basic design patterns for data access and some more advanced ones. To go beyond those, you may start on the road leading to a full Object-Relational Mapping tool (ORM), using such patterns as Metadata Mapping and Query Object (mentioned in the previous chapter). Even if you don't, you may run into another source of complexity: the need to optimize queries.

21.4.2 Optimizing queries

Object-oriented approaches to database access emphasize programmer efficiency over program efficiency. For example, you may not need all the columns every time you use a `find()` method to retrieve an object. It's more efficient to have a query that asks only for the columns you need. What should you do?

Similarly, here's a question that was asked in a discussion forum: what if you need to delete 100 objects in one go? Deleting one at a time is obviously inefficient.

The answer to these questions is simple, and the principle is the same: if you need an extra query for performance, make an extra method in the data access class to run that query. Just make sure you really need it. If you need to delete 100 objects often, and it really is slow to do it one by one, this could be worth it. But don't optimize just to optimize. Don't optimize a query that obviously doesn't need it, such as one that gets only a single row from the database. (There might be exceptions to that. For example, there are column types in MySQL that are capable of storing 4GB of data. But the average database row is relatively small.)

21.5 SUMMARY

The simplest approaches to storing objects in a database sidestep the problems of mismatch between objects and relational data by dealing with data in the form of plain, non-object-oriented variables and arrays. This reduces the need to package the data. Fowler's Table Data Gateway pattern uses an object-oriented, or at least encapsulated, mechanism to store and retrieve the data, but keeps the data representations simple and similar to those in the database.

A slightly more advanced alternative—of which Active Record is the most popular example—is to let objects store themselves in the database by adding methods to the object classes to handle insert, update, and delete operations. The methods take the data from the object, insert it into SQL queries, and run the queries.

It's also possible to handle object persistence with objects that are specialized for the purpose, as in Fowler's Data Mapper pattern. This goes a long way toward making object persistence transparent to the application.

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

53999

9 781932 394757

MANNING

$39.99 / Can $51.99