CHAPTER 13

Database Patterns

Most web applications of any complexity handle persistence to a greater or lesser extent. Shops must recall their products and their customer records. Games must remember their players and the state of play. Social networking sites must keep track of your 238 friends and your unaccountable liking for boy-bands of the ’80s and ’90s. Whatever the application, the chances are it’s keeping score behind the scenes. In this chapter, I look at some patterns that can help.

This chapter will cover

- *The Data Layer interface*: Patterns that define the points of contact between the storage layer and the rest of the system
- *Object watching*: Keeping track of objects, avoiding duplicates, automating save and insert operations
- *Flexible queries*: Allowing your client coders to construct queries without thinking about the underlying database
- *Creating lists of found objects*: Building iterable collections
- *Managing your database components*: The welcome return of the Abstract Factory pattern

The Data Layer

In discussions with clients, it’s usually the presentation layer that dominates. Fonts, colors, and ease of use are the primary topics of conversation. Amongst developers it is often the database that looms large. It’s not the database itself that concerns us; we can trust that to do its job unless we’re very unlucky. No, it’s the mechanisms we use to translate the rows and columns of a database table into data structures that cause the problems. In this chapter, I look at code that can help with this process.

Not everything presented here sits in the Data layer itself. Rather I have grouped some of the patterns that help to solve persistence problems. All of these patterns are described by one or more of Clifton Nock, Martin Fowler, and Alur et al.

Data Mapper

If you thought I glossed over the issue of saving and retrieving `Venue` objects from the database in the “Domain Model” section of Chapter 12, here is where you might find at least some answers. The Data Mapper pattern is described by both Alur et al in *Core J2EE Patterns* (as Data Access Object) and Martin Fowler in *Patterns of Enterprise Application Architecture* (in fact, Data Access Object is not an exact match, as it generates data transfer objects, but since such objects are designed to become the real thing if you add water, the patterns are close enough).

As you might imagine, a data mapper is a class that is responsible for handling the transition from database to object.

The Problem

Objects are not organized like tables in a relational database. As you know, database tables are grids made up of rows and columns. One row may relate to another in a different (or even the same) table by means of a foreign key. Objects, on the other hand, tend to relate to one another more organically. One object may contain another, and different data structures will organize the same objects in different ways, combining and recombining objects in new relationships at runtime. Relational databases are optimized to manage large amounts of tabular data, whereas classes and objects encapsulate smaller focussed chunks of information.

This disconnect between classes and relational databases is often described as the object-relational impedance mismatch (or simply impedance mismatch).

So how do you make that transition? One answer is to give a class (or a set of classes) responsibility for just that problem, effectively hiding the database from the domain model and managing the inevitable rough edges of the translation.

Implementation

Although with careful programming, it may be possible to create a single `Mapper` class to service multiple objects, it is common to see an individual `Mapper` for a major class in the Domain Model.

Figure 13–1 shows three concrete `Mapper` classes and an abstract superclass.

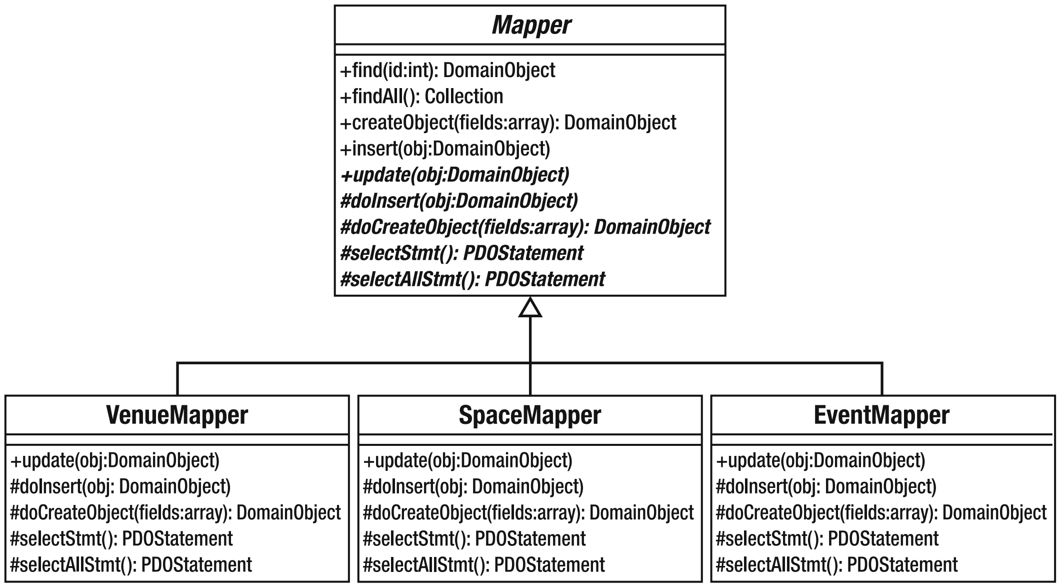
Mapper
+find(id:int): DomainObject
+findAll(): Collection
+createObject(fields:array): DomainObject
+insert(obj:DomainObject)
+update(obj:DomainObject)
#doInsert(obj:DomainObject)
#doCreateObject(fields:array): DomainObject
#selectStmt(): PDOStatement
#selectAllStmt(): PDOStatement
VenueMapper
+update(obj:DomainObject)
#doInsert(obj: DomainObject)
#doCreateObject(fields:array): DomainObject
#selectStmt(): PDOStatement
#selectAllStmt(): PDOStatement
SpaceMapper
+update(obj:DomainObject)
#doInsert(obj: DomainObject)
#doCreateObject(fields:array): DomainObject
#selectStmt(): PDOStatement
#selectAllStmt(): PDOStatement
EventMapper
+update(obj:DomainObject)
#doInsert(obj: DomainObject)
#doCreateObject(fields:array): DomainObject
#selectStmt(): PDOStatement
#selectAllStmt(): PDOStatement

Figure 13–1. *Mapper classes*

In fact, since the Space objects are effectively subordinate to Venue objects, it may be possible to factor the SpaceMapper class into VenueMapper. For the sake of these exercises, I'm going to keep them separate.

As you can see, the classes present common operations for saving and loading data. The base class stores common functionality, delegating responsibility for handling object-specific operations to its children. Typically, these operations include actual object generation and constructing queries for database operations.

The base class often performs housekeeping before or after an operation, which is why Template Method is used for explicit delegation (calls from concrete methods like insert() to abstract ones like doInsert(), etc.). Implementation determines which of the base class methods are made concrete in this way, as you will see later in the chapter.

Here is a simplified version of a Mapper base class:

```
namespace woo\mapper;
//...

abstract class Mapper {
    protected static $PDO;
    function __construct() {

        if ( ! isset(self::$PDO) ) {
            $dsn = \woo\base\ApplicationRegistry::getDSN( );
            if ( is_null( $dsn ) ) {
                throw new \woo\base\AppException( "No DSN" );
            }
            self::$PDO = new \PDO( $dsn );
            self::$PDO->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
        }
    }

    function find( $id ) {
        $this->selectStmt()->execute( array( $id ) );
        $array = $this->selectStmt()->fetch( );
        $this->selectStmt()->closeCursor( );
        if ( ! is_array( $array ) ) { return null; }
        if ( ! isset( $array['id'] ) ) { return null; }
        $object = $this->createObject( $array );
        return $object;
    }

    function createObject( $array ) {
        $obj = $this->doCreateObject( $array );
        return $obj;
    }

    function insert( \woo\domain\DomainObject $obj ) {
        $this->doInsert( $obj );
    }

    abstract function update( \woo\domain\DomainObject $object );
    protected abstract function doCreateObject( array $array );
    protected abstract function doInsert( \woo\domain\DomainObject $object );
    protected abstract function selectStmt();
}
```

The constructor method uses an ApplicationRegistry to get a DSN for use with the PDO extension. A standalone singleton or a request-scoped registry really come into their own for classes like this. There isn't always a sensible path from the control layer to a Mapper along which data can be passed. Another way of managing mapper creation would be to hand it off to the Registry class itself. Rather than instantiate it, the mapper would expect to be *provided* with a PDO object as a constructor argument.

```
namespace woo\mapper;
//...
abstract class Mapper {
    protected $PDO;
    function __construct( \PDO $pdo ) {
        $this->pdo = $pdo;
    }
}
```

Client code would acquire a new VenueMapper from Registry using \woo\base\Request Registry::getVenueMapper( ). This would instantiate a mapper, generating the PDO object too. For subsequent requests, the method would return the cached mapper. The trade-off here is that you make Registry much more knowledgeable about your system, but your mappers remain ignorant of global configuration data.

The insert() method does nothing but delegate to doInsert(). This would be something that I would factor out in favor of an abstract insert() method were it not for the fact that I know that the implementation will be useful here in due course.

find() is responsible for invoking a prepared statement (provided by an implementing child class) and acquiring row data. It finishes up by calling createObject(). The details of converting an array to an object will vary from case to case, of course, so the details are handled by the abstract doCreateObject() method. Once again, createObject() seems to do nothing but delegate to the child implementation, and once again, I'll soon add the housekeeping that makes this use of the Template Method pattern worth the trouble.

Child classes will also implement custom methods for finding data according to specific criteria (I will want to locate Space objects that belong to Venue objects, for example).

You can take a look at the process from the child's perspective here:

```
namespace woo\mapper;
//...

class VenueMapper extends Mapper {
    function __construct() {
        parent::__construct();
        $this->selectStmt = self::$PDO->prepare(
                            "SELECT * FROM venue WHERE id=?");
        $this->updateStmt = self::$PDO->prepare(
                            "update venue set name=?, id=? where id=?");
        $this->insertStmt = self::$PDO->prepare(
                            "insert into venue ( name )
                             values( ? )");
    }

    function getCollection( array $raw ) {
        return new SpaceCollection( $raw, $this );
    }
    protected function doCreateObject( array $array ) {
        $obj = new \woo\domain\Venue( $array['id'] );
        $obj->setname( $array['name'] );
        return $obj;
    }

    protected function doInsert( \woo\domain\DomainObject $object ) {
        print "inserting\n";
        debug_print_backtrace();
        $values = array( $object->getName() );
        $this->insertStmt->execute( $values );
        $id = self::$PDO->lastInsertId();
        $object->setId( $id );
    }

    function update( \woo\domain\DomainObject $object ) {
        print "updating\n";
        $values = array( $object->getName(), $object->getId(), $object->getId() );
        $this->updateStmt->execute( $values );
    }

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

Once again, this class is stripped of some of the goodies that are still to come. Nonetheless, it does its job. The constructor prepares some SQL statements for use later on. These could be made static and shared across `VenueMapper` instances, or as described earlier, a single `Mapper` object could be stored in a `Registry`, thereby saving the cost of repeated instantiation. These are refactorings I will leave to you!

The `Mapper` class implements `find()`, which invokes `selectStmt()` to acquire the prepared `SELECT` statement. Assuming all goes well, `Mapper` invokes `VenueMapper::doCreateObject()`. It's here that I use the associative array to generate a `Venue` object.

From the point of view of the client, this process is simplicity itself:

```
$mapper = new \woo\mapper\VenueMapper();
$venue = $mapper->find( 12 );
print_r( $venue );
```

The `print_r()` method is a quick way of confirming that `find()` was successful. In my system (where there is a row in the `venue` table with ID `12`), the output from this fragment is as follows:

```
woo\domain\Venue Object
(
    [name:woo\domain\Venue:private] => The Eyeball Inn
    [spaces:woo\domain\Venue:private] =>
    [id:woo\domain\DomainObject:private] => 12
)
```

The `doInsert()` and `update()` methods reverse the process established by `find()`. Each accepts a `DomainObject`, extracts row data from it, and calls `PDOStatement::execute()` with the resulting information. Notice that the `doInsert()` method sets an ID on the provided object. Remember that objects are passed by reference in PHP, so the client code will see this change via its own reference.

Another thing to note is that `doInsert()` and `update()` are not really type safe. They will accept any `DomainObject` subclass without complaint. You should perform an `instanceof` test and throw an `Exception` if the wrong object is passed. This will guard against the inevitable bugs.

Once again, here is a client perspective on inserting and updating:

```
$venue = new \woo\domain\Venue();
$venue->setName( "The Likey Lounge-yy" );
// add the object to the database
$mapper->insert( $venue );
// find the object again – just prove it works!
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
// alter our object
$venue->setName( "The Bibble Beer Likey Lounge-yy" );
// call update to enter the amended data
$mapper->update( $venue );
// once again, go back to the database to prove it worked
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
```

Handling Multiple Rows

The `find()` method is pretty straightforward, because it only needs to return a single object. What do you do, though, if you need to pull lots of data from the database? Your first thought may be to return an array of objects. This will work, but there is a major problem with the approach.

If you return an array, each object in the collection will need to be instantiated first, which, if you have a result set of 1,000 objects, may be needlessly expensive. An alternative would be to simply return an array and let the calling code sort out object instantiation. This is possible, but it violates the very purpose of the `Mapper` classes.

There is one way you can have your cake and eat it. You can use the built-in `Iterator` interface.

The `Iterator` interface requires implementing classes to define methods for querying a list. If you do this, your class can be used in `foreach` loops just like an array. There are some people who say that iterator implementations are unnecessary in a language like PHP with such good support for arrays. Tish and piffle! I will show you at least three good reasons for using PHP's built-in `Iterator` interface in this chapter.

Table 13–1 shows the methods that the `Iterator` interface requires.

Table 13–1. Methods Defined by the Iterator Interface

Name	Description
`rewind()`	Send pointer to start of list.
`current()`	Return element at current pointer position.
`key()`	Return current key (i.e., pointer value).
`next()`	Return element at current pointer and advance pointer.
`valid()`	Confirm that there is an element at the current pointer position.

In order to implement an `Iterator`, you need to implement its methods and keep track of your place within a dataset. How you acquire that data, order it, or otherwise filter it is hidden from the client.

Here is an Iterator implementation that wraps an array but also accepts a Mapper object in its constructor for reasons that will become apparent:

```
namespace woo\mapper;
//...

abstract class Collection implements \Iterator {
    protected $mapper;
    protected $total = 0;
    protected $raw = array();

    private $result;
    private $pointer = 0;
    private $objects = array();

    function __construct( array $raw=null, Mapper $mapper=null ) {
        if ( ! is_null( $raw ) && ! is_null( $mapper ) ) {
            $this->raw = $raw;
            $this->total = count( $raw );
        }
        $this->mapper = $mapper;
    }

    function add( \woo\domain\DomainObject $object ) {
        $class = $this->targetClass();
        if ( ! ($object instanceof $class ) ) {
            throw new Exception("This is a {$class} collection");
        }
        $this->notifyAccess();
        $this->objects[$this->total] = $object;
        $this->total++;
    }

    abstract function targetClass();

    protected function notifyAccess() {
        // deliberately left blank!
    }
    private function getRow( $num ) {
        $this->notifyAccess();
        if ( $num >= $this->total || $num < 0 ) {
            return null;
        }
        if ( isset( $this->objects[$num]) ) {
            return $this->objects[$num];
        }

        if ( isset( $this->raw[$num] ) ) {
            $this->objects[$num]=$this->mapper->createObject( $this->raw[$num] );
            return $this->objects[$num];
        }
    }

    public function rewind() {
       $this->pointer = 0;
    }

   public function current() {
       return $this->getRow( $this->pointer );
   }

   public function key() {
       return $this->pointer;
   }

   public function next() {
        $row = $this->getRow( $this->pointer );
        if ( $row ) { $this->pointer++; }
        return $row;
   }

   public function valid() {
       return ( ! is_null( $this->current() ) );
   }
}
```

The constructor expects to be called with no arguments or with two (the raw data that may eventually be transformed into objects and a mapper reference).

Assuming that the client has set the `$raw` argument (it will be a `Mapper` object that does this), this is stored in a property together with the size of the provided dataset. If raw data is provided an instance of the `Mapper` is also required, since it's this that will convert each row into an object.

If no arguments were passed to the constructor, the class starts out empty, though note that there is the `add()` method for adding to the collection.

The class maintains two arrays: `$objects` and `$raw`. If a client requests a particular element, the `getRow()` method looks first in `$objects` to see if it has one already instantiated. If so, that gets returned. Otherwise, the method looks in `$raw` for the row data. `$raw` data is only present if a `Mapper` object is also present, so the data for the relevant row can be passed to the `Mapper::createObject()` method you encountered earlier. This returns a `DomainObject` object, which is cached in the `$objects` array with the relevant index. The newly created `DomainObject` object is returned to the user.

The rest of the class is simple manipulation of the `$pointer` property and calls to `getRow()`. Apart, that is, from the `notifyAccess()` method, which will become important when you encounter the Lazy Load pattern.

You may have noticed that the `Collection` class is abstract. You need to provide specific implementations for each domain class:

```
namespace woo\mapper;
//...


class VenueCollection
        extends Collection
        implements \woo\domain\VenueCollection {

    function targetClass( ) {
        return "\woo\domain\Venue";
    }
}
```

The `VenueCollection` class simply extends `Collection` and implements a `targetClass()` method. This, in conjunction with the type checking in the super class's `add()` method, ensures that only `Venue` objects can be added to the collection. You could provide additional checking in the constructor as well if you wanted to be even safer.

Clearly, this class should only work with a `VenueMapper`. In practical terms, though, this is a reasonably type-safe collection, especially as far as the Domain Model is concerned.

There are parallel classes for Event and Space objects, of course.

Note that `VenueCollection` implements an interface: `woo\domain\VenueCollection`. This is part of the Separated Interface trick I will describe shortly. In effect, it allows the `domain` package to define its requirements for a `Collection` independently of the `mapper` package. `Domain` objects hint for `woo\domain\VenueCollection` objects and not `woo\mapper\VenueCollection` objects, so that, at a later date, the `mapper` implementation might be removed. It could then be replaced with an entirely different implementing class without many changes within the domain package.

Here is the `\woo\domain\VenueCollection` interface, together with its siblings.

```
namespace woo\domain;



interface VenueCollection extends \Iterator {
    function add( DomainObject $venue );
}

interface SpaceCollection extends \Iterator {
    function add( DomainObject $space );
}

interface EventCollection extends \Iterator {
    function add( DomainObject $event );
}
```

Figure 13–2 shows some Collection classes.

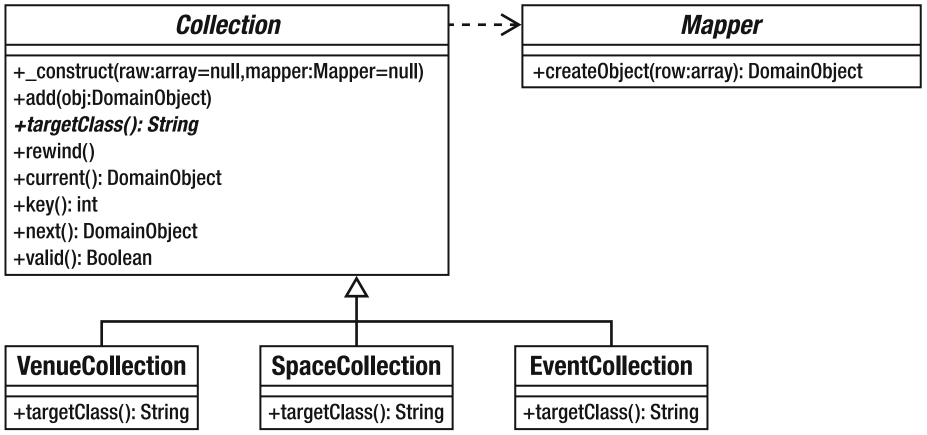

Figure 13–2. *Managing multiple rows with collections*

Because the Domain Model needs to instantiate Collection objects, and because I may need to switch the implementation at some point (especially for testing purposes), I provide a factory class in the Domain layer for generating Collection objects on a type-by-type basis. Here's how I get an empty VenueCollection object:

```
$collection = \woo\domain\HelperFactory::getCollection("woo\\domain\\Venue");
$collection->add( new \woo\domain\Venue( null, "Loud and Thumping" ) );
$collection->add( new \woo\domain\Venue( null, "Eeezy" ) );
$collection->add( new \woo\domain\Venue( null, "Duck and Badger" ) );

foreach( $collection as $venue ) {
    print $venue->getName()."\n";
}
```

With the implementation I have built here, there isn't much else you can do with this collection, but adding elementAt(), deleteAt(), count(), and similar methods is a trivial exercise. (And fun, too! Enjoy!)

The DomainObject superclass is a good place for convenience methods that acquire collections.

```
// namespace woo\domain;
// ...

// DomainObject

    static function getCollection( $type ) {
        return HelperFactory::getCollection( $type );
    }

    function collection() {
        return self::getCollection( get_class( $this ) );
    }
```

The class supports two mechanisms for acquiring a Collection object: static and instance. In both cases, the methods simply call HelperFactory::getCollection() with a class name. You saw the static getCollection() method used in the Domain Model example Chapter 12. Figure 13–3 shows the HelperFactory. Notice that it can be used to acquire both collections and mappers.

A variation on the structure displayed in Figure 13–3 would have you create interfaces within the domain package for Mapper and Collection which, of course would need to be implemented by their mapper counterparts. In this way, domain objects can be completely insulated from the mapper package (except within the HelperFactory itself, of course). This basic pattern, which Fowler calls Separated Interface, would be useful if you knew that some users might need to switch out the entire mapper package and replace it with an equivalent. If I were to implement Separated Interface, getFinder() would commit to return an instance of a Finder interface, and my Mapper objects would implement this. However, in most instances, you can leave this refinement as a possible future refactor. In these examples, getFinder() returns Mapper objects pure and simple.

In light of all this, the Venue class can be extended to manage the persistence of Space objects. The class provides methods for adding individual Space objects to its SpaceCollection or for switching in an entirely new SpaceCollection.

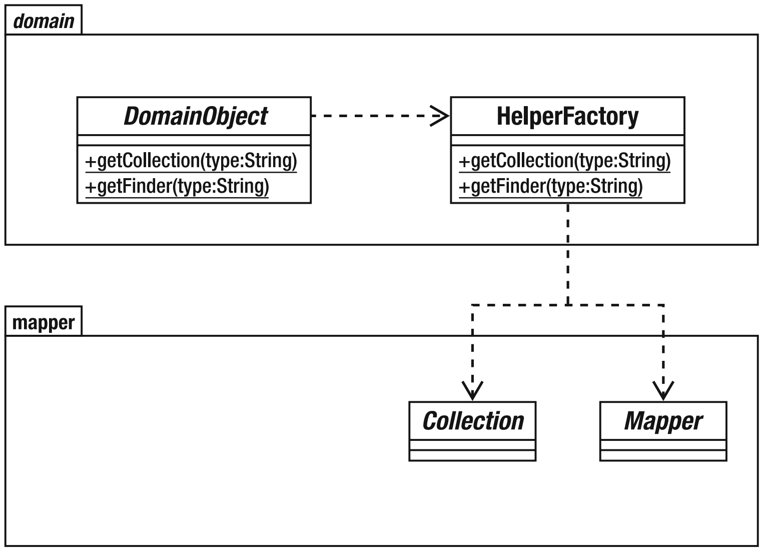

Figure 13–3. Using a factory object as an intermediary to acquire persistence tools

```
// Venue
// namespace woo\domain;
// ...

    function setSpaces( SpaceCollection $spaces ) {
        $this->spaces = $spaces;
    }

    function getSpaces() {
        if ( ! isset( $this->spaces ) ) {
            $this->spaces = self::getCollection("woo\\domain\\Space");
        }
        return $this->spaces;
    }

    function addSpace( wSpace $space ) {
        $this->getSpaces()->add( $space );
        $space->setVenue( $this );
    }
```

The `setSpaces()` operation is really designed to be used by the `VenueMapper` class in constructing the `Venue`. It takes it on trust that all `Space` objects in the collection refer to the current `Venue`. It would be easy enough to add checking to the method. This version keeps things simple though. Notice that I only instantiate the `$spaces` property when `getSpaces()` is called. Later on, I'll demonstrate how you can extend this lazy instantiation to limit database requests.

The `VenueMapper` needs to set up a `SpaceCollection` for each `Venue` object it creates.

```
// VenueMapper
// namespace woo\mapper;
// ...

    protected function doCreateObject( array $array ) {
        $obj = new w\woo\domain\Venue( $array['id'] );
        $obj->setname( $array['name'] );
        $space_mapper = new SpaceMapper();
        $space_collection = $space_mapper->findByVenue( $array['id'] );
        $obj->setSpaces( $space_collection );
        return $obj;
    }
```

The `VenueMapper::doCreateObject()` method gets a `SpaceMapper` and acquires a `SpaceCollection` from it. As you can see, the `SpaceMapper` class implements a `findByVenue()` method. This brings us to the queries that generate multiple objects. For the sake of brevity, I omitted the `Mapper::findAll()` method from the original listing for `woo\mapper\Mapper`. Here it is restored:

```
//Mapper
// namespace woo\mapper;
// ...

  function findAll( ) {
        $this->selectAllStmt()->execute( array() );
        return $this->getCollection(
            $this->selectAllStmt()->fetchAll( PDO::FETCH_ASSOC ) );
    }
```

This method calls a child method: `selectAllStmt()`. Like `selectStmt()`, this should contain a prepared statement object primed to acquire all rows in the table. Here's the `PDOStatement` object as created in the `SpaceMapper` class:

```
// SpaceMapper::__construct()
        $this->selectAllStmt = self::$PDO->prepare(
                            "SELECT * FROM space");
//...
        $this->findByVenueStmt = self::$PDO->prepare(
                            "SELECT * FROM space where venue=?");
```

I included another statement here, `$findByVenueStmt`, which is used to locate `Space` objects specific to an individual `Venue`.

The `findAll()` method calls another new method, `getCollection()`, passing it its found data. Here is `SpaceMapper::getCollection()`:

```
    function getCollection( array $raw ) {
        return new SpaceCollection( $raw, $this );
    }
```

A full version of the `Mapper` class should declare `getCollection()` and `selectAllStmt()` as abstract methods, so all mappers are capable of returning a collection containing their persistent domain objects. In order to get the `Space` objects that belong to a `Venue`, however, I need a more limited collection. You have already seen the prepared statement for acquiring the data; now, here is the `SpaceMapper::findByVenue()` method, which generates the collection:

```
    function findByVenue( $vid ) {
        $this->findByVenueStmt->execute( array( $vid ) );
        return new SpaceCollection(
            $this->findByVenueStmt->fetchAll(), $this );
    }
```

The findByVenue() method is identical to findAll() except for the SQL statement used. Back in the VenueMapper, the resulting collection is set on the Venue object via Venue::setSpaces().

So Venue objects now arrive fresh from the database, complete with all their Space objects in a neat type-safe list. None of the objects in that list are instantiated before being requested.

Figure 13–4 shows the process by which a client class might acquire a SpaceCollection and how the SpaceCollection class interacts with SpaceMapper::createObject() to convert its raw data into an object for returning to the client.

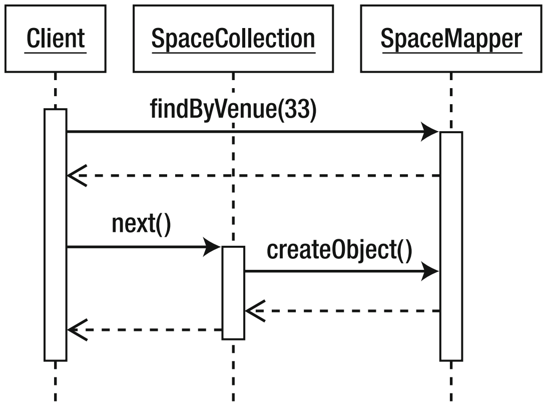

Figure 13–4. *Acquiring a SpaceCollection and using it to get a Space object*

Consequences

The drawback with the approach I took to adding Space objects to Venue ones is that I had to take two trips to the database. In most instances, I think that is a price worth paying. Also note that the work in Venue::doCreateObject() to acquire a correctly populated SpaceCollection could be moved to Venue::getSpaces() so that the secondary database connection would only occur on demand. Here's how such a method might look:

```
// Venue
// namespace woo\domain;
// ...

    function getSpaces() {
        if ( ! isset( $this->spaces ) ) {
            $finder = self::getFinder( 'woo\\domain\\Space' );
            $this->spaces = $finder->findByVenue( $this->getId() );
        }
        return $this->spaces;
    }
```

If efficiency becomes an issue, however, it should be easy enough to factor out SpaceMapper altogether and retrieve all the data you need in one go using an SQL join.

Of course, your code may become less portable as a result of that, but efficiency optimization always comes at a price!

Ultimately, the granularity of your `Mapper` classes will vary. If an object type is stored solely by another, then you may consider only having a `Mapper` for the container.

The great strength of this pattern is the strong decoupling it effects between the Domain layer and database. The `Mapper` objects take the strain behind the scenes and can adapt to all sorts of relational twistedness.

Perhaps the biggest drawback with the pattern is the sheer amount of slog involved in creating concrete `Mapper` classes. However, there is a large amount of boilerplate code that can be automatically generated. A neat way of generating the common methods for `Mapper` classes is through reflection. You can query a domain object, discover its setter and getter methods (perhaps in tandem with an argument naming convention), and generate basic `Mapper` classes ready for amendment. This is how all the `Mapper` classes featured in this chapter were initially produced.

One issue to be aware of with mappers is the danger of loading too many objects at one time. The `Iterator` implementation helps us here, though. Because a `Collection` object only holds row data at first, the secondary request (for a `Space` object) is only made when a particular `Venue` is accessed and converted from array to object. This form of lazy loading can be enhanced even further, as you shall see.

You should be careful of ripple loading. Be aware as you create your mapper that the use of another one to acquire a property for your object may be the tip of a very large iceberg. This secondary mapper may itself use yet more in constructing its own object. If you are not careful, you could find that what looks on the surface like a simple find operation sets off tens of other similar operations.

You should also be aware of any guidelines your database application lays down for building efficient queries and be prepared to optimize (on a database-by-database basis if necessary). SQL statements that apply well to multiple database applications are nice; fast applications are much nicer. Although introducing conditionals (or strategy classes) to manage different versions of the same queries is a chore, and potentially ugly in the former case, don't forget that all this mucky optimization is neatly hidden away from client code.

Identity Map

Do you remember the nightmare of pass-by-value errors in PHP 4? The sheer confusion that ensued when two variables that you thought pointed to a single object turned out to refer to different but cunningly similar ones? Well, the nightmare has returned.

The Problem

Here's some test code created to try out the Data Mapper example:

```
$venue = new \woo\domain\Venue();
$venue->setName( "The Likey Lounge" );
$mapper->insert( $venue );
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
$venue->setName( "The Bibble Beer Likey Lounge" );
$mapper->update( $venue );
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
```

The purpose of this code was to demonstrate that an object that you add to the database could also be extracted via a `Mapper` and would be identical. Identical, that is, in every way except for being the *same* object. I cheated this problem by assigning the new `Venue` object over the old. Unfortunately, you won't always have that kind of control over the situation. The same object may be referenced at several different times within a single request. If you alter one version of it and save that to the database, can you be sure that another version of the object (perhaps stored already in a `Collection` object) won't be written over your changes?

Not only are duplicate objects risky in a system, they also represent a considerable overhead. Some popular objects could be loaded three or four times in a process, with all but one of these trips to the database entirely redundant.

Fortunately, fixing this problem is relatively straightforward.

Implementation

An identity map is simply an object whose task it is to keep track of all the objects in a system, and thereby help to ensure that nothing that should be one object becomes two.

In fact, the Identity Map itself does not prevent this from happening in any active way. Its role is to manage information about objects. Here is a simple Identity Map:

```
namespace woo\domain;
//...

class ObjectWatcher {
    private $all = array();
    private static $instance;

    private function __construct() { }

    static function instance() {
        if ( ! self::$instance ) {
            self::$instance = new ObjectWatcher();
        }
        return self::$instance;
    }

    function globalKey( DomainObject $obj ) {
        $key = get_class( $obj ).".".$obj->getId();
        return $key;
    }

    static function add( DomainObject $obj ) {
        $inst = self::instance();
        $inst->all[$inst->globalKey( $obj )] = $obj;
    }

    static function exists( $classname, $id ) {
        $inst = self::instance();
        $key = "$classname.$id";
        if ( isset( $inst->all[$key] ) ) {
            return $inst->all[$key];
        }
        return null;
    }
}
```

Figure 13–5 shows how an Identity Map object might integrate with other classes you have seen.

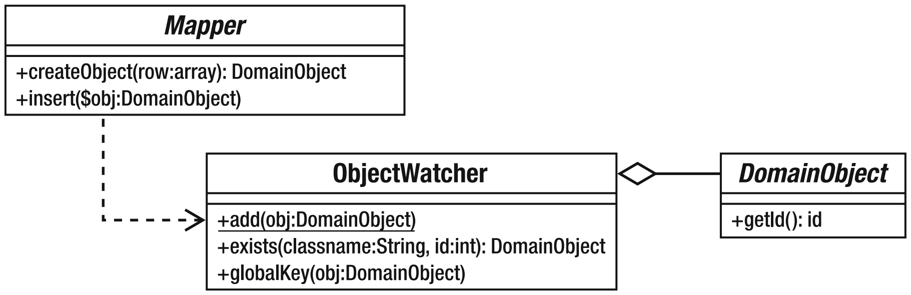
Mapper
+createObject(row:array): DomainObject
+insert($obj:DomainObject)
ObjectWatcher
+add(obj:DomainObject)
+exists(classname:String, id:int): DomainObject
+globalKey(obj:DomainObject)
DomainObject
+getId(): id

Figure 13–5. *Identity Map*

The main trick with an Identity Map is, pretty obviously, identifying objects. This means that you need to tag each object in some way. There are a number of different strategies you can take here. The database table key that all objects in the system already use is no good because the ID is not guaranteed to be unique across all tables.

You could also use the database to maintain a global key table. Every time you created an object, you would iterate the key table's running total and associate the global key with the object in its own row. The overhead of this is relatively slight, and it would be easy to do.

As you can see, I have gone for an altogether simpler approach. I concatenate the name of the object's class with its table ID. There can be no two objects of type `woo\domain\Event` with an ID of 4, so my key of `woo\domain\Event.4` is safe enough for my purposes.

The `globalKey()` method handles the details of this. The class provides an `add()` method for adding new objects. Each object is labeled with its unique key in an array property, `$all`.

The `exists()` method accepts a class name and an `$id` rather than an object. I don't want to have to instantiate an object to see whether or not it already exists! The method builds a key from this data and checks to see if it indexes an element in the `$all` property. If an object is found, a reference is duly returned.

There is only one class where I work with the `ObjectWatcher` class in its role as an Identity Map. The `Mapper` class provides functionality for generating objects, so it makes sense to add the checking there.

```
// Mapper
namespace woo\mapper;
// ...

 private function getFromMap( $id ) {
        return \woo\domain\ObjectWatcher::exists
                ( $this->targetClass(), $id );
    }

    private function addToMap( \woo\domain\DomainObject $obj ) {
        return \woo\domain\ObjectWatcher::add( $obj );
    }

    function find( $id ) {
        $old = $this->getFromMap( $id );
        if ( $old ) { return $old; }
        // work with db
        return $object;
    }
    function createObject( $array ) {
        $old = $this->getFromMap( $array['id']);
        if ( $old ) { return $old; }
        // construct object
        $this->addToMap( $obj );
        return $obj;
    }

    function insert( \woo\domain\DomainObject $obj ) {
        // handle insert. $obj will be updated with new id
        $this->addToMap( $obj );
    }
```

The class provides two convenience methods: addToMap() and getFromMap(). These save the bother of remembering the full syntax of the static call to ObjectWatcher. More importantly, they call down to the child implementation (VenueMapper, etc.) to get the name of the class currently awaiting instantiation.

This is achieved by calling targetClass(), an abstract method that is implemented by all concrete Mapper classes. It should return the name of the class that the Mapper is designed to generate. Here is the SpaceMapper class's implementation of targetClass():

```
    protected function targetClass() {
        return "woo\\domain\\Space";
    }
```

Both find() and createObject() first check for an existing object by passing the table ID to getFromMap(). If an object is found, it is returned to the client and method execution ends. If, however, there is no version of this object in existence yet, object instantiation goes ahead. In createObject(), the new object is passed to addToMap() to prevent any clashes in the future.

So why am I going through part of this process twice, with calls to getFromMap() in both find() and createObject()? The answer lies with Collections. When these generate objects, they do so by calling createObject(). I need to make sure that the row encapsulated by a Collection object is not stale and ensure that the latest version of the object is returned to the user.

Consequences

As long as you use the Identity Map in all contexts in which objects are generated from or added to the database, the possibility of duplicate objects in your process is practically zero.

Of course, this only works *within* your process. Different processes will inevitably access versions of the same object at the same time. It is important to think through the possibilities for data corruption engendered by concurrent access. If there is a serious issue, you may need to consider a locking strategy. You might also consider storing objects in shared memory or using an external object caching system like Memcached. You can learn about Memcached at http://danga.com/memcached/ and about PHP support for it at http://www.php.net/memcache.

Unit of Work

When do you save your objects? Until I discovered the Unit of Work pattern (written up by David Rice in Martin Fowler's *Patterns of Enterprise Application Architecture*), I sent out save orders from the Presentation layer upon completion of a command. This turned out to be an expensive design decision.

The Unit of Work pattern helps you to save only those objects that need saving.

The Problem

One day, I echoed my SQL statements to the browser window to track down a problem and had a shock. I found that I was saving the same data over and over again in the same request. I had a neat system of composite commands, which meant that one command might trigger several others, and each one was cleaning up after itself.

Not only was I saving the same object twice, I was saving objects that didn't need saving.

This problem then is similar in some ways to that addressed by Identity Map. That problem involved unnecessary object loading; this problem lies at the other end of the process. Just as these issues are complementary, so are the solutions.

Implementation

To determine what database operations are required, you need to keep track of various events that befall your objects. Probably the best place to do that is in the objects themselves.

You also need to maintain a list of objects scheduled for each database operation (insert, update, delete). I am only going to cover insert and update operations here. Where might be a good place to store a list of objects? It just so happens that I already have an `ObjectWatcher` object, so I can develop that further:

```
// ObjectWatcher
// ...
    private $all = array();
    private $dirty = array();
    private $new = array();
    private $delete = array(); // unused in this example
    private static $instance;
// ...
    static function addDelete( DomainObject $obj ) {

        $self = self::instance();

        $self->delete[$self->globalKey( $obj )] = $obj;

    }

    static function addDirty( DomainObject $obj ) {
        $inst = self::instance();
        if ( ! in_array( $obj, $inst->new, true ) ) {
            $inst->dirty[$inst->globalKey( $obj )] = $obj;
        }
    }

    static function addNew( DomainObject $obj ) {
        $inst = self::instance();
        // we don't yet have an id
        $inst->new[] = $obj;
    }

    static function addClean( DomainObject $obj ) {
        $self = self::instance();
        unset( $self->delete[$self->globalKey( $obj )] );
        unset( $self->dirty[$self->globalKey( $obj )] );
        $self->new = array_filter( $self->new,
                function( $a ) use ( $obj ) { return !( $a === $obj ); }
                );
    }

    function performOperations() {
        foreach ( $this->dirty as $key=>$obj ) {
            $obj->finder()->update( $obj );
        }
        foreach ( $this->new as $key=>$obj ) {
            $obj->finder()->insert( $obj );
        }
        $this->dirty = array();
        $this->new = array();
    }
```

The `ObjectWatcher` class remains an Identity Map and continues to serve its function of tracking all objects in a system via the `$all` property. This example simply adds more functionality to the class.

You can see the Unit of Work aspects of the `ObjectWatcher` class in Figure 13–6.

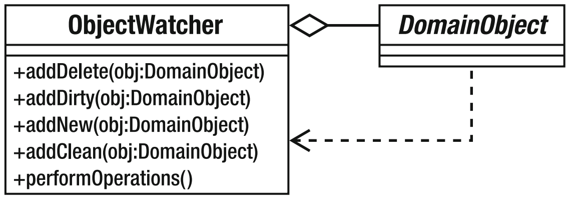
ObjectWatcher
+addDelete(obj:DomainObject)
+addDirty(obj:DomainObject)
+addNew(obj:DomainObject)
+addClean(obj:DomainObject)
+performOperations()
DomainObject

Figure 13–6. *Unit of Work*

Objects are described as “dirty” when they have been changed since extraction from the database. A dirty object is stored in the `$dirty` array property (via the `addDirty()` method) until the time comes to update the database. Client code may decide that a dirty object should not undergo update for its own reasons. It can ensure this by marking the dirty object as clean (via the `addClean()` method). As you might expect, a newly created object should be added to the `$new` array (via the `addNew()` method). Objects in this array are scheduled for insertion into the database. I am not implementing delete functionality in these examples, but the principle should be clear enough.

The `addDirty()` and `addNew()` methods each add an object to their respective array properties. `addClean()`, however, *removes* the given object from the `$dirty` array, marking it as no longer pending update.

When the time finally comes to process all objects stored in these arrays, the `performOperations()` method should be invoked (probably from the controller class, or its helper). This method loops through the `$dirty` and `$new` arrays either updating or adding the objects.

The `ObjectWatcher` class now provides a mechanism for updating and inserting objects. The code is still missing a means of adding objects to the `ObjectWatcher` object.

Since it is these objects that are operated upon, they are probably best placed to perform this notification. Here are some utility methods I can add to the `DomainObject` class. Notice also the constructor method.

```
// DomainObject
namespace woo\domain;
//...
```

```
abstract class DomainObject {
   private $id = -1;

   function __construct( $id=null ) {
       if ( is_null( $id ) ) {
           $this->markNew();
       } else {
           $this->id = $id;
       }
   }

   function markNew() {
       ObjectWatcher::addNew( $this );
   }

   function markDeleted() {
       ObjectWatcher::addDelete( $this );
   }

   function markDirty() {
       ObjectWatcher::addDirty( $this );
   }

   function markClean() {
       ObjectWatcher::addClean( $this );
   }

   function setId( $id ) {
       $this->id = $id;
   }

   function getId( ) {
       return $this->id;
   }

   function finder() {
       return self::getFinder( get_class( $this ) );
   }

   static function getFinder( $type ) {
       return HelperFactory::getFinder( $type );
   }
   //...
```

Before looking at the Unit of Work code, it is worth noting that the `Domain` class here has `finder()` and `getFinder()` methods. These work in exactly the same way as `collection()` and `getCollection()`, querying a simple factory class, `HelperFactory`, in order to acquire `Mapper` objects when needed. This relationship was illustrated in Figure 13–3.

As you can see, the constructor method marks the current object as new (by calling `markNew()`) if no `$id` property has been passed to it. This qualifies as magic of a sort and should be treated with some caution. As it stands, this code slates a new object for insertion into the database without any intervention from the object creator. Imagine a coder new to your team writing a throwaway script to test some domain behavior. No sign of persistence code there, so all should be safe enough, shouldn't it? Now imagine these test objects, perhaps with interesting throwaway names, making their way into persistent storage. Magic is nice, but clarity is nicer. It may be better to require client code to pass some kind of flag into the constructor in order to queue the new object for insertion.

I also need to add some code to the Mapper class:

```
// Mapper
    function createObject( $array ) {
        $old = $this->getFromMap( $array['id']);
        if ( $old ) { return $old; }
        $obj = $this->doCreateObject( $array );
        $this->addToMap( $obj );
        $obj->markClean();
        return $obj;
    }
```

Because setting up an object involves marking it new via the constructor's call to ObjectWatcher::addNew(), I must call markClean(), or every single object extracted from the database will be saved at the end of the request, which is not what I want.

The only thing remaining to do is to add markDirty() invocations to methods in the Domain Model classes. Remember, a dirty object is one that has been changed since it was retrieved from the database. This is the one aspect of this pattern that has a slightly fishy odor. Clearly, it's important to ensure that all methods that mess up the state of an object are marked dirty, but the manual nature of this task means that the possibility of human error is all too real.

Here are some methods in the Space object that call markDirty():

```
namespace woo\domain;

//...

class Space extends DomainObject {

//...

    function setName( $name_s ) {
        $this->name = $name_s;
        $this->markDirty();
    }

    function setVenue( Venue $venue ) {
        $this->venue = $venue;
        $this->markDirty();
    }
```

Here is some code for adding a new Venue and Space to the database, taken from a Command class:

```
        $venue = new \woo\domain\Venue( null, "The Green Trees" );
        $venue->addSpace(
            new \woo\domain\Space( null, 'The Space Upstairs' ) );
        $venue->addSpace(
            new \woo\domain\Space( null, 'The Bar Stage' ) );

        // this could be called from the controller or a helper class
        \woo\domain\ObjectWatcher::instance()->performOperations();
```

I have added some debug code to the ObjectWatcher, so you can see what happens at the end of the request:

```
inserting The Green Trees
inserting The Space Upstairs
inserting The Bar Stage
```

Because a high-level controller object usually calls the `performOperations()` method, all you need to do in most cases is create or modify an object, and the Unit of Work class (`ObjectWatcher`) will do its job just once at the end of the request.

Consequences

This pattern is very useful, but there are a few issues to be aware of. You need to be sure that all modify operations actually do mark the object in question as dirty. Failing to do this can result in hard-to-spot bugs.

You may like to look at other ways of testing for modified objects. Reflection sounds like a good option there, but you should look into the performance implications of such testing— the pattern is meant to improve efficiency, not undermine it.

Lazy Load

Lazy Load is one of those core patterns most Web programmers learn for themselves very quickly, simply because it's such an essential mechanism for avoiding massive database hits, which is something we all want to do.

The Problem

In the example that has dominated this chapter, I have set up a relationship between `Venue`, `Space`, and `Event` objects. When a `Venue` object is created, it is automatically given a `SpaceCollection` object. If I were to list every `Space` object in a `Venue`, this would automatically kick off a database request to acquire all the `Events` associated with each `Space`. These are stored in an `EventCollection` object. If I don't wish to view any events, I have nonetheless made several journeys to the database for no reason. With many venues, each with two or three spaces, and with each space managing tens, perhaps hundreds, of events, this is a costly process.

Clearly, we need to throttle back this automatic inclusion of collections in some instances.

Here is the code in `SpaceMapper` that acquires `Event` data:

```
protected function doCreateObject( array $array ) {
    $obj = new \woo\domain\Space( $array['id'] );
    $obj->setname( $array['name'] );
    $ven_mapper = new VenueMapper();
    $venue = $ven_mapper->find( $array['venue'] );
    $obj->setVenue( $venue );
    $event_mapper = new EventMapper();
    $event_collection = $event_mapper->findBySpaceId( $array['id'] );
    $obj->setEvents( $event_collection );
    return $obj;
}
```

The `doCreateObject()` method first acquires the `Venue` object with which the space is associated. This is not costly, because it is almost certainly already stored in the `ObjectWatcher` object. Then the method calls the `EventMapper::findBySpaceId()` method. This is where the system could run into problems.

Implementation

As you may know, a Lazy Load means to defer acquisition of a property until it is actually requested by a client.

As you have seen, the easiest way of doing this is to make the deferral explicit in the containing object. Here's how I might do this in the Space object:

```
    // Space
    function getEvents() {
        if ( is_null($this->events) ) {
            $this->events = self::getFinder('woo\\domain\\Event')
                ->findBySpaceId( $this->getId() );
        }
        return $this->events;
    }
```

This method checks to see whether or not the `$events` property is set. If it isn't set, then the method acquires a finder (that is, a `Mapper`) and uses its own `$id` property to get the `EventCollection` with which it is associated. Clearly, for this method to save us a potentially unnecessary database query, I would also need to amend the `SpaceMapper` code so that it does not automatically preload an `EventCollection` object as it does in the preceding example!

This approach will work just fine, although it is a little messy. Wouldn't it be nice to tidy the mess away?

This brings us back to the `Iterator` implementation that goes to make the `Collection` object. I amalready hiding one secret behind that interface (the fact that raw data may not yet have been used to instantiate a domain object at the time a client accesses it). Perhaps I can hide still more.

The idea here is to create an `EventCollection` object that defers its database access until a request is made of it. This means that a client object (such as `Space`, for example) need never know that it is holding an empty `Collection` in the first instance. As far as a client is concerned, it is holding a perfectly normal `EventCollection`.

Here is the `DeferredEventCollection` object:

```
namespace woo\mapper;
//...

class DeferredEventCollection extends EventCollection {
    private $stmt;
    private $valueArray;
    private $run=false;

    function __construct( Mapper $mapper, \PDOStatement $stmt_handle,
                          array $valueArray ) {
        parent::__construct( null, $mapper );
        $this->stmt = $stmt_handle;
        $this->valueArray = $valueArray;
    }

    function notifyAccess() {
        if ( ! $this->run ) {
            $this->stmt->execute( $this->valueArray );
            $this->raw = $this->stmt->fetchAll();
            $this->total = count( $this->raw );
        }
        $this->run=true;
    }
}
```

As you can see, this class extends a standard `EventCollection`. Its constructor requires `EventMapper` and `PDOStatement` objects and an array of terms that should match the prepared statement. In the first instance, the class does nothing but store its properties and wait. No query has been made of the database.

You may remember that the `Collection` base class defines the empty method called `notifyAccess()` that I mentioned in the "Data Mapper" section. This is called from any method whose invocation is the result of a call from the outside world.

`DeferredEventCollection` overrides this method. Now if someone attempts to access the `Collection`, the class knows it is time to end the pretense and acquire some real data. It does this by calling the `PDOStatement::execute()` method. Together with `PDOStatement::fetch()`, this yields an array of fields suitable for passing along to `Mapper::createObject()`.

Here is the method in `EventMapper` that instantiates a `DeferredEventCollection`:

```
// EventMapper
namespace woo\mapper;
// ...
function findBySpaceId( $s_id ) {
        return new DeferredEventCollection(
                    $this,
                    $this->selectBySpaceStmt, array( $s_id ) );
    }
```

Consequences

Lazy loading is a good habit to get into, whether or not you explicitly add deferred loading logic to your domain classes.

Over and above type safety, the particular benefit of using a collection rather than an array for your properties is the opportunity this gives you to retrofit lazy loading should you need it.

Domain Object Factory

The Data Mapper pattern is neat, but it does have some drawbacks. In particular a `Mapper` class takes a lot on board. It composes SQL statements; it converts arrays to objects and, of course, converts objects back to arrays, ready to add data to the database. This versatility makes a `Mapper` class convenient and powerful. It can reduce flexibility to some extent, however. This is especially true when a mapper must handle many different kinds of query or where other classes need to share a `Mapper`'s functionality. For the remainder of this chapter, I will decompose Data Mapper, breaking it down into a set of more focused patterns. These finer-grained patterns combine to duplicate the overall responsibilities managed in Data Mapper, and some or all can be used in conjunction with that pattern. They are well defined by Clifton Nock in *Data Access Patterns* (Addison Wesley 2003), and I have used his names where overlaps occur.

Let's start with a core function: the generation of domain objects.

The Problem

You have already encountered a situation in which the `Mapper` class displays a natural fault line. The `createObject()` method is used internally by `Mapper`, of course, but `Collection` objects also need it to create domain objects on demand. This requires us to pass along a `Mapper` reference when creating a

Collection object. While there's nothing wrong with allowing callbacks (as you have seen in the Visitor and Observer patterns,), it's neater to move responsibility for domain object creation into its own type. This can then be shared by Mapper and Collection classes alike.

The Domain Object Factory is described in *Data Access Patterns*.

Implementation

Imagine a set of Mapper classes, broadly organized so that each faces its own domain object. The Domain Object Factory pattern simply requires that you extract the createObject() method from each Mapper and place it in its own class in a parallel hierarchy. Figure 13–7 shows these new classes:

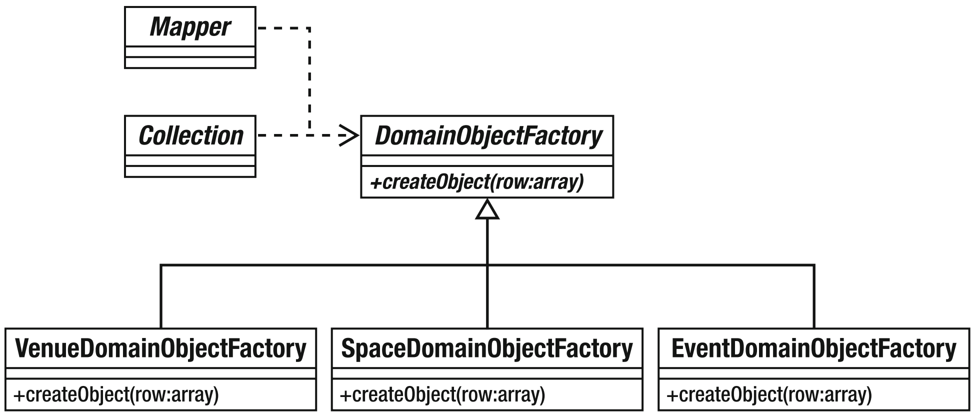

Figure 13–7. *Domain Object Factory classes*

Domain Object Factory classes have a single core responsibility, and as such, they tend to be simple:

```
namespace woo\mapper;
// ...

abstract class DomainObjectFactory {
    abstract function createObject( array $array );
}
```

Here's a concrete implementation:

```
namespace woo\mapper;
// ...
class VenueObjectFactory extends DomainObjectFactory {
    function createObject( array $array ) {
        $obj = new \woo\domain\Venue( $array['id'] );
        $obj->setname( $array['name'] );
        return $obj;
    }
}
```

Of course, you might also want to cache objects to prevent duplication and prevent unnecessary trips to the database as I did within the Mapper class. You could move the addToMap() and getFromMap()

methods here, or you could build an observer relationship between the ObjectWatcher and your createObject() methods. I'll leave the details up to you. Just remember, it's up to you to prevent clones of your domain objects running amok in your system!

Consequences

The Domain Object Factory decouples database row data from object field data. You can perform any number of adjustments within the createObject() method. This process is transparent to the client, whose responsibility it is to provide the raw data.

By snapping this functionality away from the Mapper class, it becomes available to other components. Here's an altered Collection implementation, for example:

```
namespace woo\mapper;
// ...

abstract class Collection {
    protected $dofact;
    protected $total = 0;
    protected $raw = array();

    // ...

    function __construct( array $raw=null, ➥
\woo\mapper\DomainObjectFactory $dofact=null ) {
        if ( ! is_null( $raw ) && ! is_null( $dofact ) ) {
            $this->raw = $raw;
            $this->total = count( $raw );
        }
        $this->dofact = $dofact;
    }
// ...
```

The DomainObjectFactory can be used to generate objects on demand:

```
        if ( isset( $this->raw[$num] ) ) {
            $this->objects[$num]=$this->dofact->createObject( $this->raw[$num] );
            return $this->objects[$num];
        }
```

Because Domain Object Factories are decoupled from the database, they can be used for testing more effectively. I might, for example, create a mock DomainObjectFactory to test the Collection code. It's much easier to do this than it would be to emulate an entire Mapper object (you can read more about mock and stub objects in Chapter 18).

One general effect of breaking down a monolithic component into composable parts is an unavoidable proliferation of classes. The potential for confusion should not be underestimated. Even when every component and its relationship with its peers is logical and clearly defined, I often find it challenging to chart packages containing tens of similarly named components.

This is going to get worse before it gets better. Already, I can see another fault line appearing in Data Mapper. The Mapper::getCollection() method was convenient, but once again, other classes might want to acquire a Collection object for a domain type, without having to go to a database facing class. So I have two related abstract components: Collection and DomainObjectFactory. According to the domain object I am working with, I will require a different set of concrete implementations: VenueCollection and VenueDomainObjectFactory, for example, or SpaceCollection and SpaceDomainObjectFactory. This problem leads us directly to the Abstract Factory pattern of course.

Figure 13–8 shows the `PersistenceFactory` class. I'll be using this to organize the various components that make up the next few patterns.

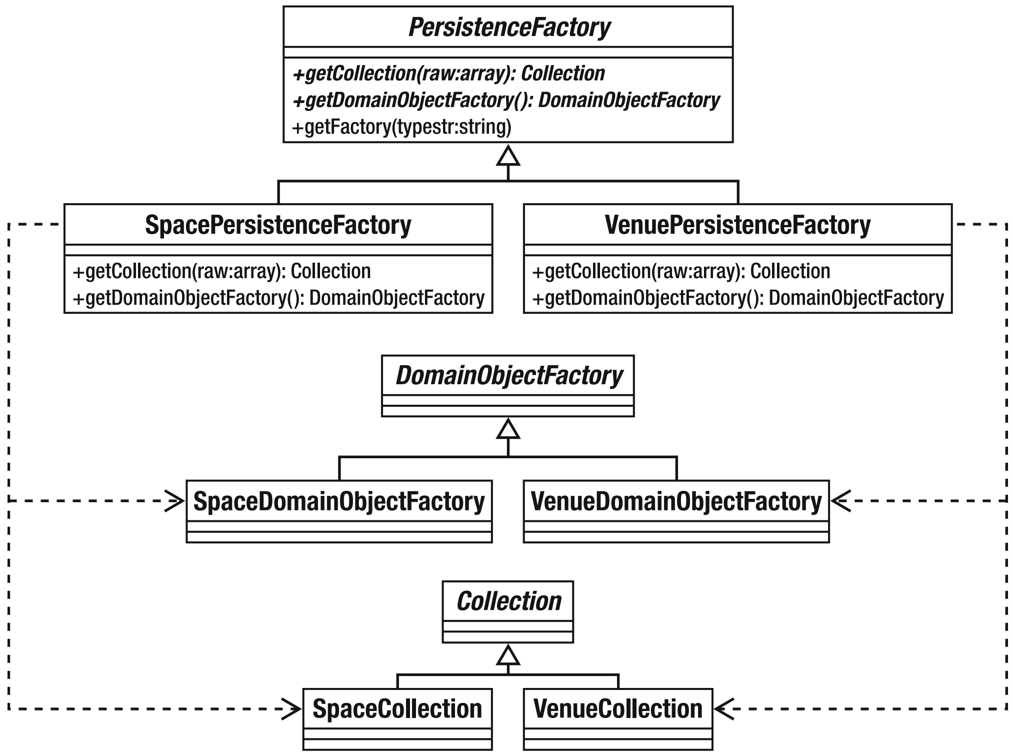
PersistenceFactory
+getCollection(raw:array): Collection
+getDomainObjectFactory(): DomainObjectFactory
+getFactory(typestr:string)
SpacePersistenceFactory
+getCollection(raw:array): Collection
+getDomainObjectFactory(): DomainObjectFactory
VenuePersistenceFactory
+getCollection(raw:array): Collection
+getDomainObjectFactory(): DomainObjectFactory
DomainObjectFactory
SpaceDomainObjectFactory
VenueDomainObjectFactory
Collection
SpaceCollection
VenueCollection

Figure 13–8. *Using the Abstract Factory pattern to organize related components*

The Identity Object

The mapper implementation I have presented here suffers from a certain inflexibility when it comes to locating domain objects. Finding an individual object is no problem. Finding all relevant domain objects is just as easy. Anything in between, though, requires you to add a special method to craft the query (`EventMapper::findBySpaceId()` is a case in point).

An identity object (also called a Data Transfer Object by Alur et al.) encapsulates query criteria, thereby decoupling the system from database syntax.

The Problem

It's hard to know ahead of time what you or other client coders are going to need to search for in a database. The more complex a domain object, the greater the number of filters you might need in your query. You can address this problem to some extent by adding more methods to your `Mapper` classes on a case-by-case basis. This is not very flexible, of course, and can involve duplication as you are required to craft many similar but differing queries both within a single Mapper class and across the mappers in your system.

An identity object encapsulates the conditional aspect of a database query in such a way that different combinations can be combined at runtime. Given a domain object called Person, for example, a client might be able to call methods on an identity object in order to specify a male, aged above 30 and below 40, who is under 6 feet tall. The class should be designed so conditions can combined flexibly (perhaps you're not interested in your target's height, or maybe you want to remove the lower age limit). An identity object limits a client coder's options to some extent. If you haven't written code to accommodate an income field, then this cannot be factored into a query without adjustment. The ability to apply different combinations of conditions does provide a step forward in flexibility, however. Let's see how this might work:

Implementation

An identity object will typically consist of a set of methods you can call to build query criteria. Having set the object's state, you can pass it on to a method responsible for constructing the SQL statement.

Figure 13–9 shows a typical set of IdentityObject classes.

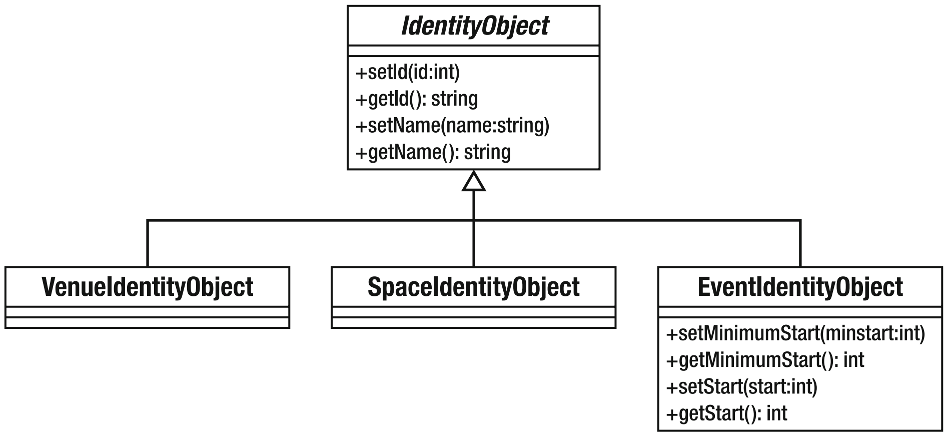

Figure 13–9. *Managing query criteria with identity objects*

You can use a base class to manage common operations and to ensure that your criteria objects share a type. Here's an implementation which is simpler even than the classes shown in Figure 13–9:

```
namespace woo\mapper;
//...

class IdentityObject {
    private $name = null;
    function setName( $name ) {
        $this->name=$name;
    }

    function getName() {
        return $this->name;
    }
}

class EventIdentityObject
    extends IdentityObject {
    private $start = null;
    private $minstart = null;

    function setMinimumStart( $minstart ) {
        $this->minstart = $minstart;
    }

    function getMinimumStart() {
        return $this->minstart;
    }

    function setStart( $start ) {
        $this->start = $start;
    }

    function getStart() {
        return $this->start;
    }
}
```

Nothing's too taxing here. The classes simply store the data provided and give it up on request. Here's some code that might use `SpaceIdentityObject` to build a `WHERE` clause:

```
$idobj = new EventIdentityObject();
$idobj->setMinimumStart( time() );
$idobj->setName( "A Fine Show" );
$comps = array();
$name = $idobj->getName();
if ( ! is_null( $name ) ) {
    $comps[] = "name = '{$name}'";
}
$minstart = $idobj->getMinimumStart();
if ( ! is_null( $minstart ) ) {
    $comps[] = "start > {$minstart}";
}

$start = $idobj->getStart();
if ( ! is_null( $start ) ) {
    $comps[] = "start = '{$start}'";
}

$clause = " WHERE " . implode( " and ", $comps );
```

This model will work well enough, but it does not suit my lazy soul. For a large domain object, the sheer number of getters and setters you would have to build is daunting. Then, following this model, you'd have to write code to output each condition in the `WHERE` clause. I couldn't even be bothered to handle all cases in my example code (no `setMaximumStart()` method for me), so imagine my joy at building identity objects in the real world.

Luckily, there are various strategies you can deploy to automate both the gathering of data and the generation of SQL. In the past, for example, I have populated associative arrays of field names in the base class. These were themselves indexed by comparison types: greater than, equal, less than or equal to. The child classes provide convenience methods for adding this data to the underlying structure. The SQL builder can then loop through the structure to build its query dynamically. I'm sure implementing such a system is just a matter of coloring in, so I'm going to look at a variation on it here.

I will use a fluent interface. That is a class whose setter methods return object instances, allowing your users to chain objects together in fluid, language-like way. This will satisfy my laziness, but still, I hope, give the client coder a flexible way of defining criteria.

I start by creating `woo\mapper\Field`, a class designed to hold comparison data for each field that will end up in the `WHERE` clause:

```
namespace woo\mapper;

class Field {
    protected $name=null;
    protected $operator=null;
    protected $comps=array();
    protected $incomplete=false;

    // sets up the field name (age, for example)
    function __construct( $name ) {
        $this->name = $name;
    }

    // add the operator and the value for the test
    // (> 40, for example) and add to the $comps property
    function addTest( $operator, $value ) {
        $this->comps[] = array( 'name' => $this->name,
            'operator' => $operator, 'value' => $value );
    }

    // comps is an array so that we can test one field in more than one way
    function getComps() { return $this->comps; }

    // if $comps does not contain elements, then we have
    // comparison data and this field is not ready to be used in
    // a query
    function isIncomplete() { return empty( $this->comps); }
}
```

This simple class accepts and stores a field name. Through the `addTest()` method the class builds an array of `operator` and `value` elements. This allows us to maintain more than one comparison test for a single field. Now, here's the new `IdentityObject` class:

```
namespace woo\mapper;

class IdentityObject {
    protected $currentfield=null;
    protected $fields = array();
    private $and=null;
    private $enforce=array();

    // an identity object can start off empty, or with a field
    function __construct( $field=null, array $enforce=null ) {
        if ( ! is_null( $enforce ) ) {
            $this->enforce = $enforce;
        }
        if ( ! is_null( $field ) ) {
            $this->field( $field );
        }
    }

    // field names to which this is constrained
    function getObjectFields() {
        return $this->enforce;
    }

    // kick off a new field.
    // will throw an error if a current field is not complete
    // (ie age rather than age > 40)
    // this method returns a reference to the current object
    // allowing for fluent syntax
    function field( $fieldname ) {
        if ( ! $this->isVoid() && $this->currentfield->isIncomplete() ) {
            throw new \Exception("Incomplete field");
        }
        $this->enforceField( $fieldname );
        if ( isset( $this->fields[$fieldname] ) ) {
            $this->currentfield=$this->fields[$fieldname];
        } else {
            $this->currentfield = new Field( $fieldname );
            $this->fields[$fieldname]=$this->currentfield;
        }
        return $this;
    }

    // does the identity object have any fields yet
    function isVoid() {
        return empty( $this->fields );
    }

    // is the given fieldname legal?
    function enforceField( $fieldname ) {
        if ( ! in_array( $fieldname, $this->enforce ) &&
             ! empty( $this->enforce ) ) {
            $forcelist = implode( ', ', $this->enforce );
            throw new \Exception("{$fieldname} not a legal field ($forcelist)");
        }
    }

    // add an equality operator to the current field
    // ie 'age' becomes age=40
    // returns a reference to the current object (via operator())
    function eq( $value ) {
        return $this->operator( "=", $value );
    }

    // less than
    function lt( $value ) {
        return $this->operator( "<", $value );
    }
    // greater than
    function gt( $value ) {
        return $this->operator( ">", $value );
    }


    // does the work for the operator methods
    // gets the current field and adds the operator and test value
    // to it
    private function operator( $symbol, $value ) {
        if ( $this->isVoid() ) {
            throw new \Exception("no object field defined");
        }
        $this->currentfield->addTest( $symbol, $value );
        return $this;
    }

    // return all comparisons built up so far in an associative array
    function getComps() {
        $ret = array();
        foreach ( $this->fields as $key => $field ) {
            $ret = array_merge( $ret, $field->getComps() );
        }
        return $ret;
    }
}
```

The easiest way to work out what's going on here is to start with some client code and work backward.

```
$idobj->field("name")->eq("The Good Show")
      ->field("start")->gt( time() )
                      ->lt( time()+(24*60*60) );
```

I begin by creating the `IdentityObject`. Calling `add()` causes a `Field` object to be created and assigned as the `$currentfield` property. Notice that `add()` returns a reference to the `identity` object. This allows us to hang more method calls off the back of the call to `add()`. The comparison methods `eq()`, `gt()`, and so forth each call `operator()`. This checks that there is a current `Field` object to work with, and if so, it passes along the operator symbol and the provided value. Once again, `eq()` returns an object reference, so that I can add new tests or call `add()` again to begin work with a new field.

Notice the way that the client code is almost sentence-like: field `"name"` equals `"The Good Show"` and field `"start"` is greater than the current time, but less than a day away.

Of course, by losing those hard-coded methods, I also lose some safety. This is what the `$enforce` array is designed for. Subclasses can invoke the base class with a set of constraints:

```
namespace woo\mapper;

class EventIdentityObject extends IdentityObject {
    function __construct( $field=null ) {
       parent::__construct( $field,
          array('name', 'id','start','duration',  'space' ) );
    }
}
```

The EventIdentityObject class now enforces a set of fields. Here's what happens if I try to work with a random field name:

```
PHP Fatal error:  Uncaught exception 'Exception' with message 'banana not a ➥
legal field (name, id, start, duration, space)'...
```

Consequences

Identity objects allow client coders to define search criteria without reference to a database query. They also save you from having to build special query methods for the various kinds of find operation your user might need.

Part of the point of an identity object is to shield users from the details of the database. It's important, therefore, that if you build an automated solution like the fluent interface in the preceding example, the labels you use should refer explicitly to your domain objects and not to the underlying column names. Where these differ, you should construct a mechanism for aliasing between them.

Where you use specialized entity objects, one for each domain object, it is useful to use an abstract factory (like PersistenceFactory described in the previous section) to serve them up along with other domain object related objects.

Now that I can represent search criteria, I can use this to build the query itself.

The Selection Factory and Update Factory Patterns

I have already pried a few responsibilities from the Mapper classes. With these patterns in place a Mapper does not need to create objects or collections. With query criteria handled by Identity Objects, it must no longer manage multiple variations on the find() method. The next stage is to remove responsibility for query creation.

The Problem

Any system that speaks to a database must generate queries, but the system itself is organized around domain objects and business rules rather than the database. Many of the patterns in this chapter can be said to bridge the gap between the tabular database and the more organic, treelike structures of the domain. There is, however, a moment of translation—the point at which domain data is transformed into a form that a database can understand. It is at this point that the true decoupling takes place.

Implementation

Of course, you have seen some of this functionality before in the Data Mapper pattern. In this specialization, though, I can benefit from the additional functionality afforded by the identity object pattern. This will tend to make query generation more dynamic, simply because the potential number of variations is so high.

Figure 13–10 shows my simple selection and update factories.

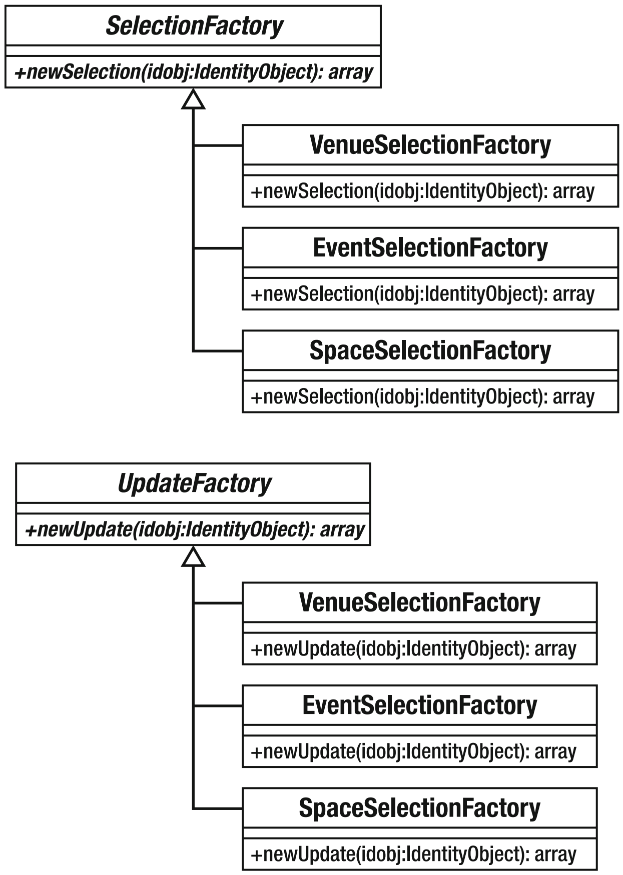
SelectionFactory
+newSelection(idobj:IdentityObject): array
VenueSelectionFactory
+newSelection(idobj:IdentityObject): array
EventSelectionFactory
+newSelection(idobj:IdentityObject): array
SpaceSelectionFactory
+newSelection(idobj:IdentityObject): array
UpdateFactory
+newUpdate(idobj:IdentityObject): array
VenueSelectionFactory
+newUpdate(idobj:IdentityObject): array
EventSelectionFactory
+newUpdate(idobj:IdentityObject): array
SpaceSelectionFactory
+newUpdate(idobj:IdentityObject): array

Figure 13–10. *Selection and update factories*

Selection and update factories are, once again, typically organized so that they parallel the domain objects in a system (possibly mediated via identity objects). Because of this, they are also candidates for my `PersistenceFactory`: the Abstract Factory I maintain as a one-stop shop for domain object persistence tools. Here is an implementation of a base class for update factories:

```
namespace woo\mapper;

abstract class UpdateFactory {

    abstract function newUpdate( \woo\domain\DomainObject $obj );

    protected function buildStatement( $table, array $fields, array $conditions=null ) {
        $terms = array();
        if ( ! is_null( $conditions ) ) {
            $query  = "UPDATE {$table} SET ";
            $query .= implode ( " = ?,", array_keys( $fields ) )." = ?";
            $terms = array_values( $fields );
            $cond = array();
            $query .= " WHERE ";
            foreach ( $conditions as $key=>$val ) {
                $cond[]="$key = ?";
                $terms[]=$val;
            }
            $query .= implode( " AND ", $cond );
        } else {
            $query  = "INSERT INTO {$table} (";
            $query .= implode( ",", array_keys($fields) );
            $query .= ") VALUES (";
            foreach ( $fields as $name => $value ) {
                $terms[]=$value;
                $qs[]='?';
            }
            $query .= implode( ",", $qs );
            $query .= ")";
        }
        return array( $query, $terms );
    }
}
```

In interface terms, the only thing that this class does is define the `newUpdate()` method. This will return an array containing a query string, and a list of terms to apply to it. The `buildStatement()` method does the generic work involved in building the update query, with the work specific to individual domain objects handled by child classes. `buildStatement()` accepts a table name, an associative array of fields and their values, and a similar associative array of conditions. The method combines these to create the query. Here's a concrete `UpdateFactory` class:

```
namespace woo\mapper;
//...

class VenueUpdateFactory extends UpdateFactory {

    function newUpdate( \woo\domain\DomainObject $obj ) {
        // not type checking removed
        $id = $obj->getId();
        $cond = null;
        $values['name'] = $obj->getName();
        if ( $id > -1 ) {
            $cond['id'] = $id;
        }
        return $this->buildStatement( "venue", $values, $cond );
    }
}
```

In this implementation, I work directly with a `DomainObject`. In systems where one might operate on many objects at once in an update, I could use an identity object to define the set on which I would like to act. This would form the basis of the `$cond` array, which here only holds `id` data.

`newUpdate()` distills the data required to generate a query. This is the process by which object data is transformed to database information.

Notice that the newUpdate() method will accept any DomainObject. This is so that all UpdateFactory classes can share an interface. It would be a good idea to add some further type checking to ensure the wrong object is not passed in.

You can see a similar structure for SelectionFactory classes. Here is the base class:

```
namespace woo\mapper;

//...

abstract class SelectionFactory {
    abstract function newSelection( IdentityObject $obj );

    function buildWhere( IdentityObject $obj ) {
        if ( $obj->isVoid() ) {
            return array( "", array() );
        }
        $compstrings = array();
        $values = array();
        foreach ( $obj->getComps() as $comp ) {
            $compstrings[] = "{$comp['name']} {$comp['operator']} ?";
            $values[] = $comp['value'];
        }
        $where = "WHERE " . implode( " AND ", $compstrings );
        return array( $where, $values );
    }
}
```

Once again, this class defines the public interface in the form of an abstract class. newSelection() expects an IdentityObject. Also requiring an IdentityObject but local to the type is the utility method buildWhere(). This uses the IdentityObject::getComps() method to acquire the information necessary to build a WHERE clause, and to construct a list of values, both of which it returns in a two element array.

Here is a concrete SelectionFactory class:

```
namespace woo\mapper;

//...

class VenueSelectionFactory extends SelectionFactory {

    function newSelection( IdentityObject $obj ) {
        $fields = implode( ',', $obj->getObjectFields() );
        $core = "SELECT $fields FROM venue";
        list( $where, $values ) = $this->buildWhere( $obj );
        return array( $core." ".$where, $values );
    }
}
```

This builds the core of the SQL statement and then calls buildWhere() to add the conditional clause. In fact, the only thing that differs from one concrete SelectionFactory to another in my test code is the name of the table. If I don't find that I require unique specializations soon, I will refactor these subclasses out of existence and use a single concrete SelectionFactory. This would query the table name from the PersistenceFactory.

Consequences

The use of a generic identity object implementation makes it easier to use a single parameterized SelectionFactory class. If you opt for hard-coded identity objects—that is, identity objects which consist of a list of getter and setter methods—you are more likely to have to build an individual SelectionFactory per domain object.

One of the great benefits of query factories combined with identity objects is the range of queries you can generate. This can also cause caching headaches. These methods generate queries on the fly, and it's difficult to know when you're duplicating effort. It may be worth building a means of comparing identity objects so that you can return a cached string without all that work. A similar kind of database statement pooling might be considered at a higher level too.

Another issue with the combination of patterns I have presented in the latter part of this chapter is the fact that they're flexible, but they're not *that* flexible. By this, I mean they are designed to be extremely adaptable within limits. There is not much room for exceptional cases here, though. Mapper classes, while more cumbersome to create and maintain, are very accommodating of any kind of performance kludge or data juggling you might need to perform behind their clean APIs. These more elegant patterns suffer from the problem that, with their focused responsibilities and emphasis on composition, it can be hard to cut across the cleverness and do something dumb but powerful.

Luckily, I have not lost my higher level interface—there's still a controller level where I can head cleverness off at the pass if necessary.

What's Left of Data Mapper Now?

So, I have stripped object, query, and collection generation from Data Mapper, to say nothing of the management of conditionals. What could possibly be left of it? Well, something that is very much like a mapper is needed in vestigial form. I still need an object that sits above the others I have created and coordinates their activities. It can help with caching duties and handle database connectivity (although the database-facing work could be delegated still further). Clifton Nock calls these data layer controllers domain object assemblers.

Here is an example:

```
namespace woo\mapper;

//...

class DomainObjectAssembler {
    protected static $PDO;

    function __construct( PersistenceFactory $factory ) {
        $this->factory = $factory;
        if ( ! isset(self::$PDO) ) {
            $dsn = \woo\base\ApplicationRegistry::getDSN( );
            if ( is_null( $dsn ) ) {
                throw new \woo\base\AppException( "No DSN" );
            }
            self::$PDO = new \PDO( $dsn );
            self::$PDO->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
        }
    }

    function getStatement( $str ) {
        if ( ! isset( $this->statements[$str] ) ) {
            $this->statements[$str] = self::$PDO->prepare( $str );
        }
        return $this->statements[$str];
    }

    function findOne( IdentityObject $idobj ) {
        $collection = $this->find( $idobj );
        return $collection->next();
    }

    function find( IdentityObject $idobj ) {
        $selfact = $this->factory->getSelectionFactory(  );
        list ( $selection, $values ) = $selfact->newSelection( $idobj );
        $stmt = $this->getStatement( $selection );
        $stmt->execute( $values );
        $raw = $stmt->fetchAll();
        return $this->factory->getCollection( $raw );
    }

    function insert( \woo\domain\DomainObject $obj ) {
        $upfact = $this->factory->getUpdateFactory(  );
        list( $update, $values ) = $upfact->newUpdate( $obj );
        $stmt = $this->getStatement( $update );
        $stmt->execute( $values );
        if ( $obj->getId() < 0 ) {
            $obj->setId( self::$PDO->lastInsertId()  );
        }
        $obj->markClean();
    }
}
```

As you can see, this is not an abstract class. Instead of itself breaking down into specializations, it uses the `PersistenceFactory` to ensure that it gets the correct components for the current domain object.

Figure 13–11 shows the high-level participants I built up as I factored out `Mapper`.

Aside from making the database connection and performing queries, the class manages `SelectionFactory` and `UpdateFactory` objects. In the case of selections, it also works either with a `Collection` class or directly with a `DomainObjectFactory` to generate return values.

From a client's point of view, acquiring a `DomainObjectFactory` is easy. It's simply a matter of getting the correct concrete `PersistenceFactory` object:

```
$factory = \woo\mapper\PersistenceFactory::getFactory("woo\\domain\\Venue" );
$finder = new \woo\mapper\DomainObjectAssembler( $factory );
```

Although, of course, it would be even easier to add a `getFinder()` method to the `PersistenceFactory` itself and transform the previous example into a one-liner like this:

```
$finder = \woo\mapper\PersistenceFactory::getFinder( 'woo\\domain\\Venue' );
```

I'll leave that to you, however.

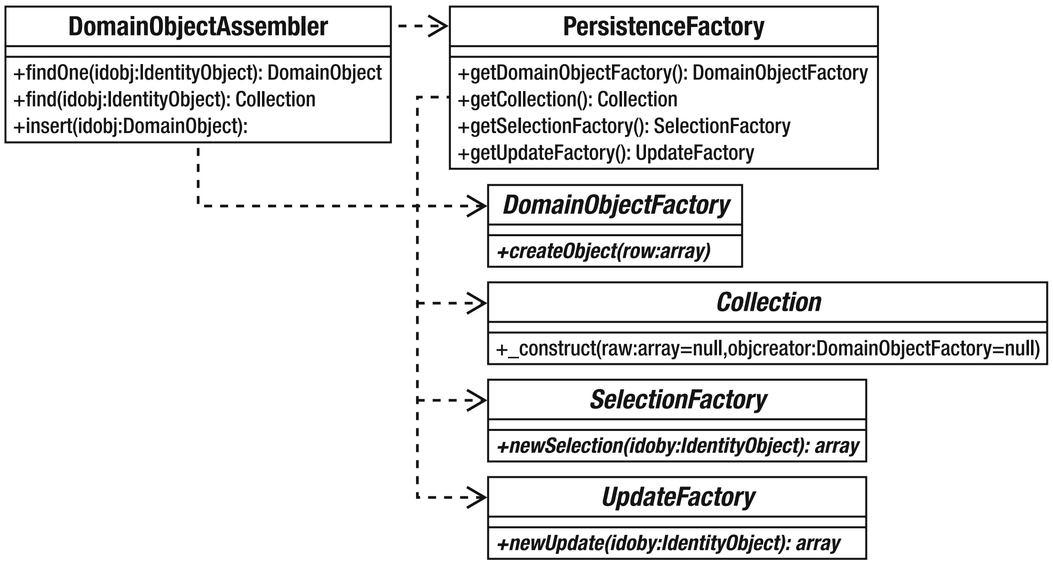
DomainObjectAssembler
+findOne(idobj:IdentityObject): DomainObject
+find(idobj:IdentityObject): Collection
+insert(idobj:DomainObject):
PersistenceFactory
+getDomainObjectFactory(): DomainObjectFactory
+getCollection(): Collection
+getSelectionFactory(): SelectionFactory
+getUpdateFactory(): UpdateFactory
DomainObjectFactory
+createObject(row:array)
Collection
+_construct(raw:array=null,objcreator:DomainObjectFactory=null)
SelectionFactory
+newSelection(idoby:IdentityObject): array
UpdateFactory
+newUpdate(idoby:IdentityObject): array

Figure 13–11. *Some of the persistence classes developed in this chapter*

A client coder might then go on to acquire a collection of Venue objects:

```
$idobj = $factory->getIdentityObject()->field('name')
        ->eq('The Eyeball Inn');
$collection = $finder->find( $idobj );


foreach( $collection as $venue ) {
    print $venue->getName()."\n";
}
```

Summary

As always, the patterns you choose to use will depend on the nature of your problem. I naturally gravitate toward a Data Mapper working with an identity object. I like neat automated solutions, but I also need to know I can break out of the system and go manual when I need to, while maintaining a clean interface and a decoupled database layer. I may need to optimize an SQL query, for example, or use a join to acquire data across multiple tables. Even if you're using a complex pattern-based third-party framework, you may find that the fancy object-relational mapping on offer does not do quite what you want. One test of a good framework, and of a good home-grown system, is the ease with which you can plug your own hack into place without degrading the overall integrity of the system as a whole. I love elegant, beautifully composed solutions, but I'm also a pragmatist!

Once again, I have covered a lot in this chapter. We examined the following patterns:

- *Data Mapper*: Create specialist classes for mapping Domain Model objects to and from relational databases.
- *Identity Map*: Keep track of all the objects in your system to prevent duplicate instantiations and unnecessary trips to the database.
- *Unit of Work*: Automate the process by which objects are saved to the database, ensuring that only objects that have been changed are updated and only those that have been newly created are inserted.
- *Lazy Load*: Defer object creation, and even database queries, until they are actually needed.
- *Domain Object Factory*: Encapsulate object creation functionality.
- *Identity Object*: Allow clients to construct query criteria without reference to the underlying database.
- *Query (Selection and Update) Factory*: Encapsulate the logic for constructing SQL queries.
- *Domain Object Assembler*: Construct a controller that manages the high-level process of data storage and retrieval.

In the next chapter, we take a welcome break from code, and I'll introduce some of the wider practices that can contribute to a successful project.