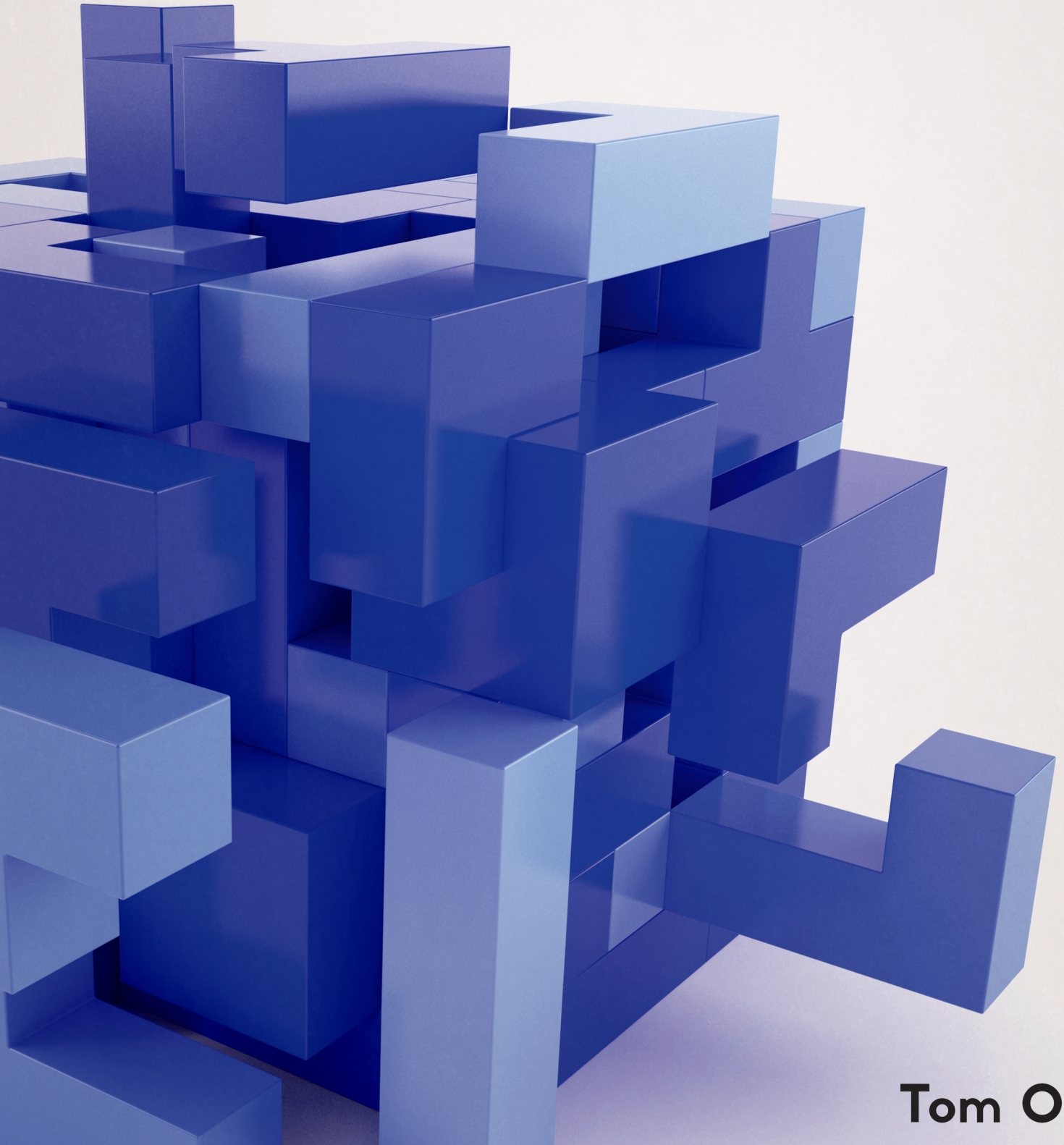
Modern Application Development in PHP



Tom Oram

Modern Application Development with PHP

Tom Oram

This book is for sale at http://leanpub.com/modern-application-development-with-php

This version was published on 2015-01-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Tom Oram by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Checkout out Modern Application Development in PHP, a book on current PHP development practices.

The suggested hashtag for this book is #ModernAppDevInPHP.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ModernAppDevInPHP

This book is dedicated to my mother Lesley Robinia. You inspired and supported me with everything I have done. Miss you mum!

9th May 1950 - 26th October 2014

Contents

Preface

I have been working professionally with PHP since 2000. In that time I have seen it grow from a basic scripting language which allowed you to hack your ideas together in version 3, to something that resembled a fully featured language in version 4, through to actually becoming that fully featured language in version 5.

One of the fantastic features of PHP is that it's so easy to get up and running quickly; you can easily add some dynamic elements to your existing HTML page with very little knowledge or experience. However, this is also a problem. Since anyone can start piecing together snippets of PHP code to get something to work, there are many people out there who are writing bad PHP code as they have never learnt how to program properly. Badly written code is often broken, very hard to add new functionality to, hard to maintain and rarely scalable.

In this book I hope to incorporate many programming methodologies, techniques and tools and apply them in the context of PHP to create a well designed application. These include Object Oriented Programming, Design Patterns, elements from Domain Driven Design, Refactoring, The SOLID Principals, Test Driven Development and Behaviour Driven Development. If these are all new to you, then this will be quite a ride.

I do not intend to cover any of these topics extensively. There are already many great books and resources covering all of them already and written by far cleverer people than myself. Therefore, I encourage you to read and learn more about all these subjects further. This book should, however, provide a great introduction to many of these methodologies, techniques and tools and show you how to get started quickly using them with PHP.

This book aims to introduce you to what you need to start building a well designed, manageable, and extensible medium size PHP application. The approach and process of working throughout this book should also fit very well into an *Agile* process of working. However, it will only deal with the actual development and architecture of the project rather than the full planning, communication and team management aspects included in the Agile way of working.

The architecture of the application we will be building in this book is a fairly common approach. It will not be based on any frameworks, but we will make use of some later on to see how frameworks can be very useful tools. In my mind this is how modern frameworks should be used. This architecture and approach to PHP application design is one we often use at the company I work for and is a tried and tested approach.

So finally, who is this book for? Firstly this is not an introduction to PHP. I assume you are already familiar with the language; you should be able to use *classes* and at least understand what an *interface* is. You have probably built at least one medium scale application but may not really know a lot about software design principles. If you have less experience than this you may struggle to keep

up. If you have more then there may be less to be learnt from this book but hopefully it will still be helpful.

Source Code

The example source code for this book is included as part of the Leanpub purchase. You can download it via your Leanpub Dashboard[1].

Discussing Book Content

I have created a Google Group which can be used by readers of this book to discuss the ideas and content. This group can be found at:

https://groups.google.com/forum/#!forum/modern-application-development-in-php

Getting in Contact

I'd love to hear from you! Whether it's about something in the book which you didn't understand, think I could improve or didn't agree with, or if you'd like to share how you do things differently, or even just to talk about software design in general then please get in touch.

You can find me on:

Twitter: @tomphp Email: tom@x2k.co.uk Skype: x2kmusic

Thanks

Many people have contributed to the creation of this book, as well as to my journey to get to this point of creating it. I would like to say thanks to:

My parents for inspiring and encouraging me to do positive things in my life. My collegues Felix and Rob for our daily debates and mutual encouragement to develop our skills further. John and Jamie for employing me in this industry and John for all his mentoring. George and Ollie for engaging with lengthy technical phone calls. Matthew for his inspiration and support to create this book. Rob, Felix, Lowri, Loki, Peter, Lee and Steph for pointing out errors and typos.

[1] https://leanpub.com/dashboard

Prerequisites

This section of the book makes sure you have all the knowledge and tools which will be needed to progress through the rest of the book. It will cover a lot of topics briefly; the idea is to give you enough information on each topic for you to progress through this book, however, each topic is pretty deep so I don't want to try and cover any of them completely. Many of them also have great books covering that topic exclusively and I encourage you to learn more about each of them yourself.

Terms and Conventions

Some Terms

Before getting started, I'd just like to introduce a few terms which I'll be using in this book. If you're used to reading about software development and architecture, then these may seem quite obvious. But if this is you're first real entry point into these topics, then it's probably useful to point these out now.

Stakeholder

When I talk about the *stakeholder*, I'm referring to the person (or persons) responsible for requesting the features you have to build into the application. If you're a freelancer and you're building a application for one of your clients, or if you work for a development company and you're working on a team building an application for one of the company's clients, then the client is the *stakeholder*. If your working in-house at a company, and you're developing software for the employees, and the management team are telling you what needs to be built: then the management team are the stakeholders.

Basically, the stakeholders are the people who will pay you or your company if you build what they want. They choose what features are needed, and you have to keep them happy!

Domain

The *domain* is the real world process or environment which you are trying to translate into software. If you are building an e-commerce application, then the domain is the sales process.

Every project's domain will be unique. The sales process of a small manufacturer with 10 products will be different to that of a company selling thousands of products from multiple warehouses.

The *domain* is considered to be the *problem space*. This is because it contains the problems which you need to solve in order to model it in software.

Domain Model

The *domain model* is the solutions to the domain's *problems*, it is considered to be the *solutions space*. The *domain model* is the core of your application and it models just the *domain*. It does not include the User Interface, the Database Layer, a Framework, API requests; it is a pure model of the data and logic in the *domain*.

A bit later on I'll introduce Domain Driven Design which is a process that holds the design of the *domain model* at utmost importance when creating software.

Conventions

In this book I will often instruct you to run commands in the terminal. Whenever I show a command to be run, it will be in fixed width text and preceded by a '$' symbol, like this:

```
$ ls -l
```

When executing the command, do not include the `$` symbol. For the example above, simply type `ls -l` into the terminal window, then press the `Enter` key on your keyboard.

Other times I will show the output of a command executed in the terminal. In this case I'll use the same fixed with font, but there will be no preceding `$`, like so:

```
total 13640
-rw-rw-r-- 1 tom tom      142 Aug 20 23:49 LICENSE.md
drwxrwxr-x 4 tom tom     4096 Oct 18 14:15 manuscript
-rw-rw-r-- 1 tom tom      956 Aug 20 23:56 README.md
-rw-rw-r-- 1 tom tom 13952786 Sep 24 22:25 tags
```

Finally, there are times where I'll show the command and the output in one block of text. The commands will be preceded by the `$` symbol, and the output will be in the lines which follow without the `$` symbol:

```
$ ls -l
total 13640
-rw-rw-r-- 1 tom tom      142 Aug 20 23:49 LICENSE.md
drwxrwxr-x 4 tom tom     4096 Oct 18 14:15 manuscript
-rw-rw-r-- 1 tom tom      956 Aug 20 23:56 README.md
-rw-rw-r-- 1 tom tom 13952786 Sep 24 22:25 tags
```

Everything else should hopefully be self explanatory.

Next we'll look at the tools you will need to work through this book.

The Development Environment

First up I will be using a Linux system in all my examples, therefore most of the content in terms of using the command line and configuration should translate directly if you're a Mac OS X user. If you're a Windows user things may be a bit different but shouldn't be too hard to work out.

Requirements

The only real requirement for working through this book is that you are using PHP 5.5 or above and the SQLite extension. Throughout this book we will be using PHP both from the command line and in a webserver environment.

Install PHP and SQLite on Ubuntu easily by running:

```
1 $ sudo apt-get install php5-cli php5-sqlite
```

When working in a web environment I will be using a tool called Vagrant, which runs the development environment in a virtual machine on the computer. This removes the need to set up and configure a webserver (and database servers) directly on the development computer. This is the way I would recommend working. However, if you do want to manually set up the relevant webserver and database servers on your development machine then you can do that, but you'll have to work that out yourself.

Vagrant

Vagrant is a neat little tool. It allows many developers working on the same project to run a local copy of the project's environment easily without having to install all the project's dependencies on their development machines. It does this by building and running a virtual machine from a config file included in the project.

Vagrant itself simply instructs a virtualisation provider on what type of virtual machine to create and then uses a configuration automation system to configure that virtual machine.

Different providers, such as VMWare, are available for use with Vagrant but I will be using VirtualBox.

Also, different configuration automation systems, such as Chef and Puppet, can be used with Vagrant. I am choosing to use Ansible just because I prefer the syntax.

Installing

In order to use Vagrant you will need to install:

- Vagrant[2]
- VirtualBox[3]
- Ansible[4]

I recommend that you install both Vagrant and VirtualBox by downloading the distribution packages directly from their websites so you get the current versions.

With Ansible, you need to make sure you have an up to date version. On Ubuntu I tend to install it via Rodney Quillo's PPA[5] since it's more up to date than the version in the Software Center.

Creating a Vagrant Config for PHP Development

Once you have Vagrant, VirtualBox and Ansible installed, it's time to build a Vagrant configuration.

It's not hard to build a Vagrant config file by hand, or to build the config automation scripts. However, it is a bit tedious, takes some learning, and is not really something this book intends to cover. Luckily there are some fantastic online tools available which make this process a lot easier. Since we want to build a PHP development environment and we're going to use Ansible. We will use a fantastic tool called Phansible[6] to create our config for us.

Phansible

First, open http://www.phansible.com/ in your browser. You will see a form asking questions about the development environment that you want to create. For this example choose the following options:

Options	Value
Operating System	Ubuntu Trusty Tahr (14.04) 64
Name	VagrantExample
IP Address	192.168.5.10
Memory	512
Shared Folder	./
Webserver	Apache + PHP5
PHP Version	5.6

[2] http://www.vagrantup.com/
[3] http://www.virtualbox.org/
[4] http://www.ansible.com/
[5] https://launchpad.net/~rquillo/+archive/ubuntu/ansible
[6] http://www.phansible.com/

You can use a different local IP address if you want, but remember it, since we'll need to use it shortly.

Ignore the **Database** and **Package** settings for now, and finally choose an appropriate **Timezone**.

Next, click the **Generate** button at the bottom of the form, and save the generated `.zip` file to your hard drive. I saved my file to `/home/tom/Downloads/phansible_VagrantExample.zip`.

Creating the Project

Now that we have generated the Vagrant configuration, let's create a PHP project and add the Vagrant configuration to it.

At your terminal, `cd` to where ever you want to create your project:

```
1 $ cd /home/tom/Projects
```

Next, create a directory for your new project, and `cd` into it:

```
1 $ mkdir VagrantExample
2 $ cd VagrantExample
```

Now create a file inside this directory called `index.php`, and add the following content, using your favourite IDE or text editor:

index.php

```
1 <?php
2
3 echo 'Hello wonderful World!';
```

Next, unpack the contents of the Vagrant configuration `.zip` file that we downloaded from the Phansible website earlier,

```
1 $ unzip /home/tom/Downloads/phansible_VagrantExample.zip
```

Finally, while still inside the project's directory and with your computer connected to the Internet, run:

```
$ vagrant up
```

This may take some time, and you may be prompted to enter your password to allow Vagrant to `sudo` to update some config files. Just be patient.

When it is done, open your browser and enter the IP address we selected earlier into the location bar like so:

`http://192.168.5.10/`

If all has gone to plan you should see `Hello wonderful World!` displayed on the page! Success! - we have created a PHP development environment for our project without installing or configuring a webserver on our local machine.

Once you have marvelled in the glory of Vagrant you can shut down the virtual machine by running the following command at your terminal:

```
$ vagrant halt
```

Shutting Down

I have found that if I forget to shutdown my virtual machines before I try to shut down my computer, it hangs during the shut down process. If you have a solution to this problem I'd love to hear it!

Getting up to Speed with PHP

As I said previously, this book assumes you are already familiar with PHP. In this chapter I will quickly cover a few newer additions to PHP, as well as a few tools and techniques which you will need to know about to continue with this book.

I don't intend to go into anything in too much depth. It will contain just enough information the things we will be using. Therefore, I encourage you to research them further yourself.

Namespaces

If *namespaces* are new to you then the easiest analogy I can think of is that they are like folders for your code. Using them means you can have 2 or more classes with the same name in different *namespaces* - in the same way that you can have 2 or more files with the same name in different folders.

Without covering all the details of PHP namespaces here, I will quickly cover the aspects of them which we will be using.

Firstly, in order to define a class inside a specific namespace, you use the `namespace` statement on the first line of the file containing the class. Like so:

Class defined inside namespace

```
1  <?php
2
3  namespace MyApp\Entity;
4
5  class Contact
6  {
7      // ...
8  }
```

This defines the class `Contact` inside the `Entity` *namespace* which is inside the `MyApp` *namespace*. Namespaces are separated by backslashes.

To use a class inside code which in the same namespace, you can simply refer to it by name:

Using a class defined in the current namespace

```
<?php

namespace MyApp\Entity;

$contact = new Contact();
```

To use a class inside code in a different namespace, you can refer to it by its *Fully Qualified Class Name* (FQCN). Like so:

Referencing a class by its FQCN

```
<?php

namespace MyApp;

$contact = new \MyApp\Entity\Contact();
```

You can also refer to a class in a sub-namespace relative to the current namespace. Like so:

Referencing a class by its relative namespace

```
<?php

namespace MyApp;

$contact = new Entity\Contact();
```

Finally, you can pull a class from a different namespace into scope with the `use` statement. Add this at the top of the file just after the `namespace` statement. This is the way I prefer in most situations - here's an example:

The use statement

```php
<?php

namespace MyApp;

use MyApp\Entity\Contact;

$contact = new Contact();
```

One final thing: if you want to use a class from one namespace in another namespace which already has a class with the same name in it, then you can rename the one you're importing with the `as` keyword. Like so:

Aliasing an imported class

```php
<?php

namespace MyApp\Form;

use MyApp\Entity\Contact as ContactEntity;

class Contact
{
    public function __construct(ContactEntity $entity)
    {
        // ...
    }

    // ...
}
```

It's all pretty simple really right? That's everything you'll need to know about *namespaces* to continue with this book. However, if you do want to learn more about them, then the full documentation for PHP namespaces can be found in the manual[7].

Typehints

PHP is a dynamically typed language. It has a few basic, scalar types:

[7] http://php.net/manual/en/language.namespaces.php

Scalar types in PHP

```
<?php

42; // integer

12.5; // float or double

'abc123'; // string

false; // boolean
```

It also has *arrays*, *callables*, *resources* and any user defined *classes* or *interfaces*, which are all also types.

Being a dynamically typed language means that a variable or function argument can contain any type of value at any time (this also goes for function return types):

Dynamic typing example

```
<?php

function fn($p)
{
    return $p;
}

class C1
{
}

class C2
{
}

$x = 42; // $x contains an integer

$x = 12.5; // now $x contains a float

$y = fn(123); // $p in fn() contains an integer and $y contains an integer

$y = fn(false); // $p in fn() contains a boolean and $y contains a boolean

```

```
$c = new C1(); // $c contains a C1 instance

$c = new C2(); // $c contains a C2 instance
```

In contrast: in a statically typed language a variable, function argument or return value can only ever be the type it is defined to contain. If another type is assigned it will either cause an error, or it will get converted. Here's a C++ version of the last example:

Static typing example (in C++)

```
int fn(int p) {
    return p;
}

class C1 {
};

class C2 {
};

int main() {
    int x = 42; // x contains an integer

    x = 12.5; // x contains 12, it keeps only the integer part

    y = fn(123); // p in fn() contains an integer and y contains an integer

    y = fn(false); // p in fn() contains 0 and y contains 0

    C1 *c = new C1(); // c contains a C1 instance

    c = new C2(); // this is an error as c can only contains instances of C1
}
```

Statically typed languages have great benefits. Because you always know what type everything is, there's never a chance of you doing something to a variable which is not allowed to be done the type it contains. On the other hand, dynamically typed languages let you get on and do things quickly and without having to worry about how to work with type constraints.

Since static typing does have benefits, PHP introduced *typehints* on function arguments. *Typehints* allow you to specify exactly what user defined type a function accepts for each parameter; PHP will throw an `InvalidArgumentException` exception if the wrong type is given:

PHP typehint example

```php
<?php

class C1
{
}

class C2
{
}

function fn(C1 $c)
{
}

fn(new C1()); // works perfectly

fn(new C2()); // error

fn(5); // error
```

Frustratingly PHP does not allow typehints for scalar types or function return values (yet).

However, even though PHP is a dynamically typed language you should still strive to keep your typing sensible. This means that if you create a variable that contains a specific type, you should try not to reuse it by assigning a new value of a different type to it. Also, don't call methods on objects if they are not in the typehinted *interface*. If you do PHP will produce be an error but it's really not good practice:

Mis-using an interface in PHP

```php
<?php

interface Fooer
{
    public function doFoo();
}

class FooBar implements Fooer
{
    public function doFoo()
    {
```

```
12      }
13
14      public function doBar()
15      {
16      }
17  }
18
19  function performAction(Fooer $f)
20  {
21      // This is fine, doFoo() is defined in the Fooer interface
22      $f->doFoo();
23
24      // Don't do this, $f is a Fooer and doBar() is not defined
25      // in the Fooer interface
26      $f->doBar();
27  }
28
29  performAction(new FooBar());
```

Throughout this book I will be writing PHP code as if I'm writing in a statically typed language 90% of the time - using typehint whenever possible. However, PHP is still a dynamic language and some times it's helpful to take advantage of this; whenever I do this I will point it out and explain my reason for choosing to do so.

Front Controllers

The front controller is a design pattern for web applications which involves creating a single entry point into your application. It requires you to configure your webserver to redirect all requests to a single PHP script which then processes the request and decides what content to display.

Let's take a look at an example:

The Traditional Approach

First, let's look at the traditional approach of using PHP. Create 2 files inside an empty folder, the first one we'll call `page1.php`:

page1.php

```
<?php

echo 'You are on page 1';
```

And the second one we'll call `page2.php`

page2.php

```
<?php

echo 'You are on page 2';
```

Next open a terminal and `cd` into the directory containing these files, Now, use the following command to start up PHP's built in webserver:

```
$ php -S localhost:8080
```

Now if you open your browser and go to `http://localhost:8080/page1.php` then you will see `You are viewing page 1`. If you then go to `http://localhost:8080/page2.php` you will see `You are viewing page 2`.

When you are done, press `CTRL+C` in your terminal to stop the webserver.

This is the approach that you normally first learn when starting out with PHP. There's nothing wrong with this approach, but, in general using a *front controller* is better so let's take a look at that.

Single Entry Point

Create a new folder and this time create a single file called `index.php` containing:

index.php

```
<?php

echo 'You are looking at: ' . $_SERVER['REQUEST_URI'];
```

Then, in the terminal `cd` to this new directory. This time start the PHP built in webserver with the name of the file we want to use as the application entry point. Like so:

```
$ php -S localhost:8080 index.php
```

Again open your browser and visit `http://localhost:8080/page1` and you will see `You are looking at: /page1`, and again go to`http://localhost:8080/page2` and you will see `You are looking at: /page2`.

So, as you can see, anything you type after the `http://localhost:8080` is redirected to the `index.php` file, and you can use the `$_SERVER` superglobal to get the actual URI requested.

The Simplest Front Controller in the World

Next, let's modify the `index.php` file to look like this:

index.php

```
<?php

switch ($_SERVER['REQUEST_URI']) {
    case '/page1':
        echo 'You are viewing page 1';
        break;

    case '/page2':
        echo 'You are viewing page 2';
        break;

    default:
        header('HTTP/1.0 404 Not Found');

        echo '<html>'
            . '<head><title>404 Not Found</title></head>'
            . '<body><h1>404 Not Found</h1></body>'
            . '</html>';
}
```

Now if we go to our browser and go to`http://localhost:8080/page1` or`http://localhost:8080/page2` then they work as expected. Also, going to `http://localhost:8080/anything-else` now shows a 404 message.

Obviously this is a pretty pointless and limiting front controller, but hopefully you now understand the theory behind it.

Stop the webserver again by pressing `CTRL+C` in the terminal.

Front Controllers using Apache

In order to use a front controller with Apache you need to tell it where to find the PHP script to be used for the application entry point. As of Apache version `2.2.16`, you can simply do this by adding the following to the `.htaccess` file in your document root:

.htaccess

```
1 FallbackResource /index.php
```

Standards

As a language gets more powerful it often allows many ways and approaches to achieve the same thing. Each person then has their own preferences of how they personally like to do things.

On one hand this is great: it allows programmers to be expressive and to write their code in the way which best fits how they think, lay it out in they way which looks the most aesthetically pleasing to them, and structure it in the way which they find easiest to navigate.

On the other hand this becomes a total nightmare: when you are working with several libraries, all written by different programmers who each have their own way of doing all things. You have to learn each of the different approaches to efficiently navigate and understand each author's code. Also, it may make it tricky for some of the libraries to happily interact with each other.

For the reasons just mentioned, programmers get together in groups and create *standards*, these are a nice middle ground which everyone is *mostly happy* with. You'll often find that you won't agree with everything defined in a standard, but by putting that to one side and accepting it you reap the benefits of having your code being much more consistent with all the other users of the standard's code - as well as any tools which have been built to work with that standard.

PHP-FIG

Introducing PHP-FIG! PHP-FIG or the *PHP Framework Interoperability Group* is a group built up of various key people in the PHP community who have got together and started to build some standards for using PHP. Many PHP software projects have now adopted or are adopting many of these standards. I fully recommend you do the same!

At the time of writing this there are 5 published standards:

Standard	Description
PSR-0	Autoloading Standard
PSR-1	Basic Coding Standard
PSR-2	Coding Style Guide (implies PSR-1)
PSR-3	Logger Interface
PSR-4	Improved Autoloading (an extension to PSR-0)

And a 3 more in discussion:

Standard	Description
PSR-5	PHPDoc
PSR-6	Caching Interface
PSR-7	HTTP Message Interfaces

In this book I will be using PSR-0 or PSR-4 for all code (except in some small examples) and PSR-2 for coding style. I will explain a bit more about these in the next few sections.

For more Information on PHP-FIG visit the website[8].

PHP The Right Way

PHP The Right Way[9] is not a standard as such, it's simply a website which lists lots of things about how PHP should be used if you are serious about writing good code. It contains lots of fantastic advice and I highly recommend studying it.

Docblocks

Docblocks are comments which contain *annotations* which can be added to your code to make it possible to generate documentation about your codebase automatically. One such document generation tool is phpDocumentor[10]. This can be fantastically useful if you are building a library for others to use, since you can easily generate great API documentation using it.

There is however another use for it; we've already talked about typehints which, by enforcing the types of function arguments, provide an extra level of *documentation* to people reading the code; it helps them to understand it quicker. They also help IDEs provide auto completion functionality while you're writing the code. This is great, but so far PHP has only gone half way - as I said earlier there are no *typehints* for function return values or for scalar types. Also, the type of a *variable* or *property* cannot be defined. Therefore, I've made it a habit to document these by using PHPdoc tags. I hope one day PHP will add more complete *typehinting*.

The format of docblocks are fairly standard now, but there's work to fully standardise it with PSR-5.

Here's a little example of how I'll be using docblocks in the code in this book:

[8] http://www.php-fig.org/
[9] http://www.phptherightway.com/
[10] http://www.phpdoc.org/

Using docblocks to annotate types

```php
<?php

class Example
{
    /** @var string */
    private $name;

    /** @var Email */
    private $email;

    /** @param string $name */
    public function addCustomer($name, Email $email)
    {
        // ...
    }

    /** @return Customer[] */
    public function getCustomers()
    {
        // ...
    }
}
```

Docblocks allow you to add much more detail than I've shown here. You can give descriptions and details for the *file*, the *class*, any *variables*, *properties* or *parameters*, etc. But since I don't want to generate an API document for this codebase, I'm only using it to specify the types which cannot be specified directly in PHP.

The Autoloader

You have may have never created a PHP *autoloader*, you may have never ever heard of one, but if you've every build an application using a recent PHP framework you have probably used one. However, if you're not aware of the autoloader and you're including all your different classes and functions by using PHP's `require`, `require_once`, `include` and `include_once` statements, then you need to STOP and read this section now!

The autoloader is a system in PHP where you can create a callback function that will be called if you try to use a class which has not yet been defined. This callback receives the name of the class trying to be used as a parameter. The function can then use the name to lookup and *require* the file needed to provide the class definition.

In order to use an autoloader callback you can either define a function called `__autoload` which will provide the autoloading logic. Or, you can use the more recent and more flexible `spl_autoload_register` function to register your autoloading function.

Now you know what an autoloader is there's some good news: there's no actual need to write the autoloader function yourself, there's a wonderful tool called Composer which can take care of that for you. However, if you do want to look into autoloading in more detail you can read about it in the manual[11].

PSR-0 - Autoloading Standard

Before talking about Composer I'd like to first introduce PSR-0. PSR-0 is a standard which was designed to make it easy to find the files where given classes are defined.

The basic rules of PSR-0 are as follows:

- There is exactly one class defined per file.
- The file name is the same name (and case) as the name of the class defined inside it, with `.php` appended to it.
- The file exists in a directory structure which fully matches the namespace in which the class is defined.

The full PSR-0 specification can be read on the PHP-FIG website[12].

Example

A file located at `/home/tom/projects/AutoloadExample/src/MyApp/Entity/Contact.php` would contain the following class definition:

PSR-0 compliant class

```
1  <?php
2
3  namespace MyApp\Entity;
4
5  class Contact
6  {
7      // ...
8  }
```

Whatever comes before the root namespace in the file path (in this case `/home/tom/projects/AutoloadExample/src`) is not important, so long as the FQCN is mirrored in the folder structure up to the class name.

[11] http://php.net/manual/en/language.oop5.autoload.php
[12] http://www.php-fig.org/psr/psr-0/

PSR-4 - Improved Autoloading Standard

PSR-4 improves on the PSR-0 standard. It removes some old, obsolete features and allows a namespace *prefix* to be defined. If your whole application exists under a single namespace this remove the need of having a directory level for that namespace.

Using the previous PSR-0 example: if the prefix `MyApp` is chosen then the `Contact` class definition can remain exactly the same, but the file is instead stored in `/home/tom/projects/AutoloadExample/src/Entity/Conta`

Composer

Composer is a dependency manager for PHP. It allows you to specify all the libraries and tools that your PHP project depends on in a simple JSON file. It will then fetch the correct versions of those dependencies (and all their dependencies) into your project.

This means that it's easy to distribute your project without including its 3rd party dependencies. While making it very easy for users or developers working on the project to easily install them themselves. It also makes it easy to quickly update to newer versions of dependencies.

Composer installs the dependencies locally to the project in a directory called `vendor`, rather than installing them globally onto the system. This is a definite plus, as it means you can run many projects on the same system, all working with different versions of their dependencies and without getting in to a mess.

Now this is all very interesting, but you might have no intention of using any external libraries or tools with your new project. So is Composer still useful?

The answer is most definitely yes:

- Firstly, there are lots of great development tools which can be installed via Composer. You should be using these tool even if you don't intend on using 3rd party libraries.
- Secondly, you may not intend on using 3rd party libraries but if you start off using Composer from the beginning, you can always change your mind and add a dependency very easily later on.
- Thirdly, Composer provides a nice and easy to set up autoloader for PHP. By simply adding a few lines of JSON to your project, your autoloader is set up and ready to go.

While there are plenty of people out there who will have a good reason not to use Composer, in my opinion if you don't have one, then you should definitely be using it in your projects.

So, that's a little intro on what Composer can do for you. You can find out about all its features and settings in the documentation[13] but, to save you the hassle of reading it all now let's have a look at a little example of the basics.

[13] https://getcomposer.org/

Composer Example

Installing

The various installation options for Composer can be found on the website[14].

You can either install a copy locally to your project which means your use it by running:

```
$ ./composer.phar
```

Or you can install it globally on your system and rename it to `composer`. I have done it this way so on my system I run Composer by simply typing:

```
$ composer
```

If you have installed it differently from me you will need to adjust my instructions accordingly.

Setting up the Autoloader

To start off, create a new project directory and `cd` into it:

```
$ mkdir ComposerExample
$ cd ComposerExample
```

If you want to use Composer locally you'll want to install it inside this directory now. However, if you have installed it global already then you can just carry on.

Next up, create a file in the project folder called `composer.json` and add the following content:

composer.json

```
{
    "autoload": {
        "psr-0": {
            "ComposerExample\\": "./src"
        }
    }
}
```

What we have told Composer to do here is set up its autoloader, to locate any classes in the `ComposerExample` namespace, by using the PSR-0 file structure inside a directory in our project called `src`.

Next, tell Composer to apply these settings with the following command (remember you will need to adjust it if you have installed Composer locally):

[14] https://getcomposer.org/

```
$ composer install
```

Once it has finished, you will notice it has created a new directory called `vendor` and another file called `composer.lock`. If you're are using a *source control system* like *git* (and you really should be!), then you should instruct it to ignore the `vendor` directory from the repository. The `composer.lock` file should be added to the repository though.

Next up, create a directory structure for our PSR-0 classes to go in:

```
$ mkdir -p src/ComposerExample
```

Then create a file called `src/ComposerExample/HelloApplication.php` with the following content:

src/ComposerExample/HelloApplication.php

```php
<?php

namespace ComposerExample;

class HelloApplication
{
    public function run()
    {
        echo "Hello beautiful World!\n";
    }
}
```

Finally, create a file called `run.php` containing:

run.php

```php
<?php

// Load up Composer's autoloader
require_once __DIR__ . '/vendor/autoload.php';

// This class will loaded automatically
$app = new \ComposerExample\HelloApplication();
$app->run();
```

Now to see it work run:

```
$ php run.php
```

Ta da!

Composer also supports PSR-4. To use it instead simply use `psr-4` in the `composer.json`. When doing this everything in the `src` folder will have the prefix (`ComposerExample`) applied to the namespace so the `ComposerExample` directory level would have to be removed.

Adding a Dependency

Next let's dress it up a bit using a 3rd party library. I had a little hunt around for something interesting to try and found Maxime Bouroumeau-Fuseau's ConsoleKit library.

First up, let's add it to the project as a dependency by updating our `composer.json` file to contain the following:

composer.json

```
{
    "require": {
        "maximebf/consolekit": ">=1.0.0"
    },
    "autoload": {
        "psr-0": {
            "ComposerExample\\": "./src"
        }
    }
}
```

Then we tell Composer to download its new dependency by running:

```
$ composer update
```

This should download the ConsoleKit package, which will now be ready to use. Let's update our `ComposerExample\HelloApplication` class to look like this:

src/ComposerExample/HelloApplication.php

```php
<?php

namespace ComposerExample;

use ConsoleKit\Console;

class HelloApplication extends Console
{
    public function run()
    {
        $console = new Console();
        $console->addCommand('ComposerExample\\HelloCommand');
        $console->run();
    }
}
```

And let's add a new class called `ComposerExample\HelloCommand` like so:

src/ComposerExample/HelloCommand.php

```php
<?php

namespace ComposerExample;

use ConsoleKit\Command;
use ConsoleKit\Colors;

class HelloCommand extends Command
{
    public function execute(array $args, array $options = array())
    {
        $this->writeln('Hello green World!', Colors::GREEN);
    }
}
```

Now to try to run it:

```
$ php run.php hello
```

There we have it, we've simply added a dependency to our app and made use of it. Composer has done all the hard work of downloading it and setting up the autoloader required to find it.

Adding Development Tools

Composer's `require` section lets you define the requirements your project needs to run. It also has a `require-dev` section which is for dependencies which you want to use for development only - testing tools for example.

CodeSniffer[15] is a tool which checks that your code follows a given coding style, lets add it to our project. To use it update the `composer.json` file to include the CodeSniffer development dependency:

composer.json

```
{
    "require": {
        "maximebf/consolekit": ">=1.0.0"
    },
    "require-dev": {
        "squizlabs/php_codesniffer": "1.*"
    },
    "autoload": {
        "psr-0": {
            "ComposerExample\\": "./src"
        }
    }
}
```

Once again, tell Composer to update its dependencies by running:

```
$ composer update
```

After it has finished, CodeSniffer is ready to be used. When Composer installs any tools it will install the executable files in a local directory, by default this directory is `vendor/bin`.

We can now run CodeSniffer by with the following command:

```
$ vendor/bin/phpcs --standard=psr2 src
```

If all the code in our `src` directory conforms to the PSR-2 coding style then CodeSniffer should have completed without and errors.

To make life easier, your can add `vendor/bin` to your operating system's `PATH` variable so you can execute your tools more easily. On Linux you do this by adding the following to your `.bashrc` file in your home directory:

```
PATH=./vendor/bin:$PATH
```

After you have done this you should be able to simple run:

[15] https://github.com/squizlabs/PHP_CodeSniffer

```
$ phpcs --standard=psr2 src
```

Keeping Logic and Display Code Separate

PHP lets you easily mix text output (usually HTML) with your logic. This makes PHP a really useful and powerful web templating language, but it also makes it very easy to write hideous code which mixes application logic in with the HTML output like so:

Mixing logic and display code

```
<?php

$repository = new CustomerRepository();

?>

<h1>List Customers</h1>

<?php

function escape($string)
{
    return htmlentities($string);
}

if ($_POST['search']) {
    try {
        $customers = $repository->getMatching($_POST['search']);
    } catch (LoadingException $e) {
        die('The was an error');
    }
}

if (count($customers)) { ?>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Email Address</th>
                <th>Phone Number</th>
            </tr>
        </thead>
```

```
        <tbody>
            <?php foreach ($customers as $customer) {
                echo '<tr>';
                echo '<td>' . escape($customer->name) . '</td>';
                $customer->printEmail();
                ?>
                    <td><?php echo escape($customer->phone); ?></td>
                </tr>
            <?php } ?>
        </tbody>
    </table>
<?php } else {  ?>
    <p>No customers found.</p>
<?php } ?>
```

I think we can all agree that is pretty ugly, but if it's left to get out of control it can get a lot uglier!

The solution to this is to not mix your HTML and PHP code together and instead maintain a clear separation between the two. This also allows designers to work on the user interface without have to understand the codebase.

There are templating libraries for PHP such as Twig[16] and Smarty[17] which introduce their own template tags for using in your HTML templates. You can also use PHP itself to do the templating, but if you do you really should maintain the discipline of keeping logic and view code separate; it may even be worth using different file extensions to keep it clear - `.php` for logic and `.phtml` for HTML templates is often used.

The choice between using PHP or a dedicated templating library can often be down to who is going to have access to modify the view templates. If your designers are in house, trustworthy and trained, then using PHP can be the easiest approach. However, if you are out sourcing the design work to people you trust less, then using a templating engine means they cannot compromise the security of the application by adding bad PHP code into the templates.

Coding Style

Coding style is simply the way you layout and format your code. In the previous Standards section I talked about how they were introduced to maintain a consistent approach to using a programming language between many developers. Using a consistent coding style is one element of this.

Coding style standards define things like:

- How many spaces should be used to indent code.

[16] http://twig.sensiolabs.org/
[17] http://www.smarty.net/

- If the opening brace for a function's body goes on the same line as the function definition, or on the line after.
- If variables be named using `camelCase` or `snake_case`
- etc.

As with all standards, it's unlikely that you'll find one which you agree with every bit of. Even so, rather that creating your own perfect one which no one else uses, you should use a well used one you like mostly.

At the moment the best one to use for PHP in my opinion is the one defined by PSR-2 as many people have adopted it. All application code I present in this book will follow the PSR-2 standard - with a couple of exceptions:

1. Unit Tests

When writing unit tests I follow PSR-2 apart from 2 elements - both regarding the method names for the tests.

- Firstly, instead of using `camelCase` for test method names I use `snake_case`. This is because the method names are sentences, and with `snake_case` it's easier to separate the words visually when reading it.
- Secondly I'll omit the `public` access specifier as it's the default in PHP and keeps the line shorter with long test method names.

Unit test coding style example

```
1  <?php
2
3  namespace spec;
4
5  use PhpSpec\ObjectBehavior;
6
7  class ExampleSpec extends ObjectBehavior
8  {
9      function it_adds_2_numbers_together()
10     {
11         $this->add(5, 2)->shouldReturn(7);
12     }
13 }
```

2. Template Code

When writing template code using PHP I prefer to keep it looking as close to HTML as possible. I try to keep the code inside PHP tags to single expressions and I use the `foreach :/endforeach`, `if :/endif`, etc. style of code blocks instead of using *braces* as I think they are easier to match up in this context. Here's an example:

Neat display template code example

```
<h1>List Customers</h1>

<?php if (count($customers)) : ?>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Email Address</th>
                <th>Phone Number</th>
            </tr>
        </thead>
        <tbody>
            <?php foreach ($customers as $customer) : ?>
                <tr>
                    <td><?php echo escape($customer->name); ?></td>
                    <td><?php echo escape($customer->email); ?></td>
                    <td><?php echo escape($customer->phone); ?></td>
                </tr>
            <?php endforeach; ?>
        </tbody>
    </table>
<?php else : ?>
    <p>No customers found.</p>
<?php endif; ?>
```

Methodologies, Techniques and Tools

In this chapter I want to introduce the methodologies, techniques and tools incorporated in this book. None of these are exclusive to PHP and therefore they are worth learning about regardless of which language you're developing in. Also, none of these are brand new to PHP - in fact any serious PHP development company will be using at least some, if not all of these already. While some of these are new and have been developed in recent years, many of them have been developed over the last one, two or more decades.

As in the previous chapter, I don't want to go into any real depth with any of these topics. I just want to give a short introduction or primer on each subject. This should provide enough of an understanding to work through this book, but as always, I do encourage you dig deeper into each of the subjects yourself. I will provide references for each topic of good places to start looking when you want to learn more.

Object Oriented Programming (OOP)

Sometimes I think there is a misconception that if you use classes in your code then you are doing OOP. This is not the case. OOP is an approach to modelling which involves grouping your business model down into objects, then grouping the related data and behaviours (methods) of these objects in your code as *classes*. This style of programming can even be done in languages which have no concepts of classes, and classes can be used ways which really don't represent good OO code.

This is not a book on OO design but we will be using it extensively. I will be explaining my choices for doing the things I do and as a result, if you're coming in cold to the subject you will probably get a good feel for OOP from this book alone. Even so I think this is a subject which you should study in more detail. If you do already have some experience of OOP, but you're not an expert, then I hope you will really have a lot to gain from this book.

Before moving on I just want to quickly cover a couple of OO topics:

Encapsulation

Objects consist of *state* (their *properties*) and a *public interface* (all **public** *methods* and *properties*). Generally there are rules as to what are the valid values for any object's *state*. Encapsulation is making state private so that it can only be changed via the *public interface's* methods.

The point I want to make here is: that you should design your public methods so that there is no way that the object can be put into an invalid state.

Let's look at a couple of examples:

Example 1

An object to represent an email address should not be able to contain a value which could not be a valid email address. In this case it would make sense to throw an exception if the email address provided does not look like a valid email address:

Well defined email address object example

```php
<?php

class EmailAddress
{
    /** @var string */
    private $address;

    /** @param string $address */
    public function __construct($address)
    {
        if (strpos($address, '@') === false) {
            throw new InvalidArgumentException(
                'Email addresses must contain an @ symbol'
            );
        }

        $this->address = $address;
    }

    public function __toString()
    {
        return $this->address;
    }
}
```

In the example above we chose the simple rule that *"anything with an @ symbol in it could be an email address"*. Obviously in production code this would need to be more well defined.

If you study the code above you will find that there is no way that you can create an instance of `EmailAddress` with an email address which does not contain an `@` symbol. This is good design!

Example 2

Now consider implementing a collection of words which maintains a count of how many words it contains. This collection class will have 2 properties: *the list of words* and *the count*. In order to add words to the collection we must add the word to the list and increment the count. First let's try this with 2 separate methods:

Bad state consistency example

```php
<?php

class WordCollection
{
    /** @var string[] */
    private $words = [];

    /** @var int */
    private $count = 0;

    /** @param string $word */
    public function addWord($word)
    {
        $this->words[] = $word;
    }

    public function incrementCounter()
    {
        $this->count++;
    }

    /** @return string[] */
    public getWords()
    {
        return $this->words;
    }

    /** @return int */
    public getNumberOfWords()
    {
        return $this->count;
    }
}
```

Now you can probably see what's wrong with that straight away, but let me explain for completeness. Take the following code:

```php
<?php

$collection = new WordCollection();

$collection->addWord('hello');
$collection->incrementCounter();
```

It looks OK and `$collection` is left in a valid state at the end right? But is it always in a valid state? Look again:

```php
<?php

$collection = new WordCollection();

// words in list = 0
// counter = 0

$collection->addWord('hello');

// words in list = 1
// counter = 0
// Oh dear!

$collection->incrementCounter();

// words in list = 1
// counter = 1
```

There's a point in the middle where the state of the object is invalid, we do fix it in the next line of code but what if a developer forgot to increment the counter? It could lead to a nasty bug!

The solution is simple - design the class so it can never be put into an invalid state:

Good state consistency example

```php
<?php

class WordCollection
{
    /** @var string[] */
    private $words = [];

    /** @var int */
```

```
9       private $count = 0;
10
11      /** @param string $word */
12      public function addWord($word)
13      {
14          $this->words[] = $word;
15
16          // increment the counter when the word is added
17          $this->count++;
18      }
19
20      /** @return string[] */
21      public getWords()
22      {
23          return $this->words;
24      }
25
26      /** @return int */
27      public getNumberOfWords()
28      {
29          return $this->count;
30      }
31  }
```

SORTED!

Inheritance vs. Composition

Inheritance is a very useful and powerful tool in OOP but it's often misused. The most common misuse of inheritance is using it to bring common functionality into 2 unrelated classes.

An example could be:

A bird and an aeroplane both fly, so it might seem reasonable to inherit both a `Bird` class and an `Aeroplane` class from a `FlyingThing` superclass:

Misused Inheritance

```php
<?php

abstract class FlyingThing
{
    public function startFlying()
    {
        // ...
    }

    // ...
}

class Bird extends FlyingThing
{
}

class Aeroplane extends FlyingThing
{
}
```

There are a few reasons why this is a problem:

Firstly, we only have single inheritance in PHP which means we can't extend from more than one super class. Birds can communicate with each other and aeroplanes (well the pilots) can communicate with each other too. If we wanted to add a `CommunicatingThing` class then we couldn't inherit from it as well as `FlyingThing` (without creating a horrible `CommunicatingFlyingThing` class instead).

Next up, birds start flying by *flapping their wings* whereas an aeroplane is *propelled by its engines*. Now we have 2 different types of `FlyingThing`. The same goes for communicating: birds do it by *tweeting* and aeroplanes do it by *radio*.

Finally, we've categorised birds and aeroplanes, which are 2 very different things, together as *flying things* and *communicating things* - which we've had to make up. A much more sensible category for an aeroplane might be a *vehicle*, and for a bird maybe an *animal*.

So how to fix it? So far we've been talking about *is a* relationships; *a bird **is a** flying thing*. Inheritance is all about *is a* relationships, but we also have *has a* relationships. What might be better would be if we said *"a bird **has a** flight system"* or *"an aeroplane **has a** communication system"*. When talking about **has a** relationships we are talking about *composition*. With this in mind Let's take a look at a better example of how to model this:

Using composition

```php
<?php

class FlightSystem
{
    public function startFlying()
    {
        // ...
    }

    // ...
}

class CommunicationSystem
{
    /** @param string $message */
    public function sendMessage($message)
    {
        // ...
    }

    // ...
}

class Bird
{
    /** @var FlightSystem */
    private $flightSystem;

    /** @var CommunicationSystem */
    private $communicationSystem;

    public function __construct()
    {
        $this->flightSystem = new FlightSystem();
        $this->communicationSystem = new CommunicationSystem();
    }

    public function startFlying()
    {
        $this->flightSystem->startFlying();
    }
```

```
42
43      /** @param string $message */
44      public function sendMessage($message)
45      {
46          $this->communicationSystem->setMessage($message);
47      }
48  }
49
50
51  class Aeroplane
52  {
53      /** @var FlightSystem */
54      private $flightSystem;
55
56      /** @var CommunicationSystem */
57      private $communicationSystem;
58
59      public function __construct()
60      {
61          $this->flightSystem = new FlightSystem();
62          $this->communicationSystem = new CommunicationSystem();
63      }
64
65      public function startFlying()
66      {
67          $this->flightSystem->startFlying();
68      }
69
70      /** @param string $message */
71      public function sendMessage($message)
72      {
73          $this->communicationSystem->setMessage($message);
74      }
75  }
```

What's more, once we start to realise that a bird's flight system is very different from an aeroplane's, then we can start having different flight systems which can still be instructed to start flying in the same way by making use of an *interface*:

Flexible flight systems

```
interface FlightSystem
{
    public function startFlying();
}

class FlappingFlightSystem implements FlightSystem
{
    public function startFlying()
    {
        // ...
    }
}

class PropelledFlightSystem implements FlightSystem
{
    public function startFlying()
    {
        // ...
    }
}

class Bird
{
    // ...

    public function __construct()
    {
        $this->flightSystem = new FlappingFlightSystem();
        // ...
    }

    // ...
}

class Aeroplane
{
    // ...

    public function __construct()
    {
        $this->flightSystem = new PropelledFlightSystem();
```

```
42          // ...
43      }
44
45      // ...
46  }
```

The general rule here is: determine if you are really modelling an **is a** relationship or a **has a** relationship. Don't bend your model to fit your code, rather make your code fit the model. When done properly you should find you actually use inheritance pretty rarely.

Design Patterns

Design Patterns are tried and tested solutions to various problems programmers often face. There's several common design patterns which are very well known, these are called things like the *visitor pattern* and the *strategy pattern*.

The de facto resource on this subject is the book titled *Design Patterns: Elements of Reusable Object-Oriented Software* by the Gang of Four[ˆthe-gang-of-four]. Robert C. Martin says this book is *"probably the most important book in software engineering which has been written in the last 30 years"*. This book is definitely a *must read* for all programmers. However, you can also find all the common design patterns listed and explained on Wikipedia[18].

If you'd prefer to learn about Design Patterns from a PHP point of view then Brandon Savage has written a book on the subject called Practical Design Patterns in PHP[19].

I'm not going to go into any more details on design patterns here, I just wanted you to know what they are for now. As we work though this book we will be using various design patterns, and as they come up I shall point out which one we are using and explain the choice to do so.

Value Objects & Immutability

When we say something is *immutable* we mean that its value is fixed and can not be changed. This means a class is either *mutable* or *immutable* depending on whether its public interface contains methods which change its internal state. Let's see an example of each:

[18] http://en.wikipedia.org/wiki/Software_design_pattern#Classification_and_list

[19] http://practicaldesignpatternsinphp.com/

Mutable email address class

```php
<?php

class MutableEmailAddress
{
    /** @var string */
    private $address;

    /** @var string */
    public function __construct($address)
    {
        $this->address = $address;
    }

    /** @var string */
    public function setAddress($address)
    {
        $this->address = $address;
    }

    public function __toString()
    {
        return $this->address;
    }
}
```

Immutable email address class

```php
<?php

class ImmutableEmailAddress
{
    /** @var string */
    private $address;

    /** @var string */
    public function __construct($address)
    {
        $this->address = $address;
    }

```

```
14      public function __toString()
15      {
16          return $this->address;
17      }
18  }
```

We call *immutable* classes *value objects*. This is because an instance of a given class represents a value and only that value. In the same way that we can say `5` will always be `5`, we can say that an `ImmutableEmailAddress` object containing the email address `bill@microsoft.com` will always contain the email address `bill@microsoft.com` - therefore this object represents the *value* of that email address.

Value objects are used to make the type of things explicit and should be used to classify anything which can be classified:

An Example

Instead of using a *float* for temperate use a `Temperature` value object - even if it simply contains a single *float* value. You might however also want to store the unit (Fahrenheit or Celsius) in the `Temperature` object - a value object can contain more than `1` scalar value. Also, the unit of temperature is classifiable itself, therefore make that a value object too.

Your value objects should only be able to be created with valid values, we don't want anyone to be able to create a temperature of *"30 degrees **hotness**"*. This should be enforced in the class definition!

For completeness, let's look a good implementation of what we've just discussed:

Temperature; a good use of value objects

```
1   <?php
2
3   class TemperatureUnit
4   {
5       const CELSIUS    = 'c';
6       const FAHRENHEIT = 'f';
7
8       /** @var string */
9       private $value;
10
11      /** @param string $value */
12      public function __construct($value)
13      {
14          $this->assertValueIsValid($value);
15
```

```
16          $this->value = $value;
17      }
18
19      /** @return string */
20      public function getValue()
21      {
22          return $this->value;
23      }
24
25      private function assertValueIsValid($value)
26      {
27          if (!in_array($value, [self::CELSIUS, self::FAHRENHEIT])) {
28              throw new InvalidArgumentException(
29                  'A temperature unit must be celsius or fahrenheit'
30              );
31          }
32      }
33  }
34
35  class Temperature
36  {
37      /** @var float */
38      private $degrees;
39
40      /** @var TemperatureUnit */
41      private $unit;
42
43      /** @param float $degrees */
44      public function __construct($degrees, TemperatureUnit $unit)
45      {
46          $this->degrees = $degrees;
47          $this->unit    = $unit;
48      }
49
50      /** @return float */
51      public function getDegrees()
52      {
53          return $this->degrees;
54      }
55
56      /** @return TemperatureUnit */
57      public function getUnit()
```

```
58      {
59          return $this->unit;
60      }
61  }
```

The benefits

There are several benefits to using value objects:

Confidence

With a value object you know that it can't change unexpectedly. Using our `MutableEmailAddress` class from earlier consider the following code:

Unexpected change of value

```
1   <?php
2
3   function sendEmail(MutableEmailAddress $address)
4   {
5       mail($address->getValue(), 'Hello', '...');
6       $address->setValue('steve@apple.com');
7   }
8
9   function logEmailSent(MutableEmailAddress $address)
10  {
11      echo "A message was sent to " . $address->getValue();
12  }
13
14  $address = new MutableEmailAddress('bill@microsoft.com');
15
16  sendEmail($address);    // send email to bill@microsoft.com
17  logEmailSent($address); // but logs "A message was sent to steve@apple.com"
```

Looking at just the last 3 lines of code in the example, you would expect the message would get sent to Bill and the log message would reflect that, but it doesn't!

If we had used an `ImmutableEmailAddress` object instead then the `$address->setValue('steve@apple.com');` line of code would not have been allowed and would have caused an error. Therefore, we could be certain that our last 3 lines of code would work as expected.

Consistency

If we create a function that takes a `Temperature` as an argument we can be sure that the value we get is a valid `Temperature`. We don't need to write any defensive code to check that the unit is valid or that degrees are specified as a floating point value, because all that has been taken care of in the class definitions of our value objects.

Documentation

When you open a file and look at the code, it's much easier to know the type of a parameter that was typehinted in the method argument list. This is really a benefit of typehints rather than of value objects. However, by using value objects abundantly you will have more types to typehint for.

Entities

Entities make up an important part of the business model of most applications. Unlike value objects they are *mutable*. They are also the objects which are often persisted and make up the ongoing state of the application.

Entities have *identities*. They can have all their properties changed but they continue to maintain the same identity through out their lifetime. A simple analogy of an entity object is a person, they can change their name but they are still the same person. This means that the equality of entities determined by the equality of their *identities* - unlike *value objects* the equality of their other properties are not important when determining equality.

Here's a simple example of what a simple entity class make look like:

Example entity class

```php
<?php

class Customer
{
    /** @var CustomerId */
    private $id;

    /** @var PersonName */
    private $name;

    /** @var EmailAddress */
    private $email;

    public function __construct(
        CustomerId $id,
```

```
16          PersonName $name,
17          EmailAddress $email
18      ) {
19          $this->id    = $id;
20          $this->name  = $name;
21          $this->email = $email;
22      }
23
24      public function changeName(PersonName $newName)
25      {
26          $this->name = $newName;
27      }
28
29      public function changeEmail(EmailAddress $newAddress)
30      {
31          $this->email = $newAddress;
32      }
33
34      /** @return CustomerId */
35      public function getId()
36      {
37          return $this->id;
38      }
39
40      /** @return Name */
41      public function getName()
42      {
43          return $this->name;
44      }
45
46      /** @return EmailAddress */
47      public function getEmail()
48      {
49          return $this->email;
50      }
51  }
```

Dependency Injection (DI) & Inversion of Control (IoC)

Dependency Injection is simply explained by saying: if you have a class that relies on another class, rather than letting it create that other class internally, *inject* the dependency into it.

Example **without** dependency injection:

No dependency injection example

```php
<?php

class Foo
{
    public function doFoo()
    {
        // ...
    }
}

class Bar
{
    /** @var Foo */
    private $fooer;

    public function __construct()
    {
        /*
         * Bar is in control of creating a Foo.
         *
         * As a result bar can only ever use a Foo and never a subtype of
         * Foo.
         *
         * This is known as "tight coupling" because Bar can not exist without
         * Foo
         */
        $this->fooer = new Foo();
    }

    public function doBar()
    {
        $this->fooer->doFoo();
    }
}
```

Example **with** dependency injection:

Dependency injection example

```php
<?php

class Foo
{
    public function doFoo()
    {
        // ...
    }
}

class Bar
{
    /** @var Foo */
    private $fooer;

    /**
     * The Foo must be injected into the instances of Bar.
     *
     * It is also possible to inject a subtype of Foo.
     */
    public function __construct(Foo $fooer)
    {
        $this->fooer = $fooer
    }

    public function doBar()
    {
        $this->fooer->doFoo();
    }
}
```

That's it, *Dependency Injection* is as easy as that, but why do it?

Substituting Behaviour

By injecting dependencies you're promoting the development of flexible and reusable code. Since you can not only can you inject the dependent type but also any subtype. This makes it possible to extend your code without actually modifying it - by simply injecting a different subtype of a dependency.

Example

Take this simple email address printer:

Email printer using dependency injection

```php
<?php

class EmailAddress
{
    /** @var string */
    private $address;

    /** @param string $address */
    public function __construct($address)
    {
        $this->address = (string) $address;
    }

    public function __toString()
    {
        return $this->address;
    }
}

interface EmailAddressRenderer
{
    /** @return string */
    public function render(EmailAddress $email);
}

class PlainTextRenderer implements EmailAddressRenderer
{
    public function render(EmailAddress $email)
    {
        return (string) $email;
    }
}

class EmailAddressPrinter
{
    /** @var EmailAddressRenderer */
    private $renderer;

```

```
39      public function __construct(EmailAddressRenderer $renderer)
40      {
41          $this->renderer = $renderer;
42      }
43
44      public function printAddress(EmailAddress $address)
45      {
46          echo $this->renderer->render($address) . "\n";
47      }
48  }
```

To use our class we'd simply write something like this:

Email printer example use

```
1  <?php
2
3  $address = new EmailAddress('bill@microsoft.com');
4
5  $printer = new EmailAddressPrinter(new PlainTextRenderer());
6
7  $printer->printAddress($address);
```

This works great! But then we're told we need to print out the email address on a web page as a `mailto` link. Since we've injected our `EmailAddressRenderer` into the printer rather than letting the printer create it, we can now inject anything which implements the `EmailAddressRenderer` interface.

Let's add a `HTMLRenderer`:

HTML email address renderer

```
1  <?php
2
3  class HTMLRenderer implements EmailAddressRenderer
4  {
5      public function render(EmailAddress $email)
6      {
7          return sprintf('<a href="mailto:%1$s">%1$s</a>', htmlentities($email));
8      }
9  }
```

Our code to make use of this would now look like this:

HTML email printer example use

```php
<?php

$address = new EmailAddress('bill@microsoft.com');

$printer = new EmailAddressPrinter(new HTMLRenderer());

$printer->printAddress($address);
```

Simples! Notice how in order to change the way it behaves we have not altered the original code in any way, instead we've simply extended it. Here we've written our code in such a way that even though the printer uses a renderer, it does not have control over how that renderer is created. Instead it is uses the one given. This is known as *inversion of control*.

Inversion Of Control

We've just seen an example of *inversion of control* so we know what it is. But again, what is it useful for?

Well…

- Using it creates extensible code.
- It removes the need for your business model to know about implementation details such as: how things are displayed or how messages are sent. Instead, it allows those details to be plugged in; this is often done using the Adapter[20] design pattern.
- It allows your code to be easily tested as small, independent units. This is possible through the use of *test doubles*.

Testing

Each class in your codebase may depend on on other classes, each of which may then depend on even more classes. As your codebase grows, testing anything becomes quite hard as you need create all the dependencies of the class you are trying to test. Also, each of these dependencies will have their own dependencies which will need to be created as well. In this situation tests get very complex and also fragile; if you were to change anything then lots of your tests could break, and it could take a long time to get them passing again.

By using *dependency injection*, *inversion of control* and *interfaces*, you get code which doesn't depend on fixed dependencies but rather on abstractions (the interface) of the dependency. When your code is written like this your dependencies are considered to be *loosely coupled*.

[20] http://en.wikipedia.org/wiki/Adapter_pattern

When your dependencies are *loosely coupled* you can create *test doubles*. These are very simple and *predictable* versions of a class's dependencies. These can be injected into the class instead of the real dependencies, allowing the class to be tested in isolation. We'll look at this in more detail in the next chapter.

Dependency Injection Containers (DIC) & Service Managers

These are tools which can manage the creation of objects and ensure that the required dependencies get injected into them on creation. I'm not going to talk about these here but will cover them in more detail when we need to use one.

All I will add now is that there are many DIC and *Service Manager* libraries available should you ever decide you need one.

The SOLID Principles

The SOLID Principles are the first 5 principles from a list compiled by Robert C. Martin of *Principles of Object Oriented Design*[21]. These 5 principles are:

Single Responsibility Principle (SRP)
: A class should only have a single responsibility - don't create classes which do a lot of different things.

Open Closed Principle (OCP)
: Your application should be *open for extension* but *closed for modification*. We saw an example of this in action in the Dependency Injection section.

Liskov Substitution Principle (LSP)
: This states objects can be replaced by subtypes of themselves and the program should still work. What this means is: when you extend a class you should make sure its public interface still behaves in the expected way, making it possible to use the class in place of its parent anywhere in the program.

Interface Segregation Principle (ISP)
: This principle states that if something has a dependency but only uses a subset of that dependency's public interface, then that subset should be made into a separate interface. The new interface then becomes the dependency.

Dependency Inversion Principle (DIP)
: Classes should not depend on other concrete classes, but rather on abstractions (i.e. *abstract classes* and *interfaces*). Again, we looked at this earlier on in the email printer example.

[21] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

More in-depth details can be found on Robert Martin's article: Principles of OOD[22].

If you stick to these principles you will create fantastically modular, extensible and testable code.

Functional Programming (FP)

History

PHP is naturally heavily biased towards the *imperative programming*[23] paradigm. Imperative programs are basically sequences of statements which are executed in order, changing the program state they run. The imperative style of programming has been the most popular style for several decades now.

There are however other paradigms, one of which is *functional programming*[24]. Functional Programming pre-dates Imperative Programming, but Imperative Programming still won the popularity race up until recently. Once it stopped being possible for computer CPUs to get any faster, the manufacturers decided to start adding more cores instead. So rather than trying to have a single thread of instructions executing faster and faster, we have multiple threads executing simultaneously.

The imperative style of programming is very well suited towards single threads of execution because statements always run in the correct order. With multiple threads however, one thread might run faster than another, and if they are both trying to read and change the same state then there's no guarantee that it will happen in the right order. To make this work using imperative programming you need to introduces *locks* to keep track of and synchronise how the threads are interacting with the shared state. This is tricky and is not fun to do!

The functional style of programming, makes this a lot simpler. You can tell the computer to execute 2 different functions in 2 different threads and be sure that they will behave as expected, and not conflict with each other. For this reason, functional programming and functional programming languages are gaining a lot of popularity at the moment. Also, many imperative languages are adding more and more functional style features.

So What is Functional Programming?

Functional programming has its roots in lambda calculus[25], and is based on the idea that whenever you call a given function with the same set of arguments, you will always get the same result. Some other aspects of functional programming are:

- Functions have no side effects - they never modify or read global/persistent state. These are considered *pure functions*.

[22] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
[23] http://en.wikipedia.org/wiki/Imperative_programming
[24] http://en.wikipedia.org/wiki/Functional_programming
[25] http://en.wikipedia.org/wiki/Lambda_calculus

- There is no assignment - once a variable has been initialised with a value it will always be that value.
- Functions can accept other functions as arguments and return new functions. These are known as *higher order functions*.

I'm not going to go much deeper into functional programming in this book, I only really want to show what *pure functions* are. I will be applying some functional programming knowledge during the building of our application but I'm not going to highlight it too much.

Pure Functions

As previously mentioned, *pure functions* are functions which have no side effects. Let's look at some examples of pure functions:

All these functions will always return the same value when given the same arguments. And they never alter persistent state:

Pure functions

```php
<?php
function add($a, $b)
{
    return $a + $b;
}

function fahrenheitToCelsius($fahrenheit)
{
    return ($fahrenheit - 32) / 1.8;
}

function applyTwiceAndAdd(callable $fn, $value)
{
    return $fn($value) + $fn($value);
}
```

The following functions are not pure. If called twice with the same arguments the result may not be the same:

Impure functions

```php
<?php
$counter = 0;

function incrementCounter()
{
    global $counter;

    $counter++;
}

function getCount()
{
    global $counter;

    return $counter;
}

function getCountAndIncrement()
{
    global $counter;

    return $counter++;
}

function threeTimesRand($max)
{
    return 3 * rand(0, $max);
}

function readFile($filename)
{
    return file_get_contents($filename);
}
```

Now some functional programming languages do not have assignment statements to allow the changing of the value of a variable - but PHP does! It is possible to make use of this inside a function, while still providing a functionally pure interface to it:

Pure function with local state

```php
<?php

function applyNTimesAndAdd(callable $fn, $n, $value)
{
    $total = 0;

    for ($count = 0; $count < $n; $count++) {
        $total += $fn($value);
    }

    return $total;
}
```

While `applyNTimesAndAdd` does have internal state, it is never persisted beyond each execution of the function. Therefore, this can still be considered a pure function - even though the implementation uses imperative code.

Now lets move this theory inside objects. Obviously we can have methods which are pure functions:

Methods as pure functions

```php
<?php

class RandomFunctions
{
    public function add($a, $b)
    {
        return $a + $b;
    }

    public function fahrenheitToCelsius($fahrenheit)
    {
        return ($fahrenheit - 32) / 1.8;
    }

    public function applyTwiceAndAdd(callable $fn, $value)
    {
        return $fn($value) + $fn($value);
    }
}
```

Of course this hasn't really achieved much other than grouping some functions together.

However, as soon as methods start to make use of properties, they can no longer be considered as pure functions. This is because the properties persist beyond method calls:

Impure methods

```php
<?php

class Counter
{
    private $counter = 0;

    public function incrementCounter()
    {
        $this->counter++;
    }

    public function getCount()
    {
        return $this->counter;
    }

    public function getCountAndIncrement()
    {
        return $this->counter++;
    }
}
```

Since the behaviour of the methods in the `Counter` class now depends on the `$counter` property, none of the methods in the class could be considered pure.

Now thinking back to the `applyNTimesAndAdd` function, can we do something similar in the context of an object? Take a look at this example and consider whether you think it behaves in a functional way:

```php
<?php

class ResultAdder
{
    private $fn;

    private $total;

    private $value;

    public function applyNTimesAndAdd(callable $fn, $n, $value)
    {
        $this->fn = $fn
        $this->total = 0;
        $this->value $value;

        for ($count = 0; $count < $n; $count++) {
            $this->applyFunction();
        }

        return $this->total;
    }

    private function applyFunction()
    {
        $fn = $this->fn;

        $fn($this->value);
    }
}
```

What do you think? If you study it carefully you will notice that: even though properties are being used inside the class, every time a method in the public interface is invoked, the properties are being reset. This means that the `applyNTimesAndAdd` method does indeed appear to be working in a functional way.

Classes like this one are often created by the use of the Replace Method with Method Object[26] refactoring.

[26] http://refactoring.com/catalog/replaceMethodWithMethodObject.html

Learn More About FP

Learning a functional programming language is not only fascinating, but it also gives you a whole new set of tools which you can apply in any language. Also, as we're get more and more cores in our computers, it's going to become increasingly important over the next few years! So, if you're not already familiar with functional programming I really recommend taking a look at it in the not so distant future - it will be a very rewarding experience!

There are many resources on functional programming which can be easily found. Two notable ones are:

Structure and Interpretation of Computer Programs (SICP)
: This is one of the most recommended text books covering functional programming. It was written by MIT professors Harold Abelson & Gerald Jay Sussman with Julie Sussman, and was formerly used there as a text book. It can also be read for free online[27].

Functional Programming in PHP[28]
: This is a much easier book to read, written by Simon Holywell. It introduces functional programming to PHP programmers, along with some useful functional libraries which are available for PHP.

Command Query Separation (CQS)

Command Query Separation is more or what the name implies:

Methods are classified as either *commands* or a *queries* (but not both). A *command* is a method which changes the state of the object. Whereas, a *query* is a method which returns a value from the object. Also, a *command* must not return a value, and a query must not change the object's state.

Let's take another quick look at the `Counter` class we made earlier:

Command Query Separation Example

```
1  <?php
2
3  class Counter
4  {
5      /** @var int */
6      private $counter = 0;
7
8      /**
9       * This method is a COMMAND since it updates the state
```

[27] http://mitpress.mit.edu/sicp/
[28] http://www.functionalphp.com/

```
10          * and doesn't return a value.
11          */
12         public function incrementCounter()
13         {
14             $this->counter++;
15         }
16
17         /**
18          * This method is a QUERY since it returns a value
19          * but does not alter the object's state.
20          *
21          * @return int
22          */
23         public function getCount()
24         {
25             return $this->counter;
26         }
27
28         /**
29          * This method alerts the state and returns a value so it
30          * is not a separate COMMAND or QUERY.
31          *
32          * Such methods should be avoided.
33          *
34          * @return int
35          */
36         public function getCountAndIncrement()
37         {
38             return $this->counter++;
39         }
40     }
```

CQS means we can repeatedly query our object (to make assertions, display, or what ever other reason), while being confident that we are not inadvertently making changes to its state in the process. Generally, with the objects inside your model this is a good approach to try to adhere to.

What might seem an exception to this rule is, methods working in a functional way. However, since the state they modify is not used again, it does not actually break this rule.

Naming

When writing code, you have to decide on the names of many things. These include *variables*, *classes* and *methods*. Choosing the names of these carefully, makes the difference between code which is

easily understandable, and which is completely indecipherable.

When choosing names, make them descriptive, so that the intent of your code is clear. If your code is well written and the names are chosen well, there should be no need for any comments in your code. Robert C. Martin goes as far as saying that choosing to adding a comment to your code, is accepting that you have failed as a programmer to solve the problem clearly. That does sound a bit harsh, but the underlying message is that your code should be self descriptive. It's OK to use comments when something cannot be made clear in code, but those situations should be very, very rare!

One very simple rule for naming is: *use nouns for variable and class names and use verbs for method names*. It's not quite as black and white as that, but it's a very good place to start.

For some more interesting talk on naming I thoroughly recommend having a read of Mathias Verraes' blog[29].

Make Method Names Describe Intent

When creating methods, make the intent of the method's purpose clear in terms of the business language.

Say for example: a customer moves house and needs to have their address updated in the system. What might seem like an obvious approach would be to add a `setAddress` method to the `Customer` class. This works but it's not as informative as it could be. A better method name might be 'moveHouse'.

So far so good, but what if the customer's address was entered into the system incorrectly and it just needs to be amended? Does it make sense to call `moveHouse` to update it? Of course not! So we add a `amendPostalAddress` method also:

Good method names

```
1  <?php
2
3  class Customer
4  {
5      /** @var string */
6      private $name;
7
8      /** @var EmailAddress */
9      private $emailAddress;
10
11     /** @var PostalAddress */
12     private $postalAddress;
13
```

[29] http://verraes.net/

```
14      /** @param string $name */
15      public function __construct(
16          $name,
17          EmailAddress $emailAddress
18          PostalAddress $postalAddress
19      ) {
20          $this->name          = $name;
21          $this->emailAddress  = $emailAddress;
22          $this->postalAddress = $postalAddress;
23      }
24
25      public function moveHouse(PostalAddress $newAddress)
26      {
27          $this->postalAddress = $newAddress;
28      }
29
30      public function amendPostalAddress(PostalAddress $amendedAddress)
31      {
32          $this->postalAddress = $amendedAddress;
33      }
34
35      /** @return string */
36      public function getName()
37      {
38          return $this->name;
39      }
40
41      /** @return EmailAddress */
42      public function getEmailAddress()
43      {
44          return $this->emailAddress;
45      }
46  }
```

This is definitely more descriptive, but it's essentially 2 methods which do the same thing - surely there's no point in that you might think. But there is! The 2 action are actually 2 separate tasks, by separating them we make it very clear to anyone reading the code that there are 2 different events which result in the change of a customer's address data. This also makes it possible to easily extend the system: say, for example, we are told that emails should be sent to the customer when their address changes:

- When they move house, we need to send them an email congratulating them on moving into a new home.

- When amending the address, we just want to notify them that their details have been updated.

Because we've separated the 2 actions, this is very easy to add on. For this example let's use the Decorator pattern[30][31]:

Emailing decorator

```php
<?php

class NotifyingCustomerDecorator extends Customer
{
    /** @var Customer */
    private $customer;

    /** @var Mailer */
    private $mailer;

    public function __construct(Customer $customer, Mailer $mailer)
    {
        $this->customer = $customer;
        $this->mailer   = $mailer;
    }

    public function moveHouse(PostalAddress $newAddress)
    {
        $this->customer->moveHouse($newAddress);

        $this->mailer->send(
            $this->customer,
            'Congratulations on moving into your new house!',
            '...'
        );
    }

    public function amendPostalAddress(PostalAddress $newAddress)
    {
        $this->customer->amendPostalAddress($newAddress);

        $this->mailer->send(
```

[30] http://en.wikipedia.org/wiki/Decorator_pattern

[31] The reason I chose to use the *Decorator pattern* here instead of just extending `Customer` was just to introduce the pattern. The actual reason why you would choose to use this pattern, would be if you wanted to wrap the `Customer` class when extensions which could be stacked on top of each other.

```
            $this->customer,
            'We have updated your details on our system',
            '...'
        );
    }

    /** @return string */
    public function getName()
    {
        return $this->customer->getName();
    }

    /** @return EmailAddress */
    public function getEmailAddress()
    {
        return $this->customer->getEmailAddress();
    }
}
```

Pretty neat huh? Once again we have easily extended the system to add new functionality, without modifying the existing code. Therefore, we are obeying the Open Closed Principle.

Refactoring

Refactoring is the process of restructuring your code without changing its external behaviour. The purpose of refactoring is to try to make the code easier to understand, manage and extend. As programmers, we should be refactoring all the time while we write our code.

While you can get quite a long way refactoring your code by just using common sense, all the common refactorings have been named and catalogued. To learn more about refactoring I don't think there's any better recommendation than to read Martin Fowler's book titled *Refactoring*.

Also, much like with design patterns, lots of information can be found about the different refactorings online. In fact, Martin Fowler also has website called http://refactoring.com/ which has a catalogue of refactorings[32] on it.

As I write this book I'll be refactoring all the example code as I go. Sadly you'll often not get to see this process as the book will just contain the finished, refactored result. That said you will see the code evolve throughout the book as functionality is added, and that will include some refactoring. I'll also cover it a bit more in the next chapter.

[32] http://refactoring.com/catalog/

Object Calisthenics

Object Calisthenics is an idea suggested by Jeff Bay in *The ThoughtWorks Anthology* book. It consists of 9 rules to help write better *Object Oriented* code.

These rules are:

Only One Level Of Indentation Per Method
: Code with multiple indents gets tricky to read and follow. Also, there's a greater chance of having to scroll the page to the right which is a nuisance. This can be avoided by *extracting methods* and using *guard clauses*.

Don't Use The ELSE Keyword
: There are some cases where `else` can be useful, but more often than not you can find a neater way. This again, produces code which is easier to read.

Wrap All Primitives And Strings
: This pretty much means: use value objects instead of *scalar* types in PHP. As I've already said, this is particularly useful in PHP when combined with *typehinting*. Always question the use of a scalar property in a domain object.

First Class Collections
: Rather than having arrays properties in your classes, create a collection object. Quite often there are actions which you want to apply across a collection, these actions really belong as part of the collection rather than the containing class. Doing this also makes collections reusable.

One Dot Per Line
: In PHP this should really be *Two Arrows Per Line* since we use `->` instead of `.`, and the `$this->` must be used inside methods to access things in class scope. What this rule actually means is - don't call methods on objects returned from other methods - creating a chain of method calls like so: `$this->getX()->doY()->doZ()`. This rule ties in very tightly with the Law of Demeter[33].

Don't Abbreviate
: Use descriptive names for everything, so other people (or yourself in 6 months time) reading the code can clearly understand its intent. This ties in with the previous Naming section.

Keep All Entities Small
: Simply put - don't let your classes get to long. If they are starting to get big then there are probably more classes inside which can be extracted out.

[33] http://en.wikipedia.org/wiki/Law_of_Demeter

No Classes With More Than Two Instance Variables

Nice idea but a bit extreme in my opinion! Still it's worth keeping in mind as if you're adding lots of properties to your classes, then are probably some composite types hiding in there which should be extracted.

No Getters/Setters/Properties

The idea here is to tell the class to perform an action and let it get on with it; rather than getting values out, changing them and setting them again - often referred to as tell, don't ask[34]. Sometimes a getter or setter is needed, but always try your best to find a better way.

I can't say I stick to all of these rules 100% of the time. But, by trying to follow them as much as possible, you will write much cleaner and more manageable, *Object Oriented* code.

For more details on these rules take a look at William Durand's blog post titled Object Calisthenics[35].

Also, Guilherme Blanco's slides on the subject[36] are worth a look if you want to see the rules applied in PHP.

Automated Testing

Automated tests are simply tests which can be run to confirm the logic in your program is behaving as expected.

Automated test can be:

- Test code which executes and verifies the code being tested.
- Scripts written in a test language, which executes and verifies the code via a testing framework.
- Code which automates interaction with the user interface or API.
- Scripts written in a test language, which automate interaction with the user interface or API via a testing framework.

There are different types of test. These are categorised depending on what they are testing. For example you have:

Name	Purpose
Unit Tests	Test small, isolated units of code.
Integration Tests	Test the way that several units of code work together.
Acceptance Tests	Tests which prove to the stakeholder that the required functionality has been implemented.

[34] http://martinfowler.com/bliki/TellDontAsk.html

[35] http://williamdurand.fr/2013/06/03/object-calisthenics/

[36] http://www.slideshare.net/guilhermeblanco/object-calisthenics-applied-to-php

Tests provide confidence in your codebase, and confidence in the ability to add to and modify the code. Also, when refactoring, tests let you know that you haven't made a mistake and broken the logic. When adding new or modifying existing functionality, tests give you confidence that you haven't broken a different feature in the process.

Tests are great! You should have them! One often cited argument for not writing tests, is the extra time is takes to write them. What people who say this don't know about, is all the debugging time it saves. Also, if you use TDD then the process of creating the tests is not separate from the writing of the code, so it's not an extra task which has to be done.

Test Driven Development (TDD)

I'm going to be very brief here as the next chapter is going to go into it in much more depth.

Simply put, *Test Driven Development* is a discipline which involves using the tests to guide what production code is actually written - rather than writing the code, then working out how to test it. Through this process, TDD encourages you to write much cleaner and more modular code than you might write without it. It also gives you much higher confidence in the coverage of your test suite.

If you're new to TDD and want to start learning how to apply it, then the next chapter will get you started. Also, someone really worth checking out is *Chris Hartjes* aka *The Grumpy Programmer* - he preaches TDD to PHP programmers, has written a great book called The Grumpy Programmer's Guide to Building Testable Applications to PHP[37], and has videos available from http://grumpy-learning.com/

Behaviour Driven Development (BDD)

Behaviour Driven Development is an extension to TDD. It uses testing tools which encourage the language used in the tests to be written in a declarative way - conveying intent from a business perspective. Throughout this book we will be using BDD extensively.

Uncle Bob's Clean Code

Robert C. Martin, aka *Uncle Bob*, is a very outspoken and published programmer who has strong ideals about how high quality code should be written. His books include:

- Clean Code: A Handbook of Agile Software Craftsmanship
- The Clean Coder: A Code of Conduct for Professional Programmers
- Agile Software Development, Principles, Patterns, and Practices

[37] https://leanpub.com/grumpy-testing

He also has a fantastic video series called Clean Code[38].

Through his books and videos he covers all the topics we've looked at in this chapter in a lot more depth. If you really want to improve the way you work he is definitely someone to whom you should be paying attention.

Domain Driven Design (DDD)

Domain Driven Design is much more than how to write code. It includes the full process from understanding the business (*domain*), to translating that into software. As the name implies, DDD's driving factor is the *domain* (the business we are modelling), and the fact that as developers we have to accept that no one understands the *domain* as much as the *domain experts* (the people who understand and use the business we are modelling). When doing DDD we do not try to squeeze the domain we are modelling into our software development world's terms. Rather, we aim to transfer the domain language into our software's code.

The first stage of DDD, is sitting down with the *domain experts* and starting to learn about their domain. During this time, we start to crate a *ubiquitous language*. This is made up of the words and phrases that we can use to discuss the domain in the *domain expert's* terms. This *ubiquitous language* should then be used in all following discussions, as well as being transferred into the actual source code of the application.

While DDD is about the full process, from analysing and understanding the domain through to actually modelling it, there is a certain style of code and set of design patterns which it makes use of. These include use of value objects, entities, aggregates, the repository pattern and more.

Since DDD has a far broader scope than just the writing of the code, this book doesn't really cover it in any depth. However, the code produced in this book is heavily influence by this style of design.

If you really want to understand DDD, you want to start off by reading Eric Evans' book *Domain-Driven Design*.

Another person to keep an eye on regarding this subject in the PHP community is Mathias Verraes[39].

Command Query Responsibility Segregation (CQRS)

A lot of the techniques I have talked about have about so far, have been about building a rock solid *domain model* which cannot be broken by creating objects in invalid states. All these careful checks are very important when state is being changed and manipulated. However, often we just want to view the state. In this circumstance the construction of a complex domain model may be considered unnecessarily computationally expensive. For this reason we could consider a bomb proof domain model to be *optimised for change*.

[38] http://cleancoders.com/

[39] http://verraes.net/

Now consider this: most web applications have far more hits where state is just viewed, than they do when state is being updated. Take a basic web shop, you view tens of pages of search results, lists, products and reviews, before adding the product you want to the cart then checking out. Here the adding to the cart and checkout processes are the parts which actually update state, all the browsing actions are read only. We could assume that maybe only 1 in 5 hits change state, and we've already established that building a complex model for reading only is overly expensive. This is where CQRS comes in.

Before I explain CQRS let me introduce one more point about application architecture. I've already introduced how a robust domain model is the key, central part to an application. However, I've not said how it is interacted with. On top of the domain layer you might typically build a layer of *application service*, *use case* or *transaction* classes, which talk to the domain model to perform a specific action.

As an example, you might have a `ChangeCustomersEmail` transaction which loads up a `Customer` entity from the storage, changes the email address, then stores it. You might also might have a `ListCustomers` transaction which returns a list of customers on the system.

Now let's say the site is getting busy and growing, and it's getting slower and slower. It needs to scale, but how? Well since it's running on a domain model which is *optimised for changing*, and we also know that most hits are *read only*, it would make sense to make those actions *optimised for reading*. To do this you could separate out all *transactions* which are read only, and instead of building a complicated model from a complex data store for them. You could create a thin read layer, which reads it's information from a version of the data store which is optimised for the required queries. The result of this is you would have some *transactions* which modify state by talking to the complex domain model and updating the master data store. We call these *commands*. Then you have another set of *transactions*, which read from a *denormalised* version of the data store, to display the information quickly for reading only. We call these *queries*.

That's it, CQRS is all about splitting your application in half. One half is optimised for updating, the other for reading. There is more to CQRS regarding scaling across multiple servers, but what we have just talked about is the main gist of it from the software architecture point of view.

In this book we will not be building a full CQRS implementation (maybe that could be the sequel). However, I do feel that it is worthwhile making the distinction between the *command* and *query* transactions in our application. My reason for this is, that without the extra effort of building a full CQRS solution, we still make it clear which *transactions* change state. Should the day come that the application needs to be scaled up in a big way and the decision to implement CQRS is taken, this will make it just that little bit easier.

If you want to learn more about CQRS there's a few great articles by Greg Young in the documents section of http://cqrs.wordpress.com/

Agile

Agile is an approach to software development which is embodied in methodologies such as *Extreme Programming (XP)*[40] and *SCRUM*[41]. It accepts the fact that, getting from the stakeholder's idea to the final product, is not something that can be set as a rigid path from the beginning. As the project progresses, the stakeholder's ideas evolve and technical challenges might affect the way the project progresses.

With *agile* development, the stakeholder is heavily involved with the development process, and the project is broken down into manageable sized tasks. The stakeholder prioritises the tasks, then the development team works in small bursts of time, to get the tasks completed and delivered to the stakeholder - in order of priority. In XP these bursts of time are called *iterations*, whereas in SCRUM they're known as *sprints*. After each *iteration*/*sprint*, the stakeholder and development team review what has been done and decide what to tackle next.

Agile is about the approach to work and the idea that things can change as they develop. In this book I won't be discussing this any further, but I will be presenting the code examples as if we're working in an Agile environment. What this means is: the code I present in an earlier chapter may well change, be replaced or even be deleted in a later chapter. The reason for this is that I don't just want to present an application architecture as an end product of the book. Rather, I want to go through the process of how we get to that point.

A great book to check out on Agile style development is Kent Beck's *Extreme Programming Explained*.

User Stories

A user story is a short description of one of the tasks that a user will do when interacting with the system:

> A customer can view their previous orders.

These are created as part of the planning and analysis process. They are often written on small cards called *story cards*. *Acceptance tests* are then decided on for a story, then that feature can be implemented.

One thing to note is that a story is not a specification! It's intentionally vague and it's purpose is to exist as a *request for conversation*. When it is time to implement a story, the team (which includes the stakeholder) should discuss the story and decide on the details - this will result in the generation of the *acceptance tests*. For the purpose of this book, I'll provide the story and tests as though this conversation has already taken place, I'll do this for each part of the application we implement.

[40] http://en.wikipedia.org/wiki/Extreme_programming

[41] http://en.wikipedia.org/wiki/Scrum_(software_development)

Creating user stories is not something I will be covering. For this I really recommend Mike Cohn's rather fantastic book on the subject: *User Stories Applied*.

An Introduction to Testing and TDD

In this chapter I'm going to introduce some testing tools which we will be using, as well as give a brief introduction to TDD. This is an optional chapter so if you're already familiar with these tools and TDD then feel free to skip it. However, if you are new to TDD, or have not been doing it for long, then this chapter is definitely worth reading.

Like most of the things I've covered so far, this chapter is just here to serve as a brief introduction. It aims to give you what you will need to work through the rest of this book. As always, I recommend you research further in to everything covered here - there are many wonderful resources available.

Types of Test

The 2 main types of test which we will be using during the process of building our application are: *acceptance tests* and *unit tests*.

Acceptance Tests

Acceptance tests are high level tests which confirm to the developer **and** the stakeholder, that the required features have been implemented and are working properly. Since these tests are about producing the features that the stakeholder requests, they are ideally produced with the stakeholder's presence and input and in a form which they can understand.

A typical example might be to write something down like this:

> **Listing customers when there are none will return an empty list**
>
> Given there are no customers
>
> When I ask to list customers
>
> Then I should get an empty list

These simple rules could then be converted easily into automated tests using any assert based test framework, like so:

xUnit style acceptance test example:

```
function test_that_listing_customers_when_there_are_none_will_return_an_empty_li\
st()
{
    // Given there are no customers
    $customerRepository = new CustomerRepository();
    $customerRepository->clear();

    // When I ask to list customers
    $lister = new CustomerLister($customerRepository);
    $customers = $lister->listCustomers();

    // Then I should get an empty list
    $this->assertEquals([], $customers);
}
```

This works. The comments are there to explain the test in terms the stakeholder can understand, and the code checks the conditions are met. That said, it does mix up the stakeholder's language with the programmer's, and it relies on the comments being present and correct. Therefore, if you are working with the stakeholder to produce acceptance tests, then there is a better tool for the job. It's called *Cucumber*, and in PHP it's implemented by a tool called Behat[42].

Cucumber scripts are tests written in a language that the stakeholder can read and write in also. Tests are grouped as *features*, which are a collection of *scenarios*. These *scenarios* look very much like the original test rules we started with:

Behat acceptance test example

```
Feature: List customers
    In order to view a list of customers on the system
    As an administrator
    I need to be able to see lists of the customers information

    Scenario: Listing customers when there are none will return an empty list
        Given there are no customers
        When I ask to list customers
        Then I should get an empty list
```

The language used in the Behat tests can include any phrases that you want. The meanings of these phases (called *snippets*) are then implemented in PHP in classes know as *contexts*. Here's an example:

[42] http://behat.org/

Behat context example

```php
<?php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

/**
 * Behat context class.
 */
class FeatureContext implements SnippetAcceptingContext
{
    /**
     * @var CustomerRepository
     */
    private $customerRepository;

    /**
     * @var mixed
     */
    private $result;

    /**
     * Initializes context.
     *
     * Every scenario gets its own context object.
     * You can also pass arbitrary arguments to the context constructor through
     * behat.yml.
     */
    public function __construct()
    {
        $this->customerRepository = new CustomerRepository();
    }

    /**
     * @Given there are no customers
     */
    public function thereAreNoCustomers()
    {
        $this->customerRepository->clear();
    }
```

```
42
43      /**
44       * @When I ask to list customers
45       */
46      public function iAskToListCustomers()
47      {
48          $lister = new CustomerLister($this->customerRepository);
49          $this->result = $lister->listCustomers();
50      }
51
52      /**
53       * @Then I should get an empty list
54       */
55      public function iShouldGetAnEmptyList()
56      {
57          if ([] !== $this->result) {
58              throw new Exception('Result was not an empty array.');
59          }
60      }
61  }
```

The different *snippets* can then be reused in other *scenarios* and *features*.

In this book, I will be providing the *features* for each new bit of functionality that we add. We'll then use Behat to run the tests.

Unit Tests

Unit tests are for the development team's benefit. They are not written with the stakeholder's help; in fact they may event not know they exist. Instead, they are created by the developer as the code is written, and they test little bits of code as isolated *units*. A unit is a small piece of code which performs a specific task. Often, we treat each class as a unit, but there are situations where multiple classes, or a subsection of a single class, could be considered a unit. Saying that, PHPSpec (the main tool we will be using for unit testing) does enforce that each class is a *unit*. As a result, this will be the way that we shall mostly be working.

When doing TDD, the tests are actually written **before** each new bit of code is written. This process, when used properly, actually influences what code is written. Sometimes producing surprisingly elegant solutions.

There are 2 main types of *unit testing* tools, these are: *xUnit* style tools and *BDD* style tools.

xUnit Frameworks

xUnit test tools are call so because each language has at least one of these tools available and they are generally named *SomethingUnit* (e.g. JUnit for Java, CppUnit for C++). The origin of these names comes from *Smalltalk's SUnit*, which was created by Kent Beck and was the first testing framework of this kind. Of course, PHP has such a framework, and as expected it is called *PHPUnit*. *PHPUnit* is probably PHP's most well used unit testing framework and was created by Sebastian Bergmann.

xUnit frameworks are based on assertions. Each framework provides a set of available assertion functions. These are used to check that the results our code produces are what we expect. Some examples of the assertions functions available in *PHPUnit* are:

PHPUnit assertions example

```
1 $this->assertEquals(7, $subject->getValue());
2 $this->assertSame($expectedOwner, $subject->getOwner());
3 $this->assertFalse($subject->isBroken());
4 $this->assertTrue($subject->isWonderful());
5 $this->assertNull($subject->getNothing());
6 // ...
```

A full list of *PHPUnit's* assertions can be found in Appendix A[43] of the manual.

While we will not be using PHPUnit as the main *unit testing* framework in this book, we will be making use of it at some point for certain tests. We will also make use if its assertion library in our *Behat Contexts* for simplicity.

BDD Frameworks

The second type of unit testing frameworks are BDD *spec* frameworks. These are used in a very similar way to *xUnit* frameworks, but the language used in a bit different. Instead of *asserting* a test's expectations, they are described using the word *should*. The idea is that the tests describe the functionality of a unit, rather than just check it's functionality. I will be using PHPSpec[44] which is a great, but *highly opinionated*, BDD unit testing tool for PHP.

Using PHPSpec, the assertions we looked at for PHPUnit would written like this instead:

[43] https://phpunit.de/manual/current/en/appendixes.assertions.html
[44] http://www.phpspec.net/

PHPSpec tests example

```
$this->getValue()->shouldReturn(7);
$this->getOwner()->shouldReturn($expectedOwner);
$this->shouldNotBeBroken();
$this->shouldBeWonderful();
$this->getNothing()->shouldReturn(null);
```

The first difference you should notice is that they simply read as better sentences than the *PHPUnit* assertions.

There is a second difference though: assertions allow you to assert the values of anything. Whereas, PHPSpec's *should** methods only work on return values from the unit being tested or on *test double* methods. This means that you are much more limited in how you can test with *PHPSpec*, however, this is not necessarily a bad thing: it forces you to write good code and I like this (it's not for everyone though).

Red, Green, Refactor: The TDD Unit Testing Process

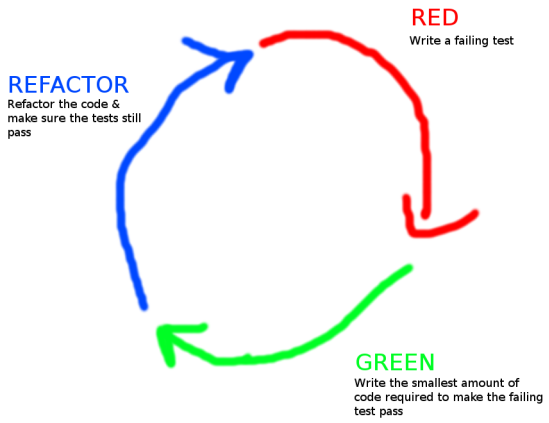
When doing TDD, there is a process that defines the order in which things should be done. This order is known as **Red**, **Green**, **Refactor**:

Red Red means that when we run our test suite we see a failing test. This is the first step in writing code: *write a failing test*. The test we write should be next smallest step we can take in our implementation which will cause the test suite to fail. Once the test is written, you must run the test suite and you **must see it fail**.

Green
The green step is making the failing test pass. In order to do this you only write **the smallest amount of code needed to make the failing test pass**. Then we run the test suite again and see that all tests pass.

Refactor
The refactor step is the point at which the code can, and should, be refactored. No functionality should be altered here, you simply tidy up the code. After any refactorings have been made, you must run the test suite to make sure you've not broken anything.

After the *Refactor* step you go back to the **Red** step and write the next failing test. This process is repeated until the solution is complete.

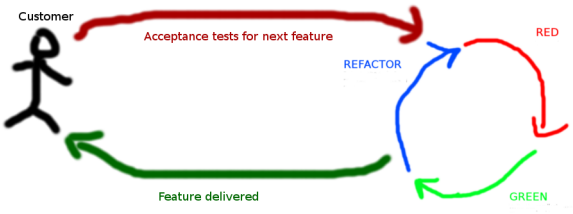


The Red, Green, Refactor Cycle

I will run though a demonstration of this process shortly in the Unit Testing With PHPSpec section.

The Double Feedback Loop

Acceptance tests and *unit tests* support the development process in a *double feedback loop*. *Acceptance tests* come from the *stakeholder* in order of priority. The *developers* then implement the features using *unit tests* with the *red, green, refactor cycle*, until the *acceptance tests* pass. At this point, the feature can be delivered to the *stakeholder* and the next set of *acceptance tests* will be produced.



The Double Feedback Loop

Given, When, Then

We've already seen the use of these 3 words earlier in the Acceptance Tests section. However, these 3 steps apply to all types of tests. The *given* part sets up the preconditions, the *when* part performs

the action being tested, and the *then* part checks the result. Whenever you are writing a test, it is a good idea to maintain clear grouping into these 3 stages.

We've seen this in Behat already. It's highlighted and enforce by the *Cucumber* language. Now lets take a look an example in a *PHPUnit* and a *PHPSpec* test:

Given, When, Then example using PHPUnit

```
// Inside AgeCalcuatorTest
function test_calculate_the_age_of_a_customer()
{
    // Given there is a customer whose date of birth is 1982-03-15
    // And today is 2014-09-03
    $customer = new Customer('customer name', Date::fromDate('1982-03-15'));
    $today = Date::fromDate('2014-09-03');

    // When I calcuate the age of the customer
    $calculator = new AgeCalculator($today);
    $age = $calculator->calculateAge($customer);

    // Then the result should be 32
    $this->assertEquals(32, $age);
}
```

Given, When, Then example using PHPSpec

```
// Inside AgeCalcuatorSpec
function it_can_calculate_the_age_of_a_customer()
{
    // Given there is a customer whose date of birth is 1982-03-15
    // And today is 2014-09-03
    $customer = new Customer('customer name', Date::fromDate('1982-03-15'));
    $this->beConstructedWith(Date::fromDate('2014-09-03'));

    // When I calcuate the age of the customer
    $age = $calculator->calculateAge($customer);

    // Then the result should be 32
    $age->shouldBe(32);
}
```

The comments are not actually necessary, and it's also fine to compound the various expressions (particularly the *when* and *then*) but the point is not to mix up the order of these stages. I like to use vertical whitespace to separate the setting up of the preconditions, from the tests. A more condensed version of the *PHPSpec* example would be:

Condensed Given, When, Then example using PHPSpec

```
// Inside AgeCalcuatorSpec
function it_can_calculate_the_age_of_a_customer()
{
    $customer = new Customer('customer name', Date::fromDate('1982-03-15'));
    $this->beConstructedWith(Date::fromDate('2014-09-03'));

    $calculator->calculateAge($customer)->shouldReturn(32);
}
```

I think that looks pretty neat and is very easy to read and understand what is supposed to happen.

Acceptance Testing with Behat

We've already seen *Behat* and what *Cucumber* tests look like. Now let's have a go at setting up a project using them. To start off create a new folder for the project and `cd` into it:

```
$ mkdir behat-example
$ cd behat-example
```

Next up, let's add Behat to the project. We've already seen how to set up the `composer.json` file but there is a simpler way: instead, we can create the `composer.json`, add Behat to it, and run `composer update`, all in one command like so:

```
$ composer require behat/behat:3.* --dev
```

Once it has finished, Behat will be installed in the project and the `behat` command will be available to us. Next, we need to tell Behat to initialise the project - to do this simply run:

```
$ behat --init
```

This command creates a directory called `features`. This is where we'll put the tests. It also creates directory inside `features` called `bootstrap`, which contains a file called `FeatureContext.php`. This is the default context, and is where we can add our *snippets*.

Now let's add a feature by creating a file called `features/list-books.feature`:

features/list-books.feature

```
Feature: List books
    In order to list books
    As a reader
    I must be able to view a list of all books stored on the system

    Scenario: Display an empty list
        Given there are no books
        When I list all books
        Then I should see an empty list
```

Next, run Behat:

```
$ behat
```

This should produce the following output:

```
Feature: List books
    In order to list books
    As a reader
    I must be able to view a list of all books stored on the system

  Scenario: Display an empty list   # features/list-books.feature:6
    Given there are no books
    When I list all books
    Then I should see an empty list

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.03s (9.00Mb)

--- FeatureContext has missing steps. Define them with these snippets:

    /**
     * @Given there are no books
     */
    public function thereAreNoBooks()
    {
        throw new PendingException();
    }
```

```
24
25      /**
26       * @When I list all books
27       */
28      public function iListAllBooks()
29      {
30          throw new PendingException();
31      }
32
33      /**
34       * @Then I should see an empty list
35       */
36      public function iShouldSeeAnEmptyList()
37      {
38          throw new PendingException();
39      }
```

This is Behat saying that it doesn't understand the *snippets* which we have used in the test. It also gives some template code which can be copied and pasted into our feature context to implement these *snippets*. We could do that, but we can actually get Behat to do this for us by running:

```
1  $ behat --append-snippets
```

If you look in `features/bootstrap/FeatureContext.php` after this command has run, you will see the template code has been added for the 3 new *snippets*. At this point we can run Behat again:

```
1  $ behat
```

This time it informs us that the first *snippet* has no content and needs to be implemented (the others have been skipped). Let's implement the first *snippet*.

Update `features/bootstrap/FeatureContext.php` so that it looks like this:

features/bootstrap/FeatureContext.php

```php
<?php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

use BehatExample\BookList;

/**
 * Behat context class.
 */
class FeatureContext implements SnippetAcceptingContext
{
    /**
     * @var BookList
     */
    private $bookList;

    /**
     * Initializes context.
     *
     * Every scenario gets its own context object.
     * You can also pass arbitrary arguments to the context constructor through
     * behat.yml.
     */
    public function __construct()
    {
        $this->bookList = new BookList();
    }

    /**
     * @Given there are no books
     */
    public function thereAreNoBooks()
    {
        $this->bookList->clear();
    }

    /**
     * @When I list all books
```

```
42          */
43         public function iListAllBooks()
44         {
45             throw new PendingException();
46         }
47
48         /**
49          * @Then I should see an empty list
50          */
51         public function iShouldSeeAnEmptyList()
52         {
53             throw new PendingException();
54         }
55     }
```

You can try running Behat again, but it will fail since there is no `BookList` class. To create this quickly set up the autoloader by updating the `composer.json` to look like this:

composer.json

```
{
    "require-dev": {
        "behat/behat": "3.*"
    },
    "autoload": {
        "psr-0": {
            "BehatExample\\": "src/"
        }
    }
}
```

Next, create the directory for the source code and update Composer:

```
$ mkdir -p src/BehatExample
$ composer update
```

Then, create the `BookList` class:

src/BehatExample/BookList.php

```
<?php

namespace BehatExample;

class BookList
{
    public function clear()
    {
    }
}
```

Running Behat at this point, will show the first *snippet* passing and inform that the second snippet needs to be implemented. Let's implement the last 2 *snippets* at once: update the `features/bootstrap/FeatureConte` to contain the following:

features/bootstrap/FeatureContext.php

```
<?php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

use BehatExample\BookList;

/**
 * Behat context class.
 */
class FeatureContext implements SnippetAcceptingContext
{
    /**
     * @var BookList
     */
    private $bookList;

    /**
     * @var mixed
     */
    private $result;

```

```
25      /**
26       * Initializes context.
27       *
28       * Every scenario gets its own context object.
29       * You can also pass arbitrary arguments to the context constructor through
30       * behat.yml.
31       */
32      public function __construct()
33      {
34          $this->bookList = new BookList();
35      }
36
37      /**
38       * @Given there are no books
39       */
40      public function thereAreNoBooks()
41      {
42          $this->bookList->clear();
43      }
44
45      /**
46       * @When I list all books
47       */
48      public function iListAllBooks()
49      {
50          $this->result = $this->bookList->getBooks();
51      }
52
53      /**
54       * @Then I should see an empty list
55       */
56      public function iShouldSeeAnEmptyList()
57      {
58          if ([] !== $this->result) {
59              throw new Exception('Result was incorrect.');
60          }
61      }
62  }
```

Again, we've not implemented `BookList::getBooks()` so let's do that too:

src/BehatExample/BookList.php

```php
<?php

namespace BehatExample;

class BookList
{
    public function clear()
    {
    }

    /** @return array */
    public function getBooks()
    {
        return [];
    }
}
```

Run Behat and see the test pass:

```
$ behat
Feature: List books
    In order to list books
    As a reader
    I must be able to view a list of all books stored on the system

  Scenario: Display an empty list
    Given there are no books
    When I list all books
    Then I should see an empty list

1 scenario (1 passed)
3 steps (3 passed)
0m0.01s (9.10Mb)
```

Before we move on, let's quickly add another test to the feature by appending the following to `features/list-books.feature`:

features/list-books.feature

```
    Scenario: Books are listed in alphabetical order
        Given there is a book called "Domain Driven Design" by "Eric Evans"
        And there is a book called "Refactoring" by "Martin Fowler"
        And there is a book called "Design Patterns" by "The Gang of Four"
        When I list all books
        Then I should see:
            | title                | author           |
            | Design Patterns      | The Gang of Four |
            | Domain Driven Design | Eric Evans       |
            | Refactoring          | Martin Fowler    |
```

Again, if you run Behat it will say that there are new, unknown *snippets*. As before, add these by running:

```
$ behat --append-snippets
```

Open up `features/bootstrap/FeatureContext.php` again, and update the 2 new methods which have been added to the class, so that they look like this:

features/bootstrap/FeatureContext.php

```
    /**
     * @Given there is a book called :title by :author
     */
    public function thereIsABookCalledBy($title, $author)
    {
        $this->bookList->add($title, $author);
    }

    /**
     * @Then I should see:
     */
    public function iShouldSee(TableNode $table)
    {
        if ($table->getHash() !== $this->result) {
            throw new Exception('Result was incorrect.');
        }
    }
```

Finally, update `BehatExample\BookList` to contain a working implementation:

src/BehatExample/BookList.php

```php
<?php

namespace BehatExample;

class BookList
{
    /** @var array */
    private $books = [];

    public function clear()
    {
        $this->books = [];
    }

    /**
     * @param string $title
     * @param string $author
     */
    public function add($title, $author)
    {
        $this->books[] = [
            'title'  => $title,
            'author' => $author
        ];
    }

    /** @return array */
    public function getBooks()
    {
        $list = $this->books;

        usort($list, function ($a, $b) {
            return $a['title'] > $b['title'];
        });

        return $list;
    }
}
```

Then run Behat to see the tests pass:

```
$ behat
Feature: List books
    In order to list books
    As a reader
    I must be able to view a list of all books stored on the system

  Scenario: Display an empty list
    Given there are no books
    When I list all books
    Then I should see an empty list

  Scenario: Books are listed in alphabetical order
    Given there is a book called "Domain Driven Design" by "Eric Evans"
    And there is a book called "Refactoring" by "Martin Fowler"
    And there is a book called "Design Patterns" by "The Gang of Four"
    When I list all books
    Then I should see:
      | title                | author           |
      | Design Patterns      | The Gang of Four |
      | Domain Driven Design | Eric Evans       |
      | Refactoring          | Martin Fowler    |

2 scenarios (2 passed)
8 steps (8 passed)
0m0.02s (9.15Mb)
```

Mink

AUTHOR NOTE: Coming soon...

The Mink Context

AUTHOR NOTE: Coming soon...

Which Language to use in Interface Tests

AUTHOR NOTE: Coming soon...

Unit Testing with PHPSpec

Now let's take a look at *unit testing*. We'll use *PHPSpec* for this. As we work through building the application in the next section of this book, I'll be showing the process of *acceptance testing*.

However, I will not be showing the process of *unit testing* in the examples (except in certain circumstances where there is something to learn from it). The reason for this is, that it would be a long winded, pointless and excessive process to document, and it would detract from the topic of this book. That said, all the example code will have been written using the full TDD process, and you should do the same. This also means, that all the example code for this book will contain complete unit tests which you can study for yourself.

Since we're not going to be highlighting the unit testing process when building the application, we'll go through an example now to get familiar with the process.

Example

For our TDD example we'll create an algorithm which finds the greatest common divisor of 2 numbers.

Setup

Let's start by creating a new project:

```
$ mkdir phpspec-example
$ cd phpspec-example
```

And configure Composer for the project:

composer.json

```
{
    "require-dev": {
        "phpspec/phpspec": "2.*@dev"
    },
    "autoload": {
        "psr-0": {
            "PhpspecExample\\": "src/"
        }
    }
}
```

```
$ composer install
```

Now we can start the TDD Cycle:

First of all, we'll tell PHPSpec to create a new Spec file. To do this run:

```
$ phpspec desc PhpspecExample\\GreatestCommonDivisorFinder
```

This will create a test file called `spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php`. To run it use:

```
$ phpspec run --format=pretty
```

PHPSpec will say it can't find the `PhpspecExample\GreatestCommonDivisorFinder` class and offer to create it, say **Yes** and run will create it and run the tests again. This time the test should pass.

Output Formatting

The `--format=pretty` option is not necessary. I have added it because it produces nice output which I can paste into this book.

There are other formats available, these are: **progress** (the default), **html**, **pretty**, **junit** and **dot**.

At this point 2 files have been created for us. The test:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```php
<?php

namespace spec\PhpspecExample;

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class GreatestCommonDivisorFinderSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType('PhpspecExample\GreatestCommonDivisorFinder');
    }
}
```

And the class we are going to implement:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
<?php

namespace PhpspecExample;

class GreatestCommonDivisorFinder
{
}
```

The `it_is_initializable` test has served it's purpose now so you can remove that from the test.

Red

Now we need to think what the simplest condition we can test is, which will make progress towards the final goal.

Let's start with the condition that when both numbers are the same, then that number is the greatest common divisor.

Add the following test to the spec file:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
    function it_returns_the_number_if_both_values_are_that_number()
    {
        $this->findGreatestDivisor(5, 5)->shouldReturn(5);
    }
```

Now we have to see this fail, so run:

```
$ phpspec run --format=pretty
```

Again PHPSpec says that the `findGreatestDivisor` method does not exist and offers to create it. Say **yes**, this will add the empty method to `PhpspecExample\GreatestCommonDivisorFinder` and run it again. At this point you should see some red:

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  x returns the number if both values are that number
        expected [integer:5], but got null.

----  failed examples

        PhpspecExample/GreatestCommonDivisorFinder
  10  x returns the number if both values are that number
        expected [integer:5], but got null.


1 specs
1 examples (1 failed)
11ms
```

This tells us that the test was expecting the value `5` to be returned, but `null` was returned instead. Time to make make this test go green...

Green

Remember that in the green stage, we must only add the smallest amount of code to make the test pass. In this case all we need to do is return the value `5`. Update the `PhpspecExample\GreatestCommonDivisorFinder` method and re-run the tests:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
    public function findGreatestDivisor($a, $b)
    {
        return 5;
    }
```

This time we see the green result:

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number


1 specs
1 examples (1 passed)
9ms
```

Refactor

This is the point when we do our refactoring. However, right now there's not much to refactor, so let's move straight on to the next failing test.

Red

For this next test, let's check that when the first number is a divisor of the second number, then the first number should be returned. To do this add the following test:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
    function it_returns_the_first_number_if_it_is_a_divisor_of_the_second()
    {
        $this->findGreatestDivisor(3, 9)->shouldReturn(3);
    }
```

Now run PHPSpec to see red:

```
$ phpspec run --format=pretty

     PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number
  15  x returns the first number if it is a divisor of the second
        expected [integer:3], but got [integer:5].

----  failed examples

        PhpspecExample/GreatestCommonDivisorFinder
  15  x returns the first number if it is a divisor of the second
```

```
13          expected [integer:3], but got [integer:5].
14
15
16  1 specs
17  2 examples (1 passed, 1 failed)
18  12ms
```

The first test continues to pass, but the second one does not since our implementation currently only returns the integer `5`.

Green

Again, using the smallest change possible, we can easily change this to green by just returning the first number:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
1      public function findGreatestDivisor($a, $b)
2      {
3          return $a;
4      }
```

Now when running PHPSpec we see that both tests are successfully passing:

```
1  $ phpspec run --format=pretty
2
3        PhpspecExample\GreatestCommonDivisorFinder
4
5    10  ✓ returns the number if both values are that number
6    15  ✓ returns the first number if it is a divisor of the second
7
8
9  1 specs
10 2 examples (2 passed)
11 11ms
```

Refactor

There's still nothing to refactor yet. So on to the next failing test...

Red

Let's now try the previous test the other way around: when the second number is a divisor of the first, the second number should be returned.

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
    function it_returns_the_second_number_if_it_is_a_divisor_of_the_first()
    {
        $this->findGreatestDivisor(9, 3)->shouldReturn(3);
    }
```

Run PHPSpec again (**you have to see it fail!**):

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number
  15  ✓ returns the first number if it is a divisor of the second
  20  x returns the second number if it is a divisor of the first
        expected [integer:3], but got [integer:9].

----  failed examples

        PhpspecExample/GreatestCommonDivisorFinder
  20  x returns the second number if it is a divisor of the first
        expected [integer:3], but got [integer:9].


1 specs
3 examples (2 passed, 1 failed)
13ms
```

Green

To make this one pass we can simply return the smaller of the two numbers:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
    public function findGreatestDivisor($a, $b)
    {
        return min($a, $b);
    }
```

Yet again, we run the tests and see the successful results:

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number
  15  ✓ returns the first number if it is a divisor of the second
  20  ✓ returns the second number if it is a divisor of the first


1 specs
3 examples (3 passed)
10ms
```

Refactor

Still nothing to refactor so let's move on...

Red

How about if there's no common divisor other than then number 1? Let's write a test for that:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
    function it_returns_1_if_there_is_no_greater_divisor()
    {
        $this->findGreatestDivisor(3, 5)->shouldReturn(1);
    }
```

As always, run the tests and see it fail:

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number
  15  ✓ returns the first number if it is a divisor of the second
  20  ✓ returns the second number if it is a divisor of the first
  25  x returns 1 if there is no greater divisor
        expected [integer:1], but got [integer:3].

----  failed examples

```

```
13         PhpspecExample/GreatestCommonDivisorFinder
14   25  x returns 1 if there is no greater divisor
15         expected [integer:1], but got [integer:3].
16
17
18 1 specs
19 4 examples (3 passed, 1 failed)
20 13ms
```

Green

This one is a tiny bit more complicated. To solve it let's say that if the lower of the 2 numbers is not a factor of one of the numbers, then return 1:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
1     public function findGreatestDivisor($a, $b)
2     {
3         $divisor = min($a, $b);
4
5         if ($a % $divisor !== 0 || $b % $divisor !== 0) {
6             $divisor = 1;
7         }
8
9         return $divisor;
10    }
```

Run the tests:

```
1  $ phpspec run --format=pretty
2
3        PhpspecExample\GreatestCommonDivisorFinder
4
5    10  ✓ returns the number if both values are that number
6    15  ✓ returns the first number if it is a divisor of the second
7    20  ✓ returns the second number if it is a divisor of the first
8    25  ✓ returns 1 if there is no greater divisor
9
10
11 1 specs
12 4 examples (4 passed)
13 11ms
```

So far, so good!

Refactor

Finally, something to refactor! There is a duplication of the logic to check if the `$divisor` variable is a factor of a variable. Let's extract it out with the extract method[45] refactoring like so:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
<?php

namespace PhpspecExample;

class GreatestCommonDivisorFinder
{
    private $divisor;

    public function findGreatestDivisor($a, $b)
    {
        $this->divisor = min($a, $b);

        if (!$this->divisorIsFactorOf($a) || !$this->divisorIsFactorOf($b)) {
            $this->divisor = 1;
        }

        return $this->divisor;
    }

    private function divisorIsFactorOf($target)
    {
        return $target % $this->divisor === 0;
    }
}
```

Now run the tests to make sure they still pass:

[45] http://refactoring.com/catalog/extractMethod.html

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number
  15  ✓ returns the first number if it is a divisor of the second
  20  ✓ returns the second number if it is a divisor of the first
  25  ✓ returns 1 if there is no greater divisor


1 specs
4 examples (4 passed)
11ms
```

Red

OK. Next let's try a divisor that is not `1`, or either of the numbers themselves:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
    function it_returns_a_divisor_of_both_numbers()
    {
        $this->findGreatestDivisor(6, 9)->shouldReturn(3);
    }
```

Watch the test fail:

```
$ phpspec run --format=pretty

      PhpspecExample\GreatestCommonDivisorFinder

  10  ✓ returns the number if both values are that number
  15  ✓ returns the first number if it is a divisor of the second
  20  ✓ returns the second number if it is a divisor of the first
  25  ✓ returns 1 if there is no greater divisor
  30  x returns a divisor of both numbers
        expected [integer:3], but got [integer:1].

----  failed examples

        PhpspecExample/GreatestCommonDivisorFinder
  30  x returns a divisor of both numbers
```

```
16          expected [integer:3], but got [integer:1].
17
18
19  1 specs
20  5 examples (4 passed, 1 failed)
21  14ms
```

Green

We actually only have to make a very small change here to make this pass: we simply change the `if` to a `while`, and decrement the divisor in the loop:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
1      public function findGreatestDivisor($a, $b)
2      {
3          $this->divisor = min($a, $b);
4
5          // leanpub-insert-start
6          while (!$this->divisorIsFactorOf($a) || !$this->divisorIsFactorOf($b)) {
7              $this->divisor--;
8          }
9          // leanpub-insert-end
10
11         return $this->divisor;
12     }
```

Run the tests:

```
1  $ phpspec run --format=pretty
2
3        PhpspecExample\GreatestCommonDivisorFinder
4
5    10  ✓ returns the number if both values are that number
6    15  ✓ returns the first number if it is a divisor of the second
7    20  ✓ returns the second number if it is a divisor of the first
8    25  ✓ returns 1 if there is no greater divisor
9    30  ✓ returns a divisor of both numbers
10
11
12 1 specs
13 5 examples (5 passed)
14 11ms
```

Everything has passed! And, if you study the code you should see we've reached a final solution to *find the greatest common divisor or 2 numbers.*

If you really want to double check it, you could add this extra test, but it will pass straight away:

spec/PhpspecExample/GreatestCommonDivisorFinderSpec.php

```
    function it_returns_the_greatest_divisor_of_both_numbers()
    {
        $this->findGreatestDivisor(12, 18)->shouldReturn(6);
    }
```

Refactor

Looking at the code that we've produced, I'd say it's pretty neat already, and doesn't require any additional refactoring. Maybe we could just add some *docblocks* for type information.

The Result

Here is this final result. A complete solution to our original problem, achieved by TDD:

src/PhpspecExample/GreatestCommonDivisorFinder.php

```
<?php

namespace PhpspecExample;

class GreatestCommonDivisorFinder
{
    /** @var int */
    private $divisor;

    /**
     * @param int $a
     * @param int $b
     *
     * @return int
     */
    public function findGreatestDivisor($a, $b)
    {
        $this->divisor = min($a, $b);

        while (!$this->divisorIsFactorOf($a) || !$this->divisorIsFactorOf($b)) {
            $this->divisor--;
```

```
22          }
23
24          return $this->divisor;
25      }
26
27      /**
28       * @param int $target
29       *
30       * @return bool
31       */
32      private function divisorIsFactorOf($target)
33      {
34          return $target % $this->divisor === 0;
35      }
36  }
```

Test Doubles

I've already mentioned *test doubles*, and how they can be used to help test units of code in isolation by replacing the unit's dependencies. Now let's take a look at how this is actually done.

Types of Test Doubles

Test doubles take on different forms depending on how the are used. Each of these different types are used to solve a different problem. Let's take a look at the different types.

Dummy

A *dummy* is the simplest *test double*. It simply serves as a place holder for a dependency. It's essentially an empty class which implements the dependency's interface. *Dummies* are used when the unit being tested has a required dependency, but the dependency is not needed for the functionality being tested.

Dummy test double example

```php
/**
 * Method to be tested.
 *
 * @param number $a
 * @param number $b
 *
 * @return number
 */
public function addValues(Writer $writer, $a, $b)
{
    $writer->write("Adding $a and $b");

    return $a + $b;
}

/**
 * The test.
 *
 * Here we only want to test the addition takes place and don't care
 * about the writer. However, the writer is required, so we provide a dummy in
 * its place.
 */
function test_dummy()
{
    $writer = new DummyWriter();
    $examples = new Examples();

    $this->assertEquals(5, $examples->addValues($writer, 2, 3));
}

/**
 * The dummy writer.
 *
 * No functionality is requried, it just needs to implement the Writer
 * interface.
 */
final class DummyWriter implements \Writer
{
    public function write($string)
    {
    }
```

```
42 }
```

Stub

A *stub* is the next simplest *test double*. It simply returns constant values from the methods in the public interface. *Stubs* are used to isolate the unit under test, from the complexity of its dependency, or for testing an interface or abstract class which has no implementation available.

Stub test double example

```
1  /**
2   * The method being tested.
3   *
4   * @return int
5   */
6  public function doubleInput(Reader $reader)
7  {
8      return $reader->readInt() * 2;
9  }
10
11 /**
12  * The test.
13  *
14  * We don't want to use a real Reader to check the logic, so we use a stub
15  * which returns a constant value, which we can reliably check.
16  */
17 function test_stub()
18 {
19     $reader = new StubReader();
20     $examples = new Examples();
21
22     $this->assertEquals(14, $examples->doubleInput($reader));
23 }
24
25 /**
26  * The stub reader.
27  *
28  * There is no logic in a stub, it simply returns constant values.
29  */
30 final class StubReader implements Reader
31 {
32     public function readInt($src = self::STDIN)
33     {
```

```
34          return 7;
35      }
36  }
```

Fake

A *fake* contains logic which emulates a simplified and reliable version of the dependency. *Fakes* are used when the dependency is suitably complex, that a stub is not powerful enough.

Fake test double example

```
1   /**
2    * The method being tested.
3    *
4    * Here Reader::readInt() is called multiple times with different parameters.
5    */
6   public function addFromInputAndFile(Reader $reader, $filename)
7   {
8       return $reader->readInt(Reader::STDIN) + $reader->readInt($filename);
9   }
10
11  /**
12   * The test.
13   *
14   * We're using a fake as we want different values to be returned depending
15   * on the parameters provided.
16   */
17  function test_fake()
18  {
19      $reader = new FakeReader();
20      $examples = new Examples();
21
22      $this->assertEquals(
23          7,
24          $examples->addFromInputAndFile($reader, 'file.txt')
25      );
26  }
27
28  /**
29   * The fake reader.
30   *
31   * Here we return a 2 if the reader is instructed to read from STDIN, otherwise
32   * it returns 5.
```

```
 */
final class FakeReader implements Reader
{
    public function readInt($src = self::STDIN)
    {
        return $src == self::STDIN ? 2 : 5;
    }
}
```

Mock

A *mock* is like a *fake*, but unlike a *fake mocks* are aware that they are part of a test suite. Rather than being just a simplified implementation of the dependency's interface, they contain testing related logic and code.

When crafting *test doubles* by hand the situation for creating a *mock* comes up very rarely. However, as we'll see in a moment, *test doubles* can be created automatically using *mocking frameworks*. In this case all test doubles are technically *mocks*, regardless of the actual way they are being used. For this reason, the term *mock* is often used interchangeably with *test double*.

Spy

A *spy* is a type of *mock* which records the actions which have been performed on it. After executing the code under test, a spy can be asked if it was interacted with as expected. *Spies* are used to verify a dependency has be used in the correct way.

Spy test double example

```
/**
 * Method being tested.
 *
 * @param number $a
 * @param number $b
 *
 * @return number
 */
public function addValues(Writer $writer, $a, $b)
{
    $writer->write("Adding $a and $b");

    return $a + $b;
}

```

```php
/**
 * The test.
 *
 * This time, we want to make sure that the writer was instructed to write a
 * message.
 */
function test_spy()
{
    $writer = new SpyWriter();
    $examples = new Examples();

    $examples->addValues($writer, 2, 3);

    $this->assertTrue(
        $writer->hasWriteBeenCalledWith('Adding 2 and 3'),
        '$writer->write("Adding 2 and 3") should have been called'
    );
}

/**
 * The spy write test double.
 *
 * The spy records the message that the write function was called with.
 */
final class SpyWriter implements \Writer
{
    private $message;

    public function write($message)
    {
        $this->message = $message;
    }

    /**
     * @param string $message
     *
     * @return bool
     */
    public function hasWriteBeenCalledWith($message)
    {
        return $this->message == $message;
    }
```

```
58 }
```

Expectations

Expectations serve a similar purpose to *spies*, except rather than recording what methods have been called so they can be checked later, the methods required to be called are defined first. Not all mocking framework's provide both *spies* and *expectations*, but generally they will support at least one of these features.

The one thing which using *expectations* does, is mix up the *given, when, then* order. For this reason, I prefer to use *spies* when possible.

Mocking Frameworks

There are many mocking frameworks available for PHP. PHPUnit has one built in, PhpSpec uses one called Prophecy[46], and there are others available too. The main reasons for using a different framework are, either because it provides some extra features, or because they produce tests which you think are easier to understand.

Some examples of other mocking frameworks are Mockery[47], Phake[48] and Facebook's FBMock[49].

Now, let's quickly take a look at the previous test double examples, this time using PHPUnit's and PHPSpec's Prophecy framework's mocking facilities.

PHPUnit

PHPUnit's built in mocking facilities are quite verbose, and often considered to not be the easiest to read. It also doesn't support *spies*, so you have to use *expectations*. For these reasons, alternative mocking frameworks are often chosen when working with PHPUnit.

Here's the examples with PHPUnit:

[46] https://github.com/phpspec/prophecy
[47] https://github.com/padraic/mockery
[48] http://phake.readthedocs.org/en/latest/
[49] https://github.com/facebook/FBMock

tests/MockExampleTest.php

```php
<?php

namespace tests;

use Examples;
use Reader;

class MockExampleTest extends \PHPUnit_Framework_TestCase
{
    function test_dummy()
    {
        $examples = new Examples();

        $writer = $this->getMock('Writer');

        $this->assertEquals(5, $examples->addValues($writer, 2, 3));
    }

    function test_stub()
    {
        $examples = new Examples();

        $reader = $this->getMock('Reader');

        $reader->expects($this->any())
               ->method('readInt')
               ->will($this->returnValue(7));

        $this->assertEquals(14, $examples->doubleInput($reader));
    }

    function test_fake()
    {
        $examples = new Examples();

        $reader = $this->getMock('Reader');

        $reader->expects($this->at(0))
               ->method('readInt')
               ->with($this->equalTo(Reader::STDIN))
               ->will($this->returnValue(2));
```

```
42
43          $reader->expects($this->at(1))
44                 ->method('readInt')
45                 ->with($this->equalTo('file.txt'))
46                 ->will($this->returnValue(5));
47
48          $this->assertEquals(
49              7,
50              $examples->addFromInputAndFile($reader, 'file.txt')
51          );
52      }
53
54      function test_expectation()
55      {
56          $examples = new Examples();
57
58          $writer = $this->getMock('Writer');
59
60          $writer->expects($this->once())
61                 ->method('write')
62                 ->with($this->equalTo('Adding 2 and 3'));
63
64          $examples->addValues($writer, 2, 3);
65      }
66  }
```

PHPSpec & Prophecy

Prophecy is very much a part of PHPSpec. By using PHPSpec, you are choosing to use Prophecy also. That said, Prophecy doesn't have to be used with PHPSpec and can be used with other test frameworks also.

PHPSpec makes mocking exceptionally simple. You don't need to call any special methods to create a mock, rather you just provide a *typehinted* parameter to the test method, and PHPSpec will create and inject the mock automatically.

Here's the same examples with PHPSpec:

spec/ExamplesSpec.php

```php
<?php

namespace spec;

use PhpSpec\ObjectBehavior;
use Reader;
use Writer;

class ExamplesSpec extends ObjectBehavior
{
    function it_uses_mock_as_dummy(Writer $writer)
    {
        $this->addValues($writer, 2, 3)->shouldReturn(5);
    }

    function it_uses_mock_as_stub(Reader $reader)
    {
        $reader->readInt()->willReturn(7);

        $this->doubleInput($reader)->shouldReturn(14);
    }

    function it_uses_mock_as_fake(Reader $reader)
    {
        $reader->readInt(Reader::STDIN)->willReturn(2);
        $reader->readInt('file.txt')->willReturn(5);

        $this->addFromInputAndFile($reader, 'file.txt')->shouldReturn(7);
    }

    function it_uses_mock_as_spy(Writer $writer)
    {
        $this->addValues($writer, 2, 3);

        $writer->write('Adding 2 and 3')->shouldHaveBeenCalled();
    }

    function it_can_set_expectations_on_mocks(Writer $writer)
    {
        $writer->write('Adding 2 and 3')->shouldBeCalled();

```

```
42          $this->addValues($writer, 2, 3);
43      }
44  }
```

Katas

In order to develop and hone your programming skills, the concept of *katas* has become a popular way to practice them. A *kata* (like a martial arts kata) is a sequence of steps which are performed repeatedly in order to drill and perfect the skills and techniques which they contain.

Some popular katas are *Prime Factors*, *The Bowling Game*, *String Calculator* and *Roman Numerals*. You can easily find the details and demonstrations of these katas in many languages with a quick web search. That said Ciaran McNulty has put up excellent videos of the Prime Factors[50] and Roman Numerals[51] katas using PHPSpec on Vimeo. I think these are well worth a watch.

[50] http://vimeo.com/74529780

[51] http://vimeo.com/88289877

Building the Application

This is the main part of the book. Here we'll work through the complete process of building a robust, extensible and scalable application from scratch.

Getting Started

The Application

A brief description of the application we are going to build is as follows:

> The aim is to produce a website, which allows *users* of the site to *view* and *rate* cocktail recipes submitted by other users. They can also *submit their own*.
>
> Any *visitors* to the site can view the list of recipes *sorted by rating*. However, a *visitor* must *register* as a *user*, with a *username*, *email* and *password*, in order to *rate* or *submit recipes*.
>
> A *recipe* consists of the *cocktail name*, the *method* and the list of *measured ingredients*, which consists of the *ingredient* and *amount*. The *recipe* must also keep track of the *user* who *submitted* it.
>
> Ratings will be *star* ratings, with *users* being able to *rate* a recipe with 1 to 5 *stars*.
>
> *Quantities* can be entered as either *millilitres (ml)*, *fluid ounces (fl oz)*, *teaspoons (tsp)* or just a number.
>
> The cocktail *ingredients* available are limited to a selection which can only be added to by an *administrator*.

Now we've got a basic understanding of the application we are going to build, let's take a quick look at the list of user stories. These are presented in order of priority.

- A visitor can view a list of recipes
- A visitor can view a recipe
- A visitor can register and become a user
- A visitor can login to become a user
- A user can rate a recipe
- A user can add a recipe
- An administrator can add an ingredient

We will proceed to implement each of these stories in order. This may seem like a very basic application, but it will provide enough functionality to give a good example of how to start building a well designed, extensible application. Also, because we will be emulating an *agile* process while building the application, details and requirements may change as it progresses, and extra features may be requested.

Creating the Project

Before jumping in, let's quickly set up a project. Start by creating a directory to build the application in:

```
$ mkdir cocktail-rater
$ cd cocktail-rater
```

Then add the following `composer.json`

composer.json

```
{
    "require": {
        "php": ">=5.5"
    },
    "require-dev": {
        "behat/behat": "3.*",
        "phpunit/phpunit": "4.2.*",
        "phpspec/phpspec": "2.*@dev"
    },
    "autoload": {
        "psr-4": {
            "CocktailRater\\": "src/"
        }
    }
}
```

We're using a PSR-4 autoloader here. Using PSR-4 means everything can exist in a `CocktailRater` top level namespace, but we can avoid creating an extra directory level for it.

PHPSpec Version

You may have noticed that the PHPSpec requirement is for a development version. The reason for this is: there are some features which we will be using which are not in the stable release yet. When this changes I will update the book.

Now run Composer to install the test tools:

```
$ composer install
```

We can also configure it to format its output using the *pretty* formatter by default. That way we don't need to put it on the command line every time we run it. To do this, create a file called `phpspec.yml` with the following contents:

phpspec.yml

```
formatter.name: pretty
```

Finally, initialise Behat so we're ready to start development:

```
$ behat --init
```

All done! Now we can start.

The First Story

Let's look at the first story. Here's the card:

> A visitor can view a list of recipes
>
> - Displays an empty list if there are no recipes
> - Recipes display the *name of the cocktail*, *the rating* and the *name of the user who submitted it*
> - The list should be presented in *descending* order of *rating*

From this information, we can add the following feature file to the project:

features/visitors-can-list-recipes.feature

```
Feature: A visitor can view a list of recipes
    In order to view a list of recipes
    As a visitor
    I need to be able get a list of recipes

    Scenario: View an empty list of recipes
        Given there are no recipes
        When I request a list of recipes
        Then I should see an empty list

   Scenario: Viewing a list with 1 recipe
        Given there's a recipe for "Mojito" by user "tom" with 5 stars
        When I request a list of recipes
        Then I should see a list of recipes containing:
            | name   | rating | user |
            | Mojito | 5.0    | tom  |
```

```
17
18      Scenario: Recipes are sorted by rating
19          Given there's a recipe for "Daquiri" by user "clare" with 4 stars
20          And there's a recipe for "Pina Colada" by user "jess" with 2 stars
21          And there's a recipe for "Mojito" by user "tom" with 5 stars
22          When I request a list of recipes
23          Then I should see a list of recipes containing:
24              | name        | rating | user  |
25              | Mojito      | 5.0    | tom   |
26              | Daquiri     | 4.0    | clare |
27              | Pina Colada | 2.0    | jess  |
```

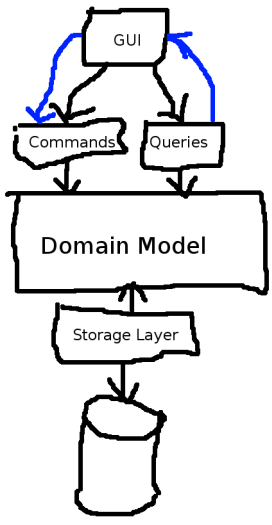
If you try to run Behat with this feature, it will say that the context has missing steps. To add the required snippets run:

```
1  $ behat --append-snippets
```

Now we can start working to get these *scenarios* to pass.

Application Structure

Before jumping straight into writing code, let's just take a small moment to take a look at the structure we plan to use to build the application.



Proposed Application Structure

The core part of the application will be the *domain model*, this will consist of our modelled interpretation of the business rules. It will have no knowledge of how or where the data is stored, the user interface or any non-business related implementation details. To achieve this level of separation we'll use *inversion of control* to let the other layers *plug in* to the domain layer.

Behind the *domain model* there will be a storage implementation layer for our chosen storage system. The storage system has not yet been decided so we'll make use of SQLite until we have chosen which one to use. The reasons for using SQLite are that, it allows the use of a database file without needing to set up a database server, and it's easier to use than writing our own file-based storage system.

In chapter 3 I introduced CQRS and stated that while we are not going to implement it in our application, we will make a distinction between *command* and *query* interactions within the application. Therefore, in front of the *domain model* we'll have a layer of *commands* and *queries*. All interactions with the *domain model* from the *UI* will go through these.

Finally, we'll have the UI website. We'll start off by mocking this up with some basic HTML, but as our application becomes more complete, we can make use of a modern MVC[52] framework. Again, we won't worry about which one until later on.

Scenario: View an empty list of recipes

Let's start off by getting the first scenario to pass. As a quick reminder here it is:

View an empty list of recipes

```
1     Scenario: View an empty list of recipes
2         Given there are no recipes
3         When I request a list of recipes
4         Then I should see an empty list
```

We're going to use TDD to create our code from the *outside in*. What I mean by this is: rather than trying to build the model and then get it to do what we need it to do, we'll start with what we want it to do and let that help create the model.

Fleshing out the FeatureContext

Behat has already added the required *snippet* templates to the `FeatureContext`, so let's try to pencil in what we want to happen. Take a look at the code I have added first, then I'll explain it:

[52]The Model View Controller design pattern.

features/bootstrap/FeatureContext.php

```php
<?php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Behat\Tester\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;
use CocktailRater\Application\Query\ListRecipes;
use CocktailRater\Application\Query\ListRecipesHandler;
use CocktailRater\Application\Query\ListRecipesQuery;
use CocktailRater\Application\Query\ListRecipesQueryHandler;
use CocktailRater\Testing\Repository\TestRecipeRepository;
use PHPUnit_Framework_Assert as Assert;

/**
 * Behat context class.
 */
class FeatureContext implements SnippetAcceptingContext
{
    /** @var RecipeRepository */
    private $recipeRepository;

    /** @var mixed */
    private $result;

    /**
     * Initializes context.
     *
     * Every scenario gets its own context object.
     * You can also pass arbitrary arguments to the context constructor through
     * behat.yml.
     */
    public function __construct()
    {
    }

    /**
     * @BeforeScenario
     */
    public function beforeScenario()
    {
        $this->recipeRepository = new TestRecipeRepository();
```

```
42      }
43
44      /**
45       * @Given there are no recipes
46       */
47      public function thereAreNoRecipes()
48      {
49          $this->recipeRepository->clear();
50      }
51
52      /**
53       * @When I request a list of recipes
54       */
55      public function iRequestAListOfRecipes()
56      {
57          $query = new ListRecipesQuery();
58          $handler = new ListRecipesHandler($this->recipeRepository);
59
60          $this->result = $handler->handle($query);
61      }
62
63      /**
64       * @Then I should see an empty list
65       */
66      public function iShouldSeeAnEmptyList()
67      {
68          $recipes = $this->result->getRecipes();
69
70          Assert::assertInternalType('array', $recipes);
71          Assert::assertEmpty($recipes);
72      }
73
74      /**
75       * @Given there's a recipe for :arg1 by user :arg2 with :arg3 stars
76       */
77      public function theresARecipeForByUserWithStars($arg1, $arg2, $arg3)
78      {
79          throw new PendingException();
80      }
81
82      /**
83       * @Then I should see a list of recipes containing:
```

```
84          */
85      public function iShouldSeeAListOfRecipesContaining(TableNode $table)
86      {
87          throw new PendingException();
88      }
89  }
```

The thinking I have used here goes something like this:

> In order to list recipes we'll create a query object, then somehow we'll process that query to get the result. This process will involve fetching all existing recipes and returning the result.

The first line of our test states: *"Given there are no recipes"*. We're going to use the *Repository* design pattern for the storing of objects. So, to make this test pass, we've got to ensure that the *Repository* for storing recipes is empty. I've also stated that we're not going to worry about what storage system we will be using until later. So in the mean time, we can create a simple test repository, which we'll use to emulate the repository functionality. I've decided to name this `CocktailRater\Testing\Repository\TestRecipeRepository`.

With this information, the first thing we need to do is create an instance of this repository. I've done this in the `beforeScenario` method in the `FeatureContext`.

Annotations

You may have noticed that I've added `@BeforeScenario` to the docblock for this method. This is known as an *annotation* and is required to inform Behat to run this method before it runs each *scenario*.

Annotation strings in the docblock start with the `@` symbol. Behat uses annotations for several things - you will see that each snippet function has a `@Given`, `@When` or `@Then` annotation. Again, this is not just a comment, but is actually required by Behat in order to work.

Then, in the `thereAreNoRecipes` method, we *clear* the repository to ensure there are no recipes currently stored.

The next line of the test states: *"When I request a list of recipes"*. For this we create the query object, run it and store the result. I've decided that the *running* of the query will be done by a *query handler*, and therefore, we'll use the verb *handle* to run it. Also, we know that the *query handler* will need to fetch recipes from the *repository*, so we pass this to the *handler* via the constructor. All this is put into action in the `iRequestAListOfRecipes` method in the `FeatureContext`.

Finally, the last line of the test says: *"Then I should see an empty list"*. To make this pass, we'll simple check the value in the query result. In order to make a Behat snippet fail, it must throw an exception.

However, rather than writing our own checking methods, we can make use of the *assert* methods provided by PHPUnit. For this test we've used 2 asserts, one to check the result is an *array*, and the second to check it's empty.

At this point, if you try to run Behat you'll see PHP error messages saying we've referenced classes which don't exist. To fix this lets add the classes…

Writing the Code

The first line of the test requires the *repository*, and that it has a method called `clear`. Let's start by creating that:

src/Testing/Repository/TestRecipeRepository.php

```
1  <?php
2
3  namespace CocktailRater\Testing\Repository;
4
5  use CocktailRater\Domain\Repository\RecipeRepository;
6
7  final class TestRecipeRepository
8  {
9      public function clear()
10     {
11     }
12 }
```

Final

You may have noticed the use of the `final` keyword. For now I'm just going to say that I add this to my classes by default, this is not required but is my preference. I'll explain the reason for this a bit later on.

Next up let's create the `ListRecipesQuery`. A query class will contain the parameters for the query. In this case there are none, so the class simply looks like this:

src/Application/Query/ListRecipesQuery.php

```
<?php

namespace CocktailRater\Application\Query;

final class ListRecipesQuery
{
}
```

Now for the interesting bit: the `ListRecipesHandler`. From looking at the `FeatureContext`, this needs to take a *repository* as a constructor parameter, the *query* as a parameter to the `handle` method, and return some object which has a `getRecipes` method.

Here we don't want to depend on our test repository, so we'll create an *interface* which will be used in its place. For the return value, we'll create a class called `CocktailRater\Application\Query\ListRecipesResult`.

Without further ado, here it is:

src/Application/Query/ListRecipesHandler.php

```
<?php

namespace CocktailRater\Application\Query;

use CocktailRater\Domain\Repository\RecipeRepository;

final class ListRecipesHandler
{
    public function __construct(RecipeRepository $repository)
    {
    }

    /** @return ListRecipesResult */
    public function handle(ListRecipesQuery $query)
    {
        return new ListRecipesResult();
    }
}
```

At this point we've created all the classes that were referenced from the `FeatureContext`, but this last one has just introduced 2 more: the `RecipeRepository` and the `ListRecipesResult`. Let's add them to the project also (this is what I was referring to when I said we'd work from the *outside in*):

src/Application/Query/ListRecipesResult.php

```
<?php

namespace CocktailRater\Application\Query;

final class ListRecipesResult
{
    /** @return array */
    public function getRecipes()
    {
        return [];
    }
}
```

src/Domain/Repository/RecipeRepository.php

```
<?php

namespace CocktailRater\Domain\Repository;

interface RecipeRepository
{
}
```

The `ListRecipesResult` class simply returns an empty list from `getRecipes`. This is all it needs to do to make the test pass.

The `RecipeRepository` interface currently has no methods. This is because the only method currently existing in our test repository is `clear`, however this method is only relevant for the tests so there is no requirement for it in the actual application.

Now there's only one thing left to do. The `ListRecipesHandler` class requires a `RecipeRepository` to be provided to the constructor, but in the `FeatureContext` we've provided a `TestRecipeRepository`. To make this work we need to make the test repository implement the interface:

src/Testing/Repository/TestRecipeRepository.php

```php
<?php

namespace CocktailRater\Testing\Repository;

use CocktailRater\Domain\Repository\RecipeRepository;

final class TestRecipeRepository implements RecipeRepository
{
    public function clear()
    {
    }
}
```

At this point, we should be able to run Behat and see the first *scenario* pass:

```
$ behat
Feature: A visitor can view a list of recipes
    In order to view a list of recipes
    As a visitor
    I need to be able get a list of recipes

  Scenario: View an empty list of recipes
    Given there are no recipes
    When I request a list of recipes
    Then I should see an empty list

  Scenario: Viewing a list with 1 recipe
    Given there's a recipe for "Mojito" by user "tom" with 5 stars
      TODO: write pending definition
    When I request a list of recipes
    Then I should see a list of recipes containing:
      | name   | rating | user |
      | Mojito | 5.0    | tom  |

  Scenario: Recipes are sorted by rating
    Given there's a recipe for "Daquiri" by user "clare" with 4 stars
      TODO: write pending definition
    And there's a recipe for "Pina Colada" by user "jess" with 2 stars
    And there's a recipe for "Mojito" by user "tom" with 5 stars
    When I request a list of recipes
```

```
    Then I should see a list of recipes containing:
      | name        | rating | user  |
      | Mojito      | 5.0    | tom   |
      | Daquiri     | 4.0    | clare |
      | Pina Colada | 2.0    | jess  |

3 scenarios (1 passed, 2 pending)
11 steps (3 passed, 2 pending, 6 skipped)
0m0.36s (9.95Mb)
```

Scenario: View a list with 1 recipe

We got the first scenario to pass without adding any real logic. To get the next one to pass we need to start filling in some of the blanks that we've created.

Updating the FeatureContext

Just like last time, we can start by adding some content to our 2 remaining methods in the `FeatureContext`. Here I'd just like to point out that you may find it easier to work with one at a time, but for the sake of not making this book too long, I'm condensing the processes down a bit.

features/bootstrap/FeatureContext.php

```php
<?php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Behat\Tester\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;
use CocktailRater\Application\Query\ListRecipes;
use CocktailRater\Application\Query\ListRecipesHandler;
use CocktailRater\Application\Query\ListRecipesQuery;
use CocktailRater\Application\Query\ListRecipesQueryHandler;
use CocktailRater\Domain\CocktailName;
use CocktailRater\Domain\Rating;
use CocktailRater\Domain\Recipe;
use CocktailRater\Domain\User;
use CocktailRater\Domain\Username;
use CocktailRater\Testing\Repository\TestRecipeRepository;
use PHPUnit_Framework_Assert as Assert;

/**
```

```
 * Behat context class.
 */
class FeatureContext implements SnippetAcceptingContext
{
    // ...

    /**
     * @Given there's a recipe for :name by user :user with :rating stars
     */
    public function theresARecipeForByUserWithStars($name, $user, $rating)
    {
        $this->recipeRepository->store(
            new Recipe(
                new CocktailName($name),
                new Rating($rating),
                new User(new Username($user))
            )
        );
    }

    /**
     * @Then I should see a list of recipes containing:
     */
    public function iShouldSeeAListOfRecipesContaining(TableNode $table)
    {
        $callback = function ($recipe) {
            return [
                (string) $recipe['name'],
                (float) $recipe['rating'],
                (string) $recipe['user']
            ];
        };

        Assert::assertEquals(
            array_map($callback, $this->result->getRecipes()),
            array_map($callback, $table->getHash())
        );
    }
}
```

In `theresARecipeForByUserWithStars` we're creating a new `Recipe` object. The `Recipe` needs a name, rating and user, so we can add what we think look like sensible dependencies via the

constructor. We also *store* this new object in the repository.

In the `iShouldSeeAListOfRecipesContaining` method, we compare the results returned from the query, with the table of expected results, using PHPUnit's `assertEquals`. I've also used `array_map` to ensure both arrays contain the same types since all values in Behat tables are strings.

Adding new Classes to the Model

Unit Tests

Before continuing I'd just like to point out that up until this point I've not created any unit tests. From this point on I'll be using them for all development in the *domain model*. However, I won't be showing them or the process of creating them, as it would take up too many pages. However, they're all available in the example code for the book if you want to study them.

Let's start off by adding the new classes to the model:

src/Domain/Recipe.php

```
1  <?php
2
3  namespace CocktailRater\Domain;
4
5  final class Recipe
6  {
7      /** @param string $name */
8      public function __construct(CocktailName $name, Rating $rating, User $user)
9      {
10     }
11 }
```

src/Domain/CocktailName.php

```
1  <?php
2
3  namespace CocktailRater\Domain;
4
5  final class CocktailName
6  {
7      /** @var string $value */
8      public function __construct($value)
```

```
    {
    }
}
```

src/Domain/Rating.php

```
<?php

namespace CocktailRater\Domain;

final class Rating
{
    /** @var float $value */
    public function __construct($value)
    {
    }
}
```

src/Domain/User.php

```
<?php

namespace CocktailRater\Domain;

final class User
{
    /** @var string $username */
    public function __construct(Username $username)
    {
    }
}
```

src/Domain/Username.php

```
<?php

namespace CocktailRater\Domain;

final class Username
{
    /** @param string $value */
    public function __construct($value)
    {
    }
}
```

We also need to add the `store` method to the repository interface:

src/Domain/Repository/RecipeRepository.php

```
<?php

namespace CocktailRater\Domain\Repository;

use CocktailRater\Domain\Recipe;

interface RecipeRepository
{
    public function store(Recipe $recipe);
}
```

This also means that we need to add the method to the `TestRecipeRepository`:

src/Testing/Repository/TestRecipeRepository.php

```
<?php

namespace CocktailRater\Testing\Repository;

use CocktailRater\Domain\Recipe;
use CocktailRater\Domain\Repository\RecipeRepository;

final class TestRecipeRepository implements RecipeRepository
{
```

```
10      public function store(Recipe $recipe)
11      {
12      }
13
14      public function clear()
15      {
16      }
17  }
```

Making the Scenario Pass

At this point, only the last line of the scenario should be failing. We've got the template of the model laid out, so we just need to fill in the details. To start with, let's take a look at how the query handler will work:

src/Application/Query/ListRecipesHandler.php

```
1   <?php
2
3   namespace CocktailRater\Application\Query;
4
5   use CocktailRater\Domain\Repository\RecipeRepository;
6
7   final class ListRecipesHandler
8   {
9       /** @var RecipeRepository */
10      private $repository;
11
12      public function __construct(RecipeRepository $repository)
13      {
14          $this->repository = $repository;
15      }
16
17      /** @return ListRecipesResult */
18      public function handle(ListRecipesQuery $query)
19      {
20          $result = new ListRecipesResult();
21
22          foreach ($this->repository->findAll() as $recipe) {
23              $result->addRecipe(
24                  $recipe->getName()->getValue(),
25                  $recipe->getRating()->getValue(),
26                  $recipe->getUser()->getUsername()->getValue()
```

```
            );
        }

        return $result;
    }
}
```

It's quite simple really: it fetches all recipes from the repository, adds the details of each one to the result object, then returns the result. This looks good, but we've got a bit of work to do to get it all working. First up let's update the classes in the *domain model*:

src/Domain/Recipe.php

```
<?php

namespace CocktailRater\Domain;

final class Recipe
{
    /** @var CocktailName */
    private $name;

    /** @var Rating */
    private $rating;

    /** @var User */
    private $user;

    /** @param string $name */
    public function __construct(CocktailName $name, Rating $rating, User $user)
    {
        $this->name   = $name;
        $this->rating = $rating;
        $this->user   = $user;
    }

    /** @return CocktailName */
    public function getName()
    {
        return $this->name;
    }

    /** @return Rating */
```

```
    public function getRating()
    {
        return $this->rating;
    }

    /** @return User */
    public function getUser()
    {
        return $this->user;
    }
}
```

src/Domain/CocktailName.php

```
<?php

namespace CocktailRater\Domain;

use Assert\Assertion;

final class CocktailName
{
    /** @var string */
    private $value;

    /** @param string $value */
    public function __construct($value)
    {
        Assertion::string($value);

        $this->value = $value;
    }

    /** @return string */
    public function getValue()
    {
        return $this->value;
    }
}
```

src/Domain/Rating.php

```php
<?php

namespace CocktailRater\Domain;

use Assert\Assertion;
use CocktailRater\Domain\Exception\OutOfBoundsException;

final class Rating
{
    /** @var float */
    private $value;

    /**
     * @var float $value
     *
     * @throws OutOfBoundsException
     */
    public function __construct($value)
    {
        Assertion::numeric($value);

        $this->assertValueIsWithinRange($value);

        $this->value = (float) $value;
    }

    /** @return float */
    public function getValue()
    {
        return $this->value;
    }

    /**
     * @var float $value
     *
     * @throws OutOfBoundsException
     */
    private function assertValueIsWithinRange($value)
    {
        if ($value < 1 || $value > 5) {
            throw OutOfBoundsException::numberIsOutOfBounds($value, 1, 5);
```

```
        }
    }
}
```

src/Domain/User.php

```php
<?php

namespace CocktailRater\Domain;

final class User
{
    /** @var Username */
    private $username;

    /**
     * @param string $username
     *
     * @return User
     */
    public static function fromValues($username)
    {
        return new self(new Username($username));
    }

    /** @var string $username */
    public function __construct(Username $username)
    {
        $this->username = $username;
    }

    /** @return Username */
    public function getUsername()
    {
        return $this->username;
    }
}
```

src/Domain/Username.php

```
<?php

namespace CocktailRater\Domain;

use Assert\Assertion;

final class Username
{
    /** @var string */
    private $value;

    /** @param string $value */
    public function __construct($value)
    {
        Assertion::string($value);

        $this->value = $value;
    }

    /** @param */
    public function getValue()
    {
        return $this->value;
    }
}
```

In the *domain model*, we've started to make use of Benjamin Eberlei's *Assert*[53] library. For this to work we need to install the dependency with Composer by running:

```
$ composer require beberlei/assert:@stable
```

[53] https://github.com/beberlei/assert

Using 3rd Party Libraries in the Domain Model

Adding a dependency to a 3rd party library is something that should not be done without serious consideration. A better approach is to use *Inversion of Control* to make the model depend on the library via a layer of abstraction. The Adapter[54] design pattern is a very good tool for this job.

So, with that said, why am I using the *Assert* library from within the domain model? The reason is: firstly it's a well-used and stable library made up of utility methods which have no side effects. Secondly, and more importantly, I'm using it in a way which adds, what I think, is a missing feature in the PHP language: namely typehints for scalar types and arrays.

There is an interesting discussion with Mathais Verraes on the DDDinPHP Google Group[55] about adding dependencies to 3rd party libraries to your domain model. However, the bottom line here is: before doing this you should exercise extreme consideration of what you are about to do.

One thing which may have caught your eye in the `User` class is the `fromValues` *static* method. This is known as a *named constructor*. It's a way in which we can provide alternate constructors for classes, and is one of the few valid uses of the `static` keyword. Since it maintains no state, and works in a purely *functional* way, it is a safe use of `static`. At this point `fromValues` has only been used in the unit tests, even so, I felt the neater tests were a good enough reason to add it.

Another thing we have done here, is restricted the value allowed for a rating to be between 1 and 5. If it falls outside of this range, we throw an exception. The appropriate exception to be throw here is PHP SPL's `OutOfBoundsException`. However, rather than throw it directly, we've extended it so that it can be tracked down as coming from our application. Let's take a quick look at it:

src/Domain/Exception/OutOfBoundsException.php

```
1  <?php
2
3  namespace CocktailRater\Domain\Exception;
4
5  class OutOfBoundsException extends \OutOfBoundsException
6  {
7      /**
8       * @param number $number
9       * @param number $min
10      * @param number $max
11      *
12      * @return OutOfBoundsException
13      */
```

[54] http://en.wikipedia.org/wiki/Adapter_pattern

[55] https://groups.google.com/forum/#!msg/dddinphp/YGogT1NSbO0/u22c4dgoxdEJ

```
    public static function numberIsOutOfBounds($number, $min, $max)
    {
        return new static(sprintf(
            'The number %d is out of bounds, expected a number between %d and %d\
.',
            $number,
            $min,
            $max
        ));
    }
}
```

Again you'll notice the use of a *named constructor*. I think this is a really neat way to keep the exception messages neat and tidy, and in a relevant place.

Next, let's quickly update the `ListRecipesResult` class:

src/Application/Query/ListRecipesResult.php

```
<?php

namespace CocktailRater\Application\Query;

final class ListRecipesResult
{
    /** @var array */
    private $recipes = [];

    /**
     * @param string $name
     * @param float  $rating
     * @param string $username
     */
    public function addRecipe($name, $rating, $username)
    {
        $this->recipes[] = [
            'name'     => $name,
            'rating'   => $rating,
            'user' => $username
        ];
    }

    /** @return array */
```

```
25      public function getRecipes()
26      {
27          return $this->recipes;
28      }
29  }
```

Finally, we need to update the functionality of the repository to return the list of recipes stored:

src/Domain/Repository/RecipeRepository.php

```
1   <?php
2
3   namespace CocktailRater\Domain\Repository;
4
5   use CocktailRater\Domain\Recipe;
6
7   interface RecipeRepository
8   {
9       public function store(Recipe $recipe);
10
11      /** @return Recipe[] */
12      public function findAll();
13  }
```

src/Testing/Repository/TestRecipeRepository.php

```
1   <?php
2
3   namespace CocktailRater\Testing\Repository;
4
5   use CocktailRater\Domain\Recipe;
6   use CocktailRater\Domain\Repository\RecipeRepository;
7
8   final class TestRecipeRepository implements RecipeRepository
9   {
10      /** @var Recipe[] */
11      private $recipes = [];
12
13      public function store(Recipe $recipe)
14      {
15          $this->recipes[] = $recipe;
16      }
```

```
17
18      public function findAll()
19      {
20          return $this->recipes;
21      }
22
23      public function clear()
24      {
25      }
26  }
```

As you can see, we've created an in-memory test repository. This is good enough for what we need so far.

You can now run Behat and watch the second scenario pass.

```
1   $ behat
2   Feature: A visitor can view a list of recipes
3       In order to view a list of recipes
4       As a visitor
5       I need to be able get a list of recipes
6
7     Scenario: View an empty list of recipes
8       Given there are no recipes
9       When I request a list of recipes
10      Then I should see an empty list
11
12    Scenario: Viewing a list with 1 recipe
13      Given there's a recipe for "Mojito" by user "tom" with 5 stars
14      When I request a list of recipes
15      Then I should see a list of recipes containing:
16        | name   | rating | user |
17        | Mojito | 5.0    | tom  |
18
19    Scenario: Recipes are sorted by rating
20      Given there's a recipe for "Daquiri" by user "clare" with 4 stars
21      And there's a recipe for "Pina Colada" by user "jess" with 2 stars
22      And there's a recipe for "Mojito" by user "tom" with 5 stars
23      When I request a list of recipes
24      Then I should see a list of recipes containing:
25        | name        | rating | user  |
26        | Mojito      | 5.0    | tom   |
```

```
       | Daquiri     | 4.0    | clare |
       | Pina Colada | 2.0    | jess  |
       Failed asserting that two arrays are equal.
       --- Expected
       +++ Actual
       @@ @@
        Array (
            0 => Array (
       +        0 => 'Mojito'
       +        1 => 5.0
       +        2 => 'tom'
       +    )
       +    1 => Array (
                0 => 'Daquiri'
                1 => 4.0
                2 => 'clare'
            )
       -    1 => Array (
       +    2 => Array (
                0 => 'Pina Colada'
                1 => 2.0
                2 => 'jess'
       -    )
       -    2 => Array (
       -        0 => 'Mojito'
       -        1 => 5.0
       -        2 => 'tom'
            )
        )

--- Failed scenarios:

    features/visitors-can-list-recipes.feature:18

3 scenarios (2 passed, 1 failed)
11 steps (10 passed, 1 failed)
0m0.05s (10.60Mb)
```

Scenario: Recipes are sorted by rating

You may have already noticed, that when you run Behat now most of our final scenario already passes, The only thing which fails is the order in which the recipes are listed. To fix this we can go

straight into the `ListRecipesHandler`, and sort the recipes there:

src/Application/Query/ListRecipesHandler.php

```php
<?php

namespace CocktailRater\Application\Query;

use CocktailRater\Domain\Repository\RecipeRepository;
use CocktailRater\Domain\Recipe;

final class ListRecipesHandler
{
    /** @var RecipeRepository */
    private $repository;

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

    /** @return ListRecipesResult */
    public function handle(ListRecipesQuery $query)
    {
        $result = new ListRecipesResult();

        foreach ($this->getAllRecipesSortedByRating() as $recipe) {
            $result->addRecipe(
                $recipe->getName()->getValue(),
                $recipe->getRating()->getValue(),
                $recipe->getUser()->getUsername()->getValue()
            );
        }

        return $result;
    }

    private function getAllRecipesSortedByRating()
    {
        $recipes = $this->repository->findAll();

        usort($recipes, function (Recipe $a, Recipe $b) {
            return $a->isHigherRatedThan($b) ? -1 : 1;
```

```
        });

        return $recipes;
    }
}
```

We also need to add new comparison methods to both the `Recipe` and `Rating` classes:

src/Domain/Recipe.php

```
<?php

namespace CocktailRater\Domain;

use Assert\Assertion;

final class Recipe
{
    // ...

    /** @return bool */
    public function isHigherRatedThan(Recipe $other)
    {
        return $this->rating->isHigherThan($other->rating);
    }
}
```

src/Domain/Rating.php

```
<?php

namespace CocktailRater\Domain;

use Assert\Assertion;
use CocktailRater\Domain\Exception\OutOfBoundsException;

final class Rating
{
    // ...

    /** @return bool */
    public function isHigherThan(Rating $other)
```

```
14      {
15          return $this->value > $other->value;
16      }
17  }
```

That's it, the first feature is done!

```
1   $ behat
2   Feature: A visitor can view a list of recipes
3       In order to view a list of recipes
4       As a visitor
5       I need to be able get a list of recipes
6
7     Scenario: View an empty list of recipes
8       Given there are no recipes
9       When I request a list of recipes
10      Then I should see an empty list
11
12    Scenario: Viewing a list with 1 recipe
13      Given there's a recipe for "Mojito" by user "tom" with 5 stars
14      When I request a list of recipes
15      Then I should see a list of recipes containing:
16        | name   | rating | user |
17        | Mojito | 5.0    | tom  |
18
19    Scenario: Recipes are sorted by rating
20      Given there's a recipe for "Daquiri" by user "clare" with 4 stars
21      And there's a recipe for "Pina Colada" by user "jess" with 2 stars
22      And there's a recipe for "Mojito" by user "tom" with 5 stars
23      When I request a list of recipes
24      Then I should see a list of recipes containing:
25        | name        | rating | user  |
26        | Mojito      | 5.0    | tom   |
27        | Daquiri     | 4.0    | clare |
28        | Pina Colada | 2.0    | jess  |
29
30  3 scenarios (3 passed)
31  11 steps (11 passed)
32  0m0.04s (10.48Mb)
```

Tidying Up

Now the feature is complete, let's take a little look and see if there's anything we can do to make the code a bit better.

The main thing which needs to be improved here is chaining of methods in the *query handler*. We've created have ugly lines of code like this:

```
$recipe->getUser()->getUsername()->getValue()
```

Big chains of method calls like this violate the Law of Demeter[56] which states: *you should only talk to your immediate friends*. This means you should **only** call methods or access properties of objects which are: properties of the current class, are parameters to the current method, or have been created inside the method. This law is pretty much stating the same thing as the *one dot per line* rule of Object Calisthenics. Note that PHP requires the use of `$this->` to call methods and access properties, so it's actually *two arrows per line*.

So, how to this issue? One approach might be to ask the top level class (*aggregate*) to ask the next level down to return the value, repeating down the hierarchy. Here's an example:

```
class Recipe
{
    // ...

    public function getUsername()
    {
        return $this->user->getUsernameValue();
    }
}

class User
{
    // ...

    public function getUsernameValue()
    {
        return $this->username->getValue();
    }
}
```

However, if you're going to do this for more than 2 or 3 values, the interface is going to start to get pretty bloated. Another way might be to add a method to the `Recipe` class to return all its

[56] http://en.wikipedia.org/wiki/Law_of_Demeter

values as an array or value object. There are other ways you could do this, but for this project let's use a combination of these 2 methods. If only 1 or 2 getters are required we'll consider using them, otherwise we'll use a *read* method to return an object (I prefer objects to arrays because, even though they require extra code, the content is well defined and they can be immutable, However, using an array or object with public properties, might be appropriate for your project).

Exposing Recipe Values

With this in mind, let's expose the contents of the `Recipe` class via a details value object. We do this by creating 2 new classes, one for `Recipe` and one for `User`:

src/Domain/RecipeDetails.php

```
1  <?php
2
3  namespace CocktailRater\Domain;
4
5  final class RecipeDetails
6  {
7      /** @var CocktailName */
8      private $name;
9
10     /** @var UserDetails */
11     private $user;
12
13     /** @var Rating */
14     private $rating;
15
16     public function __construct(
17         CocktailName $name,
18         UserDetails $user,
19         Rating $rating
20     ) {
21         $this->name   = $name;
22         $this->user   = $user;
23         $this->rating = $rating;
24     }
25
26     /** @return string */
27     public function getName()
28     {
29         return $this->name->getValue();
30     }
```

```
31
32      /** @return string */
33      public function getUsername()
34      {
35          return $this->user->getUsername();
36      }
37
38      /** @return float */
39      public function getRating()
40      {
41          return $this->rating->getValue();
42      }
43  }
```

src/Domain/UserDetails.php

```
1   <?php
2
3   namespace CocktailRater\Domain;
4
5   final class UserDetails
6   {
7       /** @var Username */
8       private $username;
9
10      public function __construct(Username $username)
11      {
12          $this->username = $username;
13      }
14
15      /** @return Username */
16      public function getUsername()
17      {
18          return $this->username->getValue();
19      }
20  }
```

And add the following method to `User` and `Recipe`:

CocktailRater/Domain/Recipe.php

```
    /** @return RecipeDetails */
    public function getDetails()
    {
        return new RecipeDetails(
            $this->name
            $this->user->getDetails(),
            $this->rating
        );
    }
```

CocktailRater/Domain/User.php

```
    /** @return UserDetails */
    public function getDetails()
    {
        return new UserDetails($this->username);
    }
```

Then we can update our query handler to use these like so:

CocktailRater/Application/Query/ListRecipesHandler.php

```
    /** @return ListRecipesResult */
    public function handle(ListRecipesQuery $query)
    {
        $result = new ListRecipesResult();

        foreach ($this->getAllRecipesSortedByRating() as $recipe) {
            $details = $recipe->getDetails();

            $result->addRecipe(
                $details->getName(),
                $details->getRating(),
                $details->getUsername()
            );
        }

        return $result;
    }
```

At this point, we can also remove `getUsername` from the `User` class and `getName`, `getRating` and `getUser` from the `Recipe` class.

Already this is looking a lot neater, but we're still violating the *law of demeter* at 2 levels in the *handler*. Firstly, we're calling `getDetails` on a `Recipe` objects which are not an *immediate friends* of the handler (since they fetched from a repository). Secondly, we're calling the *get* methods on the details object returned from the `Recipe` objects. Considering this is happening just at the application layer, I don't really think this is the biggest crime and therefore could be left as is. That said, let's still try to tidy it up some more.

To do this, let's get rid of all the calls to the *getters* on the details objects. We can do this by simply passing in the details object to the result class constructor. The problem with this is that is adds a dependency on the *domain model* from anywhere that a result object is used. When using a language like Java, C++ or C#, this becomes something that really needs to be fixed, since separate packages need to be able to be compiled and deployed independently. However, PHP doesn't work like that (maybe one day it will). Even so, it's probably still good practice to work this way. Also, since we don't want any other layers which talk to the application, to create *result* objects, let's make the *result* into an *interface*. Then we can have a concrete result *Data Transfer Object*, which can know about the *details* class. Because the dependency from outside is now on the interface only, it's decoupled form the *domain*.

Here's the updated code:

src/Application/Query/ListRecipesResult.php

```
1  <?php
2
3  namespace CocktailRater\Application\Query;
4
5  interface ListRecipesResult
6  {
7      /** @return array */
8      public function getRecipes();
9  }
```

src/Application/Query/ListRecipesResultData.php

```php
<?php

namespace CocktailRater\Application\Query;

use Assert\Assertion;
use CocktailRater\Domain\RecipeDetails;

final class ListRecipesResultData implements ListRecipesResult
{
    /** @var RecipeDetails[] */
    private $recipes = [];

    public function __construct(array $recipes)
    {
        Assertion::allIsInstanceOf($recipes, RecipeDetails::class);

        $this->recipes = $recipes;
    }

    /** @return array */
    public function getRecipes()
    {
        return array_map(
            function (RecipeDetails $recipe) {
                return [
                    'name'   => $recipe->getName(),
                    'rating' => $recipe->getRating(),
                    'user'   => $recipe->getUsername()
                ];
            },
            $this->recipes
        );
    }
}
```

CocktailRater/Application/Query/ListRecipesHandler.php

```
    /** @return ListRecipesResult */
    public function handle(ListRecipesQuery $query)
    {
        return new ListRecipesResultData(
            array_map(function (Recipe $recipe) {
                return $recipe->getDetails();
            },
            $this->getAllRecipesSortedByRating())
        );
    }
```

That's almost done! The handler is much neater. But we've still not quite conformed to the *Law of Demeter*, because it still gets the recipe from the *repository*. In most circumstances, particularly in the *domain model*, I'm very diligent about obeying the *Law of Demeter*. However, in this circumstance, I feel we've done enough. A good exercise is, to consider how to obey it completely in the handler, but for now I'm going to leave it as it is.

Have we gone too far?

You might be thinking to yourself that this is all a bit excessive. That we have an *aggregate*, which returns a details value object, which is then copied into a results DTO, which looks almost the same as the value object, and we have an extra interface to describe the *result* DTO. You might also think that simply passing back the *details* value object from the handler would be sufficient. Or, that even that would be too much, and the *details* class is overkill, and a simple associative array would have done. You may even be thinking this looks far too much like Java.

If you are thinking any of these things you are right! None of this is necessary. But, depending on the scale of the project, how many people are going to be working with the code, the growth expectancy of the project, and even the budget, this level of detail may be extremely valuable. What we've done here is apply best practices, we've made the code as explicit and self documented as possible. As a result future developers (and our future selves) will thank us for this.

What Next?

So far we've managed to get the first feature's tests to pass. However, we've done it in quite an isolated way by considering this single *query* on its own. In the next chapter we'll quickly add the second feature, then we can analyse the two to find similarities. We'll then use this knowledge to *refactor* what we have into a more generic form. After that we'll try to display the application's output on a page.