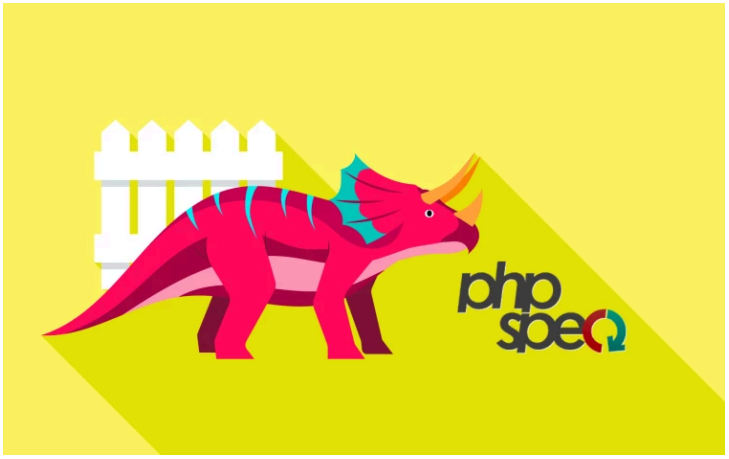
phpspec: Testing... *Designing* with a Bite



Chapter 1: Installing phpspec

Yo friends! Oh, SO glad you've made it for our phpspec tutorial! You will *not* regret it. Thing number one to know about phpspec is: it's... just... fun!

Ok, but what *is* phpspec? It's a unit testing tool... exactly like `phpunit`. Wait... that's not totally right. If you watched our PHPUnit tutorial, then you know that PHPUnit is a perfectly fine tool for unit testing your code. So... why are we even talking about phpspec?

Here's the truth: yes, phpspec *is* a tool for unit testing your code. But, that's *not* its main job. Nope, it's a tool for helping you *design* your code in a well-organized, meaningful and maintainable way. You probably already think about the design and user experience of your front-end. But, have you ever thought about the design and experience of your PHP classes?

That's phpspec's job. And yes, as a nice by-product, you *will* get unit tests. And as a *nicer* by-product, you will also *enjoy* the process - coding with phpspec is fun. Oh, and later - we'll talk about how phpspec & PHPUnit fit together - like should we use *both* in the same app? Short answer: yes!

Starting Point: Empty Project!

Ok, let's go! Just like in our PHPUnit tutorial, we're going to design & build a dinosaur park - complete with T-Rex, stegosaurus, enclosures for our dinosaurs *and*, with any lucky, some security systems that - thanks to our tests - won't fail as soon as a storm rolls in or a developer leaves early for lunch.

To make sure our dinosaurs don't *once* again rule the Earth, you should *totally* code along with me. Download the course code from this page. When you unzip it, you'll find a `start/` directory with the same code that you see here. But... well... what we have here is... nothing! Just an empty project with a `composer.json` file that also... has nothing important inside. This `tutorial` directory *does* have a few files that we'll use later - so make sure you have it.

We're starting with an empty project because phpspec is *truly* a framework-agnostic library. But don't worry - if you're a Symfony user, we'll build a structure that will be very familiar to you - with the same directories and namespaces as a Symfony app.

Installing phpspec

To get phpspec installed, open a terminal, move into the project, close Facebook, and run:

```
composer require phpspec/phpspec --dev
```

And.... ding! Just like with PHPUnit, installing phpspec means that you get a new, shiny executable! Run:

```
./vendor/bin/phpspec
```

The `phpspec` executable really only has two commands: `describe` and `run`. And we'll talk about both of them very soon.

Configuration autoload in composer.json

But first, we need just a *little* bit of configuration to get things working. The first piece of configuration... has *nothing* to do with phpspec at all! Our app has *no* PHP classes yet. But when we add some, I want to put them in the `src/` directory and prefix each namespace with `App`. That will be exactly like a Symfony project.

Open `composer.json`. To make sure Composer's autoloader knows where our classes live, we need to add some config here. This is code that you *normally* get automatically when you start, for example, a new Symfony project. But I want to show how it's done by hand so that we can *truly* understand what's going on behind the scenes.

Add `autoload`, then `psr-4`, then say that classes starting with `App\\` will live in the `src/` directory.

composer.json

```
{
// ... lines 2 - 6
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

To make Composer notice this change, find your terminal and run:

```
composer dump-autoload
```

Autoloading... done!

Configuring phpspec

Next, one of my *favorite* things about phpspec is that it generates code for you! But to do that, it *also* needs to know that our classes will live in the `src/` directory and that each namespace will start with `App`. Unfortunately, phpspec can't automatically get all this info from `composer.json`, but it's no problem.

Create a `phpspec.yml` file at the root of the project - `phpspec` automatically knows to look for this. Inside add `suites` then `default`. Like most testing tools, you can organize your tests into multiple groups, or "suites" if you want. In this tutorial, we'll stick to using the one, "default" suite.

phpspec.yml

```
suites:
    default:
// ... lines 3 - 5
```

Under this, add `namespace: App` - because all of our classes will start with the `App` namespace - and `psr4_prefix: App`. Those two lines are enough to help phpspec know *where* to generate our files.

phpspec.yml

```
// ... lines 1 - 2
        namespace: App
        psr4_prefix: App
```

And... team, we're ready to go! Next, let's create our first *specification*... ooOOOOooo. That's the file where we will *describe* how a single class should behave by writing *examples*. Woh.

Chapter 2: Buzzwords! Specification & Examples

So if phpspec is all about helping you design your classes - helping you ask: how do I want this class to look and behave? - how... does it actually do that? The idea is cool: instead of jumping straight into your code and hacking until something works... or you get sleepy... stop... step back... and instead, *first*, *describe* how you want your class to *behave*.

We do that by creating a class called a... *specification*. That's a fancy... or maybe *boring* word that means that, before coding, we will *first* create a class where we simply *describe* how our future class will work and act.

Generating the Specification

Let's see this in action. Remember the two commands of the phpspec executable? The first is `describe` - run it with `-h`:

```
./vendor/bin/phpspec describe -h
```

I'm passing `-h` to see the help details. Basically, each time you want to create a new class, you should *first* use this command to create a corresponding *specification* class. Oh, notice that forward slashes are used for the namespaces, that's just to avoid escaping problems.

Anyways, because we're building a dinosaur park, the *first* class we need is... `Dinosaur`! So let's run:

```
./vendor/bin/phpspec describe App/Entity/Dinosaur
```

I could have chosen any namespace starting with `App` - that's up to how you want to organize your code. But, if you're used to Doctrine in Symfony, this will feel familiar.

What are Examples?

Ok! One new file: `DinosaurSpec.php`. Let's go check it out! Ok - so `phpspec` creates a `spec/` directory, which is meant to have the same file structure that our classes will eventually have in `src/`.

Open the new file. Ok... these spec classes look a little weird at first - and we're going to talk a *lot* about them. The *purpose* of this class is for us to describe the behavior of our future `Dinosaur` class. On a philosophical level, we do this by writing example code: using our `Dinosaur` class as if it already existed and was finished.

spec/Entity/DinosaurSpec.php

```
// ... lines 1 - 8
class DinosaurSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType(Dinosaur::class);
    }
}
```

On a more concrete level: we describe the behavior through *examples*. Every function in this class that starts with `it_` or `its_` will be read by phpspec as an "example". They are the *key* to phpspec, and also the most complex part.

There are *two* very important things to understand about the code inside these example methods. First, and this is *truly* magic, you're supposed to use the `$this` variable as *if* we were inside of the `Dinosaur` class itself. Literally: you treat `$this` like a `Dinosaur` object - showing *examples* of how you want it to work by calling methods on that class - like `$this->getLength()` if the `Dinosaur` class had a `getLength()` method.

Hello Matchers

In addition to using `$this` to call methods that exist - or *should* exist - inside `Dinosaur`, the *second* important thing to know is that you can *also* call a huge number of methods that start with `should` or `shouldNot`. These are called "matchers" - and they are the way you *assert* that things are working correctly in phpspec.

In the one generated example function, because we're *pretending* to be inside the `Dinosaur` class, we pretend that `$this` is a `Dinosaur` object. When we call `->shouldHaveType(Dinosaur::class)`, this *asserts* that the object is an instance of that class... which, by the way, doesn't even exist yet! It's a pretty pointless test - but I usually keep it.

Oh, and the *last* strange thing about this class is that... it violates coding standards! Did you notice the missing `public` before the functions? That's totally legal in php - methods are public by default. And the method names are written using snake-case instead of camel case. Both of these things are done on purpose for one important reason: readability. We're writing PHP - but this class is meant to be a human-readable description of our future `Dinosaur` class. And right now, our specification says nothing more than a `Dinosaur` object should be... a `Dinosaur` object.

Generating the Class with run

Ready to execute the *other* phpspec command? It's called run - let's show the help details on this one too:

```
./vendor/bin/phpspec run -h
```

This is the *main* command in phpspec. Its job is to look at all of our spec files - just one right now - and all of the *example* methods inside - and verify whether or not the actual class *behaves* like we've described with that example code.

Now... you might think that's a bit crazy. After all, how can phpspec look to see if our `Dinosaur` class has the correct "behavior"? The `Dinosaur` class doesn't even exist yet! Heck, there's nothing in our `src/` directory at all! Well... let's see what happens:

```
./vendor/bin/phpspec run
```

At first, it *does* fail because `App\Entity\Dinosaur` does not exist. That's expected. But check this out: it's asking: do you want me to create it for you? This is what makes `phpspec` so fun! When it sees that you've described some behavior that's missing, it can create it for you! Let's choose yes, of course!

src/Entity/Dinosaur.php

```
// ... lines 1 - 4
class Dinosaur
{
}
```

Cool! Go look - in `src/`... there it is! It doesn't *do* anything, but... actually... our new class *now* has the *behavior* described in our spec. To prove it, re-run phpspec:

```
./vendor/bin/phpspec run
```

Woh! It works! That... does make sense. Even though we don't understand much about how the "examples" work yet, after generating the code, if you try to create an instance of a `Dinosaur` class..... you *do* get a `Dinosaur` object! Eureka!

Next: let's start creating some meaningful examples of how our class should behave and see how phpspec can help us build that.

Chapter 3: Matchers, Examples & Generation

We have an empty `Dinosaur` class. As *proud* of that empty class as I am, I think we need to start thinking about what we need this class to actually *do* - how we want it to behave! This *totally* depends on your app - and what you need to use each class for. But... hmm... let's see. I definitely want to be able to set the length on a Dinosaur - because maybe we need to render that somewhere. Oh, and if the length is *not* set, it should probably default to be 0.

Our First Example

Wait. Right there. Did you notice? I just described an *example* of how our `Dinosaur` class should work! All *we* need to do is translate that into a phpspec example! Create a new function, start it with `it_` - because that's what phpspec requires... and also because that helps us create descriptive & readable method names. How about: `it_should_default_to_zero_length()`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 8
9  class DinosaurSpec extends ObjectBehavior
10 {
↕ // ... lines 11 - 15
16     function it_should_default_to_zero_length()
17     {
↕ // ... line 18
19     }
20 }
```

Inside, remember: the goal is to *pretend* like we're inside the `Dinosuar` class. When each example is executed, `phpspec` will instantiate a `Dinosaur` object behind the scenes and we can reference it via `$this`. That's *total* and absolute magic... and we'll get to find out *exactly* how it works a bit later.

Anyways, right now: just imagine that `$this` is a `Dinosaur` object. To show an example of how the length should be 0 by default, we will *literally* write example code. For our app, I want to be able to call a `getLength()` method to get the length. Cool! Write: `$this->getLength()` as *if* that method already existed.

And because we want our Dinosaur's length to default to 0, call a matcher function to assert that: `->shouldReturn(0)`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 15
16     function it_should_default_to_zero_length()
17     {
18         $this->getLength()->shouldReturn(0);
19     }
↕ // ... lines 20 - 21
```

Matchers!

How... weird, but *cool* is that?! At any point in phpspec, you can say `->should` to call one of phpspec's *many* "matcher" functions. These are equivalent to the `assert()` functions in PHPUnit - the difference is purely style. Instead of saying `$this->assertEquals(0, $dinosuar->getLength())` like you would in PHPUnit - *boring* - you say `$this->getLength()->shouldReturn(0)`. The whole line *reads* like a clear English sentence!

And fortunately, even though phpspec is using some legit sourcery to make this all work, PhpStorm has great support for auto-completing these matcher functions. Oh, and, by the way, every matcher *always* starts with `should` or `shouldNot`. That's just a rule - and we'll learn why later.

Find your browser and go to https://phpspec.net. Click into the manual and go to the matchers section. Nice! phpspec has a *huge* number of matchers... and someone even thought to document them! Amazing! Right now, we're using one called the **Identity Matcher**, which allows you to use `shouldBe()`, `shouldBeEqualTo()` `shouldReturn()` or `shouldEqual()`. These are all different ways to compare values using `===`.

There's also a **Comparison Matcher** where you can say `shouldBeLike()` to compare values using `==`. And there are many, many, many more to geek out over. We'll learn the most useful ones along the way.

Generating the Missing Method

We *now* have one new example of how we want the `Dinosaur` class to work. Will this example already pass? Of course not! We don't even have a `getLength()` method yet! But, run phpspec anyways - and prepared to be... amazed:

```
./vendor/bin/phpspec run
```

Yes! I love failure! The `getLength()` method is not found. Oh, but check it out! Just like before, it *realizes* that we're describing some behavior that doesn't exist and asks us if we want it to do our job for us! Of course we do!

Let's go check it out! Not bad! It generates the method but, unless your version of phpspec has become self-aware - in which case... let me know what version you're using - it has *no* idea what to put *inside* the method.

src/Entity/Dinosaur.php

```
// ... lines 1 - 4
class Dinosaur
{
    public function getLength()
    {
        // TODO: write logic here
    }
}
```

And so, after phpspec generated the code, it automatically re-ran itself, but the new example *still* fails:

> *"it should default to zero length, expected integer `0` but got `null`,"*

This is the phpspec flow! One: describe behavior with an example. Two: `phpspec` generates as much as it can. Three: we fill in the logic. Four: profit!

Filling in the Logic

And... actually, the rules of TDD say that we should fill in the method with as *little* code as possible to get the test to pass - including just hardcoding a value if you can! But... more on that craziness later.

Right now, let's fill this in for real. So, hmm... because we know each `Dinosaur` can have a different length, we will probably need a `$length` property. And, ah yes, it needs to default to 0 - that's something we decided during the "spec" or "description" process. Inside the method, `return $this->length`. Oh, and to be super-cool, add the `int` return type. Viva return types!

src/Entity/Dinosaur.php

```
↕ // ... lines 1 - 4
5  class Dinosaur
6  {
7      private $length = 0;
↕ // ... line 8
9      public function getLength(): int
10     {
11         return $this->length;
12     }
13 }
```

Our class *should* now behave like our example expects. Let's see if phpspec agrees! Run it:

```
./vendor/bin/phpspec run
```

Woohoo! It passes!

Describing setLength()

But... nobody want's to visit a dinosaur park full of dinosaurs with zero length. We need a way to *set* the length. How do we want to do that? There's no right answer: it depends on your app. For example, we could make it a constructor argument. Or, it might make sense in your app to have, for example, some `updateSpecs()` method where you pass the length along with a few other things about your dinosaur. Or, you might need a simple `setLength()` method. The *cool* thing is that phpspec forces you to *think* about this - it forces you to ask: How will this class be used? Do I really need a setLength() method? Or will the user set a bunch of details all at once, and so a more descriptive `updateSpecs()` method is more clear?

Again... there's no right or wrong answer - sorry! For our app, let's create an example showing `setLength()`: `function it_should_allow_to_set_length()`. Inside, pretending that `$this` is a `Dinosaur` object, let's show how this should work: call `$this->setLength()` and pass it, how about 9. After that, we should be able to call `$this->getLength()->shouldReturn(9)`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 8
9  class DinosaurSpec extends ObjectBehavior
10 {
↕ // ... lines 11 - 20
21     function it_should_allow_to_set_length()
22     {
23         $this->setLength(9);
24
25         $this->getLength()->shouldReturn(9);
26     }
27 }
```

Done! Oh, and I want you to notice one cool thing: we now get autocomplete on the `getLength()` method! PhpStorm has *great* phpspec integration and knows that we're allowed to use `$this` like a `Dinosaur` object. We *don't* have auto-completion for `setLength()`, because that method doesn't exist yet.

Let's see what phpspec thinks about our new example:

```
./vendor/bin/phpspec run
```

Awesome! It hates it! It fails, asks us if it can generate some code, then fails again because that generated `setLength()` method is just blank. Go find the new method. Hey! It even noticed that this method should have one argument. Change it to `int $length`. Inside, `$this->length = $length`.

src/Entity/Dinosaur.php

```
↕ // ... lines 1 - 4
5  class Dinosaur
6  {
↕ // ... lines 7 - 13
14     public function setLength(int $length)
15     {
16         $this->length = $length;
17     }
18 }
```

Try it again!

```
./vendor/bin/phpspec run
```

Yes! All green! This is test-driven-development the phpspec way. Oh, and yea, we'll talk more about TDD, BDD and what all those buzzwords mean in the context of phpspec a bit later.

Next, instead of relying on the built-in matchers - like `shouldReturn()` - let's create our own *custom* matcher. Why? Because a custom matcher can help us write examples with *perfectly* natural language.

Chapter 4: Custom Inline Matcher

One of the *main* goals of a spec class is for it to communicate the behavior of our class through readable and natural language. More important than being a test, this class is meant to be *documentation*. If the function names or code inside the functions aren't naturally readable - you're at risk of angering the phpspec gods!

For example, saying `$this->getLength()->shouldReturn(9)` *does* read like a normal, English sentence. But let's pretend for a minute that this language does *not* sound clear - maybe we're using a matcher that *works*, but just feels unnatural. In that case, we can invent our *own* language. Check it out: create a new example function: `it_should_default_to_zero_length_using_custom_matcher()`.

Inside, let's show this same behavior, but in a different way - how about `$this->getLength()->shouldReturnZero()`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 8
9  class DinosaurSpec extends ObjectBehavior
10 {
↕ // ... lines 11 - 29
30     function it_should_default_to_zero_length_using_custom_matcher()
31     {
32         $this->getLength()->shouldReturnZero();
33     }
↕ // ... lines 34 - 40
41 }
```

That's great language! But, as you probably noticed, PhpStorm did *not* auto-complete that matcher function. That's because... I just made that up! There is *no* built-in matcher that allows us to say `shouldReturnZero()`.

To prove it, run spec!

```
./vendor/bin/phpspec run
```

No `returnZero` matcher found. But, if this *is* the language that is most natural, we *can* and *should* make it work. How? By creating our *own* matcher.

Overriding getMatchers()

At the top of your class... or really anywhere, go to the `Code -> Generate` menu - or Command+N on a Mac - and override a method called `getMatchers()`. We don't need to call the parent method because its empty.

This method is... kinda beautiful: just return an array where they *keys* are the custom matchers you want. Except, the *key* is *not* `shouldReturnZero()`. Nope, the name of the matcher is that string *without* the "should" or "shouldNot"

part. In other words, add `returnZero` set to a function with one argument called `$subject`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 8
9  class DinosaurSpec extends ObjectBehavior
10 {
11     public function getMatchers(): array
12     {
13         return [
14             'returnZero' => function ($subject) {
↕ // ... line 15
16             },
17         ];
18     }
↕ // ... lines 19 - 40
41 }
```

The Matcher Subject

Here's how this works: in the example, we call `getLength()`, which we know returns an integer - hopefully zero. But thanks to the magic of phpspec, we can call `shouldReturnZero()` on this value. When we do that, phpspec will call our function and pass the *length* returned from `getLength()` as the `$subject`. Complete the matcher by saying `return $subject === 0`. Our matcher function should return `true` if the `$subject` looks valid, `false` otherwise.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 13
14             'returnZero' => function ($subject) {
15                 return $subject === 0;
16             },
↕ // ... lines 17 - 42
```

So... let's try this! Go spec go!

```
./vendor/bin/phpspec run
```

It passes! Oh, and we can automatically *also* use `shouldNotReturnZero()`: every matcher is able to handle both `should` and `shouldNot`.

Better Error Message

To make sure the matcher is *really* working, in `Dinosaur`, add a bug by changing the default length to 30.

src/Entity/Dinosaur.php

```
// ... lines 1 - 4
class Dinosaur
{
    private $length = 30;
// ... lines 8 - 17
}
```

Now re-run phpspec:

```
./vendor/bin/phpspec run
```

Two examples fail - we're working on the second example. Look at the error:

> *"integer:30 expected to `returnZero()`, but it is not."*

Wow. That's... kinda bad language. phpspec is *trying* its best to tell us what went wrong in a way that makes sense... but it doesn't always work.

No problem: we can control that error. Let's refactor that code a bit: if `$subject !== 0`, then, instead of returning false, throw a new `FailureException()` with a better message:

> *"Returned value should be zero got "%s""*

and pass `$subject` for the wildcard.

Then, at the bottom `return true` to signal that everything is fine.

spec/Entity/DinosaurSpec.php

```
// ... lines 1 - 14
            'returnZero' => function ($subject) {
                if ($subject !== 0) {
                    throw new FailureException(sprintf(
                        'Returned value should be zero, got "%s"',
                        $subject
                    ));
                }

                return true;
            },
// ... lines 25 - 50
```

Try the tests again:

```
./vendor/bin/phpspec run
```

Oh, *even* with my typo on the word "got", the error is *much* better. Let's go fix that bug - change 30 back to zero - and re-run phpspec:

src/Entity/Dinosaur.php

```
// ... lines 1 - 4
class Dinosaur
{
    private $length = 0;
// ... lines 8 - 17
}
```

```
./vendor/bin/phpspec run
```

Nice! Oh, by the way, *sometimes* when you call a matcher, you may need to pass it an argument... and sometimes we don't. If we *did* pass an argument to the matcher function, it would be passed to our callback as the second argument. And if you pass two arguments to the matcher, these become arguments two and three... and so on - you can make the matcher as complex as you need.

Because the new matcher lives right inside the spec class, this is called an "inline" matcher. And as *nice* as it is, it has one major downside: the `returnZero` matcher can't be re-used in any other spec classes. So next: let's create another custom matcher that can be used in our entire app.

Chapter 5: Registering & Autoloading a Custom Matcher

Inline matchers are easy! I love that! But you can't reuse them across multiple spec files - that's a bummer. Fortunately, phpspec is awesome and so - of course - it *does* have a way to create a matcher that can be used anywhere. And... what's even *better* is that there are a *lot* of great examples to learn from.

And here's one: go to https://github.com/karriereat/phpspec-matchers - I... probably butchered that username - sorry! Anyways, this library is cool: it's just a big collection of custom matchers! Well, technically it's a phpspec *extension*, which means it's a "plugin" for phpspec.

Unfortunately, this library does *not* work with the latest version of phpspec at this time. But... who cares!? It is *still* an awesome source of inspiration for writing your own custom matchers. Each of these classes represents one matcher.

Describing a Potential Bug

Here's our next goal: our scientists are starting to grow dinosaurs for the park, but have reported a possible bug in the `Dinosaur` class! No problem! When you have a bug, the *best* thing to do is write a test for it: to describe the *correct* behavior so we can make sure our class has that.

In this case, someone reported that, when a dinosaur is created with a length of 15, sometimes... it shrinks! We've talked to our scientists and they say that *some* shrinking is ok, but a dinosaur should definitely not shrink below a length of 12. Wow, it turns out that growing a dinosaur is complex!

Let's translate this expected behavior into a new example: `function it_should_not_shrink()`. Set the length of the dinosaur to `15` - and notice that we *do* get auto-completion now that the `setLength()` method exists.

Then say, `$this->getlength()`... but... hmm. In this pretend example, the dinosaur is allowed to shrink *some* but not below 12. To reflect that, let's say: `->shouldBeGreaterThan(12)`

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 49
50     function it_should_not_shrink()
51     {
52         $this->setLength(15);
53
54         $this->getLength()->shouldBeGreaterThan(12);
55     }
56 }
```

As you probably saw, that is *not* a real matcher. So, the tests should fail. Try them:

```
./vendor/bin/phpspec run
```

They... pass? Hmm... ah! A typo! And this *proves* that each example *must* start with `it_` or `its_`. Try phpspec again:

```
./vendor/bin/phpspec run
```

There is the failure we expected.

Creating our Matcher Class

We know that we *could* create an inline matcher. But... I kinda want to be able to re-use this in other spec classes. To do that, we can create a matcher class. In the `spec/` directory, create a `Matcher` directory and then a new class: `BeGreaterMatcher`... though this class could live anywhere. The namespace should be `spec` then the directory path. So `spec\Matcher`.

spec/Matcher/BeGreaterMatcher.php

```
// ... lines 1 - 2
namespace spec\Matcher;

class BeGreaterMatcher
{

}
```

But, let's keep this class empty for now: I just want to make sure that phpspec can *see* our new matcher. How? Via its config! Open `phpspec.yml`, add a `matchers:` section and then, very simply, list your matcher: `- spec\Matcher\BeGreaterMatcher`.

phpspec.yml

```
// ... lines 1 - 5
matchers:
    - spec\Matcher\BeGreaterMatcher
```

That's it! It won't *fully* work yet of course... but let's see what happens. Run phpspec:

```
./vendor/bin/phpspec run
```

Interesting:

> *"Custom matcher `spec\Matcher\BeGreaterMatcher` does not exist."*

This is basically a "class not found" error. Copy the namespace. Yeah... that looks correct... I don't see any typos. So... what's the problem? Autoloading!

Autoloading the phpspec Directory

Open the `composer.json` file. We configured composer to be able to autoload things from the `src/` directory, but we haven't configured it to be able to autoload things from the `spec/` directory. phpspec does *not* need any autoloading to be setup to find the spec files - it handles all of that itself. But if you want to put any *other*, non-spec, classes in this directory - like a matcher - then we *do* need to set up autoloading.

No problem: copy the `autoload` section, paste and change it to `autoload-dev`. Tell composer to expect the `spec\\` namespace to live in the `spec/` directory.

composer.json

```
{
// ... lines 2 - 11
    "autoload-dev": {
        "psr-4": {
            "spec\\": "spec/"
        }
    }
}
```

To make Composer rebuild its autoloader, run:

```
composer dump-autoload
```

Cool! Try phpspec again:

```
./vendor/bin/phpspec run
```

Much better! It *does* see it, and now we get:

> *"Custom matcher spec\Matcher\BeGreaterMatcher must implement... some Matcher interface."*

Apparently all matcher classes need to implement this interface. That makes sense! Let's do that next - and finish this!

Chapter 6: Coding up the Custom Matcher

According to this friendly error, a custom matcher has one important rule: it must implement this `Matcher` interface. Cool! To see what these classes normally look like, let's cheat and dig into some of the core matchers themselves!

Peeking at the Core Matchers

Open `vendor/phpspec/phpspec/src/PhpSpec` and look in the `Matcher/` directory. Say hello to *all* the core matchers. Check out `ThrowMatcher` - a matcher we'll use later to help us test exceptions. Yep! It implements that interface. Let's also look at one we're already using a lot: `IdentityMatcher`.

Oh - instead of implementing the `Matcher` interface directly, this extends a `BasicMatcher` class, which implements that interface, but handles a lot of the low-level work. Most of the time, you'll probably want to extend this class - it just makes life easier.

Let's go! I'll close a few other files. Then, make our `BeGreaterMatcher` extend `BasicMatcher`. I'm also going to mark the class as `final`. There's no reason for that - it's just a general nice practice to mark a class as final unless you intend for it to be sub-classed. Though, marking a class as final can cause issues if you need to mock it. Either way - this is not important to get this all working.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 7
8 final class BeGreaterMatcher extends BasicMatcher
9 {
↕ // ... lines 10 - 28
29 }
```

Next, go to the `Code -> Generate` menu or Command+N on a Mac, select "Implement Methods" and implement all *four* methods that we need.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 9
10     protected function matches($subject, array $arguments): bool
11     {
12         // TODO: Implement matches() method.
13     }
14
15     protected function getFailureException(string $name, $subject, array $arguments):
   FailureException
16     {
17         // TODO: Implement getFailureException() method.
18     }
19
20     protected function getNegativeFailureException(string $name, $subject, array
   $arguments): FailureException
21     {
22         // TODO: Implement getNegativeFailureException() method.
23     }
24
25     public function supports(string $name, $subject, array $arguments): bool
26     {
27         // TODO: Implement supports() method.
28     }
↕ // ... lines 29 - 30
```

Implementing supports()

The first important method is `supports()`. Let's `var_dump()` the arguments to see what we're working with: `$name`, `$subject` and `$arguments`. Whenever *any* matcher is used inside of a spec class, phpspec loops over *all* of the matcher classes and calls `supports()` to figure out *which* one to use.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 24
25     public function supports(string $name, $subject, array $arguments): bool
26     {
27         var_dump($name, $subject, $arguments);
28     }
↕ // ... lines 29 - 30
```

Let's see what happens! Run phpspec:

```
./vendor/bin/phpspec run
```

Yep! It's dumping out *every* time any matcher is used - once for `haveType`, `returnZero` and down here for `beGreaterThan`. That's the `$name` argument. The `$subject` is `15` because, in our spec class, `getLength()` returns 15 and we're calling the matcher on that value. Finally, `$arguments` is an `array` with just one entry: 12 - because we're passing 12 as the *one* argument to the `shouldBeGreaterThan()` matcher.

The `supports()` method *really* only needs to check to make sure that the `$name` is equal to `beGreaterThan`: that's enough to tell phpspec that *we* handle that. But, a lot of times, these are written to be a bit more flexible. For

example, you could use `in_array()` to check that `$name` is one of `beGreater` or `beGreaterThan`. Then, if you want, you can even make sure the *types* of the subject and arguments are what we expect. We'll say that this matcher should only be used if `is_numeric($subject)` and if `count($arguments)` is exactly one *and* if *that* argument is also numeric *and* if it's Halloween after midnight. Kidding.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 24
25     public function supports(string $name, $subject, array $arguments): bool
26     {
27         return in_array($name, ['beGreater', 'beGreaterThan'])
28             && is_numeric($subject)
29             && count($arguments) === 1
30             && is_numeric($arguments[0]);
31     }
↕ // ... lines 32 - 33
```

Implementing Matches

So, *when* `supports()` returns `true`, phpspec will *then* call the `match()` function on top. Our job *here* is to return `true` if everything looks good, or false otherwise - just like the inline matcher. Let's `var_dump` the `$subject` and `$arguments` one more time with a `die` statement - to make sure it's called like we expect.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 9
10     protected function matches($subject, array $arguments): bool
11     {
12         var_dump($subject, $arguments);die;
13     }
↕ // ... lines 14 - 33
```

Move over and try phpspec again:

```
./vendor/bin/phpspec run
```

Yes! We *are* called with 15 as the `$subject` and the same `$arguments` array with 12 inside. Because we know there will always be exactly *one* argument, we can finish this method `return $subject > $arguments[0]`.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 9
10     protected function matches($subject, array $arguments): bool
11     {
12         return $subject > $arguments[0];
13     }
↕ // ... lines 14 - 41
```

Implementing the Failure Exception Methods

Done! If the matcher returns false, phpspec will either call `getFailureException()` or `getNegativeFailureException()` - the difference is if we're using `shouldBeGreaterThan()` or `shouldNotBeGreaterThan()`. Our job there is to return that same `FailureException` - I'll paste one in with a good message. This is the same type of object that we're throwing from our inline matcher.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 14
15     protected function getFailureException(string $name, $subject, array $arguments):
       FailureException
16     {
17         return new FailureException(sprintf(
18             'Expected %d to be greater than %d',
19             $subject,
20             $arguments[0]
21         ));
22     }
↕ // ... lines 23 - 41
```

For the other method, copy this, paste, and just tweak the language a little bit.

spec/Matcher/BeGreaterMatcher.php

```
↕ // ... lines 1 - 23
24     protected function getNegativeFailureException(string $name, $subject, array
       $arguments): FailureException
25     {
26         return new FailureException(sprintf(
27             'Expected %d to not be greater than %d',
28             $subject,
29             $arguments[0]
30         ));
31     }
↕ // ... lines 32 - 41
```

We *now* have a fully-functional custom matcher that allows us to use this new, natural, language inside of *any* spec file. Well... assuming it works - try it out:

```
./vendor/bin/phpspec run
```

All green! Next: it's time to talk about how phpspec fits into the entire world of testing. For example, there are functional tests, integration tests, and unit tests... and multiple tools like PHPUnit and Behat. Let's dive into this big mess of tools and buzzwords.

Chapter 7: phpspec? PHPUnit? BDD? TDD? Buzzwords?

Theory time! Wait, come back! Um... it's *interesting* theory... and we'll do something fun when it's over - I promise. Ok, fine: *and* I'll buy you all ice cream. Let's do this.

Functional, Integration & Unit Tests

In the world of testing, there are *three* types, and we cover each of these in our PHPUnit tutorial. The first type - unit tests - is when you're testing the code directly: you literally call methods like `setLength()` and `getLength()` on objects and assert that the values you get back are correct. The *key* thing in unit testing is that, if your object depends on another object - like a database connection, or a mailer object - you *mock* those dependencies, instead of using the real object. We'll do this later in the tutorial. Unit tests are the "pure" tests: you're testing the input and output of a method in complete isolation.

The second type of tests is called integration tests. They look and smell a lot like unit tests: you work directly with objects, call methods on them, and make sure you get back what you expect... actually *exactly* like unit tests. The *key* difference is how *dependencies* are treated. Instead of, for example, "mocking" a database connection object that the class you're testing needs, you use the *real* database connection! Crazy! And, yea, that means that your code makes *actual* database queries!

Integration tests are *super* useful when you have a lot of pieces working together and want to make sure the whole *system* "integrates" correctly. Or if you're making a complex database query and want to make sure you get back what you expect.

The *third* type of tests are functional tests. In a functional test, instead of calling methods directly on an object, you write code that commands a browser to literally go to a page, click on a link, fill out a form, and assert some text showed up on the next page. Or, if you're testing an API, you would literally make *real* HTTP requests to your API and assert that the JSON you get is what you expect. In a functional test, you're basically *using* your application as if you were an end user.

So... which tests should you write? None of them! Kidding - probably... all of them! Sometimes the "scary" or "complex" behavior you want to test lives entirely inside a single class. Use a unit test to crush that situation. Other times, integration tests are perfect when you want to check what a function does... but what it does involves database queries or a lot of other little pieces. And a lot of the time, at least for us at SymfonyCasts, we want to make sure the user experience is exactly what we want. We verify that with functional tests.

PHPUnit, Behat, phpspec?

So... which tools should we use for all of this? Well, PHPUnit can be used to do all *three* types. Behat - an awesome library we talk about in another tutorial - can *only* be used for functional tests. And... what about phpspec? Well, it can *only* do *unit* tests.

So... wait... if PHPUnit can be used for *any* test... and these other tools are more limited... why the heck are we even talking about them?

Simply: because tools like Behat and phpspec do a *better* job for the types of tests they focus on. Well, ok, that's *totally* subjective - but let me explain. Instead of "just writing tests and getting them to pass", both Behat & phpspec help you focus on the *quality* of your app. Behat forces you to think about the *external* quality of your app - by focusing you on the experience of your users *first* and coding second. That's the *key* difference between writing functional tests in Behat versus PHPUnit.

phpspec is the exact same for unit tests. Instead of just writing tests and getting them to pass, phpspec makes you think about the *design*, behavior and *purpose* of your PHP classes first. And as a nice by-product, you get unit tests.

TDD vs BDD

Got it? Great - let's move onto some buzzwords. How does all of this fit in with TDD - test-driven development versus BDD - *behavior* driven development. First, both of these aim to accomplish the same thing: creating high-quality applications. The difference is the *language* used and the *focus*. With test-driven development, you are *literally* supposed to write the tests first, and *then* write the code. You allow your tests to *drive* the code that you need to develop.

With behavior-driven development, the process is *technically* the same. But instead of starting with:

> *"Let's figure out what tests I need to write!"*

you start by thinking about the *behavior* that you want each part of your app to have. phpspec is a tool that promotes *behavior* driven development because it forces us to think about the behavior that we want each class to have - not the test.

Oh, and by the way, Behat is *another* tool that promotes BDD. The difference is that Behat is used for functional tests. Behat is "story" BDD: you write "user stories" that describe behavior. phpspec is "spec" BDD: you write specifications for your classes. Honestly, understanding all this stuff isn't *that* important - but now, you are *fully* ready to throw out these buzzwords at your next party.

The point is this: a tool like phpspec doesn't technically give you anything that PHPUnit can't give you. But it changes your focus to be the *design* of your classes instead of just writing the tests.

Next: let's talk a little bit about the super-rewarding red-green-refactor cycle and then... put a dinosaur in our terminal. You'll see.

Chapter 8: Red, Green, Refactor Cycle + More Dinosaurs

So there's *one* more little bit of theory. Well, less *theory* - more a strategy that you get to use with BDD or TDD and... it's *really* rewarding. It's called the red, green, refactor cycle. It goes like this: whenever you need to add a feature or fix a bug, you follow this simple three-step process.

First, write a test! Or, in phpspec, write an "example" showing the behavior you want. Then, run your test to make sure it's failing - that's the "red" part.

Second, write *just* enough code to get that test to pass - and *no* more. This is the green part of the cycle - and it's more interesting than it seems at first. The *key* thing with this step is that you're supposed to write *just* enough code to get your test to pass... *not* focus on writing a perfect, pretty or extensible solution.

And then, *once* the test is green, you're free to do step 3: refactor. My favorite thing about this cycle is that it gives you permission on step 2 to write bad or duplicated code! I *love* this! It lets me focus on solving the *problem*, without over-thinking the details. And if you *do* need to refactor, you can do it confidently knowing that you're not going to break anything.

Theory: Follow some of It

But ultimately... these are all just "recommendations". In the world of testing, there are a *lot* of philosophical pointers and best practices that are thrown around. At the end of the day, do whatever is best for you. Just writing *any* tests will make your app more robust.

In this tutorial, we're going to do things - more or less - the "right" way - with BDD and the red-green-refactor cycle. But sometimes I do the opposite! Sometimes I write the code first and *then* the tests. It depends on the situation and nobody is perfect. Be pragmatic.

I also don't unit test everything - actually *far* from it! Earlier, we tested the `getLength()` and `setLength()` methods. Those were great examples - but that code is *so* simple, I would *not* normally test it. I unit test a method if it scares me - and then rely on integration and functional tests to cover how all the little pieces work together.

Output Formatters

Ok, as promised, after *all* that theory, we're going to do something fun... then keep going. We already know how to run phpspec:

```
./vendor/bin/phpspec run
```

Awesome! You can *also* pass a `format` option. This accepts a number of different values, but one of the *best* is `pretty`:

```
./vendor/bin/phpspec run --format=pretty
```

Emojis! Since these check marks are super hipster, let's make phpspec use this format by default. Open `phpspec.yml` and add `formatter.name: pretty`.

phpspec.yml

```
// ... lines 1 - 8
formatter.name: pretty
```

As soon as we do that, we can remove the `format` option and *still* get those check marks.

```
./vendor/bin/phpspec run
```

The Nyan Cat Formatter

But... come on... this is a dinosaur tutorial! And so, what I *really* need while practicing BDD and the red-green-refactor cycle is a *dinosaur* to tell me if my tests are passing. Fortunately, the authors of phpspec knew this would happen, and created the `phpspec/nyan-formatters` repository. Copy the name of the library and run:

```
composer require phpspec/nyan-formatters:dev-master --dev
```

We need to use the `master` branch because it doesn't have a proper release yet that's compatible with phpspec 5, which is fine. While we're waiting for that, move back to the docs and copy the extensions code. I mentioned earlier that "extensions" are the word phpspec uses for its plugins. An extension can pretty much do anything: it can give you custom matchers, custom formatters or even change how the generated code is rendered - like to add more type-hints for arguments. There's a whole page on phpspec's docs listing some of the most popular extensions.

To activate an extension, open `phpspec.yml`, add `extensions:` and then paste the extension class name. That's it. Let's go check on the terminal... yes! It's done!

phpspec.yml

```
// ... lines 1 - 10
extensions:
    PhpSpec\NyanFormattersExtension\Extension: ~
```

The purpose of *this* extension is to give us a few new formatters. One of them is called `nyan.dino`. Ok! Run phpspec again:

phpspec.yml

```
// ... lines 1 - 7

formatter.name: nyan.dino
// ... lines 10 - 13
```

```
./vendor/bin/phpspec run
```

Hello Dino! Ok, ok - time to get back to the *real* work. Next: let's demystify all the magic behind phpspec by looking into the `ObjectBehavior` class. That's the class our spec class extends - and *it* is responsible for allowing us to use `$this` as *if* we were in a Dinosaur class. Understand how that works and you'll be unstoppable!

Chapter 9: The ObjectBehavior Magic

The *hardest* part of phpspec for me was how *weird* these spec classes look. They're... complete magic! You're supposed to pretend that the `$this` variable is a `Dinosaur` object... even though we're not in that class? And also... I guess that means that phpspec somehow instantiates a new `Dinosaur` object before it runs each example? Then, *just* when you get used to the weirdness of treating `$this` like a `Dinosaur` object and calling real methods on it... we suddenly call a matcher method - like `shouldReturn(0)`. What is going on!?

Digging into ObjectBehavior

Let's find out. Because when I finally saw how all this worked behind the scenes, I instantly felt *much* more comfortable. All of this magic starts with the base `ObjectBehavior` class. Hold Command or Ctrl and click to open that.

Ah, ok: see that `protected $object` property? Surprise! *That* is *actually* the underlying `Dinosaur` object that we're testing. Well, that's not 100% true - but imagine it is for a minute. So, at some point, phpspec instantiates a `Dinosaur` object and stores it on that property.

Pretty much *all* of the magic of this class is thanks to the `__call()` method. If you're not familiar with this method, that's great! It's a magic PHP method that you should probably *not* use - but it's perfect for phpspec. If you call a non-existent method on an object, but that class has an `__call()` method, instead of freaking out and throwing an error, PHP will instead execute `__call()` and pass it the method name and the arguments you were trying to use.

And what does `ObjectBehavior` do in this method? It basically calls that method on `$this->object` and passes *it* the arguments! *This* is why, when we say `$this->getLength()`, it works! The `getLength()` method does not exist on `ObjectBehavior`. But thanks to the `__call()` method, it *forwards* that call to the actual `Dinosaur` object. `ObjectBehavior` also has a few other methods, like `__get()` and `__set()` to forward setting properties and other stuff.

The Wrapped Object

Let's see what some of this looks like in the wild. Close that class and, in any of the examples, `var_dump($this)`. Go tests go!

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 49
50     function it_should_not_shrink()
51     {
↕ // ... lines 52 - 55
56         var_dump($this);
57     }
58 }
```

```
./vendor/bin/phpspec run
```

Interesting... As we expected, `$this` is really an instance of `DinosaurSpec`. But check out the `$object` property. I lied! It is *not* an instance of `Dinosaur`! Gasp! Nope, it's some `Subject` object from phpspec. But inside of it is something called a `WrappedObject` and inside if *it*, yep! There is the `Dinosaur` object.

So, it's a bit more complex than we thought at first, but phpspec *did* create a `Dinosaur` object and set it on that `object` property... just wrapped inside a few other objects to help the magic.

For the most part, we pretend like we're interacting directly with a `Dinosaur` object. But, if you *did* need to get the *actual*, underlying `Dinosaur` object, that's possible! Try `$this->getWrappedObject()`, then run the test again:

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 49
50     function it_should_not_shrink()
51     {
↕ // ... lines 52 - 55
56         var_dump($this->getWrappedObject());
57     }
↕ // ... lines 58 - 59
```

```
./vendor/bin/phpspec run
```

Cool! *That* gives us the *real* Dinosaur object. And it's length is *really* 15, because when we call `$this->setLength(15)`, that eventually *is* called on the real, underlying object.

Most of the time, you won't need to call `getWrappedObject()`, though there are a few edge-case exceptions. Like, imagine if our `Dinosaur` class had a method on it that started with `should`, like `shouldHandle()`. Well... that won't work. phpspec thinks that when we call anything starting with `should` or `shouldNot`, that we're trying to execute a *matcher* - not a method. Check it out:

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 49
50     function it_should_not_shrink()
51     {
↕ // ... lines 52 - 55
56         $this->shouldHandle(2);
57     }
↕ // ... lines 58 - 59
```

```
./vendor/bin/phpspec run
```

There it is: "no handle matcher found". For this edge-case, you can use `$this->callOnWrappedObject()` with `shouldHandle` and an array of arguments you want. Try it now:

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 49
50     function it_should_not_shrink()
51     {
↕ // ... lines 52 - 55
56         $this->callOnWrappedObject('shouldHandle', [2]);
57     }
↕ // ... lines 58 - 59
```

```
./vendor/bin/phpspec run
```

Nice! It fails... but with the *correct* failure: it sees that there is no `shouldHandle()` method and, actually, asks us to generate it. Choose no - we're just messing around.

Next: there's one more piece of magic we haven't talked about: when we call `$this->getLength()`, that should return 15. So then... how the heck are we able to call a method on that?

Chapter 10: The Magic of the Subject

We *now* know that the `ObjectBehavior` class forwards all method calls to the underlying `Dinosaur` object thanks to some magic methods. But, there is still one more big piece of magic. When we call `$this->getLength()`, phpspec will ultimately call `getLength()` on the `Dinosaur` object and *that* will return the integer 15. So then... what absolute madness is allowing us to call a method on that?! Let's find out!

This time `var_dump($this->getLength())`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 49
50     function it_should_not_shrink()
51     {
↕ // ... lines 52 - 55
56         var_dump($this->getLength());
57     }
58 }
```

Will this be the integer 15? An object? An emoji? Let's find out!

```
./vendor/bin/phpspec run
```

Ah! It's an instance of a `Subject` object. Stop! We *know* that object! That is the *exact* same class type that is stored on the `$object` property of our base `ObjectBehavior` class! Just like before, `Subject` is a wrapper object that gives us some magic. Let's find out how it works: type Shift+Shift and look for the `Subject.php` file.

Woh! See all these `@method` things on top? This tells PhpStorm that we can call any of these methods on this object and they will work. We need this because, if you look, these methods don't actually exist in this class! They work by magic - we'll see that in a minute.

This class works a lot like `ObjectBehavior`. Scroll down until you find the all-important `__call()` method. When we call `getLength()`, it gives us a `Subject` object. And *then* when we call `shouldBeGreaterThan()`, it's handled by `__call()`. The logic here is awesome: if the method name starts with `should`, it calls `$this->callExpectation()` - which finds and executes the correct "matcher". So, why do all matchers need to begin with `should`? Because of this line right here.

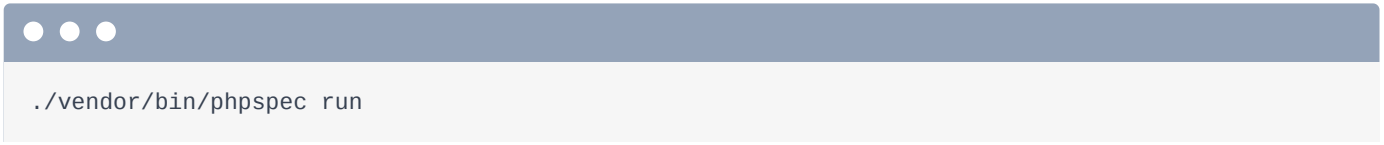
Next, if the method name starts with `beConstructedThrough` or `beConstructedWith`, it calls some code that allows us to *control* how the `Dinosaur` object is instantiated. We'll use this really soon.

And *ultimately*, if it is *not* one of those special cases, it executes code that *forwards* the call onto the underlying object and returns that value. Well, it returns the value wrapped in, yet another, `Subject` class. This is *exactly* what

happens when we call `$this->getLength()`: this last line calls `getLength()` on the `Dinosaur` object and then wraps it in a `Subject` object. Thanks to that, we can *then* call `shouldBeGreaterThan` to call our matcher.

So, yes, it *is* all magic - super impressive magic! But it's magic that's done via a couple of wrapper object and the `__call()` method.

Let's remove our debug code, and make sure the tests are still passing:



```
./vendor/bin/phpspec run
```

Cool! Now, back to testing! Next: let's learn how to "describe" a special part of our object's behavior: how it's instantiated.

Chapter 11: Describing Object Construction

Let's describe a new behavior that we need for our `Dinosaur` class. I want to be able to easily get a "description" of the Dinosaur - a string that will contain the type of dinosaur, whether or not it likes to eat people and its length.

Let's turn that into a new example: `function it_should_return_full_description()`. For this first example, we'll describe what the description should look like if we set *no* data. Let's say that there should be a new `getDescription()` method that `shouldReturn()`:

> *"The Unknown non-carnivorous dinosaur is 0 meters long"*

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 56
57     function it_should_return_full_description()
58     {
59         $this->getDescription()->shouldReturn('The Unknown non-carnivorous dinosaur is 0
   meters long');
60     }
61 }
```

Our `Dinosaur` doesn't even *have* any properties on it related to what "type" of `Dinosaur` it is, or whether it's carnivorous or non-carnivorous - but those are details for future Ryan to worry about. Let's live in the now! Run the test:

```
./vendor/bin/phpspec run
```

Yay! Failure! And it happily offers to generate that `getDescription()` method for us. Yes please! When it re-executes itself, it *still* fails: that new method is blank.

The Wonderful Lesson of Hardcoding Return Values

Ok, remember the red, green, refactor cycle? For step two, we're technically supposed to make this test green with as *little* work as possible. Challenge accepted! Let's copy this string, go into `Dinosaur`, find the new method, and - yes, I *am* about to do this - return that hardcoded string! We are *awesome* at programming! For extra credit, let's add a return type.

src/Entity/Dinosaur.php

```
↕ // ... lines 1 - 4
5  class Dinosaur
6  {
↕ // ... lines 7 - 18
19     public function getDescription(): string
20     {
21         return 'The Unknown non-carnivorous dinosaur is 0 meters long';
22     }
23 }
```

Yes, I *do* realize how silly this is. And no, I don't do this when I'm coding for real. But, there's something... beautiful about hardcoding this value: it's a reminder to focus on what we *truly* need to accomplish in this method - and to *not* over-complicate things or add extra options we don't need yet. BDD says: if you need your code to be more flexible, you'll discover that naturally when you describe some new behavior in a new example. The design of your code "emerges" naturally after writing examples and getting them to pass.

When we try phpspec again:

```
./vendor/bin/phpspec run
```

Shocking! It passes!

Describing Constructor Arguments

Head back to our `DinosaurSpec` class: to fully describe how we want the new `getDescription()` method to work, we need a few more examples. Let's see: the description contains details about the "type" or "genus" of the dinosaur - like Tyrannosaurus or Stegosaurus - and whether or not it wants to eat you. I mean, whether or not it's carnivorous.

Right now, there is no way to set this type of info on the `Dinosaur` class. We need to fix that. So... how *do* we want to set that info? For the length, we added a `setLength()` method. But, I think that the *type* of dinosaur and whether or not it's carnivorous are *so* important, that they should be passed via the constructor when instantiating a `Dinosaur`.

Let's create an example: `it_should_return_full_description_for_tyrannosaurus()`. We know that phpspec handles instantiating a new `Dinosaur` object for us so that when we call `getDescription()`, it eventually calls that method on the real object.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 61
62     function it_should_return_full_description_for_tyrannosaurus()
63     {
↕ // ... lines 64 - 67
68     }
↕ // ... lines 69 - 70
```

That's cool, but what's *cooler* is that we can control *how* it's instantiated. To do that, say `$this->beConstructedWith()` and - quite literally - pass the arguments here that we want to pass to the new `Dinosaur` object. Hmm, I think the first argument should be the dinosaur type - tyrannosaurus - and the second a boolean for whether or not it's carnivorous. Definitely `true`.

spec/Entity/DinosaurSpec.php

```
// ... lines 1 - 61
    function it_should_return_full_description_for_tyrannosaurus()
    {
        $this->beConstructedWith('Tyrannosaurus', true);
// ... lines 65 - 67
    }
// ... lines 69 - 70
```

Now... keep going like normal! Let's set a length - `$this->setLength(12)` - and then assert that `$this->getDescription()->shouldReturn()` the string

> *"The Tyrannosaurus carnivorous dinosaur should be 12 meters long"*

spec/Entity/DinosaurSpec.php

```
// ... lines 1 - 61
    function it_should_return_full_description_for_tyrannosaurus()
    {
// ... line 64
        $this->setLength(12);

        $this->getDescription()->shouldReturn('The Tyrannosaurus carnivorous dinosaur is 12 meters long');
    }
// ... lines 69 - 70
```

Perfect! And thanks to *this* new example... our hardcoded return statement? Yea... that ain't gonna work anymore.

How the Objects are Constructed

Oh, by the way, what if we had *two* `beConstructedWith()` lines with different arguments? I know, this looks silly - but this *can* happen in some cases when you use a setup function called `let()` that we'll learn about later.

Anyways, what would happen here? An error? CPU over-heating? Neither! The *last* call always wins. The *reason* is the interesting part. Behind the scenes, phpspec *delays* instantiating the object as *long* as it can. In this case, it doesn't actually instantiate the `Dinosaur` object until we call a method on it like `setLength()`. At *that* moment, phpspec realizes it needs to instantiate the object and creates it using the arguments that were passed to the last `beConstructedWith()` call.

Ok, let's run phpspec and get to the "red" part of the cycle:

```
./vendor/bin/phpspec run
```

Hey! This is cool! It says that the method `__construct()` was not found! It *realizes* that we're saying `beConstructedWith()`... but we're missing that method! And of course, it even offers to generate it. Do it!

Next, let's hook up the constructor and work with phpspec to get our examples passing.

Chapter 12: Coding & Debugging

phpspec just generated the `__construct()` method for us. Thanks buddy! Go check it out! Two cool things here. First, in `Dinosaur`, yes, it *did* add the constructor method. As a bonus, it even put it in the right place: after the properties, but above all the other public functions. phpspec, are you trying to take my job?

Second, when phpspec re-ran all of the examples, well... they're almost *all* failing now: too few arguments to `Dinosaur::__construct()`. And... that makes sense! We just *massively* changed the way that our `Dinosaur` class is designed. And so, any existing coding using that class will probably be totally borked! This is a great example of how our tests can give us feedback. They're saying:

> *"Ryan! Do you realize that you just broke all of the code in your app that creates new `Dinosaur` objects???"*

If we did that on accident... well... that would be a *pretty* good warning to get.

Implementing the Constructor

Let's get to work: the first argument should be a `string $genus` and the second `bool $isCarnivorous`. I'll press Alt+Enter and select "Initialize Fields"... which is a shortcut to create those two properties and set them. Remove the TODO.

This is cool. But... I'm going to make both of these arguments *optional*. Why? Well, it's *entirely* up to you how you want your class to work. Look back at `DinosaurSpec`. According to this example, it looks like it *should* be legal to create a `Dinosaur` object with no information. If you do, the type should be "unknown" and it should default to *not* eat you... which is kinda nice. *This* is "design by spec": we're using our examples to *drive* how the class is built.

Default `$genus` to `Unknown` and `$isCarnivorous` to `false`.

src/Entity/Dinosaur.php

```
↕ // ... lines 1 - 4
5  class Dinosaur
6  {
↕ // ... lines 7 - 8
9      private $genus;
↕ // ... line 10
11     private $isCarnivorous;
↕ // ... line 12
13     public function __construct(string $genus = 'Unknown', bool $isCarnivorous = false)
14     {
15         $this->genus = $genus;
16         $this->isCarnivorous = $isCarnivorous;
17     }
↕ // ... lines 18 - 37
38 }
```

Using --verbose

The `getDescription()` method is still wrong. But, we *did* just get a step closer, so let's try phpspec again:

```
./vendor/bin/phpspec run
```

Yep! The two strings don't match. By the way, see how it truncates the two strings? Sometimes that makes it hard to figure out what's going on. If you need more info, run phpspec with the `--verbose` option:

```
./vendor/bin/phpspec run --verbose
```

Back in `Dinosaur`, let's finish the `getDescription()` method. Wrap the string in `sprintf()` then add a few wildcards: one for the genus, one for the `non-` part and one for the length. Fill these in with `$this->genus`, a ternary to print either nothing, or `non-`, and then `$this->length`.

src/Entity/Dinosaur.php

```
// ... lines 1 - 28
    public function getDescription(): string
    {
        return sprintf(
            'The %s %scarnivorous dinsaur is %d meters long',
            $this->genus,
            $this->isCarnivorous ? '' : 'non-',
            $this->length
        );
    }
// ... lines 38 - 39
```

Oh, and let's make a typo to spice things up! Then, move over, take off the `--verbose` option and run spec:

```
./vendor/bin/phpspec run
```

It *does* fail... but... it's not exactly obvious *why*: the truncated strings look identical! *This* is when running with the verbose option is handy:

```
./vendor/bin/phpspec run --verbose
```

Much better - the typo is super obvious now. Fix that, then try it again:

```
./vendor/bin/phpspec run --verbose
```

Ah! It *still* fails! Whoops - I made a mistake in the spec file - but it's obvious. I've been using phpspec for *so* long that I can't avoid saying "should" in everything I type. Change `should be` to `is`: that's the language we want.

src/Entity/Dinosaur.php

```
// ... lines 1 - 28
    public function getDescription(): string
    {
        return sprintf(
            'The %s %scarnivorous dinsaur is %d meters long',
// ... lines 33 - 35
        );
    }
// ... lines 38 - 39
```

Try it one more time:

```
./vendor/bin/phpspec run
```

It passes! Next: with production ramping up, we need a factory for our dinosaurs. Let's see how we can describe that with phpspec.

Chapter 13: Instantiation with a static Factory Method

Ok people - the whole "dino park" idea - apparently, it's a *huge* success! *Especially* the velociraptors. Management wants us to grow more of them... a *lot* more. What could go wrong?

And because we're going to need to create velociraptors so often, passing that *whole* string to the constructor and then `true` is too much work... and spelling velociraptor is hard! So, idea time: what if we created a static factory method on `Dinosaur` to help us create them? Like a `growVelociraptor()` method! Hey! I just described some new behavior! Quick! To the example...mobile!

Telling phpspec to use a Factory Method

Add a new function: `it_should_grow_a_large_velociraptor()`. Oh, but this is tricky: we know how to control what arguments phpspec passes to the `__construct()` method when it creates the `Dinosaur`. But... in this case, we *don't* want phpspec to create the `Dinosaur` object on its own. Nope! We instead want phpspec to call our new static method and *that* will instantiate and return the `Dinosaur`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 69
70     function it_should_grow_a_large_velociraptor()
71     {
↕ // ... lines 72 - 77
78     }
79 }
```

No problem: instead of `beConstructedWith()`, call `$this->beConstructedThrough('growVelociraptor')` - that will be the name of the new method. The second argument is the array of arguments for the method. What arguments should the `growVelociraptor` have? Hmm, probably just one right now: the length. So, `array(5)`. If you need a second argument, just add another item.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 69
70     function it_should_grow_a_large_velociraptor()
71     {
72         $this->beConstructedThrough('growVelociraptor', [5]);
↕ // ... lines 73 - 77
78     }
↕ // ... lines 79 - 80
```

Ok cool! *Now* this function will work like *any* other example - except that `$this` will be the `Dinosaur` object that's returned from `growVelociraptor`. Well, actually, we should make sure it *is* a `Dinosaur`: `$this->shouldBeAnInstanceOf(Dinosaur::class)`. Oh, and how about

`$this->getGenus()->shouldBeAString()`. Ok, I'm just showing you that this function exists. In this case, we know *exactly* that `$this->getGenus()->shouldBe('Velociraptor')`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 69
70      function it_should_grow_a_large_velociraptor()
71      {
↕ // ... lines 72 - 73
74          $this->shouldBeAnInstanceOf(Dinosaur::class);
75          $this->getGenus()->shouldBeString();
76          $this->getGenus()->shouldBe('Velociraptor');
↕ // ... line 77
78      }
↕ // ... lines 79 - 80
```

Wait, but why did I use `shouldBe()` here when we've been using `shouldReturn()` until now? No reason - they're identical: use whatever feels good.

Oh, and see how PhpStorm is highlighting `getGenus()`? That's because that method doesn't exist yet. We just "discovered" that we need this method. Cool! Let's add one more check: `$this->getLength()->shouldBe(5)`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 69
70      function it_should_grow_a_large_velociraptor()
71      {
↕ // ... lines 72 - 76
77          $this->getLength()->shouldBe(5);
78      }
↕ // ... lines 79 - 80
```

You know the drill: after writing the example, run phpspec:

```
./vendor/bin/phpspec run
```

Honestly, it shouldn't even be a surprise any more when phpspec generates code for us. Choose "yes" so it generates the missing `growVelociraptor()` method. And when it re-executes... failure! A `BadMethodCallException()`. That comes from the new function. phpspec knows this should return a `Dinosaur` object... but it's not sure how. But hey! It *did*, once again, put this method in *just* the right place: below the constructor but above the public functions.

src/Entity/Dinosaur.php

```
↕ // ... lines 1 - 4
5 class Dinosaur
6 {
↕ // ... lines 7 - 18
19     public static function growVelociraptor($argument1)
20     {
21         throw new \BadMethodCallException("Mismatch between the number of arguments of the
   factory method and constructor");
22     }
↕ // ... lines 23 - 42
43 }
```

Change the argument to `int $length`, advertise that this will return an instance of `self`, and create that with `$dinosaur = new static()` passing `Velociraptor` and `true` for the `isCarnivorous` argument. Then, `$dinosaur->setLength($length)`, `return $dinosaur`.

src/Entity/Dinosaur.php

```
↕ // ... lines 1 - 18
19     public static function growVelociraptor(int $length): self
20     {
21         $dinosaur = new static('Velociraptor', true);
22         $dinosaur->setLength($length);
23 
24         return $dinosaur;
25     }
↕ // ... lines 26 - 52
```

That felt good! Let's make sure the example passes:

```
./vendor/bin/phpspec run
```

Allowing Requirements to Emerge

Wait... it failed! Of course! The `getGenus()` method doesn't exist! That's super cool: instead of planning ahead and adding this method earlier, we allowed the need for this method to "emerge" naturally. What's *especially* interesting is that, so far, the only place we need this method is in our example! What if we don't need this method in our actual app? Should we still create it? Yes. Well, let me say that differently. The code in our examples are meant to be *real* examples of how you want your class to work. If you *really* don't want a `getGenus()` method, then should write example code that doesn't use it. If you *do* use it, you need it!

So, yes phpspec, please generate that for me. Find the new method, `return $this->genus` and add the `string` return type. Try the tests again:

src/Entity/Dinosaur.php

```
// ... lines 1 - 46
    public function getGenus(): string
    {
        return $this->genus;
    }
// ... lines 51 - 52
```

```
./vendor/bin/phpspec run
```

Yes! All green! Next: as *cool* as this factory method is, we need to level up with a proper, new `DinosaurFactory` service class.

Chapter 14: Describing the Factory Service

Ok *first*, we've learned a *lot* about phpspec so far. But... we've still only described *one* class - and a pretty simple one! It's time to dig deeper and add more complexity to our app.

Here's the deal: that new `growVelociraptor()` factory method has made our life a *lot* easier because, in our pretend app, we constantly need to create new velociraptors. But now, we also need to be able to create a few other popular dinosaurs - like T-rexes and Stegosaurus! We *could* keep adding more static methods to `Dinosaur`. But to keep things organized, I'd rather put all the logic into a new class - how about `DinosaurFactory`. Or, we might choose to do this because creating a Dinosaur requires some other services - like a database object - and we can't access services from simple model classes like `Dinosaur`.

Describing a new Class

So, hey! We need a new class! Well, to say it better, it's time for us to *describe* a new class: `./vendor/bin/phpspec describe` and, for the name, how about `App/Factory/DinosaurFactory`.

```
./vendor/bin/phpspec describe App/Factory/DinosaurFactory
```

That creates one new file: `DinosaurFactorySpec`. Let's go check it out! Like last time, we get one *super* basic example for free - asserting that `$this` should be an instance of `DinosaurFactory`. That's... kinda silly... but it's enough to force some code generation! Go run phpspec:

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurFactorySpec extends ObjectBehavior
11 {
12     function it_is_initializable()
13     {
14         $this->shouldHaveType(DinosaurFactory::class);
15     }
16 }
```

```
./vendor/bin/phpspec run
```

Why, yes! I would *love* for you to generate that class for us. *Now*, the spec passes.

growVelociraptor() Example

Our *first* goal is to move the `growVelociraptor()` method into `DinosaurFactory`, but I want to follow the red, green, refactor cycle. So first, describe that functionality with a new example: `function it_grows_a_large_velociraptor()`. Then, call the method: `$dinosaur = $this->growVelociraptor(5)`.

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 16
17     function it_grows_a_large_velociraptor()
18     {
19         $dinosaur = $this->growVelociraptor(5);
↕ // ... line 20
21     }
↕ // ... lines 22 - 23
```

The Magic Behind phpspec's Subject

Eventually, after coding all of this up, we know that the `$dinosaur` variable *should* be a `Dinosaur` object. But we *also* know that phpspec adds a lot of magic. Check this out: `var_dump($dinosaur)`. Now, run phpspec:

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 16
17     function it_grows_a_large_velociraptor()
18     {
↕ // ... line 19
20         var_dump($dinosaur);
21     }
↕ // ... lines 22 - 23
```

```
./vendor/bin/phpspec run
```

First, it notices that the `growVelociraptor()` method is missing. Hit enter to generate that. Ok: scroll up to check out the dumped object. Cool! The `$dinosaur` variable is actually a `Subject` object! Right now, the underlying value is `null` because the new `growVelociraptor()` method doesn't return anything.

But more importantly, do you remember where we saw the `Subject` object earlier? It was in `DinosaurSpec`! When we call `$this->getLength()`, that returns the length, but wrapped *inside* of a `Subject` object. Why do we care? Because *that* was the magic layer that allowed us to call `->shouldReturn()`.

Inside `DinosaurFactorySpec`, it's the same thing! `growVelociraptor` will eventually return a `Dinosaur` object, but phpspec wraps that inside a `Subject` object. Thanks to that, we can call real methods on the `Dinosaur` *or* matcher methods. In other words, the `$dinosaur` in this class works pretty much exactly like the `$this` variable in `DinosaurSpec`. In fact, let's steal four lines of code from here. Paste these into the new example and change all of the `$this` to `$dinosaur`. Re-type the "r" in `Dinosaur` and hit tab so PhpStorm adds its `use` statement.

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 16
17     function it_grows_a_large_velociraptor()
18     {
↕ // ... lines 19 - 20
21         $dinosaur->shouldBeAnInstanceOf(Dinosaur::class);
22         $dinosaur->getGenus()->shouldBeString();
23         $dinosaur->getGenus()->shouldBe('Velociraptor');
24         $dinosaur->getLength()->shouldBe(5);
25     }
↕ // ... lines 26 - 27
```

Ok! The `growVelociraptor()` method is still empty, but let's see what phpspec thinks!

```
./vendor/bin/phpspec run
```

Implement the Code

And the tests are red! Step 2: make this work with as little work as possible... or at least without over-engineering it. We can cheat: copy the code from the old `growVelociraptor()` method. I'll keep this method here just as an example. Back in `DinosaurFactory`, paste, change the `new static` to `new Dinosaur`, change the argument to `int $length` and give this a `Dinosaur` return type.

src/Factory/DinosaurFactory.php

```
↕ // ... lines 1 - 6
7  class DinosaurFactory
8  {
9      public function growVelociraptor(int $length): Dinosaur
10     {
11         $dinosaur = new Dinosaur('Velociraptor', true);
12         $dinosaur->setLength($length);
13
14         return $dinosaur;
15     }
16 }
```

Try it out:

```
./vendor/bin/phpspec run
```

Refactor

Green! So *now* we get to step 3: refactor. *This* is our chance to remove duplication or improve things. For example, if I absolutely know that we will add other methods to this class - like `growTyrannosaurus()` - it might make sense to refactor some logic into a new `private` function called `createDinosaur()`. Give this 3 arguments:

`string $genus`, `bool $isCarnivorous` and `int $length`. Copy the first two lines above and make each part dynamic.

src/Factory/DinosaurFactory.php

```
↕ // ... lines 1 - 13
14     private function createDinosaur(string $genus, bool $isCarnivorous, int $length)
15     {
16         $dinosaur = new Dinosaur($genus, $isCarnivorous);
17         $dinosaur->setLength($length);
18     }
↕ // ... lines 19 - 20
```

Finally, the first method can be simplified to: `return $this->createDinosaur()`, passing `Velociraptor`, `true`, and `$length`. We *could* have wrote the code this way initially. But now we can refactor confidently because our tests will *prove* we didn't mess anything up:

src/Factory/DinosaurFactory.php

```
↕ // ... lines 1 - 8
9      public function growVelociraptor(int $length): Dinosaur
10     {
11         return $this->createDinosaur('Velociraptor', true, $length);
12     }
↕ // ... lines 13 - 20
```

```
./vendor/bin/phpspec run
```

Oh. Except... I messed something up:

> *"Return value of `DinosaurFactory::growVelociraptor()` must be an instance of `Dinosaur`, `null` returned."*

Duh! I forgot my `return` statement! And I should have added a return type too. Try it again:

src/Factory/DinosaurFactory.php

```
↕ // ... lines 1 - 13
14     private function createDinosaur(string $genus, bool $isCarnivorous, int $length):
   Dinosaur
15     {
↕ // ... lines 16 - 18
19         return $dinosaur;
20     }
↕ // ... lines 21 - 22
```

```
./vendor/bin/phpspec run
```

Now we *know* it works. To be honest, I love the red, green, refactor cycle, but I also don't *always* do it. Heck, I don't even unit test all my code - only the parts that are complex enough to keep me up at night. But I *do* take one

important lesson from it into everything I do: focus on accomplishing the behavior you need and nothing more. Keep things simple *until* they can't be. And when you get there, write a test first, *then* get crazy.

Next: we'll describe a new class that *depends* on another class. Is it finally time to talk about mocking in phpspec? Well... not so fast...

Chapter 15: Object Dependencies: To Mock, or Not?

Until now, our examples haven't really needed to involve *other* objects. For example, in `DinosaurSpec`, when we call `setLength()`, we pass a *scalar* argument. We haven't had a situation yet where the object that we're describing *depends* on another object. That's something that we need to talk a lot more about.

Pending & Skipped Examples

But first, in `DinosaurFactory`, we have this `growVelociraptor()` method. Eventually we're going to add other methods to grow other things. And just to make sure I don't forget to do that, let's create a new `it_grows_a_triceratops()` example. But I'm *not* actually ready to describe this or implement it yet... I don't know.... maybe there's a big storm coming and I need to get off the island on the last boat or something. So, just leave it blank and run phpspec:

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurFactorySpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 26
27     function it_grows_a_triceratops()
28     {
29     }
30 }
```

```
./vendor/bin/phpspec run
```

Cool! It shows up as a "todo" pending example! Once we come back after the storm, we won't forget!

One other thing you can do, which is a bit less common in apps, but still neat, is to skip tests if you're missing some sort of dependency. For example, create `it_grows_a_small_velociraptor()`.

Let's pretend like we need an outside library that contains a class called `Nanny` in order to create baby velociraptors. If that class doesn't exist, we can throw a `new SkippingException` that says:

> *"Someone needs to look over dino puppies"*

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 31
32     function it_grows_a_small_velociraptor()
33     {
34         if (!class_exists('Nanny')) {
35             throw new SkippingException('Someone needs to look over dino puppies');
36         }
↕ // ... lines 37 - 38
39     }
↕ // ... lines 40 - 41
```

So, no `Nanny` class? Skip the example. If we *do* have it, it will run like normal: `$this->growVelociraptor(1)` and, how about, `->shouldBeAnInstanceOf(Dinosaur::class)`.

spec/Factory/DinosaurFactorySpec.php

```
↕ // ... lines 1 - 31
32     function it_grows_a_small_velociraptor()
33     {
↕ // ... lines 34 - 37
38         $this->growVelociraptor(1)->shouldBeAnInstanceOf(Dinosaur::class);
39     }
↕ // ... lines 40 - 41
```

Since that's just a made-up class, when we run phpspec:

```
./vendor/bin/phpspec run
```

Yep! That one got skipped.

Describing the Enclosure

Ok: now that we have so many Dinosaurs, we should *probably* start thinking about, ya know, keeping them enclosed in some way: right now they're just wandering around the island and causing all kinds of trouble. I think we need a new `Enclosure` class that we can put the dinosaurs inside of. Oh, oh, oh! That means... it's time to *describe* a new class! Woo!

```
./vendor/bin/phpspec describe App/Entity/Enclosure
```

The idea is that, as we create & persist dinosaurs to the database, we will also create & persist Enclosures and put Dinosaur objects inside of them. Before even running this spec class, let's add our first *real* example to make sure that each Enclosure is empty by default - it would be a bit surprising if a new Enclosure automatically had a dinosaur hiding inside: `it_should_have_no_dinosaurs_by_default()`.

And, because we will probably need a way to ask what dinosaurs are inside, let's say: `$this->getDinosaurs()->shoudHaveCount(0)`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 8
9  class EnclosureSpec extends ObjectBehavior
10 {
↕ // ... lines 11 - 15
16     function it_should_have_no_dinosaurs_by_default()
17     {
18         $this->getDinosaurs()->shouldHaveCount(0);
19     }
20 }
```

Ok, good start! Head back over to your terminal and run things:

```
./vendor/bin/phpspec run
```

Enter `yes` to generate that class, and yes to generate the `getDinosaurs()` method inside of it.

Thanks to that, not *only* do we have the new `Enclosure` class, but it already has its first method!

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 4
5  class Enclosure
6  {
7      public function getDinosaurs()
8      {
9          // TODO: write logic here
10     }
11 }
```

Basic Enclosure Implementation

To get started, we probably need a `$dinosaurs` property, which will hold an array of `Dinosaur` objects. Add an array return type to the method and return `$this->dinosaurs`. Oh, and let's initialize the property to an empty array - that's exactly the behavior we're describing.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 4
5  class Enclosure
6  {
7      /** @var Dinosaur[] */
8      private $dinosaurs = [];
9
10     public function getDinosaurs(): array
11     {
12         return $this->dinosaurs;
13     }
14 }
```

We *could* have just returned a hardcoded empty array... because that *is* the minimum code we need to get the test to pass. But as you get more comfortable with phpspec, it's ok to start skipping that step - as long as you stay focused

on the behavior you need and don't allow yourself to get too fancy.

Let's make sure things are passing:

```
./vendor/bin/phpspec run
```

Perfect! Just the one, pending example.

Adding Dinosaurs to the Enclosure

Let's think a bit more about the `Enclosure`. We will definitely need a way to to add dinosaurs to it. Let's describe that! `function it_should_be_able_to_add_dinosaurs()`. And because we'll most likely be adding Dinosaurs one-by-one as they're born, I think an `addDinosaur()` method will be quite perfect: `$this->addDinosaur()` and pass that a `Dinosaur` object.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 9
10 class EnclosureSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 21
22     function it_should_be_able_to_add_dinosaurs()
23     {
24         $this->addDinosaur(new Dinosaur());
↕ // ... lines 25 - 27
28     }
29 }
```

Mock the Dinosaur Object?

But wait! This is the *first* time that we're calling a method on our object and what we need to pass to that method is... another object! Ok... so what's the big deal? Remember, in unit tests, each class is supposed to be tested in complete isolation. If you have a class that depends on a database connection, instead of passing the *real* database connection object, you're supposed to pass it a *mock* object so that it doesn't make *real* database queries and also so we can fake and control the return value of its methods.

So... question time: should we mock the `Dinosaur` object? And if so, how do we do that?

The answer is... probably no: we should *not* mock it. Whenever you need to pass an object to the object you're testing, you need to decide whether or not to mock it. And the correct answer depends on how *difficult* it is to instantiate the object and control its behavior. For example, the `Dinosaur` object is a simple model object, and it doesn't really *do* anything - it just holds data. It's easy to instantiate and, if we want its `getLength()` method to return 7 to help us test something, yea, that's super easy! Just set its length to 7!

The point is: the `Dinosaur` object is *so* simple, that mocking will work, but it will make your life harder! That's why I prefer to pass in the real object.

If this were a database connection object, something that sent emails or any other class that did some real work, I *would* mock it, and we'll talk about how to mock things in phpspec soon.

Let's copy this line so we can add two dinosaurs. And then say `$this->getDinosaurs()->shouldHaveCount(2)`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 21
22     function it_should_be_able_to_add_dinosaurs()
23     {
24         $this->addDinosaur(new Dinosaur());
25         $this->addDinosaur(new Dinosaur());
26 
27         $this->getDinosaurs()->shouldHaveCount(2);
28     }
↕ // ... lines 29 - 30
```

Ok, let's try it!

```
./vendor/bin/phpspec run
```

Woo! Sweet phpspec failure - let it generate the new method. Then, flip back and find that new method. Change the argument to `Dinosaur $dinosaur`. And inside the method, `$this->dinosaurs[] = $dinosaur`.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 4
5  class Enclosure
6  {
↕ // ... lines 7 - 14
15     public function addDinosaur(Dinosaur $dinosaur)
16     {
17         $this->dinosaurs[] = $dinosaur;
18     }
19 }
```

Did we mess anything up? Find out:

```
./vendor/bin/phpspec run
```

Definitely not... because our tests are green!

Next, let's talk about how we can test *exceptions*, including exceptions that might happen when your object is being constructed. Oh, and we'll use a cool `ObjectMatcher` that lets you test methods that return a boolean in a really smooth way.

Chapter 16: Expecting Exceptions

Sometimes your code will throw an exception... it's just how things are. Actually, *sometimes* it's super important that the *right* exception *is* thrown at the exact right time. Let's see an example - and then, see how to describe that behavior in a spec class.

If you downloaded the course code, you should have a `tutorial/` directory. Find the `Exception/` directory inside of that and copy that *whole* darn thing into `src/`. This holds two exception classes, and the first one that we're going to look at is, the very important, `NotABuffetException`. You see, we've had this problem where sometimes we accidentally put a veggiesaurus inside an `Enclosure` with a carnivorous dinosaur. And, well, the results have been... messy.

src/Exception/NotABuffetException.php

```
// ... lines 1 - 4
class NotABuffetException extends \Exception
{
    protected $message = 'Please do not mix the carnivorous and non-carnivorous dinosaurs. It will be a massacre!';
}
```

To make sure we stop doing that, we need to throw this exception if we try to mix carnivorous and non-carnivorous dinosaurs into the same Enclosure. And this is *such* an important thing, we need to make sure there is a test to *ensure* the carnage stops.

Example for an Exception

Open up `EnclousreSpec`: because we want the exception to be thrown when the `addDinosaur()` method is called. Let's say: `function it_should_not_allow_to_add_carnivorous_dinosaurs_to_non_carnivorous_enclosure()`.

spec/Entity/EnclosureSpec.php

```
// ... lines 1 - 10
class EnclosureSpec extends ObjectBehavior
{
// ... lines 13 - 30
    function it_should_not_allow_to_add_carnivorous_dinosaurs_to_non_carnivorous_enclosure()
    {
// ... lines 33 - 37
    }
}
```

Wow! That's a long name - but... ok! It's a great description for this example.

Here's the plan: we're going to add one dinosaur that's a veggie dinosaur and then add a another dinosaur that's carnivorous. And *that* should trigger the exception. Start with `$this->addDinosaur(new Dinosaur())` and make this a veggie eater by passing `false` as the second argument.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 30
31     function it_should_not_allow_to_add_carnivorous_dinosaurs_to_non_carnivorous_enclosure()
32     {
33         $this->addDinosaur(new Dinosaur('veggie-eater', false));
↕ // ... lines 34 - 37
38     }
↕ // ... lines 39 - 40
```

Now, here is the important part: *before* we call `addDinosaur()`, again, we need to tell phpspec to *expect* that there should be an exception. And it's probably no surprise that the *language* to do this is *really* natural: `$this->shouldThrow()` `NotABuffetException::class`, `->during()`, and then we tell phpspec exactly what method should trigger this: `addDinosaur` with an array of the arguments, and we only have one: `new Dinosaur()`, `Velociraptor` and `true` for the carnivorous argument.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 30
31     function it_should_not_allow_to_add_carnivorous_dinosaurs_to_non_carnivorous_enclosure()
32     {
↕ // ... lines 33 - 34
35         $this
36             ->shouldThrow(NotABuffetException::class)
37             ->during('addDinosaur', [new Dinosaur('Velociraptor', true)]);
38     }
↕ // ... lines 39 - 40
```

That's it! Let's try it out:

```
./vendor/bin/phpspec run
```

Cool! Failure because no exception was thrown.

Implementing the Code

Time for us to get to work! In `addDinosaur()`, we need to determine whether or not we're allowed to add this `Dinosaur`. Let's call a new function: `if (!$this->canAddDinosaur())` - we'll create that method in a minute - then `throw new NotABuffetException()`.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 16
17     public function addDinosaur(Dinosaur $dinosaur)
18     {
19         if (!$this->canAddDinosaur($dinosaur)) {
20             throw new NotABuffetException();
21         }
↕ // ... lines 22 - 23
24     }
↕ // ... lines 25 - 32
```

Now I can click back on `canAddDinosaur`, press Alt + Enter and click "Add Method" to create a new `private` method at the bottom. Oh, and I'm just creating this as a private method for code organization: I could have written the logic right up in the `addDinosaur()` function. But, it *is* nice to have a method called `canAddDinosaur()` - it's really clear. I'm not making it `public` because, at least so far, we don't have any need to use it outside of this class. That *also* means that we won't write an example for this function: examples are only for public methods.

Anyways, let's add a return type and then say `return count($this->dinosaurs) === 0` - because if there are no dinosaurs in this enclosure, than we can *definitely* add one - *or* we can check if the first dinosaur, index 0, has the same "diet" as the dinosaur being added, it should be allowed. So... hmm, maybe we'll call `->isCarnivorus()`.

But, wait... that method does not exist in the `Dinosaur` class. And actually, the *real* problem is that the `isCarnivorous` information is not available publicly in *any* way, except as part of the description.

This is cool! We just discovered that we need to enhance the `Dinosaur` class to get the new feature working. Before we do that, let's finish the `canAddDinosaur()` method: you should be able to add the dinosaur if the first dinosaur `->isCarnivorous()` value equals `$dinosaur->isCarnivorous()`. *If* they are compatible, this dinosaur *can* be added to the enclosure... without being eaten... hopefully.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 25
26      private function canAddDinosaur(Dinosaur $dinosaur): bool
27      {
28          return count($this->dinosaurs) === 0 ||
29              ($this->dinosaurs[0]->isCarnivorous() === $dinosaur->isCarnivorous());
30      }
↕ // ... lines 31 - 32
```

We know we're not done yet, but in the spirit of "doing as little work as possible and letting phpspec tell us what to do next", let's run it now:

```
./vendor/bin/phpspec run
```

Failure!

> *"Call to undefined method `Dinosaur::isCarnivorous()`."*

We *could* now go directly into the `Dinosaur` class and just... implement that! It's a super-easy method. But... don't we need to write an example first before we add the code? Maybe. So far, we've been testing a lot of simple getter and setter methods. You *can* test simple methods like this, but at some point a method is *so* simple that... in my opinion, testing them is overkill. Focus on testing what scares you.

But, for our great learning adventure, we *will* add some examples for this method. Why? Because it will introduce us to a really cool matcher for boolean methods. Let's check it out next.

Chapter 17: The ObjectStateMatcher

We're missing the `isCarnivorous()` method on `Dinosaur`. It will be a *really* simple method, but because it will let me show you a very special match - the `ObjectStateMatcher` - we're going to write a couple of examples for it.

Find `DinosaurSpec` and add the first new example: `it_should_be_herbivore_by_default()`. This example is meant to show that if we create a new `Dinosaur()` without passing the `isCarnivorous` argument, it should default to an herbivore. Pay careful attention to the *language* I'm about to use. Behind the scenes, because we haven't controlled the constructor arguments, we know that phpspec will create a `Dinosaur()` without *any* constructor arguments. So I'm going to say `$this->shouldNotBeCarnivorous()`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 79
80     function it_should_be_herbivore_by_default()
81     {
82         $this->shouldNotBeCarnivorous();
83     }
↕ // ... lines 84 - 90
91 }
```

That is *not* a built-in matcher... right? I mean, that's *super* specific language, and PhpStorm definitely did *not* autocomplete that for me. Well... surprise! That *is* a real matcher! Say hello to the `ObjectStateMatcher`! It's dynamic: whenever you say `shouldBeSOMETHING` or `shouldNotBeSOMETHING()`, the `ObjectStateMatcher` is activated. It parses out that "SOMETHING" part - for us the word `Carnivorous` - looks for a method called `isCarnivorous()`, and then checks that it equals true or, in our case, false, because we're using should *not*.

Let's write one more example before we see this. How about: `it_should_allow_to_check_if_dinosaur_is_carnivorous()`. Inside, use `$this->beConstructedWith()`. Remember, the `Dinosaur` class's constructor allows us to control the `$isCarnivorous` value via the *second* argument. So, we'll pass `'Velociraptor'` and `true`. Then we can say `$this->shouldBeCarnivorous()`. That's the same thing as above, but without the "not".

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 84
85     function it_should_allow_to_check_if_dinosaur_is_carnivorous()
86     {
87         $this->beConstructedWith('Velociraptor', true);
88
89         $this->shouldBeCarnivorous();
90     }
↕ // ... lines 91 - 92
```

Let's check it out!

```
./vendor/bin/phpspec run
```

Woh! The error from our two new examples is:

> *"method [array:2] not found"*

Um... cool... so what the heck does that mean? It's not a great error. But, the question at the bottom tells us what is *really* going on:

> *"Do you want me to create Dinosaur::isCarnivorous() for you?"*

This is really cool! When we say `shouldBeCarnivorous()`, the `ObjectStateMatcher` says:

> *"Hey! You're describing that your object "should be carnivorous". To figure that out, your class probably needs an `isCarnivorous()` method! So, let's create that thing!"*

Choose yes to generate it. phpspec re-runs and *does* fail - something about the `ObjectStateMatcher` expecting a boolean, but null given. That's because our new method is empty! Go find it, then return `$this->isCarnivorous` and add the `bool` return type.

src/Entity/Dinosaur.php

```
// ... lines 1 - 4
class Dinosaur
{
// ... lines 7 - 51
    public function isCarnivorous(): bool
    {
        return $this->isCarnivorous;
    }
}
```

Find your terminal and run phpspec!

```
./vendor/bin/phpspec run
```

Sweet! Everything passes! Which includes both new examples inside `DinosaurSpec` *and* the original example in `EnclosureSpec` because `Enclosure` can now use the new `isCarnivorous()` method.

ObjectStateMatcher with "shouldHave"

Now that our tests are green, we get to think about any refactoring we might want to do. Here's one piece I don't love: this logic for comparing whether or not the diet of two Dinosaurs is the same - it's just not super clear.

So, it might be nice to have a method on the `Dinosaur` class called `hasSameDietAs()`: we pass it a `Dinosaur` and it returns a boolean.

So... cool! Let's add an example for this: `it_should_allow_to_check_if_two_dinosaurs_have_same_diet`. And, check out the language `$this->shouldHaveSameDietAs(new Dinosaur())`.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 91
92     function it_should_allow_to_check_if_two_dinosaurs_have_same_diet()
93     {
94         $this->shouldHaveSameDietAs(new Dinosaur());
95     }
96 }
```

Two important things here. First, the `$this` object will be a `Dinosaur` object that's created with no constructor args - so it will be a veggiesaurus. And so, it should have the same diet as a `new Dinosaur()`, which will *also* be non-carnivorous.

Second, see this language - `shouldHaveSameDietAs()`? That language will *also* be handled by the `ObjectStateMatcher`. Yep, when you say `shouldBeSomething`, it looks for an `isSomething()` method. And if you say `shouldHaveSomething()`, it looks for a `hasSomething()` method.

One of the problems *I* originally had with the `ObjectStateMatcher` was that I was thinking about it backwards. I was thinking:

> *"Hey! I want to have a method called `isCarnivorous()`."*

Then, I would try to figure out the correct matcher method to use - like `shouldBeCarnivorous()` - so that it would look for this method. But really, we need to think about it the other direction: I shouldn't care what the method name will be called in `Dinosaur`. Nope, I can ignore that and focus on using natural language in my example: `$this->shouldBeCarnivorous()` and down here `$this->shouldHaveSameDietAs()`. Use natural language, and *then*... don't even think about the method name! Just run phpspec - it'll tell you:

```
./vendor/bin/phpspec run
```

There it is! `hasSameDietAs()`. Generate that, then go find it. This method will return a `bool`, the argument will be a `Dinosaur` object and we can `return $dinosaur->isCarnivorous() === $this->isCarnivorous()`.

src/Entity/Dinosaur.php

```
// ... lines 1 - 4
class Dinosaur
{
// ... lines 7 - 56
    public function hasSameDietAs(Dinosaur $dinosaur): bool
    {
        return $dinosaur->isCarnivorous() === $this->isCarnivorous();
    }
}
```

Let's try it!

```
./vendor/bin/phpspec run
```

We are green! Let's take that as a sign that it's safe to do a bit of refactoring inside `Enclosure`. Remove all this complicated stuff and, at the end, just say: `|| $dinosaur->hasSameDietAs($this->dinosaurs[0])`.

src/Entity/Enclosure.php

```
// ... lines 1 - 6
class Enclosure
{
// ... lines 9 - 25
    private function canAddDinosaur(Dinosaur $dinosaur): bool
    {
        return count($this->dinosaurs) === 0 || $dinosaur->hasSameDietAs($this->dinosaurs[0]);
    }
}
```

Run phpspec one more time:

```
./vendor/bin/phpspec run
```

Got it! Next, let's talk a bit more about testing exceptions and finally add some Security to our dino park.

Chapter 18: Describing for Exception Messages

In the `Exception/` directory that we copied a few minutes ago, there's another exception class: the `DinosaursAreRunningRampantException`.

src/Exception/DinosaursAreRunningRampantException.php

```
↕ // ... lines 1 - 4
5 final class DinosaursAreRunningRampantException extends \Exception
6 {
7 }
```

Here's the problem we're facing: we have these enclosures, but... they don't have any security - no electric fences, no guard towers, nothing! We need to add that capability to enclosures *and* throw this new exception *if* we try to add a `Dinosaur` to an `Enclosure` that has no active security. Because... honestly... we're having a *real* problem where people add dinosaurs to an enclosure and then just leave the door *wide* open.

In `EnclosureSpec`, let's create a new example to describe this: `it_should_not_allow_to_add_dinosaurs_to_unsecure_enclosures()`. I want you to temporarily ignore *all* the other examples that we've been working on so far, because this example is going to temporarily break... all of them.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 11
12 class EnclosureSpec extends ObjectBehavior
13 {
↕ // ... lines 14 - 40
41     function it_should_not_allow_to_add_dinosaurs_to_unsecure_enclosures()
42     {
↕ // ... lines 43 - 45
46     }
47 }
```

First... how does "security" for our enclosures need to be designed? Is it a boolean property on Enclosure so we can just turn security on or off? Something more complex? Actually, right now, it doesn't matter!

Check it out: in this example, I want to describe that if you *simply* create a new `Enclosure`, that is not enough: it's not secure. Describe that by saying `$this->shouldThrow()`. And this time, instead of passing the class name of the exception that should be thrown, I'll say `new DinosaursAreRunningRampantException()` and pass this a message: `Are you craaazy?!?`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 40
41     function it_should_not_allow_to_add_dinosaurs_to_unsecure_enclosures()
42     {
43         $this
44             ->shouldThrow(new DinosaursAreRunningRampantException('Are you craaazy?!?'))
↕ // ... line 45
46     }
↕ // ... lines 47 - 48
```

Why am I doing this differently than before? It's really up to you: you can pass the class name to `shouldThrow()` if you *only* need to make sure the exception is an instance of that class *or* if you want to make sure that the message is also correct, you can create the exception object with the message you expect.

Next, the exception should be thrown `->duringAddDinosaur(new Dinosaur())` with `Velociraptor` and `true`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 40
41     function it_should_not_allow_to_add_dinosaurs_to_unsecure_enclosures()
42     {
43         $this
↕ // ... line 44
45             ->duringAddDinosaur(new Dinosaur('Velociraptor', true));
46     }
↕ // ... lines 47 - 48
```

Oh, and this language is *also* a bit different than before. Earlier, we used `during()` and passed `addDinosaur` as an argument to that method. That's fine, but you can also use this more magical way: `duringAddDinosaur()`. It's a bit more natural because you can then pass each argument one-by-one, instead of putting them in an array. They do the same thing - so it's up to you.

The point is: we *now* have a test that describes that you *can't* just create an `Enclosure` and start putting dinosaurs into it. Somehow, security needs to be activated... whatever that means.

Let's move over and run phpspec:

```
./vendor/bin/phpspec run
```

Awesome! That *does* fail because it *is* still possible to add dinosaurs to enclosures without activating security.

Designing the Security

Ok... so how do we need enclosure security to work? The right answer depends on your dinosaur park. But the process is universal: think about what *requirements* you have. Is security just something you turn on or off - like with some `activateSecurity()` method on `Enclosure`? Or, is it more complex? Based on talking to our security experts, I've determined that we need the ability to add different types of security to different Enclosures - electric

fences around some, guard towers around others and maybe just a sign that says "Please stay inside" if we get really busy. Oh, and each `Enclosure` can have 0 or many pieces of security.

Back in the `tutorial/` directory, check out the `Entity/` directory. See that `Security` class? Copy that and put into our `src/Entity/` folder. *Not* `spec/Entity`, I'm *totally* messing this up right now... and will pay for it later.

src/Entity/Security.php

```
↕ // ... lines 1 - 4
5  class Security
6  {
7      private $name;
8
9      private $isActive;
10
11     private $enclosure;
12
13     public function __construct(string $name, bool $isActive, Enclosure $enclosure)
14     {
15         $this->name = $name;
16         $this->isActive = $isActive;
17         $this->enclosure = $enclosure;
18     }
19
20     public function getIsActive(): bool
21     {
22         return $this->isActive;
23     }
24 }
```

And, yes, yes, we're going to cheat a bit: we're going to skip the spec process for the `Security` class and start with something I've already created.

Each `Security` has a name - like "electric fence" or "guard tower" and a boolean for whether it's active or not. And we pass in the `Enclosure` that this `Security` will be attached to. For the methods - just one: `getIsActive()`.

To get the example in `EnclosureSpec` to pass, somehow, we need a way to attach `Security` objects to our `Enclosure` class. And then, when we add a dinosaur, we can check to make sure the `Enclosure` has at least one active `Security`.

Ok... cool! To hold the securities, let's create a `$securities` property and set it to an empty array. This will be an array of `Security` objects, so let's document that.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 7
8  class Enclosure
9  {
↕ // ... lines 10 - 12
13     /** @var Security[] */
14     private $securities = [];
↕ // ... lines 15 - 48
49 }
```

Now, this is interesting. If we're adding a `securities` property, shouldn't we describe this more directly with some examples that show... I don't know... some `addSecurity()` or `getSecurities()` methods? Well... maybe? Maybe because... we might not need these methods! Right now, what we *do* know is that, if there are no active securities, an exception should be thrown. And of course, we *will* need to update some of our examples from earlier once we get this working so that they also have some active security.

Anyways, down in `addDinosaur()`, let's call another new method `if (!$this->isSecurityActive())` we will throw a `new DinosaursAreRunningRampantException()` and pass it the same message that we described in our example - because we're testing for this *exact* string.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 20
21     public function addDinosaur(Dinosaur $dinosaur)
22     {
23         if (!$this->isSecurityActive()) {
24             throw new DinosaursAreRunningRampantException('Are you craaazy?!?');
25         }
↕ // ... lines 26 - 31
32     }
↕ // ... lines 33 - 50
```

In reality, this is a bit silly. In real life, I probably wouldn't care enough to test for that exact message - the class is enough.

To add the missing method, I'll put my cursor on the method name, hit Alt + Enter and click "Add Method". Cool! This will return a `bool` and inside, we can loop over the `$securities` with `$this->securities as $security`. If at least *one* `Security` object attached to this enclosure is active, then our enclosure is secure. So `if ($security->getIsActive())`, then `return true`. And if *none* of them are active, `return false`.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 38
39     private function isSecurityActive(): bool
40     {
41         foreach ($this->securities as $security) {
42             if ($security->getIsActive()) {
43                 return true;
44             }
45         }
46
47         return false;
48     }
↕ // ... lines 49 - 50
```

Okay, that should work! Move back to the terminal. Oh, see this 41? That means the example lives on line 41 of the spec class. Re-run phpspec:

```
./vendor/bin/phpspec run
```

It works! Sort of. Notice, line 41 *is* gone - that example is passing! By the way, instead of running *all* your spec classes, you can run just one by passing the filename to the command:

```
./vendor/bin/phpspec run spec/Entity/EnclosureSpec.php
```

Or, you can run just *one* example by adding colon then the line number. The example we're working on should be line 41 - yep! There it is. Try it:

```
./vendor/bin/phpspec run spec/Entity/EnclosureSpec.php:41
```

Cool! 1 passed. But if we run all of them, we have a few failures.

We made a few changes to our app that *broke* our existing examples. Next, let's think about the correct way to handle this and add a few more nice features to our `Enclosure`... including testing exceptions that happen during object construction.

Chapter 19: When Existing Tests Break & Exceptions in __construct()

The new example we just added *is* passing... but we totally broke a *bunch* of our original examples! Lame! For example, `it should be able to add dinosaurs` is getting the `DinosaursAreRunningRampantException` and that makes sense. Find `it_should_be_able_to_add_dinosaurs`. It's testing to make sure that if we add two dinosaurs, then we should have... 2 dinosaurs.

Existing Tests Broke... now what?

So, hmm: we made a change to our app and then an existing example started to fail. That's... the beauty of tests! Now we can take in all this information about which tests are failing and why they're failing and determine the best way forward. For example, we may have just accidentally introduced a bug and we want to fix that bug. Or we may discover that the tests that are failing are no longer relevant and should be removed. Or, most likely you're in a situation like this one: where you simply need to update existing tests for some new change.

Describing adding Security

For this example, the enclosure needs some security before we starting adding dinosaurs. To do that, well... we need some way to add `Security` to an `Enclosure`! And, ya know what? We should probably make it pretty easy to add security - maybe via a new constructor arg.

Let me show you what I mean - in an example! Instead of creating a totally new example function for this, I'm going to need to use this new functionality in the example that's current failing. Here's the idea: to add some basic security, say `$this->beConstructedWith(true)`. Yep, I want there to be a constructor arg that easily allows you to activate *some* type of security.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 11
12 class EnclosureSpec extends ObjectBehavior
13 {
↕ // ... lines 14 - 23
24     function it_should_be_able_to_add_dinosaurs()
25     {
26         $this->beConstructedWith(true);
↕ // ... lines 27 - 31
32     }
↕ // ... lines 33 - 52
53 }
```

Let's also add this to the other example that's failing - it's around line 32. Paste! And for the newest example we've been working on, I'll instantiate with `false`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 33
34     function it_should_not_allow_to_add_carnivorous_dinosaurs_to_non_carnivorous_enclosure()
35     {
36         $this->beConstructedWith(true);
↕ // ... lines 37 - 42
43     }
↕ // ... line 44
45     function it_should_not_allow_to_add_dinosaurs_to_unsecure_enclosures()
46     {
47         $this->beConstructedWith(false);
↕ // ... lines 48 - 51
52     }
↕ // ... lines 53 - 54
```

Ok, let's try the tests:

```
./vendor/bin/phpspec run
```

Nice! It asks us to generate the `__construct` method - yes please! A whole bunch of examples are failing - but, pfff - that's probably fine. Find `Enclosure`. Perfect!

Change the argument to `bool $withBasicSecurity` and... I don't need to, but let's give this a default value: if you pass nothing, there is no security. Next, `if ($withBasicSecurity)`, let's... add some security! I'll call a new method we haven't created yet: `$this->addSecurity()` with `new Security()` passing that fence... or I guess "fency", whatever that is... `true` to make it active and `$this` because it will be attached to *this* `Enclosure.`

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 7
8  class Enclosure
9  {
↕ // ... lines 10 - 15
16     public function __construct(bool $withBasicSecurity = false)
17     {
18         if ($withBasicSecurity) {
19             $this->addSecurity(new Security('Fence', true, $this));
20         }
21     }
↕ // ... lines 22 - 60
61 }
```

For the `addSecurity()` method, because we're not using this method anywhere outside of this class, we should technically create it as private. But, I already know that I *will* need to use it outside this class, so let's make it public: `public function addSecurity(Security $security)`. Inside, `$this->securities[] = $security`.

src/Entity/Enclosure.php

```
// ... lines 1 - 40
    public function addSecurity(Security $security)
    {
        $this->securities[] = $security;
    }
// ... lines 45 - 62
```

Phew! Okay, find your terminal and let's try this!

```
./vendor/bin/phpspec run
```

Hmm, not passing yet:

> *"Class `App\Entity\Security` not found."*

Let's see, what did I mess up? It's not clear where that error is. To get more info, use the `--verbose` flag:

```
./vendor/bin/phpspec run --verbose
```

And... ah! There it is: `Enclosure` at line 19. Find that and... ah again! Our `spec/` and `src/` directories look *so* much alike that I copied the `Security` class into the wrong spot! Move that into `src/Entity` - good job tests!

Run 'em again:

```
./vendor/bin/phpspec run
```

Now they pass.

Exception during Construction???

We're on a roll! Shall we add one more enhancement to `Enclosure`? It's now easy to create an `Enclosure` with basic security. But I *also* want the ability to pass some initial dinosaurs into the constructor. That's cool - but another programmer tried to do this last week and... oof, things got ugly. They *did* allow for initial dinosaurs to be added, but they *forgot* to check *first* to see if any security was active. Oof. Anyways, that programmer is... "unavailable" now.

Let's not make the same mistake: create a new example function: `it_should_fail_if_providing_initial_dinosaurs_without_security()`. Start with `$this->beConstructedWith(false)` and an array with one `new Dinosaur()`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 53
54     function it_should_fail_if_providing_initial_dinosaurs_without_security()
55     {
56         $this->beConstructedWith(false, [new Dinosaur()]);
↕ // ... lines 57 - 60
61     }
↕ // ... lines 62 - 63
```

Then, we just need to tell phpspec what method will cause the exception. So... wait! This is a bit different: the exception will be thrown during instantiation! Not when we call some other method.

How can we tell phpspec about that? It's *almost* the same: `$this->shouldThrow()` with `DinosaursAreRunningRampantException::class` - I don't care about testing the exact message. Then, `->duringInstantiation()`.

spec/Entity/EnclosureSpec.php

```
↕ // ... lines 1 - 53
54     function it_should_fail_if_providing_initial_dinosaurs_without_security()
55     {
↕ // ... lines 56 - 57
58         $this
59             ->shouldThrow(DinosaursAreRunningRampantException::class)
60             ->duringInstantiation();
61     }
↕ // ... lines 62 - 63
```

That's it. Let's make sure things are failing!

```
./vendor/bin/phpspec run
```

Nice and broken: there was no exception thrown. Oh, but notice one thing: this is the first time that we've added an example where we are passing a *second* argument to the constructor. But, because that method already exists, phpspec is not *quite* smart enough to automatically generate a second argument for us. Ok, then, I *guess* we'll do it by hand: add `array $initialDinosaurs = []`.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 7
8  class Enclosure
9  {
↕ // ... lines 10 - 15
16     public function __construct(bool $withBasicSecurity = false, array $initialDinosaurs =
   []) 
17     {
↕ // ... lines 18 - 24
25     }
↕ // ... lines 26 - 64
65 }
```

Next, foreach over `$initialDinosaurs as $dinosaurs` and say, `this->addDinosaur($dinosaur)`. *That* was the mistake that *other* programmer made: I'm using `addDinosaur()` instead of just setting the `$dinosaurs`

property directly because *that* method contains the security checks.

src/Entity/Enclosure.php

```
↕ // ... lines 1 - 15
16     public function __construct(bool $withBasicSecurity = false, array $initialDinosaurs =
       [])
17     {
↕ // ... lines 18 - 21
22         foreach ($initialDinosaurs as $dinosaur) {
23             $this->addDinosaur($dinosaur);
24         }
25     }
↕ // ... lines 26 - 66
```

So... that should be it! Let's try phpspec:

```
./vendor/bin/phpspec run
```

Got it. Next... it's time! It's time to talk about mocking, test doubles and all that fun, testing magic.

Chapter 20: Test Doubles

It's *finally* time to talk about one of the most critical parts of unit testing: mocking. Oh, and it's kind of the most fun part too!

To Mock or Not to Mock

Check out `EnclosureSpec`: we already had at *least* one situation where we called a method and needed to pass *another* object as an argument - a `Dinosaur` in this case. When the object you're testing has a *dependency* on another object like this, you have two options. First, you can just pass the real object, and that's what we've been doing so far. This is a simple and excellent solution when the object you're passing is easy to instantiate and doesn't have any side effects - most commonly, objects that just hold data.

The *second* solution - mocking - is perfect for all the *other* situations: when the object you're using is a pain to instantiate, its behavior is complex or its methods *do* things - like it makes database queries. In those cases, we do *not* want to use the real object: we want to mock... or create a test double... or a dummy. These are all terms that sorta describe the same thing - we'll discuss as we go along.

Creating a Mock Object

Let's see this in action! Create a new example: `function it_should_allow_to_check_if_two_dinosaurs_have_same_diet_using_stub()`. Yea, we'll discuss that word "stub" along the way.

Check this out: instead of creating a new `Dinosaur` object, add an *argument* to the example method with a `Dinosaur` type-hint. Let's `var_dump($dinosaur)` and then see what happens when we run phpspec:

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 9
10 class DinosaurSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 96
97     function it_should_allow_to_check_if_two_dinosaurs_have_same_diet_using_stub(Dinosaur
   $dinosaur)
98     {
99         var_dump($dinosaur);
100    }
101 }
```

```
./vendor/bin/phpspec run
```

Oh... interesting! It's some sort of a `Collaborator` object. But what I *really* want you to see is that, inside of it, is something called an `ObjectProphecy`. Woh, cool name.

Technically speaking, phpspec doesn't have its own mocking system - it uses a totally independent library called prophecy. Well, the truth is that the phpspec team made and maintains both libraries - but prophecy *is* its own library, and can even be used in PHPUnit.

But the point is, this is *not* a real `Dinosaur` object, it's a "fake" object that looks and *smells* like a `Dinosaur` object and one that we can completely control. And getting a mock object is easy! Just add an argument type-hinted with the class or interface you need to mock - phpspec & prophecy take care of the rest. I *love* that.

Controlling Method Return Values

So... what can we do with this `$dinosaur` mock? Well, you could take *full* control over the return value of *any* of its methods. Or you can check to make sure that one of its methods was called. We have 100% control over how this object behaves.

For this example, we're testing that the `hasSameDietAs()` method behaves correctly. We're basically doing the *same* example as before, but with a mock. And so, when our code calls `isCarnivorous()` on the mocked `Dinosaur`, we need that to return false.

Cool - let's tell our mock about this: `$dinosaur->isCarnivorous()->willReturn(false)`. I like that! It feels a *lot* like normal phpspec code! Except instead of `getGenus()->shouldBe()` to assert a return value, we're instead *training* the mock: we're *teaching* it how it should behave.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 96
97     function it_should_allow_to_check_if_two_dinosaurs_have_same_diet_using_stub(Dinosaur
   $dinosaur)
98     {
99         $dinosaur->isCarnivorous()->willReturn(false);
↕ // ... lines 100 - 101
102    }
↕ // ... lines 103 - 104
```

Now we can say `$this->shouldHaveSameDietAs($dinosaur)` - remembering that `$this` will *not* be carnivorous, because it was constructed with no arguments.

spec/Entity/DinosaurSpec.php

```
↕ // ... lines 1 - 96
97     function it_should_allow_to_check_if_two_dinosaurs_have_same_diet_using_stub(Dinosaur
   $dinosaur)
98     {
↕ // ... lines 99 - 100
101        $this->shouldHaveSameDietAs($dinosaur);
102    }
↕ // ... lines 103 - 104
```

Cool! So let's see what phpspec thinks:

```
./vendor/bin/phpspec run
```

Ha! That actually passes!

Mocks, test doubles, spies, stubs, Larry

These fake objects are called test doubles, but you'll hear them called by a number of other names as well, like `stubs`, `spies`, `mocks` and sometimes even `Larry`. When you hear these words, they're all basically referring to the same idea, though *technically*, each word - like `stub` or `spy` refer to different cool "things" that you can do with these objects.

For example, when you want to control the return value of an object, then suddenly this "fake" object is known as a stub. So, in our example, `$dinosaur` is technically a stub. Later, we're going to do things like assert that a certain method was called. Like, we could say: I want to assert that the `isCarnivorous()` method was called exactly one time. When we do *that*, the test double object will be known as a spy or a mock.

The point is: these terms are all different ways to describe the same idea of getting a fake object from phpspec and then either training it to have some sort of behavior or asserting that certain methods were called on it. To some people, this distinction is super important. For me, I can never remember the difference, and I don't care that much. Though, as we'll see later, prophecy's documentation uses these words a lot - so it's good to know a little bit about them.

But before we get there, let's add another service to our application - an `EnclosureBuilderService`. This will let us build enclosures faster and, more importantly, is going to be a kick-butt example for mocking.

Chapter 21: The EnclosureBuilderService

Because we're creating a *lot* of dinosaurs *and* a lot of enclosures, I think it might be a good idea to get organized and create a new helper service class to do all of this for us! We'll call it `EnclosureBuilderService`. You know what that means... time to describe!

```
./vendor/bin/phpspec describe App/Service/EnclosureBuilderService
```

That creates the new spec class. And *thanks* to that new spec class, we can generate the class immediately with:

```
./vendor/bin/phpspec run
```

Booya!

spec/Service/EnclosureBuilderServiceSpec.php

```
// ... lines 1 - 7
class EnclosureBuilderServiceSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType(EnclosureBuilderService::class);
    }
}
```

src/Service/EnclosureBuilderService.php

```
// ... lines 1 - 4
class EnclosureBuilderService
{
}
```

Describing the new Feature

New plan time team! Let's add a method to the service where we can pass it the number of dinosaurs we want, how much security we want, and... it will take care of the rest! Let's examplify that!

How about `function it_builds_enclosure_with_dinosaurs()`. Inside, let's see, I'd like to be able to create a new `Enclosure` by saying `$enclosure = $this->buildEnclosure()`. We'll pass that the number of security systems we want - 1 - and the number of dinosaurs we want: 2. Then we can do some basic checks, like `$enclosure->shouldBeAnInstanceOf()` to make sure an `Enclosure` is returned.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 15
16      function it_builds_enclosure_with_dinosaurs()
17      {
18          $enclosure = $this->buildEnclosure(1, 2);
19
20          $enclosure->shouldBeAnInstanceOf(Enclosure::class);
↕ // ... line 21
22      }
↕ // ... lines 23 - 24
```

Oh, and very important! I want to make sure the new `Enclosure` has *active* security: `$enclosure->isSecurityActive()`. Wait... but that's not auto-completing - I thought we added that method! Oh, it *does* exist, but it's private. We'll need to fix that in a minute. Anyways, use `$enclosure->isSecurityActive()->shouldReturn(true)`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 15
16      function it_builds_enclosure_with_dinosaurs()
17      {
↕ // ... lines 18 - 20
21          $enclosure->isSecurityActive()->shouldReturn(true);
22      }
↕ // ... lines 23 - 24
```

Simple enough! We're not asserting anything about the dinosaurs yet, but it's a good start. In `Enclosure`, make `isSecurityActive()` public: we've discovered that we *do* need to use this from outside of this class. And *because* it's public... and because I like to keep things organized, let's move it up above the private methods.

Much better. Let's try this!

```
./vendor/bin/phpspec run
```

It fails, generates the new method for us, then it fails again... because that method is empty. I hope these steps are boringly routine at this point.

Implementing the Feature

With the tests red, let's write some code! Open `EnclosureBuilderService` and... fill in the method! I'll break the arguments onto multiple lines - they're gonna be a bit long - and advertise that this returns an `Enclosure`. Change the args to `int $numberOfSecuritySystems` that defaults to 1... though it's entirely up to you if you want a default - and `int $numberOfDinosaurs = 3`. Then, `$enclosure = new Enclosure()` and we'll offload the real work to a private method: `$this->addSecuritySystems($numberOfSecuritySystems, $enclosure)`.

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 7
8  class EnclosureBuilderService
9  {
10     public function buildEnclosure(
11         int $numberOfSecuritySystems = 1,
12         int $numberOfDinosaurs = 3
13     ): Enclosure
14     {
15         $enclosure = new Enclosure();
16
17         $this->addSecuritySystems($numberOfSecuritySystems, $enclosure);
↕ // ... lines 18 - 19
20     }
↕ // ... lines 21 - 30
31 }
```

At the bottom, return `$enclosure` - we'll worry about the dinosaurs in a minute. For the `addSecuritySystems()` method, I'm going to cheat and paste that in: you can find this function on the code block on this page. Make sure to re-type the `y` on `Security` and hit tab to auto-complete that and get the `use` statement on top.

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 7
8  class EnclosureBuilderService
9  {
↕ // ... lines 10 - 21
22     private function addSecuritySystems(int $numberOfSecuritySystems, Enclosure $enclosure)
23     {
24         $securityNames = ['Fence', 'Electric fence', 'Guard tower'];
25         for ($i = 0; $i < $numberOfSecuritySystems; $i++) {
26             $securityName = $securityNames[array_rand($securityNames)];
27             $security = new Security($securityName, true, $enclosure);
28
29             $enclosure->addSecurity($security);
30         }
31     }
32 }
```

It's nothing special: it takes in the `$numberOfSecuritySystems`, does a `for` loop, chooses a random name and sets the "is active" flag to true.

We're not adding any dinosaurs yet... which is fine... because we're not asserting anything about them yet either! We'll worry about that soon. Right now, run phpspec!

```
./vendor/bin/phpspec run
```

Green! This wasn't anything new, but now we have an *awesome* problem. In `EnclosureBuilderService`, we need to create some dinosaurs... but we are *not* going to create them by hand. Nope, we already have this beautiful `DinosaurFactory` that's *great* at growing and hatching Dinosaurs! That means that `EnclosureBuilderService` will need `DinosaurFactory` as a dependency. And *that* means, in order to finish the example, we are going to need to mock `DinosaurFactory`.

Awesome. It's next.

Chapter 22: Dummies

Our new `EnclosureBuilderService` *is* building the security systems and adding them to the `Enclosure`, but it's *not* creating any dinosaurs yet. That's a boring dinosaur park! Fortunately, that should be easy! Heck, we *already* have a class that's really great at doing exactly that! The `DinosaurFactory`.

To Mock or Not?

So, hmm, thinking about the design of `EnclosureBuilderService`, we now know that it will need the `DinosaurFactory` in order to create dinosaurs. And *that* means `EnclosureBuilderService` will need a constructor function so that we can use dependency injection to pass `DinosaurFactory` into it. Ignore phpspec for a second: *this* is pure object-oriented coding: if a service like `EnclosureBuilderService` needs access to another service like `DinosaurFactory`, we will *pass* that service to it, usually via the constructor.

That is the design we'll use for `EnclosureBuilderService`. And of course, that's something that we can describe in our spec class! So far, we haven't said anything about how `EnclosureBuilderService` is instantiated, so it's being created with no arguments. Cool! Now use: `$this->beConstructedWith()` and pass it a `DinosaurFactory` object. But... how should we create the `DinosaurFactory`? Should we create it manually or mock it?

DinosaurFactory Dummy

In this case, mock it. As a rule of thumb, *if* the object you're working with is a *service* object - an object that does *work*, but doesn't hold much data, like `DinosaurFactory`, Doctrine's `EntityManager` or a class that sends emails, mock it. That's because these are usually difficult to instantiate and often have side effects, like talking to the database or sending real emails. Oof, sending emails when you run your tests is *no* fun. We want our unit tests to be isolated from all "real" systems likes that.

But if you're working with a simple model object, it's ok to create it directly. For example, in `DinosaurFactorySpec`... I mean in `EnclosureSpec`, because `Dinosaur` is so simple, we just created it ourselves!

Anyways, we need to mock `DinosaurFactory` and we already know how: add a `DinosaurFactory $dinosaurFactory` argument to the method. Thanks to that, prophecy will create a "dummy" object: one of those many words to describe that this will be an object that looks and smells like `DinosaurFactory`... but isn't *actually* a `DinosaurFactory`. Pass *this* to `beConstructedWith()`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 9
10 class EnclosureBuilderServiceSpec extends ObjectBehavior
11 {
↕ // ... lines 12 - 16
17     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
18     {
19         $this->beConstructedWith($dinosaurFactory);
↕ // ... lines 20 - 24
25     }
26 }
```

Cool! Let's not do *anything* else yet, just run phpspec and see what it thinks:

```
./vendor/bin/phpspec run
```

Woohoo! It sees that the constructor is not found and asks if we want to generate it. Of course we do! Go check it out! Change the argument to `DinosaurFactory $dinosaurFactory` and then... do nothing... yet.

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 8
9 class EnclosureBuilderService
10 {
11     public function __construct(DinosaurFactory $dinosaurFactory)
12     {
13     }
↕ // ... lines 14 - 35
36 }
```

Because... to be all "technical" about it, all we *actually* need to do to get the test to pass is have an `__construct()` method that takes one `DinosaurFactory` argument. Try the tests now:

```
./vendor/bin/phpspec run
```

Yep, green! Well, there is *one* failure, it's from line 13. This is the `it_is_initializable()` example, which is angry because it's *not* passing the required first argument. Ignore that for now and focus on running just *this* example, which is on line 18. Re-run spec with:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:18
```

Yep! This example *does* pass.

Dummy Objects Return Nothing!

Ok: *now* we need to enhance our example to describe the expected behavior for creating Dinosaurs. Basically, because we're passing "2" as the second argument, we would expect our `DinosaurFactory` to be called 2 times and for the final `Enclosure` to have 2 Dinosaurs. But... we haven't coded that yet. `var_dump($enclosure->getDinosaurs())`. This will be an empty array, right? Try it:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:18
```

Ah, we were *mostly* right: it's an instance of the all-important `Subject` object. But if you look inside, the subject property *is* an empty array. Cool! Things are working like we expect... so far.

But here's where things get weird... or cool... or something: *even* if we added the code to `EnclosureBuilderService` to use `DinosaurFactory` to create the 2 Dinosaurs and add them to the `Enclosure`, the test would still fail! What!????

Why? Because, when you create a "dummy" object like `DinosaurFactory` it's not the *real* `DinosaurFactory`. And, by default, *all* of its methods return `null` and do nothing. It's a real... dummy. Right?! So if we *did* write code here to use `DinosaurFactory` to create the dinosaurs... it wouldn't! It would return `null` and either the test would fail or, more likely, some code would blow up because it's expecting a `Dinosaur` object, not null.

Yep, if you *simply* tell prophecy to create a test double, it's referred to as a "dummy" and... it does nothing. But, there *are* two things that we *can* do to make it more awesome. Let's talk about the first one next: controlling the return value when a method is called.

Chapter 23: Stubs

There are basically two big things you can do with a test double. First, you can add behavior: you can tell it *exactly* what value to return when a certain method is called, instead of returning null. Second, you add expectations. For example, you can assert that a certain method on `DinosaurFactory` should be called a certain number of times and even with some specific arguments.

Controlling Method Return Value

Right now, we need to control the return value of the `growVelociraptor()` method. Instead of returning `null`, which will probably explode when `EnclosureBuilderService` tries to add `null` to an `Enclosure`, we need it to return a `Dinosaur` object.

Check this out: create a `$dino1` variable set to `new Dinosaur()` with `Stegosaurus` and `false`. And let's set its length to, how about, 6. *Here* is the key part: we want our `DinosaurFactory` dummy object to return *this* `Dinosaur` when somebody calls `growVelociraptor()`. I know... that's kind of confusing because this is *not* a velociraptor... but that proves my point! We can *completely* control how this behaves.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 10
11 class EnclosureBuilderServiceSpec extends ObjectBehavior
12 {
↕ // ... lines 13 - 17
18     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
19     {
↕ // ... line 20
21         $dino1 = new Dinosaur('Stegosaurus', false);
22         $dino1->setLength(6);
↕ // ... lines 23 - 31
32     }
33 }
```

Do it with `$dinosaurFactory->growVelociraptor()`. So, we *pretend* like `$dinosaurFactory` is a real object and, just like normal with phpspec, we call methods on that object and pass in real arguments. Let's say that, whenever we use the `EnclosureBuildersService`, it will *always* grow a velociraptor of length 5. Then, to control the return value, say `->willReturn($dino1)`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 17
18     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
19     {
↕ // ... lines 20 - 22
23         $dinosaurFactory->growVelociraptor(5)->willReturn(
24             $dino1
25         );
↕ // ... lines 26 - 31
32     }
↕ // ... lines 33 - 34
```

That's it! Actually, we just did *both* things that I said you could do with a test double. First, by saying `$dinosaurFactory->growVelociraptor(5)`, we've added an *assertion* that if this method is called, it *must* be passed the argument 5. If any other value is passed, the test will fail. More on that later. And second, we've controlled the return value with `->willReturn()`.

There are a few other "will" methods you can use to control the return value, and the most useful by far is just to say `->will()` and pass that a callback function. That's super useful if a method is called multiple times and you need to return different values each time. More about *that* later too.

So... let's run the test!

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:18
```

Oh! That doesn't work! Because... yea - we're now on line 19. Try it again:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

The test still passes... but we haven't *actually* added any new assertions yet.

Using the Object in EnclosureBuilderService

Let's get to work! Open `EnclosureBuilderService`. Here I'll hit Alt + Enter on `$dinosaurFactory` and select "Initialize Fields" to create and set that property. Down below, let's call a new method called `addDinosaurs()` and pass it the `$numberOfDinosaurs` argument. To add that new method, I'll put my cursor on `addDinosaur()`, hit Alt + Enter and "Add Method".

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 8
9  class EnclosureBuilderService
10 {
11     private $dinosaurFactory;
12
13     public function __construct(DinosaurFactory $dinosaurFactory)
14     {
15         $this->dinosaurFactory = $dinosaurFactory;
16     }
↕ // ... line 17
18     public function buildEnclosure(
↕ // ... lines 19 - 20
21     ): Enclosure
22     {
↕ // ... lines 23 - 25
26         $this->addDinosaurs($numberOfDinosaurs, $enclosure);
↕ // ... lines 27 - 28
29     }
↕ // ... lines 30 - 40
41     private function addDinosaurs(int $numberOfDinosaurs, Enclosure $enclosure)
42     {
↕ // ... lines 43 - 47
48     }
49 }
```

Next, copy the inside of `addSecuritySystems()`, paste it here, then clear out the inside of the loop. Change the variable to `$numberOfDinosaurs` and, very nicely, we can say `$enclosure->addDinosaur()` and pass that `$this->dinosaurFactory->growVelociraptor()`. And, remember: in the example we expected this to be called *always* with a length of `5`.

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 40
41     private function addDinosaurs(int $numberOfDinosaurs, Enclosure $enclosure)
42     {
43         for ($i = 0; $i < $numberOfDinosaurs; $i++) {
44             $enclosure->addDinosaur(
45                 $this->dinosaurFactory->growVelociraptor(5)
46             );
47         }
48     }
↕ // ... lines 49 - 50
```

Perfect! Move over and run the test again:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

It still passes. *And*, check out the wrapped object we're dumping: there are now *two* dinosaurs inside! And... interesting! They're actually the *exact* same `Dinosaur`. That makes sense: each time the `growVelociraptor()` method is called, our test double returns that *same*, one `Dinosaur` object.

This is cool because we can add a great assertion down here: `$enclosure->getDinosaurs()[0]` - to get the first `Dinosaur` - `->shouldBe($dino1)`. And, it's a little odd, but we can even check that the *second* `Dinosaur` is *also* this exact `$dino1` `Dinosaur`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 17
18     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
19     {
↕ // ... lines 20 - 30
31         $enclosure->getDinosaurs()[0]->shouldBe($dino1);
32         $enclosure->getDinosaurs()[1]->shouldBe($dino1);
33     }
↕ // ... lines 34 - 35
```

Try the test again:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Still green! This is the first, great superpower of these test doubles, or dummy objects. By adding behavior to them, we can help control *exactly* what happens inside the class we're testing. And, it often allows us to have very specific assertions.

Returning a Different Object Each Time

To make this a bit more realistic, copy `$dino1` and make a new `$dino2` - how about a `Baby Stegosaurus` with length 2. Adorable. I want to change the `$dinosaurFactory` test double so that it returns `$dino1` the *first* time `growVelociraptor()` is called and `$dino2` the second time it's called... which is a bit more how things would work in the real world. How can we do this? By passing a second argument to `willReturn()`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 17
18     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
19     {
↕ // ... lines 20 - 22
23         $dino2 = new Dinosaur('Baby Stegosaurus', false);
24         $dino2->setLength(2);
25         $dinosaurFactory->growVelociraptor(5)->willReturn(
↕ // ... line 26
27             $dino2
28         );
↕ // ... lines 29 - 34
35         $enclosure->getDinosaurs()[1]->shouldBe($dino2);
36     }
↕ // ... lines 37 - 38
```

That's it! Down below, the second object will now be `$dino2`. Try the tests one last time.

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Green! Geez - we didn't manage to break *anything* in this video - we gotta try harder! By the way, when you control the return value of a test double, it's then called a "Stub"... which is probably not that important to know, except for impressing other programmers at a party.

Next: let's talk a little bit more about this "5" argument. As it turns out, there are a *lot* of interesting things you can do with this argument. Like, what if `EnclosureBuilderService` always calls this method and passes a random number? What should we put here? Or, what if it's called multiple times with different arguments each time? Let's jump on that!

Chapter 24: Promises (control return values) & Arguments

In our spec, we said that, when we call `buildEnclosure()`, we expect *it* to call `growVelociraptor()` on `DinosaurFactory` and pass it the exact argument `5`. There was no super cool or secret reason for that: it's just how we decided to make this method work: all dinosaurs would have this same length. And when it calls `growVelociraptor()` with an argument of 5, we said that it should return `$dino1` the first time it's called and `$dino2` the second time.

Calling a Stubbed Method with a Different Argument

But... what if it weren't that simple? Go into the service. Making *every* dinosaur the same size isn't very realistic. No, let's make it more interesting - let's make the length `5 + $i`. So 5, then 6, 7 and so on.

src/Service/EnclosureBuilderService.php

```
// ... lines 1 - 40
    private function addDinosaurs(int $numberOfDinosaurs, Enclosure $enclosure)
    {
        for ($i = 0; $i < $numberOfDinosaurs; $i++) {
            $enclosure->addDinosaur(
                $this->dinosaurFactory->growVelociraptor(5 + $i)
            );
        }
    }
// ... lines 49 - 50
```

What will phpspec think of this? Our example *still* says that we expect this method to *always* be called with the argument 5. Well... let's find out!

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

It fails! And check this out! It says:

> *"Unexpected method call on `DinosaurFactory`, `growVelociraptor(6)`. Expected was `growVelociraptor(5)`"*

The *way* this fails is the *really* important part. When you call `willReturn()`, this is called a method *promise*. You're making a *promise* that the `growVelociraptor()` method will return this `$dino1` value and then this `$dino2` value. As soon as you apply even *one* promise to *one* method on a dummy object, you must apply a promise to *every* single call.

This can be a little tricky to understand at first. When we say `$dinosaurFactory->growVelociprator(5)->willReturn()`, we're *really* saying:

> *"Hey phpspec! When `growVelociraptor()` is called and passed 5 as an argument, you should return `$dino1` and `$dino2`. If any other value is passed, this promise doesn't apply."*

So, when `growVelociraptor()` is called with 6 as an argument, it looks at this promise and determines it doesn't apply. *But*, once you define even *one* promise, you *must* define a promise... basically, you must tell phpspec what to do for *every* method call and *every* possible argument.

In other words, the *simplest* way to fix this would be to add a promise for *every* argument that we'll pass - like `$dinosaurFactory->growVelociraptor(6)->willReturn()`, then 7, 8, 9 - however many you need for that example. Whenever `growVelociraptor()` is called, prophecy goes down *all* of the promises for that method and finds the *one* that fits best. If none are found... but at least one is specified... error!

Argument::any()

Of course... creating a promise for *every* possible argument is... a bit nuts. So what's the *real* fix? Remove the specific argument 5 and replace it with a special `Argument` class from prophecy and pass it `any()`. Try phpspec again:

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 18
19     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
20     {
↕ // ... lines 21 - 25
26         $dinosaurFactory->growVelociraptor(Argument::any())->willReturn(
↕ // ... lines 27 - 28
29         );
↕ // ... lines 30 - 36
37     }
↕ // ... lines 38 - 39
```

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

We're back! *This* says:

> *"Yo phpspec! It's me again, Ryan. So, when `growVelociraptor()` is called with any argument, here's what you should return."*

Thanks to this, *every* time we call `growVelociraptor()` - *regardless* of the arguments passed to it - this promise will be matched.

Other ways to Specify Arguments

Oh, and if we were passing `growVelociraptor()` *two* arguments, then we *would* need to also pass two arguments here. If you don't care, just use `Argument::any()`. Or, if you had a third argument and wanted this promise to be used no matter what values were passed, you can use `cetera()`.

But sometimes, I want to be a *bit* more specific. For example, we don't *really* want `growVelociraptor()` to be called with *any* argument: it should be an integer at least! If you *really* want to make sure that's happening correctly, you can use `Argument::type()` and pass `integer`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 25
26         $dinosaurFactory->growVelociraptor(Argument::type('integer'))->willReturn(
↕ // ... lines 27 - 39
```

This should work for us... so let's try it!

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

All green! But *now* change this to `string`. Try it again:

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 25
26         $dinosaurFactory->growVelociraptor(Argument::type('string'))->willReturn(
↕ // ... lines 27 - 39
```

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Yes! Unexpected method call on `growVelociraptor(5)`. When the integer 5 is passed, phpspec says:

> *"Hey, somebody is calling `growVelociraptor()` with an argument that doesn't match any of the promises for this method. That's probably not expected, so I'm gonna go ahead and explode. Cheers!"*

The most flexible `Argument` is called `that()`. I won't show an example, but `Argument::that()` allows you to pass a callback function. *It* passes your callback the argument and then you can do whatever logic you need to determine if this is something you're expecting or not.

Let's change this back to `type('integer')`. And *here* is where things get *truly* interesting. We know that `growVelociraptor()` is going to be called two times. And so, to take *super* crazy control of things, we can create two *separate* method promises to handle each one. If you don't fully understand how this `Argument` stuff works yet, you will next.

Chapter 25: Advanced Argument Matching

The `growVelociraptor()` method is called two times. And thanks to the two arguments to `willReturn()`, `$dino1` will be returned first, then `$dino2`. But we can control this *much* more than we are now.

Check it out: copy the beginning of the promise and paste it below. Replace the argument with 5. Then say `->willReturn($dino2)`. For the first promise, remove `$dino2`: this will now return `$dino1` *every* time its matched.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 18
19     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
20     {
↕ // ... lines 21 - 25
26         $dinosaurFactory->growVelociraptor(Argument::type('integer'))->willReturn(
27             $dino1
28         );
29         $dinosaurFactory->growVelociraptor(5)->willReturn(
30             $dino2
31         );
↕ // ... lines 32 - 38
39     }
↕ // ... lines 40 - 41
```

So... what do you think will happen now? Down here, I'm saying that when `growVelociraptor()` is called with exactly the number `5`, return `$dino2`. But up here I'm saying: if `growVelociraptor()` is called with *any* `integer`, return `$dino1`. So... when `growVelociraptor()` is called with the number 5, what's going to be returned? `$dino1`? `$dino2`? A mystery `$dino3`!? Does the order of these promises matter? Or something else? Is there a 9th planet in the solar system that we haven't discovered yet?

I don't know! Let's find out! Or at least, answer a *few* of these questions. Try phpspec:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

It fails: expected `Dinosaur` but got `Dinosaur`. Wow... umm... that's not very helpful. Re-run with `-v`:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19 -v
```

Ah! Now we can see that, on line 38, so when we're checking the first dinosaur, we expect it to be our big Stegosaurus, but it was actually our `Baby Stegosaurus`.

Arguments Matching is by "Most Specific"

This is really cool! Each time we call `growVelociraptor()`, it looks at *all* of these method promises and finds the most *specific* one. When the argument is 5, it matches *both* promises: it *is* 5, but it's also an integer. But because the second is more specific, *that* promise is used and it returns `$dino2`.

So, the *first* time we go through the loop, 5 is the argument and we return `$dino2`. The second time, the argument is 6 and *only* matches the first promise. So, `$dino1` is returned.

This means that `$dino2` is now the first item and `$dino1` is the second. How *cool* is that?! That's full, beautiful control.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 18
19     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
20     {
↕ // ... lines 21 - 36
37         $enclosure->getDinosaurs()[0]->shouldBe($dino2);
38         $enclosure->getDinosaurs()[1]->shouldBe($dino1);
39     }
↕ // ... lines 40 - 41
```

Let's take off the `-v` option and run the tests:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Got it! To prove how this works, we can even re-order the two promises. Yep, makes no difference.

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

So, whenever a method is called on a stub, prophecy will look at all the promises for that method and find the best one. And if *none* are found... but at least one exists, error!

Let's change this back to just one promise: we don't really need to be this specific. Delete the more specific call and make the other one return `$dino1` and then `$dino2` just like before. Also update the asserts to go back to the original way. Double-check that phpspec is happy:

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 18
19     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
20     {
↕ // ... lines 21 - 25
26         $dinosaurFactory->growVelociraptor(Argument::type('integer'))->willReturn(
27             $dino1,
28             $dino2
29         );
↕ // ... lines 30 - 34
35         $enclosure->getDinosaurs()[0]->shouldBe($dino1);
36         $enclosure->getDinosaurs()[1]->shouldBe($dino2);
37     }
↕ // ... lines 38 - 39
```

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Woohoo! Arguments are a *great* way to very specifically control what values are returned based on the *input*. *And* to make sure your methods are being called with the arguments you expect.

Next: instead of controlling the return value of methods, let's talk about adding *expectations* to our test doubles - like making sure that a method was called an *exact* number of times. Yep, it's mocks and spies time!

Chapter 26: Mocks & Spies - shouldBeCalledTimes()

Hey! We are *experts* when it comes to controlling the return value of any method on a test double. Here, we told it that when `growVelociraptor()` is called with any integer argument, return `$dino1` the first time and `$dino2` the second time.

By doing this, we *also* added some *expectations*. Because, as soon as you control the return value of *one* method call, then you must control the behavior of *all* method calls. And so if a different method is called or different arguments are passed to this method, the tests will fail.

But what this does *not* guarantee is that `growVelociraptor()` *was* actually called... or how many times it was called. Nope, we're saying, *if* it's called, it must be called with this argument... and here's what to return. But technically, if it were called zero times, this part would not fail!

And that's what I want to talk about now: sometimes it *is* super important to make sure a method *was* called... or was called exactly *three* times... or was *not* called! For example, what if we wanted to be *absolutely* sure that `growVelociraptor()` was called exactly two times? We can do that, and it starts the same way: `$dinosaurFactory->growVelociraptor(Argument::type('integer'))`. But then, instead of `willReturn()`, use `shouldBeCalledTimes(2)`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 18
19     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
20     {
↕ // ... lines 21 - 29
30         $dinosaurFactory->growVelociraptor(Argument::type('integer'))
31             ->shouldBeCalledTimes(2);
↕ // ... lines 32 - 38
39     }
↕ // ... lines 40 - 41
```

This feels familiar... because it's exactly like what we've been doing! It looks like a matcher! It's identical to `$this->shoudHaveType()`, for example.

Now that we're *asserting* that this method should be called twice, run phpspec:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Nice! It *passes*! What does it look like to fail? Change this to 3... and try it again:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Perfect: Expected exactly 3 calls that match `growVelociraptor()` with type integer, but 2 were made.

Re-Using the "Promise" for Stubbing & Mocking

Change this back to 2. But notice: there's some duplication here: we're repeating the `$dinosaurFactory->growVelociraptor(Argument::type('integer'))` part when we need to control the return value *and* when we want to assert how many times it was called. We can totally remove that. Chain the `->shouldBeCalledTimes()` onto the end of the first call.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 18
19     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
20     {
↕ // ... lines 21 - 25
26         $dinosaurFactory->growVelociraptor(Argument::type('integer'))->willReturn(
27             $dino1,
28             $dino2
29         )->shouldBeCalledTimes(2);
↕ // ... lines 30 - 36
37     }
↕ // ... lines 38 - 39
```

Try it!

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Nice! It's sort of a low-level detail, but when we call `$dinosaurFactory->growVelociraptor()`, that returns what's called a "method prophecy" object... and we can then add "promises" to it - that's the `willReturn()` stuff - or *predictions* - that's the `shouldBeCalledTimes()` stuff. What's interesting is that, if you call a method on a stub two times and pass it the *same* exact `Argument` stuff, prophecy will return the *same* `MethodProphecy` object. In other words, if you called `willReturn()` multiple times, the second would override the first.

If that doesn't totally make sense - forget about it - it's just some low-level coolness I wanted to mention while we were here.

Predicting After (Spies)

Anyways, when you want to add a "prediction"... basically, an expectation that a method should be called an exact number of times, you can do it in *two* different ways... and it's just a matter of style. First, you can do it like we just did: call `->shouldBeCalledTimes()` and *then* execute the code.

Or, you can put the assertion stuff *after* you run your code. Check this out: remove the `->shouldBeCalledTimes()` line. Then, anywhere after we call `buildEnclosure()`, start with `$dinosaurFactory->growVelociraptor(Argument::any())` and then `->shouldHaveBeenCalledTimes(2)`.

spec/Service/EnclosureBuilderServiceSpec.php

```
// ... lines 1 - 18
    function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory)
    {
// ... lines 21 - 25
        $dinosaurFactory->growVelociraptor(Argument::type('integer'))->willReturn(
            $dino1,
            $dino2
        );
// ... lines 30 - 37
        $dinosaurFactory->growVelociraptor(Argument::any())
            ->shouldHaveBeenCalledTimes(2);
    }
// ... lines 41 - 42
```

This does the exact same thing... it's just a different style. Oh, and I used `Argument::any()` down here instead of `type()`, but not for any special reason: I'm just showing how we can make sure that this method is called exactly 2 times, regardless of the arguments.

Let's try this!

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

We're green! Next: let's take a super-quick tour into the phpspec documentation where we'll see test doubles, dummies, mocks and spies... hiding within its text.

Chapter 27: Test Doubles, Dummies, Mocks & Spies

Now that we've seen *all* the things we can do with test doubles... we need to talk a *bit* more about some words and language that you'll see with prophecy... otherwise... it all looks a bit crazy.

Google for "php prophecy"... because, remember, this whole test double system in phpspec actually comes from prophecy.

In their documentation, they talk about *four* different types of objects! Dummy objects, stub objects, mock objects and spy objects. Woh! For me... this was... confusing! So let's walk through and see what's actually going on - it's really nothing new for us. These four different words all different ways to describe test doubles... the "things" that we just finished learning *all* about.

Dummies

First, look at dummies. A dummy object is both what I look like if you ask me to remember where I left the car keys... *and* also the object you get if you add an argument with a type-hint in phpspec to get a test double... then do absolutely nothing with it. So if we get a test double and add *no* behavior and make *no* assertions on its methods, it's called a "dummy object".

Oh, and inside of their documentation, you'll see things like `$prophecy->reveal()`. That's a detail that we don't need to worry about because phpspec takes care of that for us. Score!

Stubs

As *soon* as you start controlling even *one* return value of even *one* method... boom! This object is suddenly known as a stub. From the docs: "a stub is an object double" - *all* of these things are known as test doubles, or object doubles - that when put in a specific environment, behaves in a specific way. That's a fancy way of saying: as soon as we add one of these `willReturn()` things, it becomes a stub.

And actually, most of the documentation is spent talking about stubs and the different ways to control exactly how it behaves, including the Argument wildcarding that we saw earlier.

Mocks

If you keep reading down, the next thing you'll find are "mocks". An object becomes a mock when you call `shouldBeCalled()`. So, if you want to add an *assertion* that a method is called a certain number of times and you want to put that assertion *before* the actual code - using `shouldBeCalledTimes()` or `shouldBeCalled()` - congratulations! Your object is now known as a mock.

Spies

And *finally*, at the bottom, we have spies. A spy is the *exact* same thing as a mock, except it's when you add the expectation *after* the code - like with `shouldHaveBeenCalledTimes()`.

Putting it All Together

So there's a lot of language here... but these are all different words to describe the different things you can do with a test double. Doing nothing? It's a dummy. Controlling the return value? It's a stub. Adding an assertion to make sure a method was called? It's a mock or a spy, depending on your style. And yes, you *can* be a stub *and* a mock or spy at the same time - that's exactly what we're doing in our example.

I find these terms fun, but a little confusing and I don't think about them when I'm actually coding: I just think about what I need to get done. But now that you're familiar with them, reading the docs should be a breeze. And also, these words are used *everywhere* in the testing world - not just in phpspec - so it's kinda cool to know them. You're now part of the cool-kids testing club.

Next, let's do some more mocking... cause it's awesome! And we'll say hello to a helper method that we can use inside our spec class to run some code before *every* example.

Chapter 28: Let: The Setup Function

Now that this example is executing really well...

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

let's go back and run *all* of the examples in this spec class.

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php
```

Ah, oh yea! `it_is_initializable()` is still failing: too few arguments to the constructor: zero passed, expected one. That makes sense: we added `$this->beConstructedWith()` to our `it_builds_enclosures_with_dinosaurs()` example, but not to *this* example.

The easiest solution is just to... ya know... duplicate it! But... it does kind of make me think: it would be nice if we could call a method *before* each example was executed - a method that could, sort of, set some things up. After all, we're going to need to call `$this->beConstructedWith()` in *every* single example.

Hello let()

Unfortunately... this is not possible... and the tutorial is now over. Kidding! This is totally possible: by creating a function called `let()`. Yep! `let()` will be called once *before* each example function is executed. So, `let()` then `it_is_initializable()`, then `let()` again and `it_builds_enclosures_with_dinosaurs()`.

Inside `let()`, we can do the *exact* same things as our normal methods... meaning, if we need a test double, we can add a `DinosaurFactory $dinosaurFactory` argument. And *this* allows us to move the `beConstructedWith()` call up to `let()`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 11
12 class EnclosureBuilderServiceSpec extends ObjectBehavior
13 {
14     function let(DinosaurFactory $dinosaurFactory)
15     {
16         $this->beConstructedWith($dinosaurFactory);
17     }
↕ // ... lines 18 - 44
45 }
```

Ok, let's look at this: `let()` will be called first, and will be passed the `$dinosaurFactory` test double. But then, down in the example, we need to make sure that we get that *exact* same `DinosaurFactory` test double object

because *that* is the object we need to add behavior to.

And that is *exactly* how this will work. But, it's not just because these are both `DinosaurFactory` objects. Nope. phpspec matches by the argument *name*: because the argument is called `$dinosaurFactory` in `let()` *and* because it has the same name below, phpspec knows to pass the *same* object... instead of creating a brand-new test double.

So... unless we've mucked something up, this should make everything happy! Try it:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php:19
```

Woohoo!

Using Mocks/Spies to Guarantee Saving to the Database

Let's do *one* last thing before you all go off and take over the world with your new phpspec knowledge. One of the things I want my `EnclosureBuilderService` to do is to *save* the new `EnclosureBuilder` to the database after it finishes building it. Now, obviously, we don't have a database in this application... but who cares? We can fake it!

Open up the `tutorial/` directory: you'll have it if you downloaded the course code. In the `Service/` directory there is an `EntityManagerInterface.php` file. Copy that and put it into your `Service/` directory.

src/Service/EntityManagerInterface.php

```
// ... lines 1 - 4
interface EntityManagerInterface
{
    public function persist($object);

    public function flush();
}
```

If you're a Doctrine user, this will look familiar. But, no, I'm not recommending that you actually create or move the `EntityManagerInterface` into your app like this. We're adding this interface as a convenient way to "pretend" like Doctrine exists in our app. This interface looks *just* like the one from Doctrine, with the same `persist()` and `flush()` methods.

Guaranteeing the Object is Saved

Back in the example, I want to describe that `persist()` and `flush()` *should* be called on the `EntityManagerInterface` when we call `buildEnclosure()`: I want to guarantee that I didn't forget to save this to the database. And *that* means that our `EnclosureBuilderService` will have a second dependency. In `let()`, add a second argument: `EntityManagerInterface $entityManager`. Pass that as the second argument to `beConstructedWith()`.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 12
13 class EnclosureBuilderServiceSpec extends ObjectBehavior
14 {
15     function let(DinosaurFactory $dinosaurFactory, EntityManagerInterface $entityManager)
16     {
17         $this->beConstructedWith($dinosaurFactory, $entityManager);
18     }
↕ // ... lines 19 - 49
50 }
```

Then, down in the actual example, we want to assert that `persist()` and `flush()` were called on it. Get that same test double here by *also* saying `EntityManagerInterface $entityManager`.

At this point, we can use the mock or spy functionality... it makes no difference. I'll do it as a spy. Start with `$entityManager->persist()`. This should be passed an `Enclosure` object. So let's say `Argument::type()` with `Enclosure::class`. Then, `->shouldHaveBeenCalled()`.

Repeat this with `flush()`, except that `flush()` doesn't take any arguments.

spec/Service/EnclosureBuilderServiceSpec.php

```
↕ // ... lines 1 - 24
25     function it_builds_enclosure_with_dinosaurs(DinosaurFactory $dinosaurFactory,
   EntityManagerInterface $entityManager)
26     {
↕ // ... lines 27 - 45
46         $entityManager->persist(Argument::type(Enclosure::class))
47             ->shouldHaveBeenCalled();
48         $entityManager->flush()->shouldHaveBeenCalled();
49     }
↕ // ... lines 50 - 51
```

This time, we're not giving the test double *any* behavior: it's a *pure* spy. Try it:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php
```

Woo! It fails:

> *"no calls have been made that match `persist()` with type Enclosure, but expected at least one."*

Open `EnclosureBuilderService` and let's code that up. Start by adding the `EntityManagerInterface $entityManager` argument. I'll hit Alt + Enter to create that property and set it.

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 8
9  class EnclosureBuilderService
10 {
↕ // ... lines 11 - 12
13     private $entityManager;
↕ // ... line 14
15     public function __construct(DinosaurFactory $dinosaurFactory, EntityManagerInterface
   $entityManager)
16     {
↕ // ... line 17
18         $this->entityManager = $entityManager;
19     }
↕ // ... lines 20 - 54
55 }
```

Finally, at the bottom of the method, `$this->entityManager->persist($enclosure)` and `$this->entityManager->flush()`.

src/Service/EnclosureBuilderService.php

```
↕ // ... lines 1 - 20
21     public function buildEnclosure(
↕ // ... lines 22 - 23
24     ): Enclosure
25     {
↕ // ... lines 26 - 30
31         $this->entityManager->persist($enclosure);
32         $this->entityManager->flush();
↕ // ... lines 33 - 34
35     }
↕ // ... lines 36 - 56
```

This *looks* great. But let's run phpspec *one* last time to be sure:

```
./vendor/bin/phpspec run spec/Service/EnclosureBuilderServiceSpec.php
```

It passes! Try the entire test suite:

```
./vendor/bin/phpspec run
```

Hey! We didn't break anything!

Friends! That's it. phpspec is wonderful tool to help you write unit tests... but *really* focus on the design of your class. Yes, you *do* need to get used to a lot of the magic it does. But once you embrace the magic, the experience is *wonderful*... and the code generation isn't too bad either.

One word of warning that I always like to give people is this: just because you have a wonderful testing tool like phpspec, it doesn't mean you need to test every single thing. For example, in `DinosaurSpec`, we're doing a lot of testing on the getter and setter methods. You *can* do that... but I think it's overkill. Think of testing less as an "all-or-

nothing" sort of thing, and more of a *priority* system: make it a high priority to test, or *describe* the classes that have a lot of complexity or that scare you.

Ok, get out there, describe some *great* classes, and we'll seeya next time!