CHAPTER 11

OBJECT ORIENTED PHP

Now that you're up to speed with modern PHP syntax, it's time to look deeper into what kind of code we write with it. Throughout the next chapters, we'll zoom out to see the bigger picture. We'll start with a heavily debated topic: object-oriented programming.

Alan Kay, the inventor of the term "object-oriented programming", told a story once during a talk more than 20 years ago. You can build a dog house using only a hammer, nails, planks, and just a little bit of skill. I figure even I would be able to build it given enough time. Once you've built it you've earned the skills and know-how, and could apply it to other projects. Next, you want to build a cathedral, using the same approach with your hammer, nails, and planks. It's 100 times larger, but you've done this before — right? It'll only take a little longer.

While the scale went up by a factor of 100, its mass went up by a factor of 1,000,000 and its strength only by 10,000. Inevitably, the building will collapse. Some people plaster over the rubble, make it into a pyramid and say it was the plan all along; but you and I know what really went on.

You can watch Alan's talk here:
https://www.youtube.com/watch?v=oKg1hTOQXoY

Alan used this metaphor to explain a critical problem he saw with "modern OOP" 20 years ago. I think it still holds today: we've taken the solution to a problem — OO code — we've scaled it by a factor of 100, and expected it to work the same way. Even today, we don't think enough about architecture — which is rather crucial if you're building a cathedral — we use the OO solutions we learned without any extra thought. Most of us learned OO in isolation with small examples and rarely at scale. In most real-life projects, you cannot simply apply the patterns you've learned and expect everything to fall into place the same way it did with Animals, Cats, and Dogs.

This reckless scaling of OO code caused many people to voice their disapproval of it in recent years. I think that OOP is as good a tool as any other — functional programming being the modern-day popular contestant — *if* used correctly.

My takeaway from Alan's talk 20 years ago is that each object is a little program on its own, with its own internal state. Objects send messages between each other — packages of immutable data — which other objects can interpret and react to. You can't write all code this way, and that's acceptable — it's fine to not blindly follow these rules. Still, I have experienced the positive impact of this mindset first hand. Thinking of objects as little standalone programs, I started writing parts of my code in a different style. I hope that, now that we're going to look at object-oriented PHP, you'll keep Alan's ideas in mind. They taught me to critically look at what I took for granted as "proper OO", and learned there's more to it than you might think.

ALTERNATIVES TO OOP

I don't want to promote any tunnel vision in this book. I'm aware that there are other approaches to programming than only OOP. Functional programming, for example, has seen a tremendous increase in popularity in recent years. While I reckon that FP has its merits, PHP isn't optimised to program in a functional style. On the other hand, while OOP is the best match for PHP, all programmers

can learn valuable lessons by learning about other programming styles, like a functional one.

I'd recommend you read a book called "Thinking Functionally in PHP" by Larry Garfield. In the book, Larry clearly shows why PHP isn't the perfect language to write functional programs with, but he also explains FP's mindset, visualised in PHP. And even though you wouldn't write functional PHP production code, there's lots of knowledge we can apply to OOP as well.

THE PITFALL OF INHERITANCE

I found it difficult to believe at first, but classes and inheritance have nothing to do with OOP the way Alan envisioned it. That doesn't mean they are bad things per se, but it is good to think about their purpose and how we can use (as well as abuse) them. Alan's vision only described objects — it didn't explain how those objects were created. Classes were added later as a convenient way to manage objects, but they are only an implementation detail, not OOP's core idea. With classes came inheritance, another useful tool when used correctly. That hasn't always been the case, though. Even when you might think it's one of the pillars of object-oriented design, they are misused very often, just like the doghouse Alan tried to scale up to a cathedral.

One of OOP's acclaimed strengths is that it models our code in ways humans think about the world. In reality, though, we rarely think in terms of abstractions and inheritance. Instead of using inheritance in places where it actually makes sense, we've been abusing it to share code, and configure objects in an obscure way. I'm going to show you a great example that illustrates this problem, though I want to say upfront that it isn't my own: it's Sandi Metz's, a great teacher on the subject of OOP. Let's take a look.

Sandi's talk: https://www.youtube.com/watch?v=OMPfEXIlTVE

There's a children's nursery rhyme called "The House That Jack Built" (it's also a horror movie, but that's unrelated). It starts like this:

```
This is the house that Jack built.
```

Every iteration, there's a sentence added to it:

```
This is the malt that lay in
        the house that Jack built.
```

And next

```
This is the rat that ate
        the malt that lay in
        the house that Jack built.
```

Get it? This is the final poem:

```
This is the horse and the hound and the horn that belonged to
        the farmer sowing his corn that kept
        the rooster that crowed in the morn that woke
        the priest all shaven and shorn that married
        the man all tattered and torn that kissed
        the maiden all forlorn that milked
        the cow with the crumpled horn that tossed
        the dog that worried
        the cat that killed
        the rat that ate
        the malt that lay in
        the house that Jack built.
```

Let's code this in PHP: a program that you can ask a given iteration, and it will produce the poem up until that point. Let's do it in an OO way. We start by adding all parts into a data array within a class; let's call that class `PoemGenerator` — sounds very OO, right? Good.

```php
class PoemGenerator
{
    private static array $data = [
        'the horse and the hound and the horn that belonged to',
        'the farmer sowing his corn that kept',
        'the rooster that crowed in the morn that woke',
        'the priest all shaven and shorn that married',
        'the man all tattered and torn that kissed',
        'the maiden all forlorn that milked',
        'the cow with the crumpled horn that tossed',
        'the dog that worried',
```

```
            'the cat that killed',
            'the rat that ate',
            'the malt that lay in',
            'the house that Jack built',
        ];
    }
```

Now let's add two methods generate and phrase. generate will return the end result, and phrase is an internal function that glues the parts together.

```
class PoemGenerator
{
    // …

    public function generate(int $number): string
    {
        return "This is {$this->phrase($number)}.";
    }

    protected function phrase(int $number): string
    {
        $parts = array_slice(self::$data, -$number, $number);

        return implode("\n        ", $parts);
    }
}
```

It seems like our solution works: we can use `phrase` to take x-amount of items from the end of our data array and implode those into one phrase. Next, we use `generate`

to wrap the final result with `This is` and `.`. By the way, I implode on that spaced delimiter just to format the output a little nicer.

```
$generator = new PoemGenerator();

$generator->generate(4);

// This is the cat that killed
//         the rat that ate
//         the malt that lay in
//         the house that Jack built.
```

Exactly what we'd expect the result to be.

Then comes along... a new feature request. Let's build a random poem generator: it will randomise the order of the phrases. How do we solve this in a clean way without copying and duplicating code? Inheritance to the rescue — right? First, let's do a little

refactor: let's add a protected `data` method so that we have a little more flexibility in what it actually returns:

```
class PoemGenerator
{
    protected function phrase(int $number): string
    {
        $parts = array_slice($this->data(), -$number, $number);

        return implode("\n        ", $parts);
    }

    protected function data(): array
    {
        return [
            'the horse and the hound and the horn that belonged to',
            // …
            'the house that Jack built',
        ];
    }
}
```

Next we build our `RandomPoemGenerator`:

```
class RandomPoemGenerator extends PoemGenerator
{
    protected function data(): array
    {
        $data = parent::data();

        shuffle($data);

        return $data;
    }
}
```

How great is inheritance! We only needed to override a small part of our code, and everything works just as expected!

```
$generator = new RandomPoemGenerator();

$generator->generate(4);

// This is the priest all shaven and shorn that married
//         the cow with the crumpled horn that tossed
//         the man all tattered and torn that kissed
//         the rooster that crowed in the morn that woke.
```

Awesome!

Once again... a new feature request: an echo generator: it repeats every line a second time. So you'd get this:

```
This is the malt that lay in the malt that lay in
        the house that Jack built the house that Jack built.
```

We can solve this; inheritance — right?

Let's again do a small refactor in `PoemGenerator`, just to make sure our code stays clean. We can extract the array slicing functionality in phrase to its own method, which seems like a better separation of concerns.

```
class PoemGenerator
{
    // …

    protected function phrase(int $number): string
    {
        $parts = $this->parts($number);

        return implode("\n        ", $parts);
    }

    protected function parts(int $number): array
    {
        return array_slice($this->data(), -$number, $number);
    }
}
```

Having refactored this, implementing `EchoPoemGenerator` is again very easy:

```
class EchoPoemGenerator extends PoemGenerator
{
    protected function parts(int $number): array
    {
        return array_reduce(
            parent::parts($number),
            fn (array $output, string $line) ⇒
                [...$output, "{$line} {$line}"],
            []
        );
    }
}
```

Can we take a moment to appreciate the power of inheritance? We've created two different implementations of our original `PoemGenerator`, and have only overridden the parts that differ from it in `RandomPoemGenerator` and `EchoPoemGenerator`. We've even used SOLID principles (or so we think) to ensure that our code is decoupled so that it's easy to override specific parts. This is what great OOP is about — right?

One more time... another feature request: please make one more implementation, one that combines both the random and echo behaviour: `RandomEchoPoemGenerator`.

Now what? Which class will that one extend?

If we're extending `PoemGenerator`, we'll have to override both our `data` and `parts` methods, essentially copying code from both `RandomPoemGenerator` and

`EchoPoemGenerator`. That's bad design, copying code around. What if we extend `RandomPoemGenerator`? We'd need to reimplement `parts` from `EchoPoemGenerator`. If we'd implement `EchoPoemGenerator` instead, it would be the other way around.

To be honest, extending `PoemGenerator` and copying both implementations seems like the best solution. Since then, we're at least making it clear to future programmers that this is a thing on its own, and we weren't able to solve it any other way.

But let's be frank: whatever solution, it's all crap. We have fallen into the pitfall that is inheritance. And this, dear reader, happens so often in real-life projects: we think of inheritance as the perfect solution to override and reuse behaviour, and it always seems to work great at the start. Next comes along a new feature that causes more abstractions, and causes our code to grow out of hand. We thought we mastered inheritance but it kicked our asses instead.

So what's the problem — the *actual* problem — with our code? Doesn't it make sense that `RandomPoemGenerator` extends from `PoemGenerator`? It is a poem generator, isn't it? That's indeed the way we think of inheritance: using *"is a"*. And yes, `RandomPoemGenerator` *is a* `PoemGenerator`, but `RandomPoemGenerator` isn't *only* generating a poem now, is it?

Sandi Metz suggests the following question to identify the underlying problem: "what changed between the two — what changed during inheritance?". Well... In the case of `RandomPoemGenerator`, it's the `data` method; for `EchoPoemGenerator`, it's the `parts` method. And it just so happens that having to combine those two parts is what made our inheritance solution blow up.

Do you know what this means? It means that `parts` and `data` are something on their own. They are *more* than a protected implementation detail of our poem generator. They are what is valued by the client, they are the *essence* of our program.

So let's treat them as such.

With two separate concerns identified, we need to give them a proper name. The first one is about whether lines should be randomised or not. Let's call it the `Orderer`; it will take an original array and return a new version of it with its items sorted in a specific way.

```
interface Orderer
{
    public function order(array $data): array;
}
```

The second concern is about formatting the output - whether it should be echoed or not. Let's call this concept a `Formatter`. Its task is to receive the array of lines and format all of those lines into one string.

```
interface Formatter
{
    public function format(array $lines): string;
}
```

And here comes the magic. We're extracting this logic from our `PoemGenerator`, but we still need a way to access it from within. So let's inject both an orderer and formatter into the `PoemGenerator`:

```
class PoemGenerator
{
    public function __construct(
        public Formatter $formatter,
        public Orderer $orderer,
    ) {}

    // …
}
```

With both available, let's change the implementation details of phrase and data:

```
class PoemGenerator
{
    // …

    protected function phrase(int $number): string
    {
        $parts = $this->parts($number);

        return $this->formatter->format($parts);
    }

    protected function data(): array
    {
        return $this->orderer->order([
            'the horse and the hound and the horn that belonged to',
            // …
            'the house that Jack built',
        ]);
    }
}
```

And finally, let's implement `Orderer`:

```
class SequentialOrderer implements Orderer
{
    public function order(array $data): array
    {
        return $data;
    }
}

class RandomOrderer implements Orderer
{
    public function order(array $data): array
    {
        shuffle($data);

        return $data;
    }
}
```

As well as `Formatter`:

```
class DefaultFormatter implements Formatter
{
    public function format(array $lines): string
    {
        return implode("\n        ", $lines);
    }
}

class EchoFormatter implements Formatter
{
    public function format(array $lines): string
    {
        $lines = array_reduce(
            $lines,
            fn (array $output, string $line) ⇒
                [...$output, "{$line} {$line}"],
            []
        );

        return implode("\n        ", $lines);
    }
}
```

The default implementations, `DefaultFormatter` and `SequentialOrderer` might not do any complex operations, though still they are a valid business concern: a "sequential order" and "default format" are two valid cases needed to create the poem as we know it in its normal form.

Do you realise what just happened? You might be thinking that we're writing more code, but you're forgetting something... we can remove our `RandomPoemGenerator` and

`EchoPoemGenerator` altogether, we don't need them anymore, we can solve all of our cases, with only the `PoemGenerator`:

```
$generator = new PoemGenerator(
    new EchoFormatter(),
    new RandomOrderer(),
);
```

We can make our lives still a little easier by providing proper defaults:

```
class PoemGenerator
{
    public function __construct(
        public ?Formatter $formatter = null,
        public ?Orderer $orderer = null,
    ) {
        $this->formatter ??= new DefaultFormatter();
        $this->orderer ??= new SequentialOrderer();
    }
}
```

And using named properties, we can construct a `PoemGenerator` whatever way we want:

```
$generator = new PoemGenerator(
    formatter: new EchoFormatter(),
);

$generator = new PoemGenerator(
    orderer: new RandomOrderer(),
);

$generator = new PoemGenerator(
    formatter: new EchoFormatter(),
    orderer: new RandomOrderer(),
);
```

No more need for a third abstraction!

This is *real* object-oriented programming. I told you that OOP isn't about inheritance, and this example shows its true power. By composing objects out of other objects, we're able to make a flexible and durable solution, one that solves all of our problems in a clean way. This is what composition over inheritance is about, and it's one of the most fundamental pillars in OO.

I'll admit: I don't always use this approach when I start writing code. It's often easier to start simply during the development process and not think about abstracts or composition. I'd even say it's a great rule to follow: don't abstract too soon. The

important lesson isn't that you should always use composition. Instead, it's about identifying the problem you encounter and using the right solution to solve it.

WHAT ABOUT TRAITS?

You might be thinking about traits to solve our poem problem. You could make a `RandomPoemTrait` and `EchoPoemTrait`, implementing `data` and `phrase`. Yes, traits can be another solution, just like inheritance also is a working solution. I'm going to make the case why composition is still the better choice, but first let's show in practice what these traits would look like:

```
trait RandomPoemTrait
{
    protected function data(): array
    {
        $data = parent::data();

        shuffle($data);

        return $data;
    }
}
```

```
trait EchoPoemTrait
{
    protected function parts(int $number): array
    {
        return array_reduce(
            parent::parts($number),
            fn (array $output, string $line) ⇒
                [...$output, "{$line} {$line}"],
            []
        );
    }
}
```

You could use these to implement `RandomEchoPoemGenerator` like so:

```
class RandomEchoPoemGenerator extends PoemGenerator
{
    use RandomPoemTrait;
    use EchoPoemTrait;
}
```

Traits indeed solve the problem of code reusability; that's exactly why they were added to the language. When I mentioned the latest feature request to add `RandomEchoPoemGenerator`, I remarked that there was no clean way to solve the problem without code duplication, which was the stepping stone to search for another solution — composition. Did I deliberately ignore traits to make my point? No. While they do solve the problem of reusability, they *don't* have the added benefits we discovered when exploring composition.

First we discovered that the order and format of the poem are crucial business rules and shouldn't be treated as protected implementation details somewhere in the class.

We made `Orderer` and `Formatter` to make our code better represent the real-world problem we're trying to solve. If we're choosing traits and subclasses instead, we lose this explicitness once again.

Second, our poem example shows two parts of the `PoemGenerator` that are configurable. What if there's three or four? If we're adding two more traits, we also have to create new subclass implementations for those traits, *and* all relevant combinations with existing traits. The number of subclasses would grow exponentially. Even with our current example, there's already three subclasses: `RandomPoemGenerator`, `EchoPoemGenerator` and `RandomEchoPoemGenerator`. Composition on the other hand only requires us to add only two new classes. Our code would grow out of hand if there were more complex business rules to account for.

I'm not suggesting traits shouldn't be used at all; just like inheritance, they have their uses. What's most important is that you critically assess the pros and cons of all solutions for a given problem instead of falling back to what seems the easiest at first.

I think this reasoning applies to everything programming related, whether you're coding in an object-oriented, procedural, or functional style. OOP got a bad name because people started to scale it out of hand, without rethinking their architecture. I hope we can change that.