
Style guide for
Object Design
Matthias Noback

Style Guide for Object Design

Matthias Noback

This book is for sale at http://leanpub.com/object-design

This version was published on 2019-03-19

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

Contents

Foreword

By Ross Tuck

All programmers can appreciate the value of a good name. A name is a promise: it tells you what you can and can't expect. It helps you make a decision and move on.

But a Good Name, with capital letters, is more than just a contract. A Good Name shares something about the soul of what's described. It gives you a glimpse of the creator's intention, its raison d'être. The name Walkie-Talkie tells you both the what and the why of the thing. The name doesn't need to be accurate: fire ants aren't made of fire but they might as well be. A Good Name reveals.

I'm happy to say that the book you are now holding has an exceptionally Good Name.

You may be familiar with the style guides often used by journalists, such as the AP Stylebook or The Chicago Manual of Style. Like those books, this one offers both guidelines and guidance on achieving a clear, consistent tone across a larger team.

In following this model, Matthias's goal is humble and direct. There are no new concepts, no fancy tools, or radical breakthroughs. Matthias has simply documented what he was already doing. He has captured his approach to designing systems and distilled them into the elements of his style.

These elements are discussed in terms of existing patterns, some of which you may have heard of elsewhere. What I find wonderful in this book is the realization that these patterns seldom live in isolation. What seemed reasonable on the page will often fail in the IDE without the guarantees offered by other practices. For example, consider how difficult unit tests are without dependency injection.

And so, this Style Guide to Object Design is greater than the sum of its parts. Taken together, these patterns interlock and strengthen each other. Deeper treatments of each topic exist but what Matthias has done is take a random collection of best practices and turn them into a cohesive, recognizable style.

Publishing your own style may seem strange, even arrogant. After all, why should we follow his style rather than yours or mine? The most brutal argument suggests that like coding standards which dictate where the brackets and spaces go, it doesn't matter

which one we follow, only that we follow one. This one just has the virtue of already being documented.

Yet, I believe the intent is not to constrain the reader but to provide a reference point. Innovation happens within constraints, they say. Iterate on this style, improve on it, start here but make it your own because your circumstances are unique. After all, the only thing worse than a newspaper written like a love letter would be a love letter written like a newspaper.

If you remain unconvinced, then this book at least offers the chance to see a high-level practitioner at work. There is a remarkable honesty and vulnerability on display. There is no secret sauce, no technique withheld in these pages. What you see here is how Matthias approaches his daily work, nothing more, nothing less.

Indeed, in the process of reviewing this book, I was struck several times by the memory of watching over his shoulder as he codes, listening to him weigh each concern and select a tool. I imagined myself pointing at something that seemed out of place in his familiar style and Matthias just smiling. “Oh yes,” he’d say, “That was the interesting part.”

I hope you enjoy that same experience as much as I have.

Introduction

Between learning how to program and learning about advanced design patterns and principles, there isn't much educational material for object-oriented programmers. The books that are often recommended are hard to read and it often proves difficult to apply the theory to your everyday coding problems. Besides, most developers don't have a lot of time to read a book. So what's left is a gap in programming education materials.

Even without reading books, as a programmer, you will grow over time. You learn how to make better design choices. You collect a set of basic rules that you can pretty much always apply, freeing mental capacity to focus on other interesting areas of the code that need improving. So what can't be learned from reading software books, can in fact be learned through years and years of struggling with code.

I wrote this book to close part of the education materials gap, and to give you some suggestions that will help you write better object-oriented code. These will be mostly short and simple suggestions. Technically, there won't be a lot to master. But I find that following these suggestions (or "rules") helps moving your focus from trivial aspects of the code, to more interesting aspects that deserve more of your attention. If everyone on your team follows the same suggestions, the code in your project will have a more uniform style.

So I boldly claim that the object design rules provided in this book will improve the quality of your objects, and of your entire project. This fits well with the other goal I had in mind: that this book can be used as part of an on-boarding process for new team members. After telling them about the coding standard for the project and showing them the style guide for commit messages and code reviews, you can hand out this book and explain how your team aims for good object design in all areas of the project.

Who is this book for?

This book is for programmers with at least some basic knowledge of PHP or another object-oriented programming language. You should understand the language's possibilities regarding class design. That is, I expect you to know what it means to define a

class, instantiate it, extend it, mark it as `abstract`, define a method, call it, define parameters and parameter types, define return types, define properties and their types, etc.

I also expect you to have some experience with all of this, even it's not much. You should be able to read this book if you've just finished a basic programming course, but also if you've been working with classes for years.

Design rules for this book

Writing a book means taking a huge amount of material and shaping it into something relatively small and manageable. This is why creativity, without any constraints, is expected to produce chaos. If you mix in some constraints, the chances of success are much larger. Setting a number of constraints for yourself will help you make most of the micro-decisions along the way, preventing you from getting stuck.

Here are the constraints I introduced for this book:

- *There will be no chapter titles with the name of a principle or pattern in it.*

  It should be clear what the advice is, without having to remember what all that jargon meant.
- *Chapters are short.*

  I don't want this to be a heavy book that takes months to finish. I want the programming advice to be readily available, instead of being buried deep inside oracle-like philosophical mumblings. The advice should be clear and easy to follow.
- *Chapters have useful summaries.*

  If you want to quickly read back a piece of advice, or refer to it, you shouldn't be forced to read the whole chapter again. There should be useful summaries at the end of every chapter.
- *Code samples should come with suggestions for testing.*

  Good object design makes testing an object easier, and at the same time, the design of an object can improve by testing it in the right way. So it makes sense to show suggestions for (unit) testing next to suggestions for object design.

- *Tests are written with PHPUnit in mind.*

  Nowadays there are several useful test frameworks for PHP. However, since the most commonly used tool is still PHPUnit, I'm using it for the examples in this book. I'll take a minimalistic approach, which means that you should be able to accomplish the same style of unit testing using any other test tool.
  - There should be no dependencies between test methods.
  - There should be no `setUp()` or `tearDown()` code (in this book; in practice this is sometimes useful).
  - We shouldn't use data providers (the same applies here).
  - Only use a limited set of assertions, like `assertTrue($actual)` and `assertEquals($expected, $actual)`. For testing failure scenarios I'll assume that the function `expectException(string $exceptionClass, string $exceptionMessage, callable $function)` exists.
  - For brevity I use PHPUnit's assertion *functions* instead of the methods that are available inside `PHPUnit\Framework\TestCase`. You can make these functions globally available in your project by adding the file `vendor/phpunit/phpunit/src/Framework/Assert/Functions.php` to the `files` autoload section[1] of your `composer.json`.
- *Code samples will be written for PHP 7.4.*

  PHP 7.4 isn't even available at the time of writing, but it has been decided that it will support typed properties. It will be smart to use typed properties in your code as soon as that will be possible. For this book I assume it's already possible, which means I can save some space, leaving away doc blocks with type annotations. It should also make the book more accessible for people with a background in Java programming (or similar object-oriented programming languages).

Conventions

- I use the word "client" to represent the place where the class or method gets called. Sometimes this is also called "call site".
- I use the word "user" to represent the programmer who uses your class by instantiating it, and calling methods on it. Note that this usually isn't the user of the application as a whole.

[1] https://getcomposer.org/doc/04-schema.md#files

- In code samples, I abbreviate statements with `// ...` and expressions with `....` I use `//` or `/* ... */` to add some more context to the examples.

Overview of the contents

The book is divided into four major parts. The first part, *The lifecycle of an object*, is about creating objects and manipulating their state. We cover the ways in which you as a developer should facilitate these actions. We first make a distinction between two types of objects: services, and other objects. Then we discuss how service objects should be created, and that they shouldn't be manipulated after instantiation. Then we take a look at other objects, how they should be created, and how, in some cases, they can be manipulated afterwards.

The second part, *Using objects*, provides some rules to apply when you're adding behavior to objects. There will be two kinds of things a client can do with an object: let it perform a task, or retrieve information from it. These two use cases of an object come with different implementation rules.

The third part, *Changing the behavior of objects*, provides some advice for when it comes to changing the behavior of an object. It discusses how you can change, or enhance, the behavior of services, by composing existing objects into new objects, which expose the desired behavior.

Finally, the fourth part, *A field guide to objects*, shows you around different areas of an application, and the different types of objects you may encounter in these areas.

In the *Epilogue* I provide you with a brief overview of topics you may look into if you want to know more about object design, including some recommendations for further reading.

Acknowledgments

Before we get to it, let me first thank some people here. First of all, the 125 people who bought the book before it was even finished. This was very encouraging. Thanks for your feedback, too, in particular Сергей Лукьяненко, Iosif Chiriluta, Nikola Paunovic, Niko Mouk, Damon Jones and Mo Khaledi. A special mention goes to Rémon van de Kamp, who provided quite a list of insightful comments.

A big thank you goes to Ross Tuck and Laura Cody for performing a thorough review of this book. Thanks to your suggestions the arguments are better, the progression smoother, and the risk of misunderstanding lower.

The lifecycle of an object

In an application there will typically be two types of objects:

1. Service objects that either perform a task, or return a piece of information.
2. Objects that will hold some data, and optionally expose some behavior for manipulating or retrieving that data.

Objects of the first type will be created once, then used any number of times, but nothing can be changed about them. They have a very simple lifecycle. Once they've been created, they can run forever, like little machines with specific tasks. These objects are called services.

The second type of objects are used by the first type to complete their tasks. These objects are the materials that the services work with. For instance, a service may retrieve such an object from another service, and it would then manipulate the object and hand it over to another service for further processing. The lifecycle of a material object may therefore be more complicated than that of a service: after it has been created, it could optionally be manipulated, and it may even keep an internal event log of everything that has happened to it.

In the first part of this book, we'll cover both the creation and manipulation phase for these two types of objects. In the second part we'll look at using these objects.

1. Creating services

Objects that perform a task are often called “services”. These objects are do-ers. They often have names which indicates that: controller, renderer, calculator, etc. Service objects can be constructed by using the `new` keyword to instantiate their class, e.g. `new FileLogger()`.

In this chapter we’ll discuss all the relevant aspects of instantiating a service. You’ll learn how to deal with its dependencies, what you can and can’t do inside its constructor, and that you should be able to instantiate it once and make it reusable many times.

1.1 Inject dependencies and configuration values as constructor arguments

Services usually need other services to do their job, which are its dependencies, and they should be injected as constructor arguments:

```
interface Logger
{
    public function log(string $message): void;
}

final class FileLogger implements Logger
{
    private Formatter $formatter;

    public function __construct(Formatter $formatter)
    {
        // `Formatter` is a dependency of `FileLogger`
        $this->formatter = $formatter;
    }
```

```
    public function log(string $message): void
    {
        $formattedMessage = $this->formatter->format($message);

        // ...
    }
}

$logger = new FileLogger(new DefaultFormatter());
$logger->log('A message');
```

Making every dependency available as a constructor argument will make the service ready for use immediately after instantiation. No further setup will be required, and no mistakes can be made with that.

Sometimes a service needs some configuration values, like a location for storing files, or credentials for connecting to an external service. Inject such configuration values as constructor arguments too:

```
final class FileLogger implements Logger
{
    // ...

    private string $logFilePath;

    public function __construct(
        Formatter $formatter,
        string $logFilePath
    ) {
        // ...

        /*
         * `$logFilePath` is a configuration value which tells the
         * `FileLogger` to which file the messages should be
         * written.
         */
        $this->logFilePath = $logFilePath;
```

```
    }

    public function log(string $message): void
    {
        // ...

        file_put_contents(
            $this->logFilePath,
            $formattedMessage,
            FILE_APPEND
        );
    }
}
```

These configuration values may be globally available in your application, in some kind of a parameter bag, settings object, or otherwise large data structure containing all the other configuration values too. Instead of injecting the whole configuration object, make sure you only inject only the values that the service should have access to. In fact, only inject the values it actually needs.

Keeping together configuration values that belong together

A service shouldn't get the entire global configuration object injected, but only the values that it needs. However, some of these values will always be used together, and injecting them separately will break their natural cohesion. Take the following example where an API client gets the credentials for connecting to the API injected as separate constructor arguments:

```
final class ApiClient
{
    private string $username;
    private string $password;

    public function __construct(string $username, string $password)
    {
        $this->username = $username;
        $this->password = $password;
    }
}
```

To keep these values together, you can introduce a dedicated configuration object. Instead of injecting the username and password separately, you can inject a `Credentials` object that contains both:

```
final class Credentials
{
    private string $username;
    private string $password;

    public function __construct(string $username, string $password)
    {
        $this->username = $username;
        $this->password = $password;
    }

    public function username(): string
    {
        return $this->username;
    }

    public function password(): string
    {
        return $this->password;
    }
```

```
}

final class ApiClient
{
    private Credentials $credentials;

    public function __construct(Credentials $credentials)
    {
        $this->credentials = $credentials;
    }
}
```

1.2 Inject what you need, not where you can get it from

If a framework or library is complicated enough, it will offer you a special kind of object which holds every service and configuration value you could ever want to use. Common names for such a thing are: service locator, manager, registry, or container.

What's a service locator?

A service locator is itself a service, from which you can retrieve other services. The following example shows a service locator which has a `get()` method. When called, the locator will return the service with the given identifier, or throw an exception if the identifier is invalid:

```
final class ServiceLocator
{
    private array $services;

    public function __construct()
    {
        $this->services = [
            'logger' => new FileLogger(...)
            // We can have any number of services here...
        ];
    }

    public function get(string $identifier): object
    {
        if (!isset($this->services[$identifier])) {
            throw new LogicException(
                'Unknown service: ' . $identifier
            );
        }

        return $this->services[$identifier];
    }
}
```

In this sense, a service locator is basically a map; you can retrieve services from it, as long as you know the correct key. In practice, this key is often the name of the service class or interface that you want to retrieve.

Most often the implementation of a service locator is more advanced than the one we just saw. A service locator often knows how to instantiate all the services of an application, and it takes care of providing the right constructor arguments when doing so. It will also reuse already instantiated services, which can improve runtime performance.

Because a service locator gives you access to all of the available services in an application, it may be tempting to inject a service locator as a constructor argument and be done with it:

```
final class HomepageController
{
    private ServiceLocator $locator;

    public function __construct(ServiceLocator $locator)
    {
        /*
         * Instead of injecting the dependencies we need, we just
         * inject the whole `ServiceLocator`, from which we can
         * later retrieve any specific dependency, the moment we
         * need it.
         */
        $this->locator = $locator;
    }

    public function __invoke(Request $request): Response
    {
        $user = $this->locator->get(EntityManager::class)
            ->getRepository(User::class)
            ->getById($request->get('userId'));

        return $this->locator->get(ResponseFactory::class)
            ->create()
            ->withContent(
                $this->locator->get(TemplateRenderer::class)
                    ->render(
                        'homepage.html.twig',
                        [
                            'user' => $user
                        ]
                    ),
                'text/html'
```

```
            );
    }
}
```

This results in a lot of extra function calls in the code, which obscures the view on what the service really does. Furthermore, this service needs to have knowledge about how to retrieve dependencies. This is the opposite of the *Inversion of control* we're looking for when we use dependency injection. Finally, this service now has access to many more services that can potentially be retrieved from the service locator. Eventually this service will end up fetching all kinds of unrelated things from the service locator ad-hoc, because it doesn't push the programmer to look for a better design alternative.

Whenever a service needs another service to perform its task, it has to declare the latter explicitly as a dependency and get it injected as a constructor argument. The `ServiceLocator` in this example is not a true dependency of `HomepageController`; it's used to *retrieve* the actual dependencies. So instead of declaring the `ServiceLocator` as a dependency, the controller should declare the actual dependencies that it needs as constructor arguments, and expect them to be injected:

```
final class HomepageController
{
    private EntityManager $entityManager;
    private ResponseFactory $responseFactory;
    private TemplateRenderer $templateRenderer;

    public function __construct(
        EntityManager $entityManager,
        ResponseFactory $responseFactory,
        TemplateRenderer $templateRenderer
    ) {
        $this->entityManager = $entityManager;
        $this->responseFactory = $responseFactory;
        $this->templateRenderer = $templateRenderer;
    }

    public function __invoke(Request $request): Response
    {
```

```
        $user = $this->entityManager->getRepository(User::class)
            ->getById($request->get('userId'));

        return $this->responseFactory
            ->create()
            ->withContent(
                $this->templateRenderer->render(
                    'homepage.html.twig',
                    [
                        'user' => $user
                    ]
                ),
                'text/html'
            );
    }
}
```

We should make another iteration here. In the example we only need the `EntityManager` because we need to fetch the user repository from it. We should make it an explicit dependency instead:

```
final class HomepageController
{
    private UserRepository $userRepository;
    // ...

    public function __construct(
        UserRepository $userRepository,
        ...
    ) {
        $this->userRepository = $userRepository
        // ...
    }

    public function __invoke(Request $request): Response
    {
```

```
        $user = $this->userRepository
            ->getById($request->get('userId'));
        // ...
    }
}
```

What if I need the service and the service I retrieve from it?

Consider the following code that would need both the `EntityManager` and the `UserRepository` dependency:

```
$user = $this->entityManager
    ->getRepository(User::class)
    ->getById($request->get('userId'));
$user->changePassword($newPassword);
$this->entityManager->flush();
```

If we follow the advice to inject the `UserRepository` instead of the `EntityManager`, we end up with an extra dependency, because we'll still need that `EntityManager` for flushing (i.e. persisting) the entity.

Situations like this usually require a redistribution of responsibilities. The object that can retrieve a `User` entity might just as well be able to persist any changes that were made to it. In fact, such an object would follow an established pattern, the *Repository* pattern. Since we already have a `UserRepository` class, it would make sense to add a `flush()`, or (now that we have the opportunity to choose another name) `save()` method to it:

```
$user = $this->userRepository->getById($request->get('userId'));
$user->changePassword($newPassword);
$this->userRepository->save($user);
```

1.3 All constructor arguments should be required

Sometimes you may feel like a dependency is optional—the object could function very well without it. An example of such an optional dependency could be the `Logger` we just saw. You may consider logging to be a secondary concern for the task at hand. So to make it an optional dependency of a service, you would make it an optional constructor argument:

```
final class BankStatementImporter
{
    private ?Logger $logger;

    public function __construct(Logger $logger = null)
    {
        // $logger can be `null` or an instance of `Logger`
        $this->logger = $logger;
    }

    public function import(string $bankStatementFilePath): void
    {
        // Import the bank statement file

        // Every now and then log some information for debugging...
    }
}

/*
 * `BankStatementImporter` can be instantiated _without_ a `Logger`
 * instance:
 */
$importer = new BankStatementImporter();
```

However, this unnecessarily complicates the code inside the `BankStatementImporter` class. Whenever you want to log something, you first have to check if a `Logger` instance has indeed been provided (if you don't, you'll get a fatal error):

```
public function import(string $bankStatementFilePath): void
{
    // ...

    if ($this->logger instanceof Logger) {
        $this->logger->log('A message');
    }
}
```

To prevent this kind of workaround for optional dependencies, every dependency should be a required one.

The same goes for configuration values. You may feel like the user of a `FileLogger` doesn't really *have* to provide a path to write the log messages to, because a sensible default path exists:

```
final class FileLogger implements Logger
{
    public function __construct(
        string $logFilePath = '/tmp/app.log'
    ) {
        // ...
    }
}

/*
 * If the user omits the `$logFilePath` argument, '/tmp/app.log'
 * will be used:
 */
$logger = new FileLogger();
```

However, when someone instantiates this `FileLogger` class, it won't be immediately clear to which file the log messages will be written. The situation gets worse if the default value is buried deeper in the code:

```
final class FileLogger implements Logger
{
    private ?string $logFilePath;

    public function __construct(string $logFilePath = null)
    {
        $this->logFilePath = $logFilePath;
    }

    public function log(string $message): void
    {
        // ...

        file_put_contents(
            $this->logFilePath ?: '/tmp/app.log',
            $formattedMessage,
            FILE_APPEND
        );
    }
}
```

To figure out which file path a `FileLogger` actually uses, the user will be forced to dive into the code of the `FileLogger` class itself. Also, the default path is now an implementation detail that could easily change without the user noticing it. So instead, you should always let the user of the class provide any configuration value the object needs. If you do this for all classes, it will be very easy to find out how an object has been configured, just by looking at how it's being instantiated.

In summary: whether constructor arguments are used to inject dependencies, or to provide configuration values, in both cases constructor arguments should always be required and not have default values.

1.4 Only use constructor injection

Another trick that's used to optionally inject dependencies is to add a setter to the class, which can be called in case the user decides they want to use the dependency after all:

```php
final class BankStatementImporter
{
    private ?Logger $logger;

    public function __construct()
    {
        // The `Logger` dependency isn't a constructor argument,
    }

    public function setLogger(Logger $logger): void
    {
        // But it can be optionally provided using this setter.
        $this->logger = $logger;
    }

    // ...
}

$importer = new BankStatementImporter();

// The `BankStatementImporter` is now ready for use.

/*
 * But we can "enrich" it by injecting a `Logger` after
 * instantiation:
 */
$importer->setLogger($logger);
```

This solution comes with the same problem as described earlier: it complicates the code inside the class. Furthermore, setter injection violates two rules that we'll cover later:

1. It shouldn't be possible to create an object in an incomplete state.
2. Services should be immutable, that is, impossible to change after they have been fully instantiated.

In short, don't use setter injection, only use constructor injection.

1.5 There's no such thing as an optional dependency

All of this boils down to: "there's no such thing as an optional dependency". You either need the dependency, or you don't.

Still, consider the scenario where you really consider logging to be a secondary concern. Now that you're advised to use only constructor injection and make all constructor arguments required, what can you do about it? In many cases you can resort to a stand-in object, which looks just like the real thing, but doesn't do anything:

```
final class NullLogger implements Logger
{
    public function log(string $message): void
    {
        // Do nothing
    }
}

$importer = new BankStatementImporter(new NullLogger());
```

Such a harmless object is often called a *Null Object*, or sometimes *Dummy*.

If the injected optional dependency isn't a service, but a configuration value of some sorts, you can use a similar approach. The configuration value should still be a required argument, but you should provide a way for the user to obtain a sensible default value:

```
final class MetadataFactory
{
    public function __construct(Configuration $configuration)
    {
        // ...
    }
}

/*
 * Instead of making `MetadataFactory`'s `$configuration` argument
 * optional, provide a `Configuration` class with a sensible default
```

```
 * state:
 */
$metadataFactory = new MetadataFactory(
    Configuration::createDefault()
);
```

1.6 Make all dependencies explicit

If all of your dependencies and configuration values have been properly injected as constructor arguments, there may still be room for *hidden dependencies*. They are hidden, because they can't be recognized by a quick look at the constructor arguments.

Turn static dependencies into object dependencies

In some cases, these dependencies can be retrieved using static accessors that are globally available. Anywhere in your code you can make calls like `ServiceRegistry::get()` or `Cache::get()`. In all cases, replace these calls with constructor arguments.

```
// Before:
final class DashboardController
{
    public function __invoke(): Response
    {
        $recentPosts = [];

        if (Cache::has('recent_posts')) {
            $recentPosts = Cache::get('recent_posts');
        }

        // ...
    }
}

// After:
```

```
final class DashboardController
{
    private Cache $cache;

    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }

    public function __invoke(): Response
    {
        $recentPosts = [];

        if ($this->cache->has('recent_posts')) {
            $recentPosts = $this->cache->get('recent_posts');
        }

        // ...
    }
}
```

Turn complicated functions into object dependencies

Sometimes dependencies are hidden because they are functions, not objects. These functions are often part of the standard library of the language. For instance, `json_encode()` or `simplexml_load_file()`. There's a lot of functionality behind these functions. If you had to write the code for these functions yourself, you would introduce lots of classes to deal with all the complexity. Eventually you would inject the whole thing as a dependency into your service. This would make it a true object dependency of the service, instead of the hidden dependency that a function usually is.

Promoting these functions to become true service dependencies requires the introduction of a custom class which wraps the function call. The wrapper class is also a great starting place to add more logic around the standard library function, like providing default arguments, or improve the way it handles errors:

```php
// Before:
final class ResponseFactory
{
    public function createApiResponse(array $data): Response
    {
        // `json_encode()` is a hidden dependency:
        return new Response(
            json_encode($data, JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT),
            [
                'Content-Type' => 'application/json'
            ]
        );
    }
}

// After:

final class JsonEncoder
{
    /**
     * @throws RuntimeException
     */
    public function encode(array $data): string
    {
        try {
            /*
             * From now on, a call to `json_encode()` will always have
             * the right arguments:
             */
            return json_encode(
                $data,
                JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT
            );
        } catch (RuntimeException $previous) {
            /*
             * We can throw our own exception now, providing more
```

```
             * information that will help us with debugging:
             */
            throw new RuntimeException(
                'Failed to encode data: ' . var_export($data, true),
                0,
                $previous
            );
        }
    }
}

final class ResponseFactory
{
    private JsonEncoder $jsonEncoder;

    public function __construct(JsonEncoder $jsonEncoder)
    {
        /*
         * A `JsonEncoder` instance can now be injected as an
         * actual, explicit dependency.
         */

        $this->jsonEncoder = $jsonEncoder;
    }

    public function createApiResponse($data): Response
    {
        return new Response(
            $this->jsonEncoder->encode($data),
            [
                'Content-Type' => 'application/json'
            ]
        );
    }
}
```

Promoting the job of JSON encoding to a true object dependency of `ResponseFactory` makes it easier for the user of this class to form a picture of what it does, simply by looking at its list of constructor arguments.

Should all functions be promoted to object dependencies?

Of course, not all functions have to wrapped in objects and be injected as dependencies. For instance, functions that could easily be written inline (e.g. `array_keys()`, `strpos()`, etc.) definitely don't need to be wrapped.

To determine whether or not you should extract an object dependency you could ask the following questions:

- Would you want to replace or enhance the behavior provided by this dependency at some point?
- Is there a certain level of complexity to the behavior of this dependency, such that you couldn't achieve the same result with just a few lines of custom code?
- Is the function dealing with objects instead of just primitive-type values?

If the answers are mostly yes, you'd likely want to turn the function call into an object dependency. An added benefit is that it will be easier to describe in a test what behavior you expect from it. This will help you replace the function call with another one, some custom code, or maybe even an entire library that exposes the same behavior.

Make system calls explicit

There's a subset of functions and classes provided by the language that will also count as implicit dependencies, functions that reach out to the world outside. Examples are the `DateTime` and `DateTimeImmutable` classes, and functions like `time()` and `file_get_contents()`.

Consider the following class which depends on the system clock to get the current time:

```
final class MeetupRepository
{
    private Connection $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function findUpcomingMeetups(string $area): array
    {
        /*
         * Instantiating a `DateTimeImmutable` object with no
         * arguments will implicitly ask the system what the current
         * time is:
         */

        $now = new DateTimeImmutable();

        return $this->findMeetupsScheduledAfter($now, $area);
    }

    public function findMeetupsScheduledAfter(
        DateTimeImmutable $time,
        string $area
    ): array {
        // ...
    }
}
```

The current time isn't something that this service could derive from either the provided method arguments, or from any of its dependencies. So it's a hidden dependency. Since there is no service available to get the current time, you have to define your own:

```php
/*
 * Maybe a suitable name for this new service, that can tell us the
 * current time, would simply be "Clock":
 */

interface Clock
{
    public function currentTime(): DateTimeImmutable;
}

/*
 * The standard implementation for this service will use the
 * system's clock to return a `DateTimeImmutable` object,
 * representing the current time:
 */

final class SystemClock implements Clock
{
    public function currentTime(): DateTimeImmutable
    {
        return new DateTimeImmutable();
    }
}

final class MeetupRepository
{
    // ...
    private Clock $clock;

    public function __construct(
        Clock $clock,
        /* ... */
    ) {
        $this->clock = $clock;
    }
```

```
    public function findUpcomingMeetups(string $area): array
    {
        /*
         * Instead of "creating" the current time on the spot, we
         * can now ask the `Clock` service for it:
         */
        $now = $this->clock->currentTime();

        // ...
    }
}

$meetupRepository = new MeetupRepository(new SystemClock());
$meetupRepository->findUpcomingMeetups('NL');
```

By moving the system call ("What's the current time?") outside of the `MeetupRepository` class, we have improved the testability of the `MeetupRepository` class itself. If we'd run the tests in the original situation, the class would use the actual current time. This makes the result of the test dependent on the date and time on which the test runs. That's likely to make the test unstable, and fail after a certain date. Instead of applying patches to the problem, now that we have the `Clock` interface, we can replace the "current time" based on the system's clock with a "fixed" time, that's completely under our control:

```
/*
 * The `FixedClock` implementation of the `Clock` interface is one
 * we can use in tests. When instantiated, we have to provide a
 * `DateTimeImmutable` object which represents the "current time".
 */

final class FixedClock implements Clock
{
    private DateTimeImmutable $now;

    public function __construct(DateTimeImmutable $now)
    {
```

```
        $this->now = $now;
    }

    public function currentTime(): DateTimeImmutable
    {
        return $this->now;
    }
}

/*
 * When testing the `MeetupRepository`, we pass in a `FixedClock` as
 * constructor argument. This will make the test results fully
 * deterministic.
 */

$meetupRepository = new MeetupRepository(
    new FixedClock(
        new DateTimeImmutable('2018-12-24 11:16:05')
    )
);
$meetupRepository->findUpcomingMeetups('NL');
```

Passing in a `Clock` object as a constructor argument allows the `MeetupRepository` to request the current time. But we could also let the client of `MeetupRepository` provide the current time as a method argument of `findUpcomingMeetups()`. Then there would no longer be a need for the `Clock` dependency:

```
final class MeetupRepository
{
    public function __construct(/* ... */)
    {
        // `Clock` is no longer needed
    }

    public function findUpcomingMeetups(
        string $area,
```

```
        DateTimeImmutable $now
    ): array {
        /*
         * The current time will be provided by clients of this
         * method.
         */

        // ...
    }
}
```

After this final change we should revise our initial assessment that retrieving the current time is something we need an object dependency for. Passing the current time as a method argument turns it into contextual information that is needed to perform the task of finding upcoming meetups.

1.7 Data relevant for the task should be passed as method arguments instead of constructor arguments

As you know, a service should get all of its dependencies and configuration values injected as constructor arguments. The information about the task itself, including any relevant contextual information, should be provided as method arguments.

As a counter-example, consider an `EntityManager` that can only be used to save one entity to the database:

```
final class EntityManager
{
    private object $entity;

    public function __construct(object $entity)
    {
        $this->entity = $entity;
    }
```

```
    public function save(): void
    {
        // ...
    }
}

$user = new User(...);
$comment= new Comment(...);

$entityManager = new EntityManager($user);
$entityManager->save();

/*
 * To save another entity, we'd have to instantiate another
 * `EntityManager` as well:
 */
$entityManager = new EntityManager($comment);
$entityManager->save();
```

This wouldn't be a very useful class, since you'd have to instantiate it again and again, for every job you have for it.

Having an entity as a constructor argument may already look like an obviously bad choice of design to you. A more subtle, and more common scenario would be a service that gets the current `Request`, or the current `Session` object injected as a constructor argument:

```
final class Addressbook
{
    private Session $session;

    public function __construct(Session $session)
    {
        $this->session = $session;
    }

    public function getAllContacts(): array
```

```
    {
        return $this->select()
            ->where([
                'userId' => $this->session->getUserId(),
                'companyId' => $this->session->get('companyId')
            ])
            ->getResult();
    }
}
```

This `AddressBook` service can't be used to get the contacts of a different user or company than the one that is known to the current `Session` object. That is, it can only run in one context.

Getting part of the job details injected as constructor arguments gets in the way of making the service reusable. And the same goes for contextual data. All of this information should be passed as method arguments, in order to make the service reusable for different jobs.

A guiding question to decide whether something should be passed as a constructor argument or as a method argument, is: "Could I run this service in a batch, without requiring it to be instantiated over and over again?". Or in the case of PHP: "What if memory wouldn't be wiped after every web request; could this service be used for subsequent requests, or would it have to be reinstantiated?"

In the case of the `EntityManager` we saw earlier, it would be impossible to save multiple entities in a batch without instantiating the service again. So `$entity` should become a parameter of the `save()` method, instead of being a constructor argument:

```
final class EntityManager
{
    public function save(object $entity): void
    {
        // ...
    }
}
```

The same goes for the `AddressBook`. It couldn't be used in a batch, to get all contacts

for different users and different companies. So `getAllContacts()` should have extra arguments for the current company and user ID:

```
final class AddressBook
{
    public function getAllContacts(
        UserId $userId,
        CompanyId $companyId
    ): array {
        return $this->select()
            ->where([
                'userId' => $userId,
                'companyId' => $companyId
            ])
            ->getResult();
    }
}
```

In fact, the word “current” is a useful signal that this information is contextual information that needs to be passed as method arguments. E.g. “the current time”, “the currently logged in user ID”, “the current web request”, etc.

1.8 Don’t allow the behavior of a service to change after it has been instantiated

As we saw earlier, injecting optional dependencies later on changes the behavior of a service. This makes the service unpredictable. The same goes for methods that don’t inject dependencies, but allow you to influence the behavior of the service from the outside. For instance:

```php
final class Importer
{
    private bool $ignoreErrors = true;

    public function ignoreErrors(bool $ignoreErrors): void
    {
        $this->ignoreErrors = $ignoreErrors;
    }

    // ...
}

$importer = new Importer();

// When we use the `Importer` now, it will ignore errors...

$importer->ignoreErrors(false);

// When we use it now, it won't ignore errors anymore...
```

Make sure that this can't happen. All dependencies and configuration values should be there from the start, and it shouldn't be possible to reconfigure the service after it has been instantiated.

Another example:

```php
final class EventDispatcher
{
    private array $listeners = [];

    // You can add a new event listener for the given type of event:
    public function addListener(
        string $event,
        callable $listener
    ): void {
        $this->listeners[$event][] = $listener;
    }
```

```
    // You can also remove it again:
    public function removeListener(
        string $event,
        callable $listener
    ): void {
        $listeners = $this->listeners[$event] ?? [];

        foreach ($listeners as $key => $callable) {
            if ($callable === $listener) {
                unset($this->listeners[$event][$key]);
            }
        }
    }

    public function dispatch(object $event): void
    {
        /*
         * Any listener that hasn't been removed yet, will be
         * called:
         */

        $listeners = $this->listeners[get_class($event)] ?? [];
        foreach ($listeners as $callable) {
            $callable($event);
        }
    }
}
```

By allowing event listeners to be added and removed on-the-fly, the behavior of `EventDispatcher` will be quite unpredictable because it can even change over time. In this case, we should turn the array of event listeners into a constructor argument, and remove the `addListener()` and `removeListener()` methods:

```
final class EventDispatcher
{
    private array $listeners;

    public function __construct(array $listenersByEventName)
    {
        $this->listeners = $listenersByEventName;
    }

    // ...
}
```

Because `array` isn't a very specific type, and could basically contain anything, you should validate the `$listenersByEventName` argument before assigning it. We'll take a closer look at validation of constructor arguments later in this chapter.

1.9 Do nothing inside a constructor, only assign properties

Creating a service means injecting constructor arguments, thereby preparing the service for use. The real work will be done inside one of the object's methods. Inside a constructor, you may sometimes be tempted to do more than just assigning properties, to make the object truly ready for use. Take for example the `FileLogger` class below. Upon construction, it will prepare the log file for writing:

```
final class FileLogger implements Logger
{
    private string $logFilePath;

    public function __construct(string $logFilePath)
    {
        $logFileDirectory = dirname($logFilePath);
        if (!is_dir($logFileDirectory)) {
            // create the directory if it doesn't exist yet
            mkdir($logFiledirectory, 0777, true);
        }
```

```
        touch($logFilePath);

        $this->logFilePath = $logFilePath;
    }

    // ...
}
```

But now, instantiating a `FileLogger` will already leave a trace on the filesystem, even if you would never actually use the object to write a log message. Therefore, it's considered good object manners to not do anything inside a constructor. The only thing you should do in a service constructor is validate the provided constructor arguments, and then assign them to the object's properties:

```
final class FileLogger implements Logger
{
    private string $logFilePath;

    public function __construct(string $logFilePath)
    {
        // Only copy values into properties

        $this->logFilePath = $logFilePath;
    }

    public function log(string $message): void
    {
        $this->ensureLogFileExists();

        // ...
    }

    private function ensureLogFileExists(): void
    {
        if (is_file($this->logFilePath)) {
```

```
            return;
        }

        $logFileDirectory = dirname($this->logFilePath);
        if (!is_dir($logFileDirectory)) {
            // create the directory if it doesn't exist yet
            mkdir($logFileDirectory, 0777, true);
        }

        touch($this->logFilePath);
    }
}
```

Pushing the work outside of the constructor, deeper into the class, is one possible solution. In this case, however, we will only find out if it's possible to write to the log file at the moment the first message is written to it. Most likely, we want to hear about such problems sooner. So what we could do instead is push the work outside of the constructor: we don't want it to happen after constructing the `FileLogger`, but before:

```
final class FileLogger implements Logger
{
    private string $logFilePath;

    public function __construct(string $logFilePath)
    {
        /*
         * We expect that the log file path has already been
         * properly set up for us, so all we do here is a "safety
         * check":
         */

        if (!is_writable($logFilePath)) {
            throw new InvalidArgumentException(sprintf(
                'Log file path "%s" should be writable',
                $logFilePath
            ));
```

```
        }
        $this->logFilePath = $logFilePath;
    }

    public function log(string $message): void
    {
        // No need for a call to `ensureLogFileExists()` or anything

        // ...
    }
}

/*
 * The task of creating the log directory and file should be moved to
 * the bootstrap phase of the application itself:
 */

final class LoggerFactory
{
    public function createFileLogger(string $logFilePath): FileLogger
    {
        if (!is_file($logFilePath)) {
            $logFileDirectory = dirname($logFilePath);
            if (!is_dir($logFileDirectory)) {
                // create the directory if it doesn't exist yet
                mkdir($logFileDirectory, 0777, true);
            }

            touch($logFilePath);
        }

        return new FileLogger($logFilePath);
    }
}
```

Note that moving the log file setup code outside the constructor of `FileLogger`

changes the contract of the `FileLogger` itself. In the initial situation you could pass in any log file path, and the `FileLogger` would take care of everything (creating the directory if necessary, and checking that the file path itself is writable). In the new situation, `FileLogger` accepts a log file path, and expects that its containing directory already exists. We can push out even more to the bootstrap phase of the application and rewrite the contract of `FileLogger` to state that the client has to provide a file path to a file that already exists and is writable. In code:

```
final class FileLogger implements Logger
{
    private string $logFilePath;

    /**
     * @param string $logFilePath Absolute path to a log file that
     *                            already exists and is writable.
     */
    public function __construct(string $logFilePath)
    {
        $this->logFilePath = $logFilePath;
    }

    // ...
}

final class LoggerFactory
{
    public function createFileLogger(string $logFilePath): FileLogger
    {
        /*
         * Besides taking care of the directory, we now also make sure
         * that the log file exists and is writable:
         */
        if (!is_file($logFilePath)) {
            $logFileDirectory = dirname($logFilePath);
            if (!is_dir($logFileDirectory)) {
                // create the directory if it doesn't exist yet
```

```
                mkdir($logFileDirectory, 0777, true);
            }

            touch($logFilePath);
        }

        if (!is_writable($logFilePath)) {
            throw new InvalidArgumentException(sprintf(
                'Log file path "%s" should be writable',
                $logFilePath
            ));
        }

        return new FileLogger($logFilePath);
    }
}
```

Let's take a look at another, more subtle example of an object that does something in its constructor. Take a look at the `Mailer` class, which calls one of its dependencies inside the constructor:

```
final class Mailer
{
    private Translator $translator;
    private string $defaultSubject;

    public function __construct(Translator $translator)
    {
        $this->translator = $translator;

        // ...

        $this->defaultSubject = $this->translator
            ->translate('default_subject');
    }
```

```
    // ...
}
```

What happens if you change the order of assignments?

```
final class Mailer
{
    private Translator $translator;
    private string $defaultSubject;

    public function __construct(
        Translator $translator,
        string $locale
    ) {
        $this->defaultSubject = $this->translator
            ->translate('default_subject', $locale);

        // ...

        $this->translator = $translator;
    }

    // ...
}
```

You'll get a fatal error for calling `translate()` on `null`. This is why the rule that you should only assign properties in service constructors, comes with the consequence that the assignments could happen in any order. If it can't, you know that you're doing something in your constructor.

The constructor of this `Mailer` class is also a nice example of how contextual data, namely the "current user's locale", gets passed as a constructor argument. As you know, contextual information should be passed as a method argument instead.

1.10 Throw an exception when an argument is invalid

When a client of a class provides an invalid constructor argument, in most cases the type checker will warn you. For instance, when the argument requires a `Logger` instance and the client provides a `bool` value. However, there are other types of arguments where only relying on the type system will be insufficient. For instance, when one of the constructor arguments should be an `int`, representing some configuration flag:

```
final class Alerting
{
    private int $minimumLevel;

    public function __construct(int $minimumLevel)
    {
        $this->minimumLevel = $minimumLevel;
    }
}

$alerting = new Alerting(-99999999);
```

By simply accepting any `int` for `$minimumLevel`, you can't be sure that the provided value is realistic and can be used by the remaining code in a meaningful way. Instead, the constructor should check that the value is valid, and if it isn't, throw an exception. Only after the argument has been validated should it be assigned.

```
final class Alerting
{
    private int $minimumLevel;

    public function __construct(int $minimumLevel)
    {
        if ($minimumLevel <= 0) {
            throw new InvalidArgumentException(
                'Minimum alerting level should be greater than 0'
```

```
            );
        }
        $this->minimumLevel = $minimumLevel;
    }
}

// This will throw an `InvalidArgumentException`
$alerting = new Alerting(-99999999);
```

By throwing an exception inside the constructor, you can prevent the object from being constructed based on invalid arguments.

Instead of throwing custom exceptions, it's quite common to use reusable assertion functions for validating method and constructor arguments. We will talk about these in more detail later.

Choosing not to throw an exception could also be an option, if that won't break the object's behavior in a later stage. Consider the following example:

```
final class Router
{
    private array $controllers;
    private string $notFoundController;

    public function __construct(
        array $controllers,
        string $notFoundController
    ) {
        // Should you check if the `$controller` array is empty?
        $this->controllers = $controllers;

        $this->notFoundController = $notFoundController;
    }

    public function match(string $uri): string
    {
        foreach ($this->controllers as $pattern => $controller) {
```

```
            if ($this->matches($uri, $pattern)) {
                return $controller;
            }
        }

        return $this->notFoundController;
    }

    private function matches(string $uri, string $pattern): bool
    {
        // ...
    }
}

$router = new Router(
    [
        '/' => 'homepage_controller'
    ],
    'not-found'
);

// This will return 'homepage_controller':
$router->match('/');
```

Should you validate the `$controllers` argument here, to verify that it contains at least one URI pattern/controller name pair? Actually, you don't have to, because the behavior of the `Router` won't be broken if the `$controllers` array is empty. If you accept an empty array, and the client calls `match()`, it will just return the "not found" controller, because there are no matching patterns found for the given URI (nor any other URI). This is the behavior you'd expect from a router, so it shouldn't be considered a sign of broken logic.

However, you should validate that all the keys and values in the `$controller` array are strings. This will help you figure out programming mistakes early on:

```
final class Router
{
    // ...

    public function __construct(array $controllers)
    {
        foreach (array_keys($controllers) as $pattern) {
            if (!is_string($pattern)) {
                throw new InvalidArgumentException(
                    'All URI patterns should be provided as strings'
                );
            }
        }
        foreach ($controllers as $controller) {
            if (!is_string($controller)) {
                throw new InvalidArgumentException(
                    'All controllers should be provided as strings'
                );
            }
        }
        $this->controllers = $controllers;
    }

    // ...
}
```

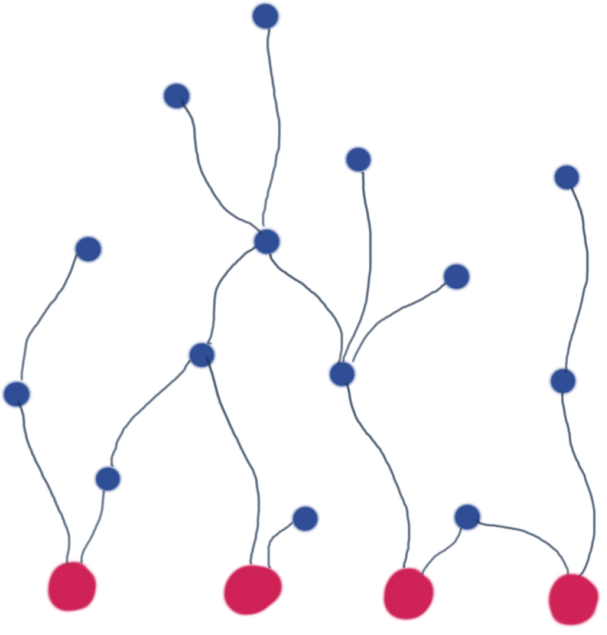
Alternatively, you may use an assertion library or custom assertion functions to validate the contents of `$controllers` (we'll discuss assertion functions later), or use the type system to check the types for you:

```
final class Router
{
    private array $controllers = [];

    public function __construct(array $controllers)
    {
        /*
         * Don't assign `$controllers` directly, but let
         * `addController()` take care of that:
         */

        foreach ($controllers as $pattern => $controller) {
            $this->addController($pattern, $controller);
        }
    }

    private function addController(
        string $pattern,
        string $controller
    ): void {
        /*
         * This private method should only be called by the
         * constructor.
         *
         * Because it has explicit `string` types for its arguments,
         * calling this method on every key/value pair in the
         * provided `$controllers` array will be the equivalent of
         * asserting that all keys and values in the array are
         * strings.
         */

        $this->controllers[$pattern] = $controller;
    }

    // ...
}
```

1.11 Define services as an immutable object graph with only a few entry points

Once the application framework calls your controller (be it a web controller or a controller for a command-line based application), you can consider every dependency to be known. For instance, the web controller needs a repository to fetch some objects from, it needs the templating engine to render a template, it needs a response factory to create a response object, etc. All these dependencies have their own dependencies, which, when carefully listed as constructor arguments, can be created at once, resulting in an often pretty large graph of objects.

If the framework decides to call a different controller, it will use a different graph of dependent objects to perform its task. The controller itself is also a service with dependencies, so you can consider controllers to be the entry points of the application's object graph.



The graph contains all services of an application, with its entry point services marked.

Most applications have something like a service container which describes how all

of the application's services can be constructed, what their dependencies are, how they can be constructed, and so on. The container behaves as a service locator, too. You can ask it to return one of its services, so you can use it. We already saw how a service locator can be used, earlier in this chapter, when we discussed the rule that you should inject the dependencies that you need, not the service locator that allows you to retrieve those dependencies.

Given the following:

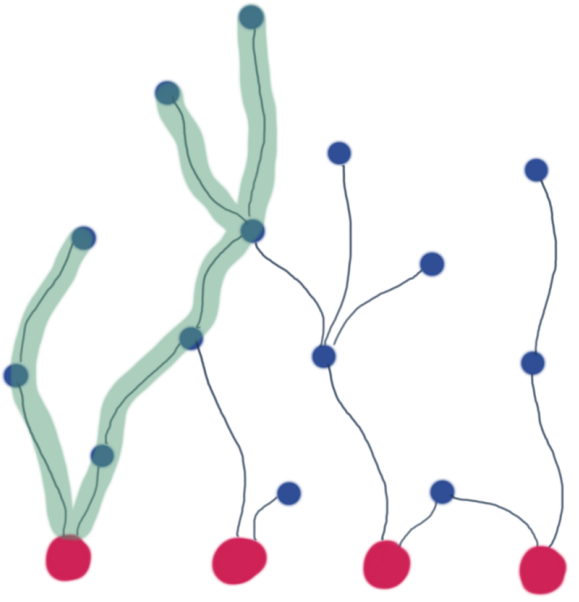
1. All the services in an application form one large object graph, and
2. The entry points will be the controllers, and from that point on,
3. No service will ever need the service locator itself to retrieve services

We should conclude that the service container only needs to have public methods for retrieving controllers. The other services defined in the container can and should remain private, because they will only be needed as injected dependencies for controllers. Translated to code, this means we could have a service container which can be used as a service locator to retrieve a controller from, and all the other service instantiation logic that's needed to produce the controller objects stays behind the scenes, in `private` methods:

```
final class ServiceContainer
{
    public function homepageController(): HomepageController
    {
        return new HomepageController(
            $this->userRepository(),
            $this->responseFactory(),
            $this->templateRenderer()
        );
    }

    private function userRepository(): UserRepository
    {
        // ...
    }
```

```
    private function responseFactory(): ResponseFactory
    {
        // ...
    }

    private function templateRenderer(): TemplateRenderer
    {
        // ...
    }

    // ...
}

/*
 * The framework could use a router to find out the right controller
 * for the current request. It then fetches the controller from the
 * service locator, and lets it handle the request:
 */

if ($uri === '/') {
    $controller = $serviceContainer->homepageController();
    $response = ($controller)($request);
    // ...
} elseif (...) {
    // retrieve and call another controller
}
```

A service container allows reuse of services, which is why, starting with the controller as an entry point, not every branch of the object graph will be completely standalone. For example, another controller may use the same `TemplateRenderer` instance as the `HomepageController`. This is why it's important to make services behave as predictably as possible. If you apply all the previously discussed rules, you will end up with an object graph that can be instantiated once, then reused many times.



Different entry points use different branches of the object graph.

1.12 Summary

Services should be created in one go, providing all their dependencies and configuration values as constructor arguments. All service dependencies should be explicit, and they should be injected as objects. All configuration values should be validated. When a constructor receives an argument that is in any way invalid, it should throw an exception.

After construction, a service should be immutable; its behavior can't be changed by calling any of its methods.

All services of an application combined will form a large, immutable object graph, often managed by a service container. Controllers are the entry points of this graph. Services can be instantiated once, reused many times.

2. Creating other objects

We mentioned earlier that there are two types of objects: services, and other objects. The second type of objects can be divided into more specific subtypes, namely *Value objects* and *Entities*. Services will create or retrieve entities, manipulate them or pass them on to other services. They will also create value objects and pass them on as method arguments, or create modified copies of them. In this sense, entities and value objects are the materials which services use to perform their tasks.

In the previous chapter we looked at how a service object should be created. In this chapter, we'll look at the rules for creating these other objects.

2.1 Require the minimum amount of data needed to behave consistently

Consider the following example:

```
final class Position
{
    private int $x;
    private int $y;

    public function __construct()
    {
    }

    public function setX(int $x): void
    {
        $this->x = $x;
    }

    public function setY(int $y): void
```

```
    {
        $this->y = $y;
    }

    public function distanceTo(Position $other): float
    {
        return sqrt(
            ($other->x - $this->x) ** 2 +
            ($other->y - $this->y) ** 2
        );
    }
}

$position = new Position();
$position->setX(45);
$position->setY(60);
```

Until we've called both `setX()` and `setY()`, the object is in an inconsistent state. We can notice this when we call `distanceTo()` before calling `setX()` or `setY()`—it won't give a meaningful answer.

Since it's crucial to the concept of a position that it has both `x` and `y` parts, we have to enforce this by making it impossible to create a `Position` object without providing values for both `x` and `y`.

```
final class Position
{
    private int $x;
    private int $y;

    public function __construct(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }

    public function distanceTo(Position $other): float
```

```
    {
        return sqrt(
            ($other->x - $this->x) ** 2 +
            ($other->y - $this->y) ** 2
        );
    }
}

/*
 * `x` and `y` have to be provided, or you won't be able to get an
 * instance of `Position`
 */
$position = new Position(45, 60);
```

This is an example of how a constructor can be used to protect a *Domain invariant*—something that's always true for a given object, based on the domain knowledge you have about the concept it represents. The domain invariant that's being protected here is: "A position has both an x and a y coordinate."

2.2 Require data that is meaningful

In the previous example, the constructor would accept any integer, positive or negative, and to infinity in both directions. Now consider another system of coordinates, where positions consists of a latitude and a longitude, which together determine a place on earth. In this case, not every possible value for latitude and longitude would be considered meaningful.

```php
final class Coordinates
{
    private float $latitude;
    private float $longitude;

    public function __construct(float $latitude, float $longitude)
    {
        $this->latitude = $latitude;
        $this->longitude = $longitude;
    }

    // ...
}

$meaningfulCoordinates = new Coordinates(45.0, -60.0);

/*
 * There's nothing that stops us from creating a `Coordinates`
 * object that doesn't make any sense at all:
 */
$offThePlanet = new Coordinates(1000.0, -20000.0);
```

Always make sure that clients can't provide data that is meaningless. What counts as meaningless can be phrased as a domain invariant, too. In this case, the invariant is: "The latitude of a coordinate is a value between -90 and 90 inclusive. The longitude of a coordinate is a value between -180 and 180 inclusive."

When you're designing your objects, let yourself be guided by these domain invariants. Collect more invariants as you go. And incorporate them in your unit tests:

```
expectException(
    InvalidArgumentException::class,
    'Latitude',
    function() {
        new Coordinates(90.1, 0.0);
    }
);
expectException(
    InvalidArgumentException::class,
    'Longitude',
    function() {
        new Coordinates(0.0, 180.1);
    }
);
// and so on
```

To make these tests pass, throw an exception in the constructor as soon as something about the provided arguments looks wrong:

```
final class Coordinates
{
    // ...

    public function __construct(float $latitude, float $longitude)
    {
        if ($latitude > 90 || $latitude < -90) {
            throw new InvalidArgumentException(
                'Latitude should be between -90 and 90'
            );
        }
        $this->latitude = $latitude;

        if ($longitude > 180 || $longitude < -180) {
            throw new InvalidArgumentException(
                'Longitude should be between -180 and 180'
            );
```

```
        }
        $this->longitude = $longitude;
    }
}
```

Although the exact order of the statements in your constructor shouldn't matter (as we discussed earlier), it's still recommended to perform the checks directly above their associated property assignments. This will make it easy for the reader to understand how the two statements are related.

In some cases it's not enough to verify that every constructor argument is valid on its own. Sometimes you may need to verify that the provided constructor arguments are meaningful together.

However, it always pays to look for ways to redesign your object and get rid of these multi-argument validations. Take for example the following class which represents a business deal between two parties, where there's a total amount of money, and it has to be divided between two parties.

```
final class Deal
{
    public function __construct(
        int $totalAmount,
        int $amountToFirstParty,
        int $amountToSecondParty
    ) {
        // ...
    }
}
```

You should at least validate the constructor arguments separately (e.g. the total amount should be larger than 0, etc.). But then there's also an invariant that spans all the arguments: the sum of what both parties get should be equal to the total amount:

```
final class Deal
{
    public function __construct(
        int $totalAmount,
        int $amountToFirstParty,
        int $amountToSecondParty
    ) {
        // ...

        if ($amountToFirstParty + $amountToSecondParty
            !== $totalAmount) {
            throw new InvalidArgumentException(...);
        }
    }
}
```

However, as you may have noted, this rule could've be enforced in a much simpler way. You could say that the total amount itself doesn't even have to be provided, as long as the client provides positive numbers for `$amountToFirstParty` and `$amountToSecondParty`. The `Deal` object can figure out on its own what the total amount of the deal was by summing these values. So the need to validate constructor arguments together disappears:

```
final class Deal
{
    private int $amountToFirstParty;
    private int $amountToSecondParty;

    public function __construct(
        int $amountToFirstParty,
        int $amountToSecondParty
    ) {
        if ($amountToFirstParty <= 0) {
            throw new InvalidArgumentException(/* ... */);
        }
        $this->amountToFirstParty = $amountToFirstParty;
```

```
        if ($amountToSecondParty <= 0) {
            throw new InvalidArgumentException(/* ... */);
        }
        $this->amountToSecondParty = $amountToSecondParty;
    }

    public function totalAmount(): int
    {
        return $this->amountToFirstParty
            + $this->amountToSecondParty;
    }
}
```

Another example where it would seem that constructor arguments have to be validated together, is the following class which represents a line:

```
final class Line
{
    public function __construct(
        bool $isDotted,
        int $distanceBetweenDots
    ) {
        /*
         * We only care about the distance if the line is a
         * dotted line. For solid lines, there's no distance to
         * be dealt with.
         */
        if ($isDotted && $distanceBetweenDots <= 0) {
            throw new InvalidArgumentException(
                'Expect the distance between dots to be positive.'
            );
        }

        // ...
```

```
    }
}
```

However, this could more elegantly be dealt with by providing the client with two distinct ways of defining a line. One which is dotted, one which is solid.

```
final class Line
{
    private bool $isDotted;
    private int $distanceBetweenDots;

    public static function dotted(int $distanceBetweenDots): Line
    {
        if ($distanceBetweenDots <= 0) {
            throw new InvalidArgumentException(
                'Expect the distance between dots to be positive.'
            );
        }

        $line = new self(...);
        $line->distanceBetweenDots = $distanceBetweenDots;
        $line->isDotted = true;

        return $line;
    }

    public static function solid(): Line
    {
        // No need to worry about `$distanceBetweenLines` here!

        $line = new self();

        $line->isDotted = false;

        return $line;
```

```
    }
}
```

These static methods are so-called *Named constructors* and we'll take a closer look at them later in this chapter.

2.3 Don't use custom exception classes for invalid argument exceptions

So far we've been throwing a generic `InvalidArgumentException` whenever a method argument didn't match our expectations. We could've used a custom exception class which extends from `InvalidArgumentException`. The advantage would be that we could catch that specific type of exception and deal with it in a specific way:

```
final class SpecificException extends InvalidArgumentException
{
}

try {
    // try to create the object
} catch (SpecificException $exception) {
    // handle this specific problem in a specific way
}
```

However, with invalid argument exceptions, you should rarely need to do that. An invalid argument means that the client is using the object in an invalid way. Most of the time this will be caused be a programming mistake. In that case you'd better fail hard and not try to recover, but fix the mistake instead.

For `RuntimeExceptions` on the other hand, it often makes more sense to use custom exception classes, because you may be able to recover from them, or convert them into a user-friendly error message. We'll discuss custom runtime exceptions and how to create them later in this chapter.

Even if you only use the generic `InvalidArgumentException` class to validate method arguments, you still need a way to distinguish between them in a unit test. Let's take another look at the `Coordinates` class and its constructor we saw earlier:

```
final class Coordinates
{
    // ...

    public function __construct(float $latitude, float $longitude)
    {
        if ($latitude > 90 || $latitude < -90) {
            throw new InvalidArgumentException(
                'Latitude should be between -90 and 90'
            );
        }
        $this->latitude = $latitude;

        if ($longitude > 180 || $longitude < -180) {
            throw new InvalidArgumentException(
                'Longitude should be between -180 and 180'
            );
        }
        $this->longitude = $longitude;
    }
}
```

We'd want to verify that clients can't pass in the wrong arguments. So we'd write a few tests, like these:

```
// Latitude can't be more than 90.0
expectException(
    InvalidArgumentException::class,
    function() {
        new Coordinates(90.1, 0.0);
    }
);
// Latitude can't be less than -90.0
expectException(
    InvalidArgumentException::class,
    function() {
```

```
        new Coordinates(-90.1, 0.0);
    }
);

// Longitude can't be more than 180.0
expectException(
    InvalidArgumentException::class,
    function() {
        new Coordinates(-90.1, 180.1);
    }
);
```

In the last test case, the `InvalidArgumentException` that gets thrown from the constructor isn't the one we'd expect it to be. Because the test case reuses an invalid value for latitude (-90.1) from the previous test case, trying to construct a `Coordinates` object will throw an exception telling us that "Latitude should be between -90.0 and 90.0". But the test was supposed to verify that the code would reject invalid values for longitude. So this basically leaves the range check for longitude uncovered in a test scenario, even though all the tests succeed.

To prevent this kind of mistake, make sure to always verify that the exception you catch in a unit test is in fact the expected one. A pragmatic option for that is to verify that the exception message contains certain predefined words:

```
expectException(
    InvalidArgumentException::class,
    // This word is supposed to be in the exception message:
    'Longitude',
    function() {
        new Coordinates(-90.1, 180.1);
    }
);
```

In our case, adding this expectation about the exception message will make the test fail. It will pass again once we provide the constructor with a sensible value for latitude.

2.4 Extract new objects to prevent domain invariants from being verified in multiple places

You'll often find the same validation logic repeated in the same class, or even in different classes:

```
final class User
{
    private string $emailAddress;

    public function __construct(string $emailAddress)
    {
        /*
         * Validate that the provided email address is valid:
         */

        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        $this->emailAddress = $emailAddress;
    }

    // ...

    public function changeEmailAddress(string $emailAddress): void
    {
        /*
         * Validate it again, if it's going to be updated:
         */

        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException(
                'Invalid email address'
```

```
            );
        }
        $this->emailAddress = $emailAddress;
    }
}

// The constructor will catch invalid email addresses
expectException(
    InvalidArgumentException::class,
    'email',
    function () {
        new User('not-a-valid-email-address');
    }
);

// create a valid `User` object first
$user = new User('valid@emailaddress.com');

// `changeEmailAddress()` will also catch invalid email addresses
expectException(
    InvalidArgumentException::class,
    'email',
    function () use ($user) {
        $user->changeEmailAddress('not-a-valid-email-address');
    }
);
```

Although you could easily extract the email address validation logic into a separate method, the better solution is to introduce a new type of object which represents a valid email address. Since we expect all objects to be valid the moment they are created, we can leave out the "valid" part from the class name, and implement it like this:

```
final class EmailAddress
{
    private string $emailAddress;

    public function __construct(string $emailAddress)
    {
        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        $this->emailAddress = $emailAddress;
    }
}
```

Wherever you encounter an `EmailAddress` object, you know it represents a value that has already been validated:

```
final class User
{
    private EmailAddress $emailAddress;

    public function __construct(EmailAddress $emailAddress)
    {
        $this->emailAddress = $emailAddress;
    }

    // ...

    public function changeEmailAddress(EmailAddress $emailAddress)
    {
        $this->emailAddress = $emailAddress;
    }
}
```

Wrapping values inside new objects called *Value objects*, is something that's useful beyond simply repeating validation logic. As soon as you notice that a method accepts a

primitive-type value (`string`, `int`, etc.), you should consider introducing a class for it. The guiding question for deciding whether or not to do this is: "Would any `string`, `int`, etc. be acceptable here?" If the answer is no, introduce a new class for this concept.

You should consider the value object class itself to be a type, just like `string`, `int`, etc. are types. That way, by introducing more objects to represent domain concepts, you're basically extending the type system. The PHP runtime will be able to support you much better, because it can do type-checking for you, and make sure that only the right types will end up being used when passing method arguments and returning values.

2.5 Extract new objects to represent composite values

When creating all these new types, you'll find that some of them naturally belong together, and always get passed together from method call to method call. For example, an amount of money always comes with the currency of the amount. When a method would receive just the amount, it wouldn't know how to deal with it.

```
final class Amount
{
    // ...
}

final class Currency
{
    // ...
}

// Amount and Currency always go together, like here:
final class Product
{
    public function setPrice(
        Amount $amount,
        Currency $currency
    ): void {
        // ...
    }
```

```
}

// And here:
final class Converter
{
    public function convert(
        Amount $localAmount,
        Currency $localCurrency,
        Currency $targetCurrency
    ): Amount
    {
        // ...
    }
}
```

In this last example, the return type is actually quite confusing. An `Amount` will be returned, but the currency of this amount is expected to match the given `$targetCurrency`. This is not evident by just looking at the types used in this method.

Whenever you notice that values belong together (or can always be found together), wrap these values into a new type. In the case of `Amount` and `Currency`, a good name for the combination of the two could be "money":

```
final class Money
{
    public function __construct(Amount $amount, Currency $currency)
    {
        // ...
    }
}
```

Using this type indicates that you want to keep these values together, although if you'd want to use them separately, you still can.

Adding more object types also leads to more typing, is that really necessary?

`100` has fewer characters than `new Amount(100)`. However, all that extra typing gives you the benefits of using object types:

- You can be certain that the data the object wraps has been validated.
- An object usually exposes additional, meaningful behaviors that make use of its data.
- An object can keep values together that belong together.
- An object helps you keep implementation details away from its clients.

If you feel like it's a hassle to create all these objects one by one based on scalar values, you can always introduce helper methods for creating them. For instance:

```
// Before:

$money = new Money(new Amount(100), new Currency('USD'));

// After:

$money = Money::fromScalars(100, 'USD');
```

You will find more about this style of creating objects in the section about named constructors.

2.6 Use assertions to validate constructor arguments

We've already seen several examples of constructors that throw exceptions when something is wrong. The general structure is always:

```
if (somethingIsWrong()) {
    throw new InvalidArgumentException(...);
}
```

These checks at the beginning of methods are called "assertions". The general idea behind an assertion is that it's a safety check. Assertions can be used to establish the situation, examine the materials, and signal if anything is wrong. For this reason, assertions are also called "pre-condition checks". Once you're past these assertions, it should be safe to perform the task at hand, with the data that has been provided.

Because you'll often write the same kind of checks in many different places, it'll be quite convenient to use an assertion library instead (for PHP: `beberlei/assert` or `webmozart/assert`). Such a library contains many assertion functions that will cover almost all situations. Some examples are:

```
Assertion::greaterThan(mixed $value, mixed $limit);
Assertion::isCallable(mixed $value);
Assertion::between(
    mixed $value,
    mixed $lowerLimit,
    mixed $upperLimit
);
// and so on
```

The question is always: "Should you verify that these assertions work in a unit test for your object?" The guiding question is: "Would it be theoretically possible for the language runtime to catch this case?" If the answer is "yes", don't write a unit test for it.

As an example, PHP doesn't have a way to set the type of an argument to "a list of <class name>". Instead, you'd have to rely on the pretty meaningless `array` type. To verify that a given array is indeed a flat list of objects of a certain type, you would use an assertion:

```
final class EventDispatcher
{
    public function __construct(array $eventListeners)
    {
        Assertion::allIsInstanceOf(
            $eventListeners,
            EventListener::class
        );

        // ...
    }
}
```

Since this is an error condition that a more evolved type-system could catch, you don't have to write a unit test that catches the `AssertionFailedException` thrown by `allIsInstanceOf()`. However, if you have to inspect a given value and check that it's within a certain range, if you have to verify the number of items in a list, etc. you do have to write a unit test that shows you've covered the edge cases. Revisiting a previous example, the domain invariant that a given latitude is always between -90 and 90 inclusive, should be verified with a test:

```
expectException(
    AssertionFailedException::class,
    'latitude',
    function() {
        new Coordinates(-90.1, 0.0)
    }
);
// etc.
```

Don't collect exceptions

Although the tools sometimes allow it, you shouldn't save up assertion exceptions and throw them as a list. Assertions are not meant to provide a convenient list of things that are wrong to the user. They are meant for the programmer, who needs to know that they are using a constructor or method in the wrong way. So as soon as you notice anything wrong, just make the object scream.

If you want to supply the user with a list of things that are wrong about the data they provided (by submitting a form, sending an API request, etc.) you should use a *Data transfer object* (DTO) and validate it instead. We'll discuss this type of object at the end of this chapter.

2.7 Don't inject dependencies, optionally pass them as method arguments

Services can have dependencies and they should be injected as constructor arguments. But other objects shouldn't get any dependencies injected, only values, value objects, or lists of them. If a value object still needs a service to perform some task, you could optionally inject it as a method argument, like in the following example:

```
final class Money
{
    private Amount $amount;
    private Currency $currency;

    public function __construct(Amount $amount, Currency $currency)
    {
        $this->amount = $amount;
        $this->curency = $currency;
    }

    public function convert(
        ExchangeRateProvider $exchangeRateProvider,
        Currency $targetCurrency
```

```
    ): Money {
        /*
         * `ExchangeRateProvider` is a method argument, not a
         * constructor argument.
         */

        $exchangeRate = $exchangeRateProvider->getRateFor(
            $this->currency,
            $targetCurrency
        );

        return $exchangeRate->convert($this->amount);
    }
}
```

It may sometimes feel a bit strange to pass a service as a method argument, so it makes sense to consider alternative implementations too. Maybe we shouldn't pass the `ExchangeRateProvider` service, but only the information we get from it: the `ExchangeRate`. This would require `Money` to expose both its internal `Amount` and `Currency` objects, but that may be a reasonable price to pay for not injecting the dependency:

```
final class ExchangeRate
{
    public function __construct(
        Currency $from,
        Currency $to,
        Rate $rate
    ) {
        // ...
    }

    public function convert(Amount $amount): Money
    {
        // ...
    }
```

```
}

$money = new Money(...);
// We retrieve the `ExchangeRate` upfront,
$exchangeRate = $exchangeRateProvider->getRateFor(
    $money->currency(),
    $targetCurrency
);
// Then use it to convert the amount we have:
$converted = $exchangeRate->convert($money->amount());
```

After moving things around one more time, we could settle for a solution that involves only exposing `Money`'s internal `Currency` object, not its `Amount` (we will get back to the topic of exposing object internals in a later chapter):

```
final class Money
{
    public function convert(ExchangeRate $exchangeRate): Money
    {
        Assertion::equals(
            $this->currency,
            $exchangeRate->fromCurrency()
        );

        return new Money(
            $exchangeRate->rate()->applyTo($this->amount),
            $exchangeRate->targetCurrency()
        );
    }
}

$money = new Money(...);
$exchangeRate = $exchangeRateProvider->getRateFor(
    $money->currency(),
    $targetCurrency
```

```
);
$converted = $money->convert($exchangeRate);
```

You could argue that this solution expresses more clearly the domain knowledge we have about money and exchange rates, like: the converted amount will be in the target currency of the exchange rate, its "source" currency will be the same currency as the currency of the original amount.

In some cases, the need for passing around services as method arguments could be a hint that the behavior should be implemented as a service instead. In the case of converting an amount of money to a given currency we might as well create a service and let it do the work, collecting all the relevant information from the `Amount` and `Currency` objects provided to it:

```
final class ExchangeService
{
    private ExchangeRateProvider $exchangeRateProvider;

    public function __construct(
        ExchangeRateProvider $exchangeRateProvider
    ) {
        $this->exchangeRateProvider = $exchangeRateProvider;
    }

    public function convert(
        Money $money,
        Currency $targetCurrency
    ): Money {
        $exchangeRate = $this->exchangeRateProvider
            ->getRateFor($money->currency(), $targetCurrency);

        return new Money(
            $exchangeRate->rate()->applyTo($money->amount()),
            $targetCurrency
        );
    }
}
```

Which solution you choose depends on how close you want to keep the behavior to the data, whether or not you think it's too much for an object like `Money` to know about exchange rates too, or how bad you think it is that object internals have to be exposed.

2.8 Use named constructors

For services it's fine to use the standard way of defining constructors, i.e. `public function __construct()`. However, for other types of objects, it's recommended to use *named constructors*. These are `public static` methods, which return an instance. They could be considered object factories.

Create from primitive type values

A common case for using named constructors is when an object can be constructed from one or more primitive-type values. This results in methods like `fromString()`, `fromInt()`, etc.

```
final class Date
{
    private const FORMAT = 'd/m/Y';
    private DateTimeImmutable $date;

    private function __construct()
    {
        // do nothing here
    }

    public static function fromString(string $date): Date
    {
        $object = new self();

        $dateTimeImmutable = DateTimeImmutable::createFromFormat(
            self::FORMAT,
            $date
        );
```

```
        /*
         * Assert that the `createFromFormat()` didn't return
         * `false`...
         */

        $object->date = $dateTimeImmutable;

        return $object;
    }
}

$date = Date::fromString('1/4/2019');
```

It's important to add a regular, but `private`, constructor method, so that clients won't be able to by-pass the named constructor you offer to them, possibly leaving the object in an invalid or incomplete state.

Wait, does this work?

It may seem strange that this `public static fromString()` method can create a new object instance and manipulate its `$date` property from the outside. After all, this property is `private`, so that shouldn't be allowed, right? You should know that scoping in PHP is class-based. So private properties can be manipulated by any object, as long as it's of the exact same class. The `fromString()` method in this example also counts as a method of the same class, which is why it can manipulate the `$date` property directly, without the need for a setter.

Don't immediately add `toString()`, `toInt()`, etc.

When you add a named constructor which creates an object based on a primitive-type value, you may feel the need for symmetry and add a method which can convert the object back to that primitive-type value. For instance, having a `fromString()` constructor may lead you to automatically provide a `toString()` method, too. Make sure you only do this once there is an actual proven need for it.

Introduce a domain-specific concept

When talking about, for instance, sales orders with a domain expert, they would never speak about "constructing" a sales order. Maybe they talk about "creating" a sales order, or they even use something more specific like "placing" a sales order. Look out for these words and use them as method names for your named constructors:

```
final class SalesOrder
{
    public static function place(...): SalesOrder
    {
        // ...
    }
}

$salesOrder = SalesOrder::place(...);
```

Optionally use the private constructor to enforce constraints

Some objects may even offer multiple named constructors, because there are different ways in which you can construct it. For example, if you want a decimal value with a certain precision, you could choose an integer value with a positive integer precision as the normalized way of representing such a number. At the same time you may want to offer clients to use their existing values which are strings or floats as input for working with such a decimal value. Using the private constructor helps to ensure that whatever construction method is chosen, the object will end up in a complete and consistent state:

```php
final class DecimalValue
{
    private int $value;
    private int $precision;

    private function __construct(int $value, int $precision)
    {
        $this->value = $value;

        Assertion::greaterOrEqualThan($precision, 0);
        $this->precision = $precision;
    }

    public static function fromInt(
        int $value,
        int $precision
    ): DecimalValue {
        return new self($value, $precision);
    }

    public static function fromFloat(
        float $value,
        int $precision
    ): DecimalValue {
        return new self(
            (int)round($value * pow(10, $precision)),
            $precision
        );
    }

    public static function fromString(string $value): DecimalValue
    {
        $result = preg_match('/^(\d+)\.(\d+)$/', $value, $matches);
        if ($result === 0) {
            throw new InvalidArgumentException(/* ... */);
        }
```

```
        $wholeNumber = $matches[1];
        $decimals = $matches[2];

        $valueWithoutDecimalSign = $wholeNumber . $decimals;

        return new self(
            (int)$valueWithoutDecimalSign,
            strlen($decimals)
        );
    }
}
```

In summary, the advantages of using named constructors are:

- They can be used to offer several ways to construct an object.
- They can be used to introduce domain-specific synonyms for creating an object.

Besides for creating entities and value objects, named constructors can be used to offer convenient ways for instantiating custom exceptions. We'll discuss these later.

2.9 Don't use property fillers

Applying all the object design rules in this book leads to objects that are in complete control of what goes in, what stays inside, and what a client can do with them. A technique that works completely against this object design style is a property filler method, which looks like this:

```
final class Position
{
    private int $x;
    private int $y;

    public static function fromArray(array $data): Position
    {
        $position = new self();
        $position->x = $data['x'];
        $position->y = $data['y'];
        return $position;
    }
}
```

This kind of method could even be turned into a generic utility that would just copy values from the `$data` array into the corresponding properties using reflection. Though it may look convenient, the object's internals are now out in the open. So always make sure that construction of an object happens in a way that's fully controlled by the object itself.

At the end of this chapter, we'll look at an exception to this rule. For *Data transfer objects*, a property filler could be a way to map for example form data onto an object. Such an object doesn't need to protect its internal data as much as an entity or a value object has to.

2.10 Don't put anything more into an object than it needs

It's common to start designing your object by thinking about what needs to go in. For services, you may end up injecting more dependencies than you need. So there, the advice would be to inject dependencies only when you need them. The same is true for other types of objects: don't require more data than is strictly needed for implementing the object's behavior.

One type of object that often ends up carrying around more data than needed is an event object, representing something that has happened somewhere in the application.

```
final class ProductCreated
{
    public function __construct(
        ProductId $productId,
        Description $description,
        StockValuation $stockValuation,
        Timestamp $createdAt,
        UserId $createdBy,
        ...
    ) {
        // ...
    }
}

// Inside the `Product` entity
$this->recordThat(new ProductCreated(
    /*
     * Passing along all the data that was available when creating
     * the product...
     */
    ...
));
```

If you don't know which event data will be important for yet-to-be-implemented event listeners, don't add anything. Just add a constructor with no arguments at all, and add more data only when this data is needed. This way, you will provide data on a need-to-know basis.

How do you know what data should actually go into an object's constructor? By designing the object in a test-driven way. And this means that you first have to know how an object is going to be used.

2.11 Don't test constructors

Writing tests for your objects, specifying their desired behavior, will let you figure out which data is actually needed at construction time, and which data can be provided

later. It will also let you figure out which data needs to be exposed later on, and which data can stay behind the scenes, as an implementation detail of the object.

As an example, let's take another look at the `Coordinates` class we saw earlier:

```
final class Coordinates
{
    // ...

    public function __construct(float $latitude, float $longitude)
    {
        if ($latitude > 90 || $latitude < -90) {
            throw new InvalidArgumentException(
                'Latitude should be between -90 and 90'
            );
        }
        $this->latitude = $latitude;

        if ($longitude > 180 || $longitude < -180) {
            throw new InvalidArgumentException(
                'Longitude should be between -180 and 180'
            );
        }
        $this->longitude = $longitude;
    }
}
```

How can we test that the constructor works?

Well, what about:

```
function it_can_be_constructed()
{
    $coordinates = new Coordinates(60.0, 100.0);

    assertIsInstanceOf(Coordinates::class, $coordinates);
}
```

This isn't very informative. In fact, it's impossible for the assertion to fail, unless the constructor has thrown an exception, which is an execution flow we're explicitly not testing here.

What is the task of the constructor? If we look at the code, it's to assign the given constructor arguments to internal object properties. So how can we be sure that this has worked? Well, we could add getters, that would allow us to find out what's inside the object's properties:

```
final class Coordinates
{
    // ...

    public function latitude(): float
    {
        return $this->latitude;
    }

    public function longitude(): float
    {
        return $this->longitude;
    }
}
```

The test then becomes:

```
function it_can_be_constructed()
{
    $coordinates = new Coordinates(60.0, 100.0);

    assertEquals(60.0, $coordinates->latitude());
    assertEquals(100.0, $coordinates->longitude());
}
```

But now we've introduced a way for internal data to get out of the object, for no other reason than to test the constructor.

Looking back at what we've done here: we've been testing constructor code after we wrote it. We've been testing this code, knowing what's going on in there, meaning the test is very close to the implementation of the class. We've been putting data into an object, without even knowing if we'd ever need that data again. In conclusion: we've done too much, too soon, and without a healthy dose of distance to the object's implementation.

The only thing that we can and should do at this point, is test that the constructor doesn't accept invalid arguments. We've discussed this before: you should verify that providing values for latitude and longitude that are outside of their acceptable ranges, triggers an exception, making it impossible to construct the `Coordinates` object.

Further down the road we'll talk more about exposing data, but for now, take the following advice:

- Only test the constructor for ways in which it should fail.
- Only pass in data as constructor arguments when you actually need it to implement any real behavior on the object.
- Only add getters to expose internal data when this data is actually needed by some other client than the test itself.

Once you start adding actual behavior to the object, you will implicitly test the happy path for the constructor anyway, because when doing so you'll need a fully instantiated object.

2.12 The exception to the rule: Data transfer objects {#the-exception-to-the-rules:-data-transfer-objects}

All of the rules described in this chapter apply to entities and value objects, which are objects where we care a lot about their consistency and the validity of the data that ends up inside. These objects can only guarantee correct behavior if the data they use is correct too.

There's another type of object which we didn't mention so far, to which most of the previous rules don't really apply. It's a type of object that you will find at the edges of the application, where data coming from the world outside is converted into a structure that the application can work with. The nature of this process requires it to behave a little different from entities and value objects.

This special type of object is also known as a *Data transfer object* (DTO). In short:

- A DTO can be created using a regular constructor.
- Its properties can be set one by one.
- All of its properties are exposed.
- Its properties contain only primitive type values.
- Properties can optionally contain other DTOs, or simple arrays of DTOs.

Use public properties

Since a DTO doesn't protect its state, and exposes all of its properties, there really is no need for getters and setters. Which means that it's quite sufficient to use `public` properties for them. Since they can be constructed in steps, and don't require a minimum amount of data to be provided, they don't need a constructor method.

DTOs are often used as command objects, matching the user's intention, and containing all the data needed to fulfill their wish. An example of such a command object is the following `ScheduleMeetup` command, which represents the user's wish to schedule a meetup, with the given title, on the given date:

```
final class ScheduleMeetup
{
    public string $title;
    public string $date;
}
```

The way you can use such an object is, for example, by first populating it with the data submitted with a form, then passing it to a service, which will actually schedule the meetup for the user:

```
final class MeetupController
{
    public function scheduleMeetupAction(Request $request): Response
    {
        // Extract the form data from the request body:
        $formData = /* ... */;

        // Create the command object using this data:
        $scheduleMeetup = new ScheduleMeetup();
        $scheduleMeetup->title = $formData['title'];
        $scheduleMeetup->date = $formData['date'];

        $this->scheduleMeetupService->__invoke($scheduleMeetup);

        // ...
    }
}
```

The service will create an entity and value objects, and eventually persist it. These objects will throw exceptions when anything is wrong about the data that was provided to them. However, such exceptions aren't really user-friendly; they can't even be easily translated to the user's language. Also, because they break the application's flow, exceptions can't be collected and returned as a list of input errors to the user.

Don't throw exceptions, collect validation errors

If you want to allow users to correct all their mistakes in one go, before resubmitting the form, you should validate the command's data before passing the object to the service that's going to handle it. One way to do this is by adding a `validate()` method to the command, which can return a simple list of validation errors. If the list is empty it means that the submitted data was valid:

```
final class ScheduleMeetup
{
    public string $title;
    public string $date;

    public function validate(): array
    {
        $errors = [];

        if ($this->title === '') {
            $errors['title'][] = 'validation.empty_title';
        }

        if ($this->date === '') {
            $errors['date'][] = 'validation.empty_date';
        }

        \DateTimeImmutable::createFromFormat('d/m/Y', $this->date);
        $errors = \DateTime::getLastErrors();
        if ($errors['error_count'] > 0) {
            $errors['date'][] = 'validation.invalid_date_format';
        }

        return $errors;
    }
}
```

Form and validation libraries may offer you more convenient and reusable tools for

validation. For instance the Symfony Form and Validator components work really well with this kind of Data transfer objects.

Use property fillers when needed

Earlier we discussed property fillers and how they shouldn't be used when working with most objects; they expose all the object's internals. In the case of a DTO, this isn't a problem at all, because a DTO doesn't protect its internals anyway. So, if it makes sense, you can add a property filler method to a DTO. For example, to copy form data or JSON request data directly into the command object. Since filling the properties is the first thing that should happen to a DTO, it makes sense to implement the property filler as a named constructor:

```
final class ScheduleMeetup
{
    public string $title;
    public string $date;

    public static function fromFormData(
        array $formData
    ): ScheduleMeetup {
        $scheduleMeetup = new self();

        $scheduleMeetup->title = $formData['title'];
        $scheduleMeetup->date = $formData['date'];

        return $scheduleMeetup;
    }
}
```

We'll revisit DTOs later when we talk about retrieving information from objects.

2.13 Summary

Objects that are not service objects don't receive dependencies as constructor arguments, but they do receive values or value objects. Upon construction, an object should

require a minimum amount of data to be provided in order to behave consistently. If any of the provided constructor arguments is invalid in some way, the constructor should throw an exception about it.

It helps to wrap primitive type arguments inside (value) objects, making it easy to reuse validation rules for these values, but also adding more meaning to the code by picking a domain-specific name for the type (i.e. class) of the value.

For objects that aren't services, constructors should be static methods, also known as named constructors, which offer yet another opportunity for introducing domain-specific names in your code.

Don't provide any more data to a constructor than is needed to make the object behave as specified by its unit tests.

A type of object for which most of these rules don't count is a *Data transfer object*. They are used to carry data provided by the world outside, and expose all their internals.

3. Manipulating objects

As you've learned in the previous chapters, services should be designed to be immutable. This means that once a service object has been created, it can never be modified. The biggest advantage is that its behavior will be predictable, and that it can be reused to perform the same task using different input.

So we know that services should be immutable objects, but what about the other types of objects: *Entities*, *Value objects*, and *Data transfer objects*?

Entities: identifiable objects which track changes and record events

Entities are the application's core objects. They represent important concepts from the business domain, like a reservation, an order, an invoice, a product, a customer, etc. They model knowledge that developers have gained about that business domain. An entity holds the relevant data, it may offer ways to manipulate that data, and it may expose some useful information based on that data:

```
final class SalesInvoice
{
    /**
     * @var Line[]
     */
    private array $lines = [];

    private bool $finalized = false;

    // You can create a sales invoice
    public static function create(/* ... */): SalesInvoice
    {
        // ...
    }
```

```
    // You can manipulate its state, e.g. by adding lines to it,
    public function addLine(/* ... */): void
    {
        if ($this->finalized) {
            throw new RuntimeException(/* ... */);
        }

        $this->lines[] = Line::create(/* ... */);
    }

    // You can finalize it,
    public function finalize(): void
    {
        $this->finalized = true;
        // ...
    }

    // And it exposes some useful information about itself
    public function totalNetAmount(): Money
    {
        // ...
    }

    public function totalVatAmount(): Money
    {
        // ...
    }
}
```

An entity may change over time, but all this time it should be the same object that undergoes all the changes. That's why an entity needs to be identifiable. When creating it we give it an identifier:

```
final class SalesInvoice
{
    private SalesInvoiceId $salesInvoiceId;

    public static function create(
        SalesInvoiceId $salesInvoiceId
    ): SalesInvoice {
        $object = new self();

        $object->salesInvoiceId = $salesInvoiceId;

        return $object;
    }
}
```

This identifier can be used by the entity's repository to save the object. Later on we can use that same identifier to retrieve it from the repository, after which it can be modified again:

```
// First we create the `SalesInvoice` and save it:
$salesInvoiceId = $this->salesInvoiceRepository->nextIdentity();
$salesInvoice = SalesInvoice::create($salesInvoiceId);
$this->salesInvoiceRepository->save($salesInvoice);

/*
 * At a later moment, we may retrieve it again to make further changes
 * to it:
 */
$salesInvoice = $this->salesInvoiceRepository->getBy($salesInvoiceId);
$salesInvoice->addLine(/* ... */);
$this->salesInvoiceRepository->save($salesInvoice);
```

Given that the state of an entity changes over time, entities are mutable objects. They come with specific rules for their implementation. The methods that change the entity's state should have a `void` return type and their names should be in the imperative form (e.g. `addLine()`, `finalize()`). They have to protect the entity against

ending up in an invalid state (e.g. `addLine()` checks that the invoice hasn't been finalized already). And they shouldn't expose all their internals, just for testing what's going on inside. Instead, an entity should keep a change log and expose that, so other objects can find out what has changed about it, and why:

```
final class SalesInvoice
{
    /**
     * @var object[]
     */
    private array $events = [];

    // You can finalize it
    public function finalize(): void
    {
        $this->finalized = true;

        $this->events[] = new SalesInvoiceFinalized(/* ... */);
    }

    /**
     * @return object[]
     */
    public function recordedEvents(): array
    {
        return $this->events;
    }
}

// In a test scenario:
$salesInvoice = SalesInvoice::create(/* ... */);
$salesInvoice->finalize();

assertEquals(
    [
        new SalesInvoiceFinalized(/* ... */)
```

```
    ],
    $salesInvoice->recordedEvents()
);

/*
 * In a service we can allow event listeners to respond to the
 * internally recorded events:
 */
$salesInvoice = $this->salesInvoiceRepository->getBy($salesInvoiceId);
$salesInvoice->finalize(/* ... */);
$this->salesInvoiceRepository->save($salesInvoice);

$this->eventDispatcher->dispatchAll(
    $salesInvoice->recordedEvents()
);
```

Value objects: replaceable, anonymous, and immutable values

Value objects are completely different objects. They are often much smaller, with just one or two properties. They can represent a domain concept, too. In that case, they represent part of an entity, or an aspect of it. For example, in the `SalesInvoice` entity, we need value objects for the ID of the sales invoice, the date on which the invoice was created, and the ID and quantity of the product on each line:

```
final class SalesInvoiceId
{
    // ...
}

final class Date
{
    // ...
}

final class Quantity
```

```
{
    // ...
}

final class ProductId
{
    // ...
}

final class SalesInvoice
{
    public static function create(
        SalesInvoiceId $salesInvoiceId,
        Date $invoiceDate
    ): SalesInvoice {
        // ...
    }

    public function addLine(
        ProductId $productId,
        Quantity $quantity
    ): void {
        $this->lines[] = Line::create(
            $productId,
            $quantity
        );
    }
}
```

As we saw in the previous chapter, value objects wrap one or more primitive-type values, and can be created by providing these values to their constructors:

```
final class Quantity
{
    public static function fromInt(
        int $quantity,
        int $precision
    ): Quantity {
        // ...
    }
}

final class ProductId
{
    public static function fromInt(int $productId): ProductId
    {
        // ...
    }
}
```

We don't need value objects to be identifiable. We don't care about the exact instance we're working with, since we don't need to track the changes that happen to a value object. In fact, we shouldn't change a value object at all. If we want to transform it to some other value, we should just instantiate a new copy, which represents the modified value. As an example, when adding two quantities, instead of changing the internal value of the original `Quantity`, we return a new `Quantity` object to represent their sum:

```
final class Quantity
{
    private int $quantity;
    private int $precision;

    private function __construct(
        int $quantity,
        int $precision
    ) {
        $this->quantity = $quantity;
```

```
        $this->precision = $precision;
    }

    public static function fromInt(
        int $quantity,
        int $precision
    ): Quantity {
        return new self($quantity, $precision);
    }

    public function add(Quantity $other): Quantity
    {
        Assertion::same($this->precision, $other->precision);

        return new self(
            $this->quantity + $other->quantity,
            $this->precision
        );
    }
}

// A quantity of 1500 with a precision of 2 represents 15.00
$originalQuantity = Quantity::fromInt(1500, 2);

// The modified quantity represents 15.00 + 5.00 = 20.00
$newQuantity = $originalQuantity->add(Quantity::fromInt(500, 2));
```

By returning a new copy instead of manipulating the existing object, we effectively make the `Quantity` value object immutable. Once created, it won't change.

Value objects don't necessarily represent domain concepts only. They can occur anywhere in the application. A value object is just any immutable object that wraps primitive-type values.

Data transfer objects: simple objects with fewer design rules

Another type of object that wraps primitive-type values is a *Data transfer object*. We already discussed it in the previous chapter. Although some prefer to implement a DTO as an immutable object, this level of protection often gets in the way of other characteristics you may be after. For instance, a tool like the Symfony Form component would want to fill the properties one by one. You also don't want to maintain, nor unit test a DTO, as it has no significant behavior (it just holds data), so you don't want it to have too many methods (i.e. getters and setters). In the end, you may settle on designing a DTO to have public properties. Although PHP doesn't offer you the option to actually make these fields read-only, imagine them to be read-only and write-once:

```
final class Line
{
    public int $productId;
    public int $quantity;
}

final class CreateSalesInvoice
{
    public string $date;

    /**
     * @var Line[]
     */
    public array $lines = [];
}
```

For data transfer objects we don't have design rules as strong as the ones for entities and value objects. For the latter design quality and data integrity are more important than for data transfer objects. This is why the design rules in this chapter apply to entities and value objects.

3.1 Prefer immutable objects

For entities it will be desirable that they can be manipulated after construction. In general, however, you should prefer objects to be immutable. In fact, most objects that

are not entities should be implemented as immutable value objects. Let's take a closer look at why you should prefer an object to be immutable.

By definition, an object can be created and then reused in different places. We can pass it on as a method argument, or even as a constructor argument, or assign the object to a property.

```
$object = new Foo();

// pass along the object
$this->someMethod($object);

// assign the object to a property
$this->someProperty = $object;

// maybe return the object...
return $object;
```

If one call site has a reference to an object, and then another call site changes some aspect of this object, it will be quite a surprise for the first call site. How can it know that the object is still useful? Maybe the initial call site doesn't even know how to deal with the new state of the object anymore.

But even within the same call site, problems related to mutability can occur. Take a look at the following code.

```
final class Appointment
{
    private DateTime $time;

    public function __construct(DateTime $time)
    {
        $this->time = $time;
    }

    public function time(): string
    {
```

```
        return $this->time->format('h:s');
    }

    public function reminderTime(): string
    {
        $oneHourBefore = '-1 hour';

        $reminderTime = $this->time->modify($oneHourBefore);

        return $reminderTime->format('h:s');
    }
}

$appointment = new Appointment(new DateTime('12:00'));

// First, we get the time of the appointment:
$time = $appointment->time(); // returns '12:00'

// Then we get the time for sending a reminder:
$reminderTime = $appointment->reminderTime(); // returns '11:00';

// Finally, we get the time of the appointment again:
$time = $appointment->time(); // also returns '11:00' now!
```

Reading code like this, it may take a lot of time to figure out why, after requesting the time for sending a reminder, the time of the appointment itself has changed as well. To prevent this kind of situation, the general rule is to design every object that is not an entity to be immutable. It'll always be safe to keep a reference to an immutable object.

Replace values instead of modifying them

If you design objects to be immutable, they show a nice similarity with primitive-type values. Consider this example:

```
$i = 1;
$i++;
```

Would you consider 1 to have been changed into 2? No, we should say that the variable `$i` previously contained 1, and now it contains 2. Integers are in fact immutable themselves. We use them, then we discard them. But we can always use them again. Also, passing them around as method arguments, or copying them into object properties, isn't considered dangerous. Every time we need some integer, we basically create a new one from the endless supply of integers. There's no shared place in the computer's memory where we keep one instance of every integer.

The same goes for objects that are implemented as immutable values. It doesn't feel like we share them anymore. And if we need the object to be different, we don't modify the object, we just create a new one. This means that, if an immutable object is inside a variable or property, and we want to change something about it, we create a new object, and store it in our variable or property.

To illustrate, let's say we implemented `Year` as an immutable object, wrapping an integer, and offering a convenient method for returning a new `Year` instance representing the next year:

```
final class Year
{
    private int $year;

    public function __construct(int $year)
    {
        $this->year = $year;
    }

    public function next(): Year
    {
        return new self($this->year + 1);
    }
}

$year = new Year(2018);
```

```
// This has no effect, since next() doesn't actually change `$year`:
$year->next();
assertEquals(new Year(2018), $year);

// Instead, we should capture the return value of `next()`:
$year = $year->next();
assertEquals(new Year(2019), $year);
```

If we keep a `Year` instance in a property of a mutable object and we want it to proceed to the next year, we should not only call `next()`, but we should also store its return value in the property that holds the current `Year` instance:

```
final class Journal
{
    private Year $currentYear;

    public function closeTheFinancialYear(): void
    {
        // ...

        $this->currentYear = $this->currentYear->next();
    }
}
```

3.2 A modifier on an immutable object should return a modified copy

So based on our findings regarding immutability, immutable objects can have methods that could be considered modifiers. But they don't actually modify the state of the object on which we call the method. Instead, such a method returns a copy of the object, but with data that matches the method's intention. So we know that the return type of the method should be the class of that object itself, just like the return type of the `next()` method from the previous example was `Year`. Then there are two basic templates for these methods. The first uses the (potentially private) constructor of the object, to create the desired copy:

```
final class Integer
{
    private int $integer;

    public function __construct(int $integer)
    {
        $this->integer = $integer;
    }

    public function plus(Integer $other): Integer
    {
        return new self($this->integer + $other->integer);
    }
}
```

Since `Integer` already has a constructor which accepts an `int` value, we can do the operation on the existing integers and pass the resulting `int` to the constructor of `Integer`.

The other option, which can sometimes be useful for immutable objects with multiple properties, is to create an actual copy of the object using the `clone` operator, and then make the desired change to it:

```
final class Position
{
    private int $x;
    private int $y;

    public function __construct(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }

    public function withX(int $x): Position
    {
        $copy = clone $this;
```

```
        $copy->x = $x;

        return $copy;
    }
}

$position = new Position(10, 20);

// The next position will be 4 steps to the left, i.e. (6, 20);
$nextPosition = $position->withX(6);
assertEquals(new Position(6, 20), $nextPosition);
```

In the previous example, `withX()` pretty much resembles a traditional setter method, allowing a client to provide a new value for a single property. There are usually better options. Make sure to look for ways to make modifier methods a bit smarter, or at least give them a name that's not technical, but more domain-oriented. You may find useful clues about how to accomplish that by looking at how clients use these methods. For example, here's a client of our `withX()` method:

```
// move 4 steps to the left
$nextPosition = $position->withX($position->x() - 4);
```

Because `Position` only has a modifier method for setting a new value for `x`, this client has to make its own calculations to determine which value it has to provide. But the client isn't really looking for a way to modify `x`, it's looking for a way to find out what the next position will be, if it takes 4 steps to the left.

Instead of letting the client make the calculations, you can let the `Position` object do it. You only need to offer a more convenient modifier method, e.g. `toTheLeft()`:

```
final class Position
{
    // ...

    public function toTheLeft(int $steps): Position
    {
        $copy = clone $this;

        $copy->x = $copy->x - $steps;

        return $copy;
    }
}

$position = new Position(10, 20);

// The next position will be (6, 20);
$nextPosition = $position->toTheLeft(4);
assertEquals(new Position(6, 20), $nextPosition);

// The original object should not have been modified
assertEquals(new Position(10, 20), $position);
```

3.3 On a mutable object, modifier methods should be command methods

Even though almost all of your objects should be immutable, there usually are some objects that are not, namely entities. As we saw at the beginning of this chapter, an entity has methods that allow it to be manipulated.

Let's take a look at another example, the `Player` class, which has a current position, encoded as values for X and Y. It's a mutable object: it has a `moveLeft()` method, which updates (actually: replaces) the player's position. The `Position` object is an immutable object, but the `Player` object itself is mutable:

```
final class Player
{
    private Position $position;

    public function __construct(Position $initialPosition)
    {
        $this->position = $initialPosition;
    }

    public function moveLeft(int $steps): void
    {
        $this->position = $this->position->toTheLeft($steps);
    }

    public function currentPosition(): Position
    {
        return $this->position;
    }
}
```

We can recognize mutability by the assignment operator in `moveLeft()`: the `$position` property gets a new value if you call this method. Another sign is the `void` return type. These two characteristics are in fact the trademarks of a so-called *Command method*.

Methods that change the state of an object should always be command methods like this. They have a name in the imperative form, are allowed to make a change to the object's internal data structures, and they don't return anything.

3.4 On an immutable object, modifier methods should have declarative names

Modifier methods on mutable objects are expected to change the state of the object, which nicely matches the traditional characteristics of a command method. For modifier methods of immutable objects we need another convention.

Imagine having the same implementation of `Position` that we saw earlier, but this time `toTheLeft()` was called `moveLeft()`:

```
final class Position
{
    // ...

    public function moveLeft(int $steps): Position
    {
        // ...
    }
}
```

Given the rule that modifier methods on mutable objects are command methods, this `moveLeft()` is confusing: it has an imperative name (`moveLeft()`), but it doesn't have a `void` return type. Without looking at the implementation, the reader will be unsure whether or not calling this method will change the state of the object.

To find a good name for modifier methods on immutable objects, you could fill in the following template: "I want this …, but …". In the case of `Position`, this becomes "I want this position, but `n` steps to the left", so `toTheLeft()` seems to be a suitable method name.

```
final class Position
{
    // ...

    public function toTheLeft(int $steps): Position
    {
        // ...
    }
}
```

Following this template you may often end up using the word "with", or using so-called participle adjectives in the past tense. For instance: "I want this quantity, but multiplied `n` times". Or: "I want this response, but with a `Content-Type: text/html` header" These are declarative names: they don't tell you what to do, but they "declare" the result of the manipulation.

When looking for good names, also aim for domain-specific, higher-level names instead of generic names from the underlying technical domain. For example, we

chose `toTheLeft()` instead of `withXDecreasedBy()`, which has a different level of abstraction.

3.5 Compare whole objects

With mutable objects you may write tests like the following:

```
$position = new Position(10, 20);
$position->moveLeft(4);
assertSame(6, $position->x());
```

As mentioned earlier, this kind of testing usually forces additional getters to be added to the class. These getters are only needed for writing the tests; no other client might be interested in them.

With immutable objects you can often resort to a different kind of assertion, one that allows the object to keep its internal data and implementation details on the inside:

```
$position = new Position(10, 20);
$nextPosition = $position->toTheLeft(4);
assertEquals(new Position(6, 20), $nextPosition);
```

`assertEquals()` will use a recursive method that compares equality of the properties of both objects, and those of the objects it keeps inside those properties, and so on. Using `assertEquals()` therefore prevents value objects from having some hidden aspect that would make two objects incomparable.

3.6 When comparing immutable objects, assert equality, not sameness

The following example shows how the `Position` class from the previous example can be used in a (mutable) `Player` class:

```
final class Player
{
    private Position $position;

    public function __construct(Position $initialPosition)
    {
        $this->position = $initialPosition;
    }

    public function moveLeft(int $steps): void
    {
        $this->position = $this->position->toTheLeft($steps);
    }

    public function currentPosition(): Position
    {
        return $this->position;
    }
}
```

A test might look like this:

```
function the_player_starts_at_a_position_and_can_move_left()
{
    $initialPosition = new Position(10, 20);
    $player = new Player($initialPosition);

    /*
     * We can get away with using `assertSame()` here - the
     * `Position` object is still the _same_ object we injected:
     */
    assertSame($initialPosition, $player->currentPosition());

    $player->moveLeft(4);

    /*
```

```
        * Here we have to use `assertEquals()`:
        */
        assertEquals(new Position(6, 20), $player->currentPosition());
}
```

When comparing immutable objects, tests shouldn't make a point of them having the same reference in memory. In fact, all that matters is the thing they represent. Again, just like we don't compare memory locations of two integers. We just say: are their values equal? So you should always use `assertEquals()` when comparing objects.

By the way, sometimes you may want to compare two objects not in a test, but in production code. In that case, you can't use `assertEquals()` of course. You could use the non-strict comparison operator `==`, but because it compares property values using `==` too[1], this may give unexpected results (because, for instance `null` would be equal to `0`, `''` and even `[]`). Using `==` in your code also triggers warnings against non-strictness in static analysis tools (for the same reason). So instead, you could implement a method on the object that can be used to compare two instances, using strict value comparison:

```
final class Position
{
    // ...

    public function equals(Position $other)
    {
        return $this->x === $other->x && $this->y === $other->y;
    }
}
```

However, most value objects really don't need an `equals()` method, and you definitely shouldn't add one to every immutable object without thinking about it. The rule for getters applies for the `equals()` method too: only add this method if some other client than a test uses it.

Also, don't make `equals()` generic by typing `$other` as `object`. You should make sure in other ways that the client of `equals()` doesn't try to compare a `Position` object to anything other than a `Position` object.

[1] http://php.net/manual/en/language.oop5.object-comparison.php

3.7 Calling a modifier method should always result in a valid object

When we talked about creating objects earlier, we discussed concepts like meaningful data and domain invariants. The same concepts can be applied to modifier methods. In fact, not only for modifier methods on immutable objects. The rules also apply to mutable objects.

A modifier method has to make sure that the client provides meaningful data, it has to protect domain invariants, and it does so in the same way as constructors do: by making assertions about the arguments that have been provided. It can thereby prevent the object from ending up in an invalid state:

```
final class TotalDistanceTraveled
{
    private int $totalDistance = 0;

    public function add(int $distance): TotalDistanceTraveled
    {
        Assertion::greaterOrEqualThan(
            $distance,
            0,
            'You cannot add a negative distance'
        );

        $copy = clone $this;
        $copy->totalDistance += $distance;

        return $copy;
    }
}

$totalDistanceTravelled = new TotalDistanceTraveled();
expectException(
    InvalidArgumentException::class,
    'distance',
```

```
    function () use ($totalDistanceTravelled) {
        $totalDistanceTravelled->add(-10);
    }
);
```

If the modifier method doesn't `clone`, but reuses the original constructor of the class, you can often reuse the validation logic that's already available. In fact, this can be good reason not to use `clone`, but always go through the constructor.

As an example, take a look at the `Fraction` class, which represents a fraction (e.g. `1/3`, `2/5`). The structure of a fraction is `[numerator]/[denominator]`. Both can be any whole number, but the denominator can never be `0`. The constructor enforces this rule already. So the modifier method `withDenominator()` only needs to forward the call to the constructor, and the rule will be verified for the input of `withDenominator()` too:

```
final class Fraction
{
    private int $numerator;
    private int $denominator;

    public function __construct(int $numerator, int $denominator)
    {
        Assertion::notEq(
            $denominator,
            0,
            'The denominator of a fraction cannot be 0'
        );

        $this->numerator = $numerator;
        $this->denominator = $denominator;
    }

    public function withDenominator($newDenominator): Fraction
    {
        /*
         * Forwarding the call to the constructor will also trigger
```

```
         * any of its assertion errors.
         */

        return new self($this->numerator, $newDenominator);
    }
}

$fraction = new Fraction(1, 2);

expectException(
    InvalidArgumentException::class,
    'denominator',
    function () use ($fraction) {
        $fraction->withDenominator(0);
    }
);
```

3.8 A modifier method should verify that the requested state change is valid

In particular on mutable objects like entities, calling a modifier method often means that the object will make some sort of state transition. It's not just about updating properties, but the change unlocks new possibilities, or blocks options that were previously available. Like the `SalesOrder` class below; once it has been marked as "delivered", it will be impossible to cancel it, since that state transition wouldn't make sense from a business perspective.

```
final class SalesOrder
{
    // ...

    public function markAsDeliverable(): void
    {
        // ...
    }

    public function markAsDelivered(Timestamp $deliveredAt): void
    {
        /*
         * You can only deliver the order if it has been "marked as
         * deliverable" earlier.
         */
    }

    public function cancel(Timestamp $cancelledAt): void
    {
        /*
         * You can't cancel an order if it has already been
         * delivered.
         */
    }

    // and so on...
}
```

Make sure that every one of these methods prevents against making an invalid state transition. You should verify this with unit tests:

```
public function a_delivered_sales_order_can_not_be_cancelled()
{
    $deliveredSalesOrder = /* ... */;
    $deliveredSalesOrder->markAsDelivered(...);

    expectException(
        LogicException::class,
        'delivered',
        function () use ($deliveredSalesOrder) {
            $deliveredSalesOrder->cancel();
        }
    );
}
```

An appropriate exception to throw here would be a `LogicException`, but you may also introduce your own exception type, like `CanNotCancelDeliveredOrder`.

If a client calls the same method twice, it requires a bit of contemplation. You could throw an exception, but in most cases it's not a big deal and you can just ignore the call.

```
public function cancel()
{
    if ($this->status->equals(Status::cancelled())) {
        // just ignore the request
        return;
    }

    // ...
}
```

3.9 Use internally recorded events to verify changes on mutable objects

We've already seen how trying to test constructors leads to adding more getters to an object than needed, only to test that what comes in, can also go out again. This isn't at

all the idea of an object, which is to hide information and implementation details. The same goes for testing modifier methods. When testing the `moveLeft()` method of the mutable `Player` object we discussed earlier, there are two options. The first option is to use a getter to verify that the current position after moving left is the position we expect it to be.

```
$player = new Player(new Position(10, 20));
$player->moveLeft(4);

assertEquals(new Position(6, 20), $player->currentPosition());
```

The other, more blunt option is to verify that the whole object is now what we expect it to be.

```
$player = new Player(new Position(10, 20));
$player->moveLeft(4);

assertEquals(new Player(new Position(6, 20)), $player);
```

It isn't all that bad of a solution, because at least we don't need that getter anymore to retrieve the current position. The main issue with this test is that it covers too much ground, and we can't easily add new behavior to the `Player` object without having to modify this test too (in particular if extra constructor arguments are added over time).

Another option could be to change `moveLeft()` a bit and make it return the new position:

```
final class Player
{
    public function moveLeft(): Position
    {
        $this->position = $this->position->toTheLeft($steps);

        return $this->position;
    }
}
```

```
$player = new Player(new Position(10, 20));
$currentPosition = $player->moveLeft(4);

assertEquals(new Position(6, 20), $currentPosition);
```

This looks clever, but is a violation of the rule that a modifier method on a mutable object should be a command method, and thus have a `void` return type. But on top of that, this test doesn't really prove that the `Player` has actually moved to the expected position. Consider for example this implementation of `moveLeft()`, for which the same test would pass as well:

```
public function moveLeft(): Position
{
    return $this->position->toTheLeft($steps);
}
```

A better way to test for changes in a mutable object is to record events inside the object, which can later be inspected These events will act like a log of the changes that happened to the object. Events are simple value objects and you can create as many of them as needed. This is the `Player` class rewritten to record `PlayerMoved` events, and expose them through its `recordedEvents()` method:

```
final class Player
{
    private $events = [];

    public function __construct(Position $initialPosition)
    {
        $this->position = $initialPosition;
    }

    public function moveLeft(int $steps): void
    {
        $nextPosition = $this->position->toTheLeft($steps);
```

```
        $this->position = $nextPosition;

        /*
         * After moving to the left, we record an event that can
         * later be used to find out what has happened inside the
         * `Player` object.
         */
        $this->events[] = new PlayerMoved($nextPosition);
    }

    public function recordedEvents(): array
    {
        return $this->events;
    }
}

// Create a new `Player` object and set an initial position for it:
$player = new Player(new Position(10, 20));

// Move it 4 steps to the left:
$player->moveLeft(4);

/*
 * We verify that the player has moved, by comparing its recorded events
 * to an expected list of events:
 */
assertEquals(
    [
        new PlayerMoved(new Position(6, 20))
    ],
    $player->recordedEvents()
);
```

You can do interesting things, like only recording events if something has actually changed. For instance, maybe you allow the player to take 0 steps. If that happens, the player hasn't really moved, or at least: the call to `moveLeft()` wouldn't really deserve

an event to be created for it:

```
public function moveLeft(int $steps): void
{
    if ($steps === 0) {
        // don't thrown an exception, but also don't record an event
        return;
    }

    $nextPosition = $this->position->toTheLeft($steps);

    $this->position = $nextPosition;

    $this->events[] = new PlayerMoved($nextPosition);
}
```

After a while, `assertEquals([...], $player->recordedEvents())` may prove not to be flexible enough to allow the implementation of the `Player` object to be changed without making existing tests fail. For example, let's see what happens if we record another event to represent the moment the player took its initial position:

```
final class PlayerTookInitialPosition
{
    // ...
}

final class Player
{
    private $events;

    public function __construct(Position $initialPosition)
    {
        $this->position = $initialPosition;

        $this->events[] = new PlayerTookInitialPosition(
            $initalPosition
```

```
        );
    }
}
```

This will break the existing test we had for moving to the left:

```
$player = new Player(new Position(10, 20));
$player->moveLeft(4);

/*
 * This assertion will fail, because the constructor now records a
 * `PlayerTookInitialPosition` event, which will also be returned by
 * `recordedEvents()`:
 */
assertEquals(
    [
        new PlayerMoved(new Position(6, 20))
    ],
    $player->recordedEvents()
);
```

One thing we could do to make this test less brittle, is to assert that the list of recorded events *contains* the expected event:

```
$player = new Player(new Position(10, 20));
$player->moveLeft(4);

assertContains(
    new PlayerMoved(new Position(6, 20)),
    $player->recordedEvents()
);
```

Isn't it a bit too pushy to introduce events in every mutable object?

As mentioned at the beginning of this chapter, almost all objects will be immutable. Those few objects that are mutable will be entities. These are objects for which it's already useful to have events (they are called "domain events" then). So in practice, adding support for recording events isn't too much to ask, it's a very natural thing to happen. I'm sure it will prove to be very useful anyway, because events are a way to respond to changes in domain objects. One type of response could be to make even more changes, or, for instance, to use event data to populate search engines, build up read models, or collect useful business insights on-the-fly.

Since the only information the `Player` exposes to its clients is a list of internal domain events, there isn't an easy way to find out the current position of the player. In practice, that's probably not very useful; we do need this information, if only to show the current position of the player on the screen. We'll get back to the topic of retrieving information from objects in a later chapter.

3.10 Don't implement fluent interfaces on mutable objects

An object has a fluent interface when its modifier methods return `$this`. If an object has a fluent interface, you can call method after method on it, without repeating the variable name of the object:

```
$queryBuilder = QueryBuilder::create()
    ->select(...)
    ->from(...)
    ->where(...)
    ->orderBy(...);
```

However, a fluent interface can be very confusing in regards to the question on which object a method gets called on. Is `QueryBuilder` immutable, then it doesn't really matter. But who knows if it's mutable? If you look at the method signatures of `QueryBuilder`, there's no way to find that out:

```
final class QueryBuilder
{
    /*
     * Do these methods update the state of the object they're called
     * on, or do they return a modified copy? Or... both?
     */

    public function select(...): QueryBuilder
    {
        // ...
    }

    public function from(...): QueryBuilder
    {
        // ...
    }

    // ...
}
```

Given that these method signatures look a lot like modifiers on immutable objects, we might assume that `QueryBuilder` is immutable. So we may also assume that we can safely reuse any intermediate stage of the `QueryBuilder` object.

```
$queryBuilder = QueryBuilder::create();

$qb1 = $queryBuilder
    ->select(...)
    ->from(...)
    ->where(...)
    ->orderBy(...);

$qb2 = $queryBuilder
   ->select(...)
   ->from(...)
```

```
    ->where(...)
    ->orderBy(...);
```

But then it turns out `QueryBuilder` isn't immutable after all, as you can see by looking at the implementation of `where()`:

```
public function where(string $clause, $value): QueryBuilder
{
    $this->whereParts[] = $clause;
    $this->values[] = $value;

    return $this;
}
```

This method looks like a modifier of an immutable object, but is in fact a regular command method, which, as a very confusing bonus, returns the current object instance after modifying it.

To avoid this confusion, don't give your mutable objects fluent interfaces. In the case of the `QueryBuilder`, it would be better off as an immutable object anyway. This would not leave its clients with an object in an unknown state:

```
public function where(string $clause, $value): QueryBuilder
{
    $copy = clone $this;

    $copy->whereParts[] = $clause;
    $copy->values[] = $value;

    return $copy;
}
```

For immutable objects, it's not a problem at all to have a fluent interface. In fact, you could say that using modifier methods as they are described in this chapter, gives you a fluent interface by definition, because every modifier will return a modified copy of itself.

```
$position = Position::startAt(10, 5)
    ->toTheLeft(4)
    ->toTheRight(2);
```

A third-party library has some object design issues, what do I do?

The `QueryBuilder` example in this section was inspired by the actual `QueryBuilder` class[2] from the Doctrine DBAL library. This is just one example of a class that doesn't follow all the rules in this book. You're likely to encounter other classes that don't (in third-party code, and in the project code itself). What to do now?

There are different trade-offs to be made. For example, do you use the "badly" designed class only inside your methods, or do instances of it get passed around between methods or even objects? In the case of the `QueryBuilder`, most likely it will only be used inside repository methods. This means it can't escape and be used in other parts of your application. This mitigates the "design risk" of using it in your project. So even if `QueryBuilder` has some design issues, there really is no need to rewrite it or work around it.

There may be other cases where an object is very confusing (is it immutable, or mutable?). A nice example is PHP's built-in `DateTime` object. An immutable alternative has been introduced for it (called `DateTimeImmutable`), but before it existed, it was already a good idea to introduce your own immutable wrapper object for it. This would make sure that the wrapped mutable `DateTime` object would never escape and cause strange state-related issues in your application.

3.11 Summary

Always prefer immutable objects, which can't be modified after they have been created. If you want to allow something to be changed about them, first make a copy and then make the change. Give these methods declarative names and use the opportunity to implement some useful behavior instead of simply allowing properties to be changed

[2] https://github.com/doctrine/dbal/blob/master/lib/Doctrine/DBAL/Query/QueryBuilder.php

to new values. Make sure that after a modifier method has been called, the object is in a valid state. To do this, accept only correct data, and make sure the object doesn't make an invalid state transition.

On mutable objects like entities, modifier methods should have a `void` return type. The changes that occur in such objects can be exposed by analyzing internally recorded events. As opposed to immutable objects, mutable objects shouldn't have a fluent interface.

Using objects

Having instantiated an object, you're ready to use it. Objects can offer useful behaviors: they can give you information, and they can perform tasks for you. Either way, these behaviors will be implemented as object methods. Before we discuss the design rules that are specific for either retrieving information or performing tasks, we'll first discuss something these methods should have in common: a template for their implementation.

4. A template for implementing methods

Whenever you design a method, you should remember this template:

```
[scope] function methodName(type name, ...): void|[return-type]
{
    [pre-conditions checks]

    [failure scenarios]

    [happy path]

    [post-condition checks]

    [return void|specific-return-type]
}
```

4.1 Pre-condition checks

The first step will be to verify that the arguments provided by the client are correct and can be used to fulfill the task at hand. Make any number of checks, and throw exceptions when anything looks off.

Pre-condition checks have the following shape:

```
if ([/* some pre-condition wasn't met */) {
    throw new InvalidArgumentException(...);
}
```

As discussed earlier, you can often use standard assertion functions for this kind of checks, like:

```
Assertion::inArray($value, ['allowed', 'values']);
```

Some of these pre-condition checks may only be needed because the type system of the language is lacking some features. For example, PHP has an `array` type, but no way to tell the engine that you only want to accept an array consisting solely of objects of a certain type. So for this, you'll need to add an assertion:

```
Assertion::allIsInstanceOf($value, EventListener::class);
```

Other checks will inspect the contents of an argument and warn the client that, for instance, they provided a value in the wrong range:

```
Assertion::greaterThan($value, 0);
```

Introduce new types to get rid of pre-condition checks

Most of these assertions will be made to validate primitive-type arguments (`int`, `string`, etc.). As we discussed earlier, it makes sense to introduce a new type (i.e. class) and move the assertions you're doing inside a regular method, to that new type's constructor:

```
// before:
public function sendConfirmationEmail(string $emailAddress): void
{
    Assertion::email($emailAddress);
    // ...
}

// after:
final class EmailAddress
{
    private string $emailAddress;

    public function __construct(string $emailAddress)
    {
        $this->emailAddress = $emailAddress;
    }
}

public function sendConfirmationEmail(
    EmailAddress $emailAddress
): void {
    // no need to validate $emailAddress anymore
}
```

This is a refactoring[1] known as "Replace primitive with object".

If no assertion fails, this means we accept all the input arguments as they are. These pre-condition checks are still superficial though, because they only inspect the values for obvious issues.

[1]Kent Beck, Martin Fowler, "Refactoring: Improving the Design of Existing Code, Second Edition", Addison-Wesley Professional, 2018.

4.2 Failure scenarios

Even if the values "look" right and therefore pass the pre-condition checks, things can still go wrong. For example, even though an email address looks valid, sending an email to it might still fail. Or, even though the client provides a positive integer, there may not be a record in the database with that ID. This means that, while running the remaining code of the method, things could still go wrong.

If something goes wrong in the method after the pre-condition checks, you should throw a different kind of exception. It won't be an exception indicating an "invalid argument". The type of the exception should indicate that an error condition occurred, which could only be detected at runtime. It's not the method itself that fails, it's some external condition that breaks the method. As an example:

```
public function getRowById(int $id): array
{
    /*
     * This could throw an InvalidArgumentException:
     */
    Assertion::greaterThan($id, 0, 'ID should be greater than 0');

    /*
     * This could cause either an InvalidArgumentException or a
     * RuntimeException to be thrown from the code that calls the
     * database:
     */
    $record = $this->db->find($id);

    /*
     * This is _our_ failure scenario: we couldn't find the record,
     * so we throw a `RuntimeException`.
     */
    if ($record === null) {
        throw new RuntimeException(sprintf(
            'Could not find record with ID "%d"',
            $id
```

```
            );
        }

        return $record;
    }
```

Down the stream, every method that gets called may have pre-condition checks too, so besides `RuntimeExceptions` originating from the vendor code that calls the database, we may also run into an `InvalidArgumentException` (or its parent: `LogicException`). Usually we just let these exceptions "bubble up"—some higher-level application error-handling mechanism should be able to deal with these errors. What matters in this part of your method is the scenarios which the method *itself* can recognize as a failure scenario.

4.3 Happy path

The happy path, or happy part of a method is where nothing is wrong, and the method is just performing its task. If you keep your methods small, like you should, you may find that often there isn't really much going on in this part. Sometimes it might even be the case that most of the code is there for dealing with failure scenarios.

4.4 Post-condition checks

Post-condition checks can be added to a method to verify that the method did what it was supposed to do. You could analyze the return value before actually returning it, or you could analyze the state of the object just before jumping out of it.

```
public function someVeryComplicatedCalculation(): int
{
    // ...
    $result = ...;

    /*
     * This post-condition check is just a safety check, a.k.a.
     * "This should never happen".
     */
    Assertion::greaterThan(0, $result);

    return $result;
}
```

In practice, most methods don't need post-condition checks. If you write tests for your methods, you already *know* that they are returning the right values, or that they are changing the object's state in the right way.

If you have strong types in your code base, and therefore don't often use primitive-type values anymore, you will find that using these types to define method parameters and return types, results in solid code that can't return something that's invalid. After all, if the return value is an object, we know that it can't exist in an invalid state.

If you're dealing with "legacy code", with lots of implicit type casting and no assertions whatsoever, you may find adding post-conditions a useful technique. They can then be used as safety checks, to make sure that there won't be any problems downstream.

Introduce new methods to get rid of post-condition checks

You could get rid of a method's post-condition checks, just like you can remove pre-condition checks, by promoting primitive-type values to proper objects and return those from your method. Another option is to wrap the method with post-condition checks in a new method, which performs these checks.

4.5 Return value

Finally, a method may return something. In fact, only query methods should do that. We'll discuss this topic in detail in the next chapter. Another good rule to keep in mind is to "return early". We've encountered this rule for exceptions already: as soon as you know something is going wrong, throw an exception about it. But the same applies to return values. As soon as you know what you will return, return it, instead of keeping the value around, skipping a few more `if` clauses and then return it.

4.6 Some rules for exceptions

Use custom exception classes only if needed

Adding a custom exception class can be very helpful, if:

1. You want to catch a specific exception type higher up.

```
try {
    // possibly throws SomeSpecific exception
} catch (SomeSpecific $exception) {
    // ...
}
```

2. If there are multiple ways to instantiate them.

```
final class CouldNotDeliverOrder extends RuntimeException
{
    public static function itWasAlreadyDelivered():
        CouldNotDeliverOrder
    {
        // ...
    }

    public static function insufficientQuantitiesInStock():
        CouldNotDeliverOrder
```

```
    {
        // ...
    }
}
```

3. If you want to use a named constructor to make it easier for the client to instantiate one.

```
final class CouldNotFindProduct extends RuntimeException
{
    public static function withId(
        ProductId $productId
    ): CouldNotFindProduct {
        return new self(sprintf(
            'Could not find a product with ID "%s"',
            $productId
        ));
    }
}

throw CouldNotFindProduct::forId(...);
```

This makes the code on the client side much cleaner. The name of the exception class combined with the name of the constructor method reads like a sentence: "Could not find product for ID ...". The message gets assembled inside the exception class instead of at the call site.

Having a custom exception class with a named constructor like this gives you the option to add more than one named constructor, making it easier to reuse the same exception class to point out different reasons for failure.

```
final class CouldNotPersistObject extends RuntimeException
{
    public static function becauseDatabaseIsNotAvailable():
        CouldNotPersistObject
    {
        return new self(...);
    }

    public static function becauseMappingConfigurationIsInvalid():
        CouldNotPersistObject
    {
        return new self(...);
    }

    // ...
}
```

Naming invalid argument or logic exception classes

Contrary to popular belief, exception class names don't need to have "Exception" in it. However, there are some naming helper sentences you could use. To indicate invalid arguments or logic errors, you could use the template "Invalid ...", e.g. `InvalidEmailAddress`, `InvalidTargetPosition`, `InvalidStateTransition`.

Naming runtime exception classes

For runtime exceptions, a very helpful rule is to finish the sentence: "Sorry, [I] ...". The words on the dots will be the name of your exception class. These will be good names, because they communicate how the system tried to perform the requested job, but couldn't finish it successfully. For example: `CouldNotFindProduct`, `CouldNotStoreFile`, `CouldNotConnect`, etc.

Use named constructors to indicate reasons for failure

If you use named constructors, you can use the name to indicate the ingredients needed to instantiate the exception, for example:

```
final class CouldNotFindStreetName extends RuntimeException
{
    public static function withPostalCode(
        PostalCode $postalCode
    ): CouldNotFindStreetName {
        // ...
    }
}
```

In other cases you may be able to use the method name to indicate the reason why something is wrong. For example:

```
final class InvalidTargetPosition extends LogicException
{
    public static function becauseItIsOutsideTheMap(
        ...
    ): InvalidTargetPosition {
        // ...
    }
}
```

Add detailed messages

Providing named constructors will be useful for clients, because the constructor of the exception, not the client itself, will set up the exception's message.

```
// Before:
final class CouldNotFindProduct extends RuntimeException
{
}

// At the call site:
throw new CouldNotFindProduct(sprintf(
    'Could not find a product with ID "%s"',
    $productId
));

// After:
final class CouldNotFindProduct extends RuntimeException
{
    public static function withId(
        ProductId $productId
    ): CouldNotFindProduct {
        return new self(sprintf(
            'Could not find a product with ID "%s"',
            $productId
        ));
    }
}

// At the call site:
throw CouldNotFindProduct::withId($productId);
```

4.7 Summary

The template for implementing methods aims at clearing the table before starting the work. You start with analyzing the provided arguments and rejecting anything that looks wrong by throwing an exception. Then you do the actual work, and deal with any failures. Finally, you wrap up, after which you may return a value to the client.

5. Retrieving information

Besides the ability to instantiate or modify an object, it may offer two other types of possibilities: you may use the object to perform a certain task, or to retrieve a piece of information from it.

5.1 Use query methods for information retrieval

Earlier, we briefly discussed command methods. These methods have a `void` return type and can be used to produce a side-effect; change state, send an email, store a file, etc. Such a method shouldn't be used for retrieving information. If you want to retrieve some information from an object, you should use a query method for it. Such a method does have a specific return type, and it's not allowed to produce any side-effects.

Take a look at the following class:

```
final class Counter
{
    private int $count = 0;

    public function increment(): void
    {
        $this->count++;
    }

    public function currentCount(): int
    {
        return $this->count;
    }
}

$counter = new Counter();
```

```
$counter->increment();

assertEquals(1, $counter->currentCount());
```

According to the rules for command and query methods, it's clear how `increment()` is a command method, because it changes the state of a `Counter` object, and `currentCount()` is a query method, because it doesn't change anything; it just returns the current value of `$count`. The good thing about this separation is that given the current state of a `Counter` object, calling `currentCount()` will always return the same answer.

Consider the following alternative implementation of `increment()`:

```
public function increment(): int
{
    $this->count++;

    return $this->count;
}
```

This method makes a change and returns information. This is confusing from a client perspective; the object changes while you just want to take a look at it.

It's better to have safe methods that can be called any time (and in fact, can be called any number of times), and other methods that are "unsafe" to call.

There are two ways to achieve this: the first thing you can do is follow the rule that a method should always be either a command or a query method. This rule is called the *Command/Query Separation principle*[1]. We've already applied it in the initial implementation of `Counter` as we saw it earlier: `increment()` was a command method, `currentCount()` a query method, and none of the methods of `Counter` were both command and query methods at the same time.

Something else you can do is make your objects immutable (as has been previously advised for almost all objects in your application).

If `Counter` would be implemented as an immutable object, `increment()` would become a modifier method, and a better, more declarative name for it would be `incremented()`:

[1]Martin Fowler, "CommandQuerySeparation" (2005), https://martinfowler.com/bliki/CommandQuerySeparation.html

```
final class Counter
{
    private int $count = 0;

    public function incremented(): Counter
    {
        $copy = clone $this;

        $copy->count++;

        return $copy;
    }

    public function currentCount(): int
    {
        return $this->count;
    }
}

assertEquals(
    1,
    (new Counter())->incremented()->currentCount()
);
assertEquals(
    2,
    (new Counter())->incremented()->incremented()->currentCount()
);
```

Is a modifier method a command or a query method?

A modifier method doesn't really return the information you're after. In fact, it returns a copy of the whole object. Once you have that copy, you can ask it questions. So modifiers don't seem to be query methods. But they aren't traditional command methods either. A command method on an immutable object would imply that it changes its state after all, which isn't the case. It only produces a new object, which isn't far from just answering a query. Although it's stretching the concept a bit, the `incremented()` method that was suggested earlier, could answer the query "give me the current count, but incremented by 1".

5.2 Query methods should have single-type return values

When a method returns a piece of information, it should return a predictable thing. So, no mixed types are allowed. Some languages don't even support it, but PHP being a dynamically typed language does. The following method will be very confusing for its users:

```
/**
 * @return string|bool
 */
public function isValid(string $emailAddress)
{
    if (/* ... */) {
        return 'Invalid email address';
    }

    return true;
}
```

If the provided email address is valid, it will return true, otherwise it will return a string. This makes it hard to use the method. Instead, make sure always use return values of a single type.

There's another situation to discuss here. Take a look at the following example of a method that doesn't have multiple return types (its single return type is `Page` object), but may alternately return `null`:

```
public function findOneBy($type): ?Page
{
}
```

This also puts a burden on the caller: they will always have to check whether the returned value is actually a `Page` object, or if it's `null` and they need to deal with that.

```
if ($page instanceof Page) {
    // `$page` is a `Page` object and can be used as such
} else {
    // `$page` is `null` and we have to decide what to do with it
}
```

Returning `null` from a method isn't always a problem. But you have to make sure that all the clients of the method will deal with this situation. Static analysis tools like PHPStan[2] can verify this, and your IDE may also help you with it, telling you when you are risking a possible "null pointer exception".

In most cases though, it pays to consider alternatives for returning `null`. For example, the following method that is supposed to retrieve a `User` entity by its ID, shouldn't return `null` if it can't find the user. It should throw an exception. After all, a client expects that `User` to exist; it's even providing the user's ID. So it won't take `null` for an answer.

```
public function getById($id): User
{
    $user = /* ... */;

    if (!$user instanceof User) {
        throw UserNotFound::withId($id);
    }

    return $user;
}
```

[2] https://github.com/phpstan/phpstan

Another alternative would be to return an object which can represent the "null" case. Such an object is called a *Null object*. Clients won't have to check for `null`, since the object has the correct type, as defined in the method signature. For example:

```
public function findOneByType(PageType $type): Page
{
    // Try to find the Page:
    $page = ...;

    if (!$page instanceof Page) {
        return new EmptyPage();
    }

    return $page;
}
```

Show the uncertainty in the name of the method

You can let a method name by an indicator of the uncertainty about whether or not the method will return a value of the expected type. In the previous examples, we used `getById()` instead of `findById()`, to communicate to the client that the method will "get" the `User`, instead of trying to find it and possibly returning empty-handed.

One last alternative for returning `null` is to return a result of the same type, but representing the empty case. If the method is expected to find and return a number of things in an array, return an empty array if you couldn't find anything:

```
public function eventListenersForEvent(string $eventName): array
{
    if (!isset($this->listeners[$eventName])) {
        /*
         * Instead of return `null`, return an empty list
         */
        return [];
    }

    return $this->listeners[$eventName];
}
```

Other return types will have different "empty" cases. For instance, if the return type is `int`, the empty return value might be `0` (or maybe `1`), for strings it might be `''` (or maybe `'N/A'`).

If you find yourself using an existing method which mixes return types, or returns `null` when it should return something more reliable, it might be a good idea to write a new method somewhere that sets things straight. In the following example, the existing `findOneByType()` method returns a `Page` object, or `null`. If we want to make sure that clients don't have to deal with the `null` case and will actually get a `Page` object, we could wrap a call to `findOneByType()` in a new method called `getOneBy`:

```
public function getOneByType(PageType $type): Page
{
    $page = $this->findOneByType($type);

    if (!$page instanceof Page) {
        // Don't return `null`, throw an exception instead:
        throw PageNotFound::withType($type);
    }

    return $page;
}
```

5.3 Avoid query methods that expose internal state

The simplest implementation for query methods is usually just to return a property of the object. These methods are known as “getters”, and they allow clients to “get” the object’s internal data.

For clients, usually the reason to get that data is to use it for further calculations, or to make a decision based on it. Since objects are better off keeping their internals to themselves, you should keep an eye on these simple getters and how their return values are used by clients. The things a client does with the information that an object provides, can often be done by the object itself too.

A first example is the `getItems()` method below, which returns the items in a shopping basket, just so the client can count them. Instead of directly exposing the items, the basket could just provide a method that counts the items for the client:

```
final class ShoppingBasket
{
    // ...

    // This method can be removed...
    public function getItems(): array
    {
        return $this->items;
    }

    // ... the moment we provide a simple count function:
    public function itemCount(): int
    {
        return count($this->items);
    }
}

// So instead of
count($basket->getItems());
```

```
// we can now do:
$basket->itemCount();
```

The naming of query methods is important too. We didn't use `getItemCount()` or `countItems()`, because these method names sound like a command, telling the object to do something. Instead, we named the method `itemCount()`, which makes it look like the item count is just some aspect of a shopping basket you can find out about.

How do you handle ambiguous naming?

What if your object has a property `$name`? The getter for this property would be called `name()`. But "name" can also be a verb. In fact, we already encountered the same potential confusion when we used the word "count", which can be a verb, too.

Although the intended meaning of words will always be up for debate, most of the ambiguity can be resolved by setting the right context. Once you establish a clear difference between query and command methods, it'll be easy to notice when a method is meant to return a piece of information (e.g. return the value of the `$name` property), or is meant to change the state of the object (e.g. change the value of the `$name` property, and return nothing). This will the reader with an important clue as to whether the word should be interpreted as a verb (which is most often the case with a command method) or as a noun (which is often the sign of query method).

Using small rewrites like this, you can make an object absorb more and more logic, thereby keeping the knowledge about the concept it represents inside of it, instead of having little bits of this knowledge all over the code base.

Another example; clients call a query method on an object, then call another method, based on the return value of that first call.

```
final class Product
{
    /*
     * A product has a setting that defines if a
     * discount percentage should be applied to it.
     * If there's not discount percentage, there
     * can still be a fixed discount.
     */

    public function shouldDiscountPercentageBeApplied(): bool
    {
        // ...
    }

    public function discountPercentage(): Percentage
    {
        // ...
    }

    public function fixedDiscountAmount(): Money
    {

    }
}

/*
 * Clients of `Product` can calculate a net amount
 * by calling `applyDiscountPercentage()` first, and
 * using its answer to either apply a discount percentage
 * or a fixed discount:
 */
$amount = new Money(/* ... */);
if ($product->shouldDiscountPercentageBeApplied()) {
    $netAmount = $product->discountPercentage()->applyTo($amount);
} else {
    $netAmount = $amount->subtract($product->fixedDiscountAmount());
```

```
}
```

A way to keep the information about how a discount should be calculated for the given product is to introduce a method with a name that exactly matches this intention: `calculateNetAmount()`:

```
final class Product
{
    public function calculateNetAmount(Money $amount): bool
    {
        // ...
    }

    private function shouldDiscountPercentageBeApplied(): bool
    {
        // ...
    }

    private function discountPercentage(): Percentage
    {
        // ...
    }

    private function fixedDiscountAmount(): Money
    {

    }
}

$amount = new Money(/* ... */);
$netAmount = $product->calculateNetAmount($amount);
```

Besides no longer repeating this logic at different call sites, this has two more advantages. First, we can stop exposing internal data like the discount percentage and the fixed discount. Second, when something changes about the calculation, it can now be changed and tested in one place.

In short: always look for ways to prevent the need for query methods which expose the object's internal data:

- Make the method smarter, adapt it to the actual need of its clients, or
- Move the call inside the object, letting it make its own decisions.

A naming convention for getters

You may have noticed that "getters" don't have the traditional "get" prefix. This convention shows how the method isn't a command method, but simply provides a piece of information. The method name is a description of the piece of information we're looking for, not an instruction for the object to "go get it" for us.

5.4 Define specific methods and return types for the queries you want to make

When you need a specific bit of information, make sure you have a specific question, and you know what the answer should look like.

As an example, if you're working on a piece of code and you need today's exchange rate for USD to EUR, you may discover that there are web services you can call to figure that out, like fixer.io[3]. So you jump in and write a bit of code that makes the call:

```
final class CurrencyConverter
{
    public function convert(Money $money, Currency $to): Money
    {
        $httpClient = new CurlHttpClient();
        $response = $httpClient->get(
            'http://data.fixer.io/api/latest?access_key=...' .
                '&base=' . $money->currency()->asString() .
                '&symbols=' . $to->asString()
```

[3] https://fixer.io/

```
        );
        $decoded = json_decode($response->getBody());
        $rate = (float)$decoded->rates->{$to->asString()};

        return $money->convert($to, $rate);
    }
}
```

There are many issues with just this tiny bit of code (we don't deal with the possible occurrence of a network failure, an error response, invalid JSON, a modified response structure, and besides, a `float` isn't the most reliable data type to use when dealing with amounts of money). However, at a conceptual level we're making way too large jumps here as well. All we needed at this point in the code is an answer to a question: "What's the current exchange rate for USD to EUR currency conversion?"

Rewriting this question in code results in two new classes: `ExchangeRates` and `ExchangeRate`. The first has a single method `exchangeRateFor()`, which represents the question that the `CurrencyConverter` wants to ask. The second class, `ExchangeRate`, represents the answer:

```
/*
 * We introduce `FixerApi::exchangeRateFor()` to represent the
 * question being asked: "What's the current exchange rate for
 * converting from ... to ...?"
 */

final class FixerApi
{
    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        $httpClient = new CurlHttpClient();
        $response = $httpClient->get(...);
        $decoded = json_decode($response->getBody());
        $rate = (float)$decoded->rates->{$to->asString()};
```

```
        return ExchangeRate::from($from, $to, $rate);
    }
}

/*
 * This is a new class that will represent the answer to the
 * question "What's the current exchange rate for ... to ...
 * currency conversion?".
 */

final class ExchangeRate
{
    public static function from(
        Currency $from,
        Currency $to,
        float $rate
    ): ExchangeRate {
        // ...
    }
}

/*
 * `CurrencyConverter` will get an `ExchangeRates` instance
 * injected, so it can find out the current exchange rate
 * when it needs to:
 */

final class CurrencyConverter
{
    private FixerApi $fixerApi;

    public function __construct(FixerApi $fixerApi)
    {
        $this->fixerApi = $fixerApi;
    }
```

```
    public function convert(Money $money, Currency $to): Money
    {
        $exchangeRate = $this->fixerApi
            ->exchangeRateFor(
                $money->currency(),
                $to
            );

        return $money->convert($exchangeRate);
    }
}
```

The "answer" class, `ExchangeRate` should be designed to be as useful as possible for the client that needs it. Potentially, this class can be reused at other call sites, but it doesn't have to be.

The important part is that the introduction of the `exchangeRateFor()` method with a specific return type improves the conversation that's going on in the code. When reading the code of `convert()`, we can clearly see there's a need for information, a question being asked, and an answer being returned, which is then used to do some more work. Note that so far we've only refactored the code; its structure has been improved, but it still has the same behavior.

5.5 Define an abstraction for queries that cross system boundaries

The question "What's the current exchange rate?" from the previous section is a question that the application itself can't answer, just by looking at what it has in memory. It needs to cross a system boundary to find the answer. In this case, it has to connect to some remote service, reachable through the network. Another example of crossing a system boundary would be when the application reaches out to the filesystem, to load or store a file. Or when it uses the system clock to find out the current time.

As soon as an application crosses a system boundary, you should always introduce an abstraction, allowing you to hide the low-level communication details of the actual calls that are going on behind the scenes.

Abstraction in this case means two things, and it can only be successful have you have both ingredients:

1. Using a service interface instead of a service class.
2. Leaving out the implementation details.

Introducing a proper abstraction will make it possible to run your code in a test scenario, without making the actual network or filesystem calls. It will also make it possible to swap out implementations without having to modify the client code; you only need to write a new implementation of the service interface.

First we'll discuss an example that fails to introduce a proper abstraction. Let's take another look at the `FixerApi` class. It makes a network call directly, using the `CurlHttpClient` class:

```
final class FixerApi
{
    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        $httpClient = new CurlHttpClient();
        $response = $httpClient->get(...);
        $decoded = json_decode($response->getBody());
        $rate = (float)$decoded->rates->{$to->asString()};

        return ExchangeRate::from($from, $to, $rate);
    }
}
```

Instead of instantiating and using this specific class, we could define an interface for it, and inject an instance of it into the `FixerApi` class:

```
/*
 * First we introduce an interface for HTTP clients:
 */

interface HttpClient
{
    public function get($url): Response;
}

/*
 * We also make sure the existing `CurlHttpClient`implements
 * this new `HttpClient` interface:
 */

final class CurlHttpClient implements HttpClient
{
    // ...
}

final class FixerApi
{
    /*
     * Now we inject the interface, not the concrete class:
     */

    public function __construct(HttpClient $httpClient)
    {
        $this->httpClient = $httpClient;
    }

    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        /*
         * We have to change the code a bit to use the new interface
```

```
            * and its `get()` method:
            */

        $response = $this->httpClient->get(...);
        $decoded = json_decode($response->getBody());
        $rate = (float)$decoded->rates->{$to->asString()};

        return ExchangeRate::from($from, $to, $rate);
    }
}
```

We can now swap out `HttpClient` implementations because we rely on the interface, not the concrete implementation. This could be useful in case you want to switch to a different HTTP client implementation some day. But we didn't abstract the most important part yet. What happens if we want to switch to a different API? It's not likely that a different API will send the same JSON response. Or maybe we want to start maintaining our own local database table with exchange rates? In that case, we wouldn't even need an HTTP client anymore.

To be able to remove the low-level implementation details, we need to pick a more abstract name that stands for what we're doing. We're looking for a way to retrieve exchange rates. Where would you get them from? From something that can "provide" them. Or from something that manages them, like a "collection". A good name for this abstraction could be `ExchangeRateProvider`, or simply `ExchangeRates`, if we look at this service like a collection of known exchange rates.

```
/**
 * We extract the "question" method and make it a public method on
 * an abstract `ExchangeRates` service:
 */
interface ExchangeRates
{
    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate;
}
```

```php
/*
 * The existing `FixerApi` class should
 * implement the new `ExchangeRates` interface:
 */

final class FixerApi implements ExchangeRates
{
    private HttpClient $httpClient;

    public function __construct(HttpClient $httpClient)
    {
        $this->httpClient = $httpClient;
    }

    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        $response = $this->httpClient->get(...);
        $decoded = json_decode($response->getBody());
        $rate = (float)$decoded->data->rate;

        return ExchangeRate::from($from, $to, $rate);
    }
}

final class CurrencyConverter
{
    private ExchangeRates $exchangeRates;

    /*
     * Instead of a `Fixer` object, we can now inject an
     * `ExchangeRates` instance:
     */
```

```
    public function __construct(ExchangeRates $exchangeRates)
    {
        $this->exchangeRates = $exchangeRates;
    }

    // ...

    private function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        /*
         * We use the new service here, to get the answer we're
         * looking for:
         */

        return $this->exchangeRates->exchangeRateFor($from, $to);
    }
}
```

As a final improvement we should inline any existing calls to that private method `exchangeRateFor()` because it's just a proxy to the `ExchangeRates` service now.

By defining an interface for the existing class, we performed step 1 of a successful abstraction. By hiding all the implementation details behind the interface, we also performed step 2, meaning we now have a proper abstraction for retrieving exchange rates. This comes with two advantages:

- We can easily switch to a different exchange rate provider. As long as the new class implements the `ExchangeRates` interface correctly, the `CurrencyConverter` won't have to be modified because it depends on the `ExchangeRates` abstraction.
- We can write a unit test for `CurrencyConverter` and inject a test double for `ExchangeRates`; one that doesn't make an internet connection. This will keep our test fast and stable.

By the way, if you know about the SOLID principles, you've already encountered a similar rule for abstraction of service dependencies, known as the *Dependency*

Inversion Principle. You can read more about it in books and articles[4] by Robert C. Martin.

Not every question deserves its own service

In the previous examples it was clear that the question "what's the exchange rate" deserved its own service. It was a question the application itself couldn't answer. In most situations, asking a question shouldn't immediately cause a new object to be introduced. There are several alternatives:

- You could introduce better variable names to improve that "conversation" that's going on inside the code.
- You could extract a private method, which represents the question and its answer (like we just did by moving the logic to the private `exchangeRateFor()` method).
- Only if the method becomes too large, needs to be tested separately, or crosses a system boundary, create a separate class for it.

This should keep the number of objects involved limited, and helps keep the code readable—you won't have to click through lots of classes to find out what's going on.

5.6 Use stubs for test doubles with query methods

The moment you introduce an abstraction for your queries, you create a useful extension point. You can change the implementation details of how exactly the "answer" will be found. Testing this logic will be easier too. Instead of only being able to test the `Currency-Converter` service when an internet connection (and the remote service) is available, you can now test the actual logic by replacing the injected `ExchangeRates` service with one that already has the answers and will supply them in a predictable manner.

[4] Robert C. Martin, "Principles of OOD", http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

```php
/*
 * This is a "fake" implementation of the `ExchangeRates` services,
 * which we can set up to return whatever exchange rates we provide
 * it with:
 */

final class ExchangeRatesFake implements ExchangeRates
{
    private array $rates = [];

    public function __construct(
        Currency $from,
        Currency $to,
        float $rate
    ) {
        $this->rates[$from->asString()][$to->asString()] =
            ExchangeRate::from($from, $to, $rate);
    }

    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        if (!isset($this->rates[$from->asString()][$to->asString()])) {
            throw new RuntimeException(
                sprintf(
                    'Could not determine exchange rate from %s to %s',
                    $from->asString(),
                    $to->asString()
                )
            );
        }
        return $this->rates[$from->asString()][$to->asString()];
    }
}
```

```
/*
 * We can then use this fake in the unit test for `CurrencyConverter`:
 */

/**
 * @test
 */
public function it_converts_an_amount_using_the_exchange_rate(): void
{
    // Set up the fake `ExchangeRates` service:
    $exchangeRates = new ExchangeRatesFake();
    $exchangeRates->setExchangeRate(
        new Currency('USD'),
        new Currency('EUR'),
        0.8
    );

    // Inject the fake service into the `CurrencyConverter`:
    $currencyConverter = new CurrencyConverter($exchangeRates);

    $converted = $currencyConverter
        ->convert(new Money(1000, new Currency('USD');

    assertEquals(new Money(800, new Currency('EUR')), $converted);
}
```

By setting up the test like this, we focus only on the logic of the `convert()` method, instead of all the logic that is involved in making the network connection, parsing the JSON response, etc. This makes the test deterministic and therefore stable.

Naming test methods

In the previous example, the test method has a name in so-called snake case: lower case, with underscores as word separators. If we'd simply follow the standard for naming methods, it should have been `itConvertsAnAmountUsingTheExchangeRate()`. Most standards would also suggest using relatively short names, but `it_converts_an_amount_using_the_exchange_rate()` is anything but short. Because they have a different purpose than regular methods, the way to go is not to submit test method to the same standard, but to set a different standard for them:

- Test method names describe object behaviors. The best description is an actual sentences.
- Because they are actual sentences, test method names will be longer than regular method names. It should still be easy to read them (so use snake case instead).

If you're not used to these rules, a good way to ease into them is to start a test method name with `it_`. This should get you in the right mood for describing a particular object behavior. Although it's a good starting point, you'll notice that not every test method makes sense starting with `it_`. For instance, `when_` or `if_` could work too.

A *Fake* is one kind of test double, which can be characterised as showing "somewhat complicated" behavior, just like the real implementation that will be used in production. When testing, you could also use a *Stub* to replace the real service. A stub is a test double which just returns hard-coded values. So whenever we'd call the `exchangeRateFor()` method, it would just return the same value:

```
/*
 * This is a sample stub implementation of `ExchangeRates`:
 */
final class ExchangeRatesStub
{
    public function exchangeRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate {
        // The return value is a hard-coded one
        return ExchangeRate::from($from, $to, 1.2);
    }
}
```

An important characteristic of stubs and fakes is that in a test scenario you can't and shouldn't make any assertions about the number of calls made to them, or the order in which those calls were made. Given the nature of query methods, they should be without side effects, so it should be possible to call them any number of times, even 0 times. Making assertions about calls made to query methods, leads to a test that doesn't keep a sufficient distance to the implementation of the class it's testing.

The opposite is the case for command methods, where you do want to verify that calls have been made, how many of them, and potentially in what order. We'll get back to this in the next chapter.

Don't use mocking tools for creating fakes and stubs

Mocking frameworks are often used to build these test doubles on-the-fly. I recommend against using these frameworks for creating fakes and stubs. They may save you a few lines of "boilerplate", but at the cost of code that is hard to read and maintain.

Even if you still prefer to use these mocking tools, I recommend using them only for creating *Dummies* (that is, test doubles that don't return anything meaningful and are only there to be passed as an unused argument). For stubs and fakes, mocking tools usually get in the way of good design. They will verify if and how many times a query method has been called. They often make refactoring harder, because method names often have to be provided as strings, and your refactoring tool may not recognize them as actual method names.

We shouldn't forget to also test the real implementation that uses an HTTP connection to retrieve exchange rates. We have to test that it does all that work correctly. But at that point we're no longer worried about testing the conversion logic itself, only that the implementation knows how to communicate well with the external service.

```
/**
 * @test
 */
public function it_retrieves_the_current_exchange_rate(): void
{
    $exchangeRates = new FixerApi();

    $exchangeRate = $exchangeRates->exchangeRateFor(
        new Currency('USD'),
        new Currency('EUR')
    );

    // verify the result
}
```

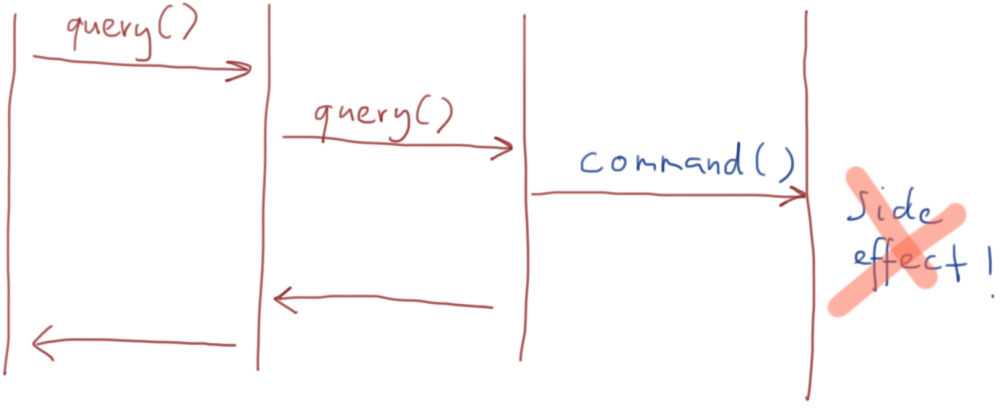
You will find that for this kind of test, you still need some way to stabilize it. You may have to set up your own exchange rate server which replicates the real one. Or you may be able to use a sandbox environment provided by the maintainers of the real service.

Note that this test doesn't count as a unit test anymore: it doesn't test the behavior of an object in memory. You could call this an *Integration test* instead, since it tests the integration of an object with the thing in the world outside that it relies on.

5.7 Query methods should use other query methods, no command methods

As we discussed, command methods can have side effects. Command methods change something, save something, send an email, etc. Query methods on the other hand won't do anything like that. They will just return a piece of information. Usually, a query method needs some collaborating objects to "build up" the requested answer. If we

get the division between command and query methods right in our code, a chain of calls that starts with a query, won't contain a call to a command method. This has to be true, because queries are supposed to have no side effects, and calling a command method somewhere in the chain will violate that rule.



There should be no calls to command methods hidden behind a query.

There are some exceptions to this rule. Consider a controller method for a web application, which can be called to register a new user. This method will have a side effect: somewhere down the chain of commands it will store a new user record in the database. This would normally force us to use a `void` return type for the controller itself. However, a web application should always return an HTTP response. So the controller will have to return at least something:

```
final class RegisterUserController
{
    private RegisterUser $registerUser;

    public function __construct(
        RegisterUser $registerUser
    ) {
        $this->registerUser = $registerUser;
    }

    public function __invoke(Request $request): Response
    {
```

```
        $newUser = $this->registerUser
            ->register($request->get('username'));

        return new Response(200, json_encode($newUser));
    }
}
```

Technically speaking, the controller violates the *Command/Query Separation* principle. But there's no way around that. At the very least we should return an empty `200 OK` response or something like that. But that won't be very useful for the frontend, which makes the "register user" `POST` request, and would like to be given a response with a JSON structure representing the newly created user.

To solve this case, you should divide the controller's "action" into two parts: registering the new user, and returning it. Preferably, you'd also determine the new ID of the user, before calling the `RegisterUser` service, so the service doesn't have to return anything at all and can be a true command method.

```
final class RegisterUserController
{
    private UserRepository $userRepository;
    private RegisterUser $registerUser;
    private UserReadModelRepository $userReadModelRepository;

    public function __construct(
        UserRepository $userRepository,
        RegisterUser $registerUser,
        UserReadModelRepository $userReadModelRepository
    ) {
        $this->userRepository = $userRepository;
        $this->registerUser = $registerUser;
        $this->userReadModelRepository = $userReadModelRepository;
    }

    public function __invoke(Request $request): Response
    {
        $userId = $this->userRepository->nextIdentifier();
```

```
        /*
         * `register()` is a command method:
         */
        $this->registerUser
            ->register($userId, $request->get('username'));

        /*
         * `getById()` is a query method:
         */
        $newUser = $this->userReadModelRepository->getById($userId);

        return new Response(200, json_encode($newUser));
    }
}
```

5.8 Summary

A query method is a method you can use to retrieve a piece of information. These methods should have a single return type. You may still return `null`, but make sure to look for alternatives, like a *Null object* or an empty list. Possibly throw an exception instead. Let query methods expose as little as possible of an object's internals.

Define specific methods and return values for every question you want to ask and every answer you want to get. Define an abstraction (an interface, free of implementation details) for these methods if the answer to the question can only be established by crossing the system's boundaries.

When testing services which use queries to retrieve information, replace them using fakes or stubs which you write yourself, and make sure not to test for actual calls being made to them.

6. Performing tasks

Besides retrieving information from objects, you can use objects to perform tasks for you. These tasks could be, for example:

- Send a reminder email,
- Save a record in the database,
- Change the password of a user,
- Store something on disk,
- And so on…

6.1 Use command methods with a name in the imperative form

We already discussed query methods and how you should use them to retrieve information. Query methods have a specific return type and no side effects, meaning that it's safe to call them several times, and that the application's state won't be any different afterwards.

For performing tasks, you should always use a command method, which has a `void` return type. The name of such a method indicates that the client can give the object the order to perform the task that the method name indicates. When looking for a good name, you should always use the imperative form, for example:

```
public function sendReminderEmail(
    EmailAddress $recipient,
    // ...
): void {
   // ...
}

public function saveRecord(Record $record): void
{
   // ...
}
```

6.2 Limit the scope of a command method, use events to perform secondary tasks

When performing a task, make sure you don't do too much in one method. These are some guiding questions to find out if a method is too large:

- Should or does the method name have "and" in it, to indicate what else it does besides its main job?
- Do all these lines of code contribute to the main job?
- Could a part of the work that this method does (in theory) be performed in a background process?

An example of a method that does too much is the following one. It changes the user's password, but also sends them an email about this:

```
public function changeUserPassword(
    UserId $userId,
    string $plainTextPassword
): void {
    $user = $this->repository->getById($userId);
    $hashedPassword = ...;
    $user->changePassword($hashedPassword);
    $this->repository->save($user);
    $this->mailer->sendPasswordChangedEmail($userId);
}
```

This is a very common scenario where the answers to the guiding questions are "yes" in all cases:

- The method name hides the fact that besides changing the user's password it will also send an email about it. It might as well have been `changeUserPasswordAndSendAnEmailAboutIt()`.
- Sending the email can't be considered the main job of this method; changing the password is.
- The email could have easily been sent in some other process that runs in the background.

One solution would be to move the email sending code to a new (public) method `sendPasswordChangedEmail()`. However, this would transfer the responsibility of calling that method to the client of `changeUserPassword()`. Considering the bigger picture, these two tasks really belong together; we just don't want to mix them in one method.

The recommended solution is to use events as the link between "changing a password" and "sending an email about it":

```
/*
 * The fact that a user has changed their password can be
 * represented by a `UserPasswordChanged` event object:
 */

final class UserPasswordChanged
{
    private UserId $userId;

    public function __construct(UserId $userId)
    {
        $this->userId = $userId;
    }

    public function userId(): UserId
    {
        return $this->userId;
    }
}

public function changeUserPassword(
    UserId $userId,
    string $plainTextPassword
): void {
    $user = $this->repository->getById($userId);
    $hashedPassword = ...;
    $user->changePassword($hashedPassword);
    $this->repository->save($user);

    /*
     * After actually changing the password, dispatch a
     * `UserPasswordChanged` event, so other services can respond to
     * it:
     */

    $this->eventDispatcher->dispatch(
```

```
        new UserPasswordChanged($userId)
    );
}

/*
 * `SendEmail` is an event listener for the `UserPasswordChanged`
 * event. When notified of the event, this listener will actually
 * send the email:
 */

final class SendEmail
{
    // ...

    public function whenUserPasswordChanged(
        UserPasswordChanged $event
    ): void {
        $this->mailer->sendPasswordChangedEmail($event->userId());
    }
}
```

You still need an event dispatcher which allows event listener services like `SendEmail` to be "registered". Most frameworks already have an event dispatcher that you can use for that, or you could write a very simple one yourself, like this one:

```
final class EventDispatcher
{
    private array $listeners;

    public function __construct(array $listenersByType)
    {
        foreach ($listenersByType as $eventType => $listeners) {
            Assertion::string($eventType);
            Assertion::allIsCallable($listeners);
        }
```

```
        $this->listeners = $listenersByType;
    }

    public function dispatch(object $event): void
    {
        $listeners = $this->listeners[get_class($event)] ?: [];
        foreach ($listeners as $listener) {
            $listener($event);
        }
    }
}

$listener = new SendEmail(...);
$dispatcher = new EventDispatcher([
    UserPasswordChanged::class =>
        [$listener, 'whenUserPasswordChanged']
]);

/*
 * Because we've registered `SendEmail` as an event listener for the
 * `UserPasswordChanged` event, dispatching an event of that type
 * will trigger a call to `SendEmail::whenUserPasswordChanged()`:
 */
$dispatcher->dispatch(new UserPasswordChanged(...));
```

Using events like this has several advantages:

- You can add even more effects, without modifying the original method.
- The original object will be more decoupled, because it doesn't get all the dependencies injected that are only needed for the effects.
- You could actually handle the effects in a background process if you wanted.

6.3 Make services immutable from the outside as well as on the inside

We already covered the rule that it should be impossible to change anything about a service's dependencies or configuration. Once it's been instantiated, a service object should be reusable for performing multiple different tasks in the same way, but using different data or a different context. There shouldn't be any risk that its behavior changes between calls. This is true for services that offer query methods, but also for ones that offer command methods.

Even if you don't offer clients a way to manipulate a service's dependencies or configuration, command methods may still change a service's state in such a way that behavior will be different for subsequent calls. For example, a service that sends out confirmation emails, may remember which users have already received such an email. No matter how many times you call the same method again, it will only send out an email once:

```
final class Mailer
{
    private array $sentTo = [];

    // ...

    public function sendConfirmationEmail(
        EmailAddress $recipient
    ): void {
        if (in_array($recipient, $this->sentTo)) {
            // Don't send the email again:
            return;
        }

        // Send the email...

        $this->sentTo[] = $recipient;
    }
}
```

```
$mailer = new Mailer(...);
$recipient = EmailAddress::fromString('info@matthiasnoback.nl');

// This will send out a confirmation email:
$mailer->sendConfirmationEmail($recipient);

// The second call won't send it again:
$mailer->sendConfirmationEmail($recipient);
```

Make sure none of your services update internal state which influences its behavior like this.

A guiding question to decide if your service behaves properly in this respect is: "Would it be possible to re-instantiate the service for every method call, and would it still show the same behavior?" For the `Mailer` class in the example, this obviously isn't true—reinstantiating it would cause multiple emails to be sent to the same recipient.

In the case of the stateful `Mailer` service, the question is: how can we prevent duplicate calls to `sendConfirmationEmail()`? Somehow the client isn't smart enough to take care of this. What if, instead of providing just one `EmailAddress`, the client could provide an already de-duplicated list of `EmailAddress` instances?

```
final class Recipients
{
    /**
     * @var EmailAddress[]
     */
    private array $emailEmailAddresses;

    /**
    * @return EmailAddress[]
     */
    public function uniqueEmailAddresses(): array
    {
        // return a de-duplicated list of of addresses
    }
```

```
}

final class Mailer
{
    public function sendConfirmationEmails(
        Recipients $recipients
    ): void {
        foreach ($recipients->uniqueEmailAddresses()
            as $emailAddress) {
            // Send the email...
        }
    }
}
```

This would certainly solve the problem and make the `Mailer` service stateless again. But instead of letting `Mailer` make that special call to `uniqueEmailAddresses()`, what we're actually looking for is a list of `Recipients` that couldn't even contain duplicate email addresses. You could most elegantly protect this domain invariant inside the `Recipients` class itself:

```
final class Recipients
{
    /**
     * @var EmailAddress[]
     */
    private array $emailAddresses;

    private function __construct(array $emailAddresses)
    {
        $this->emailAddresses = $emailAddresses;
    }

    // Start with an empty list...

    public static function emptyList(): Recipients
    {
```

```
        return new self([]);
    }

    /*
     * Any time a client wants to add an email address to it,
     * it will only be added it it's not already on the list:
     */

    public function with(EmailAddress $emailAddress): Recipients
    {
        if (in_array($emailAddress, $this->emailAddresses)) {
            // no need to add it again
            return $this;
        }

        return new self(
            array_merge($this->emailAddresses),
            [$emailAddress]
        );
    }

    // There's no need for a `uniqueEmailAddresses()` method anymore

    public function emailAddresses(): array
    {
        return $this->emailAddresses;
    }
}
```

Immutable services and service containers

Service containers are often designed to share all service instances, once they have been created. This saves the runtime from instantiating the same service again, should it be reused as a dependency of some other service. However, if a service is immutable (like it should be), this sharing isn't really needed. You could instantiate the service over and over again.

Of course, there are services in a service container that shouldn't be instantiated again every time they're used as a dependency. For instance a database connection object or any other kind of reference to a resource that needs to be created once, and then shared between dependent services. In general however, your services shouldn't need to be shared. If you've followed all of the advise so far, you're doing great already, because immutable services don't need to be shared. They can, but they don't have to.

6.4 When something goes wrong, throw an exception

The same rule for retrieving information also counts for performing tasks: when something goes wrong, don't return a special value to indicate it, throw an exception instead. As discussed, a method can have pre-condition checks which throw `InvalidArgumentExceptions` or `LogicExceptions`. For the remainder of the failure scenarios, we couldn't determine upfront if they will occur, so we'd throw a `RuntimeException`. We've already discussed the other important rules for using exceptions in the introduction to "Using objects".

6.5 Use queries to collect information, commands to take the next steps

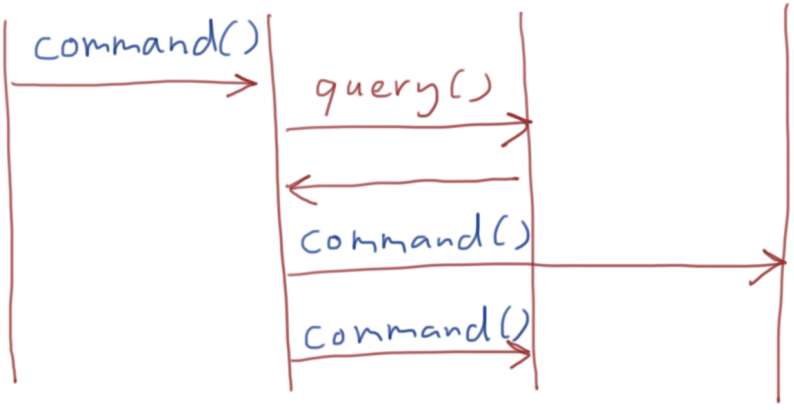
Earlier, when we discussed query methods, we saw how a chain of method calls that starts with a call to a query method, won't have a call to a *command* method inside of it. The command method may produce a side effect, which violates the rule that a query method shouldn't have any side effect.

Now that we're looking at command methods, we should note that the other way around, there's no such rule. When a chain of calls starts with a command method, it's

still possible that you'll encounter a call to a query method down the line. For instance, the `changeUserPassword()` method we saw earlier, actually starts with a query to the user repository:

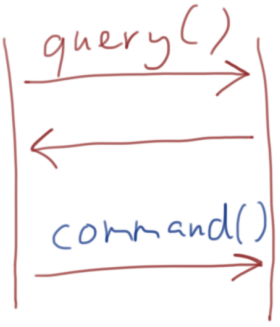
```
public function changeUserPassword(
    UserId $userId,
    string $plainTextPassword
): void {
    $user = $this->repository->getById($userId);
    $hashedPassword = ...;
    $user->changePassword($hashedPassword);
    $this->repository->save($user);
    $this->eventDispatcher->dispatch(
        new UserPasswordChanged($userId)
    );
}
```

The next method call is `changePassword()` on the user object, then another command on the repository. Inside the repository implementation, there may again be calls to command methods, but it's also possible that query methods are being called there.



Inside a command method, you may call query methods to retrieve more information.

However, when looking at how objects call each other's command and query methods, be aware of the following pattern:



Calling a query method, then a command method on the same object.

This pattern of calls often indicates a little conversation between objects that could have been happening inside the called object only. For example:

```
if ($obstacle->isOnTheRight()) {
    $player->moveLeft();
} elseif ($obstacle->isOnTheLeft()) {
    $player->moveRight();
}
```

The following is an improvement on this piece of code, where the knowledge about which action to take is now completely inside the object.

```
$player->evade($obstacle);
```

This object is able to keep this knowledge to itself, and its implementation can evolve freely, whenever it needs to show more complicated behavior.

6.6 Define an abstraction for commands that cross system boundaries

If a command method has code that reaches out across the application's own boundaries (that is, it uses a remote service, the filesystem, a system device, etc.), you should introduce an abstraction for it. For instance, here's a piece of code that publishes a message to a queue, in order for background consumers to tune into important events inside the main application:

```
final class SendMessageToRabbitMQ
{
    // ...

    public function whenUserChangedPassword(
        UserPasswordChanged $event
    ): void {
        $this->rabbitMqConnection->publish(
            'user_events',
            'user_password_changed',
            json_encode([
                'user_id' => (string)$event->userId()
            ])
        );
    }
}
```

The `publish()` method will reach out to the RabbitMQ server and publish a message to its queue, which is outside of the application's boundaries. So we should come up with an abstraction here. As discussed earlier, this requires an interface, and a higher-level concept. For example, preserving the notion that we want to queue a message, we introduce the following abstraction:

```
/*
 * `Queue` is the abstraction:
 */

interface Queue
{
    public function publishUserPasswordChangedEvent(
        UserPasswordChanged $event
    ): void;
}

/*
 * The standard `Queue` implementation is `RabbitMQQueue`, which
```

```
 * contains the code we already had:
 */

final class RabbitMQQueue implements Queue
{
    // ...

    public function publishUserPasswordChangedEvent(
        UserPasswordChanged $event
    ): void {
        $this->rabbitMqConnection->publish(
            'user_events',
            'user_password_changed',
            json_encode([
                'user_id' => (string)$event->userId()
            ])
        );
    }
}

/*
 * The event listener which is supposed to publish a message to the
 * queue whenever a `UserPasswordChanged` event has occurred, will
 * use the new abstraction as a dependency:
 */

final class SendMessageToRabbitMQ
{
    private Queue $queue;

    public function __construct(Queue $queue)
    {
        $this->queue = $queue;
    }

    public function whenUserPasswordChanged(
```

```
        UserPasswordChanged $event
    ): void {
        $this->queue->publishUserPasswordChangedEvent($event);
    }
}
```

The first step was to introduce an *abstraction*. Once you start adding more `publish...Event()` methods to `Queue`, you may start noticing similarities between these methods. Then you could apply *generalization* to make these methods more generic. This may require events to implement a certain interface that can be used for all events.

```
interface CanBePublished
{
    public function queueName(): string;
    public function eventName(): string;
    public function eventData(): array;
}

final class RabbitMQQueue implements Queue
{
    // ...

    public function publish(CanBePublished $event): void
    {
        $this->rabbitMqConnection->publish(
            $event->queueName(),
            $event->eventName(),
            json_encode($event->eventData())
        );
    }
}
```

It's generally a good idea to start with the abstraction and then wait with the generalization until you've seen about three cases that could be simplified by making the interface and the object types that are involved more generic. This prevents you from having to go back over and over again for every new case you want your abstraction to support.

6.7 Only verify calls to command methods with a mock

We already discussed that query methods shouldn't be mocked. In a unit test, you shouldn't verify the number of calls made to them. Queries are supposed to be without side effects, so you could make them many times if you want to. Allowing the implementation to do so, increases the stability of the test. If you decide to call a method twice instead of remembering its result in a variable, the test won't break.

However, when a command method makes a call to another command method, you may want to mock the latter. After all, this command is *supposed* to be called at least once (you want to verify that, because it's part of the job), but it shouldn't be called more than once (because you don't want to have its side effects being produced more than once too).

```
final class ChangePasswordService
{
    private EventDispatcher $eventDispatcher;
    // ...

    public function __construct(
        EventDispatcher $eventDispatcher,
        // ...
    ) {
        $this->eventDispatcher = $eventDispatcher;

        // ...
    }

    public function changeUserPassword(
        UserId $userId,
        string $plainTextPassword
    ): void {
        // ...

        $this->eventDispatcher->dispatch(
            new UserPasswordChanged($userId)
```

```
        );
    }
}

/**
 * @test
 */
public function it_dispatches_a_user_password_changed_event(): void
{
    $userId = ...;

    /*
     * This defines a true mock object: we verify how many times
     * (once) we expect a method to be called, and with which
     * arguments. We don't make assertions about the return value,
     * since `dispatch()` is a command method.
     */
    $eventDispatcherMock = $this->createMock(EventDispatcher::class);
    $eventDispatcherMock
        ->expects($this->once())
        ->method('dispatch')
        ->with(new UserPasswordChanged($userId));

    $service = new ChangePasswordService($eventDispatcherMock, ...);

    $service->changeUserPassword($userId, ...);
}
```

There are no regular assertions at the end of this test method, because the mock object itself will verify that our expectations were met. The test framework will ask all mock objects that were created for a single test case to do this.

If you prefer to have some actual assertions in your test case, you could use a *Spy* as test double for `EventDispatcher`. In the most generic form, a spy will remember all method calls that were made to it, including the arguments used. However, in our case, a really simple `EventDispatcher` implementation would suffice:

```
final class EventDispatcherSpy implements EventDispatcher
{
    private array $events = [];

    public function dispatch(object $event): void
    {
        /*
         * The spy just keeps a list of all the events that were
         * dispatched to it:
         */

        $this->events[] = $event;
    }

    public function dispatchedEvents(): array
    {
        return $this->events;
    }
}

/**
 * @test
 */
public function it_dispatches_a_user_password_changed_event(): void
{
    // ...
    $eventDispatcher = new EventDispatcherSpy();
    $service = new ChangePasswordService($eventDispatcher, ...);

    $service->changeUserPassword($userId, ...);

    /*
     * Now we can make an actual assertion instead of waiting for
     * the test framework to verify the methods calls on our mock:
     */
```

```
    assertEquals(
        [
            new UserPasswordChanged($userId)
        ],
        $eventDispatcher->dispatchedEvents()
    );
}
```

6.8 Summary

Performing tasks should be done with command methods. These command methods have imperative names (“Do this”, “Do that”). They should be limited in scope. Make a distinction between the main job, and the effects of this job. Dispatch events to let other services perform additional tasks. While performing its task, a command method may also call query methods to collect any information needed.

A service should be immutable from the outside, as well as on the inside. Just like with services for retrieving data, services that perform tasks should be reusable many times. If something goes wrong while performing a task, throw an exception (as soon as you know it).

Define an abstraction for commands that cross a system boundary (i.e. reach out to some remote service, database, etc.). When testing command methods that themselves call command methods, you can use a mock or a spy to test calls to these methods. You can use a mocking tool for this, or write your own spies.

7. Dividing responsibilities

We've looked at how objects can be used to retrieve information, or perform tasks. The methods for retrieving information are called query methods, the ones that perform tasks are command methods. Service objects may combine both of these responsibilities. For instance, a repository could perform the task of saving an entity to the database, and at the same time it would also be capable of retrieving an entity from the database:

```
interface PurchaseOrderRepository
{
    /**
     * @throws CouldNotSavePurchaseOrder
     */
    public function save(PurchaseOrder $purchaseOrder): void;

    /**
     * @throws CouldNotFindPurchaseOrder
     */
    public function getById(int $purchaseOrderId): PurchaseOrder;
}
```

Since saving and retrieving an entity are more or less each other's inverse operations, it's only natural to let one object have both responsibilities. However, in most other cases you will find that performing tasks and retrieving information are better off being divided amongst different objects.

7.1 Separate write models from read models

As we saw earlier, there are services, and other objects. Some of these other objects can be characterized as *Entities*, which model a particular domain concept. In doing so, they contain some relevant data, and offer ways to manipulate that data in valid and

meaningful ways. Entities can also expose data, allowing clients to retrieve information from them, whether that is exposed internal data (like the date on which an order was placed), or calculated data (like the total amount of the order).

In practice, it turns out that different clients use entities in different ways. Some clients will want to manipulate an entity's data using its command methods, while others just want to retrieve a piece of information from it using its query methods. Nevertheless, all these clients will share the same object, and potentially have access to all the methods, even when they don't need them, or should't even have access to them.

You should never pass an entity that can be modified to a client that isn't allowed to modify it. Even if the client doesn't modify it today, one day it might, and then it will be hard to find out what happened. That's why the first thing you should do to improve the design of an entity, is separate the *Write model* from the *Read model*.

We'll find out how to accomplish this by looking at an example of a `PurchaseOrder` entity. A purchase order represents the fact that a company buys a product from one of its suppliers. Once the product has been received, it's shelved in the company's warehouse. From that moment on the company has this product in stock. We'll use the same example for the remaining part of this chapter and work out different ways to improve it.

```
/*
 * For brevity, we use primitive type values, while
 * in practice, the use of value objects is recommended.
 */

final class PurchaseOrder
{
    private int $purchaseOrderId;
    private int $productId;
    private int $orderedQuantity;
    private bool $wasReceived;

    private function __construct()
    {
    }
```

```
    public static function place(
        int $purchaseOrderId,
        int $productId,
        int $orderedQuantity
    ): PurchaseOrder {
        $purchaseOrder = new self();

        $purchaseOrder->productId = $productId;
        $purchaseOrder->orderedQuantity = $orderedQuantity;
        $purchaseOrder->wasReceived = false;

        return $purchaseOrder;
    }

    public function markAsReceived(): void
    {
        $this->wasReceived = true;
    }

    public function purchaseOrderId(): int
    {
        return $this->purchaseOrderId;
    }

    public function productId(): int
    {
        return $this->productId;
    }

    public function orderedQuantity(): int
    {
        return $this->orderedQuantity;
    }

    public function wasReceived(): bool
    {
```

```
        return $this->wasReceived;
    }
}
```

In the current implementation, the `PurchaseOrder` entity exposes methods for creating and manipulating the entity (`place()` and `markAsReceived()`, as well as for retrieving information from it (`productId()`, `orderedQuantity()` and `wasReceived()`). Now take a look at how different clients use this entity. First, the `ReceiveItems` service, which will be called from a controller, passing in a raw purchase order ID:

```
final class ReceiveItems
{
    private PurchaseOrderRepository $repository;

    public function __construct(PurchaseOrderRepository $repository)
    {
        $this->repository = $repository;
    }

    public function receiveItems(int $purchaseOrderId): void
    {
        $purchaseOrder = $this->repository->getById($purchaseOrderId);

        $purchaseOrder->markAsReceived();

        $this->repository->save($purchaseOrder);
    }
}
```

Note that this service doesn't use any of the getters on `PurchaseOrder`. It's only interested in changing the state of the entity. Next, let's take a look at a controller which renders a JSON-encoded data structure detailing how much of a product the company has in stock:

```
final class StockReportController
{
    private PurchaseOrderRepository $repository;

    public function __construct(PurchaseOrderRepository $repository)
    {
        $this->repository = $repository;
    }

    public function __invoke(Request $request): Response
    {
        $allPurchaseOrders = $this->repository->findAll();

        $stockReport = [];

        foreach ($allPurchaseOrders as $purchaseOrder) {
            if (!$purchaseOrder->wasReceived()) {
                /*
                 * We didn't receive the items yet, so we
                 * shouldn't add them to the quantity-in-stock.
                 */
                continue;
            }

            if (!isset($stockReport[$purchaseOrder->productId()] )) {
                // We didn't see this product before...
                $stockReport[$purchaseOrder->productId()] = 0;
            }

            /*
             * Add the ordered (and received) quantity to the
             * quantity-in-stock:
             */
            $stockReport[$purchaseOrder->productId()]
                += $purchaseOrder->orderedQuantity;
        }
```

```
        return new JsonResponse($stockReport);
    }
}
```

This controller doesn't make any change to a `PurchaseOrder`. It just needs a bit of information from all of them. In other words, it isn't interested in the write part of the entity, only in the read part. Besides the fact that it is undesirable to expose more behavior to a client than it needs, it isn't very efficient to loop over all purchase orders of all times, to find out how much of a product the company has in stock.

The solution is to divide the entity's responsibilities. First, we create a new object that can be used for retrieving information about a purchase order:

```
final class PurchaseOrderForStockReport
{
    private int $productId;
    private int $orderedQuantity;
    private bool $wasReceived;

    public function __construct(
        int $productId,
        int $orderedQuantity,
        bool $wasReceived
    ) {
        $this->productId = $productId;
        $this->orderedQuantity = $orderedQuantity;
        $this->wasReceived = $wasReceived;
    }

    public function productId(): ProductId
    {
        return $this->productId;
    }

    public function orderedQuantity(): int
    {
```

```
        return $this->orderedQuantity;
    }

    public function wasReceived(): bool
    {
        return $this->wasReceived;
    }
}
```

This new `PurchaseOrderForStockReport` object can be used inside the controller as soon as there is a repository which can provide it. A quick and dirty solution would be to let `PurchaseOrder` return an instance of `PurchaseOrderForStockReport`, based on its internal data:

```
final class PurchaseOrder
{
    private int $purchaseOrderId
    private int $productId;
    private int $orderedQuantity;
    private bool $wasReceived;

    // ...

    public function forStockReport(): PurchaseOrderForStockReport
    {
        return new PurchaseOrderForStockReport(
            $this->productId,
            $this->orderedQuantity,
            $this->wasReceived
        );
    }
}

final class StockReportController
{
    private PurchaseOrderRepository $repository;
```

```
    public function __construct(PurchaseOrderRepository $repository)
    {
        $this->repository = $repository;
    }

    public function __invoke(Request $request): Response
    {
        // For now, we still load `PurchaseOrder` entities:
        $allPurchaseOrders = $this->repository->findAll();

        /*
         * But we immediately convert them to
         * `PurchaseOrderForStockReport` instances:
         */
        $forStockReport = array_map(
            function (PurchaseOrder $purchaseOrder) {
                return $purchaseOrder->forStockReport();
            },
            $allPurchaseOrders
        );

        // ...
    }
}
```

We can now remove pretty much all of the query methods (`productId()`, `orderedQuantity()`, `wasReceived()`) from the original `PurchaseOrder` entity. This makes it a proper write model; it isn't used by clients who just want information from it anymore:

```
final class PurchaseOrder
{
    private int $purchaseOrderId
    private int $productId;
    private int $orderedQuantity;
    private bool $wasReceived;

    private function __construct()
    {
    }

    public static function place(
        int $purchaseOrderId,
        int $productId,
        int $orderedQuantity
    ): PurchaseOrder {
        $purchaseOrder = new self();

        $purchaseOrder->productId = $productId;
        $purchaseOrder->orderedQuantity = $orderedQuantity;

        return $purchaseOrder;
    }

    public function markAsReceived(): void
    {
        $this->wasReceived = true;
    }
}
```

Removing these query methods won't do any harm to the existing clients of `PurchaseOrder` that use this object as a write model, like the `ReceiveItems` service we saw earlier:

```
final class ReceiveItems
{
    // ...

    public function receiveItems(int $purchaseOrderId): void
    {
        /*
         * This service doesn't use any query method of
         * `PurchaseOrder`:
         */
        $purchaseOrder = $this->repository->getById(
            PurchaseOrderId::fromInt($purchaseOrderId)
        );

        $purchaseOrder->markAsReceived();

        $this->repository->save($purchaseOrder);
    }
}
```

Query methods aren't forbidden

Some clients use the entity as a write model, but still need to retrieve some information from it. For instance in order to make decisions, perform extra validations, etc. Don't feel blocked to add more query methods in these cases; query methods aren't by any means forbidden. The point of this chapter is that clients that solely use an entity to retrieve information from it, should use a dedicated read model instead of a write model.

7.2 Create read models that are specific for their use cases

In the previous section, we decided to split the `PurchaseOrder` entity into a write and a read model. The write model still carries the old name, but we called the read model `PurchaseOrderForStockReport`. The extra qualification `ForStockReport` indicates that this object now serves a specific purpose. The object will be suitable for

use in a very specific context, namely the context where we arrange the data in such a way that we can produce a useful stock report for the user. The proposed solution isn't optimal yet, because the controller still needs to load all the `PurchaseOrder` entities, then convert them to `PurchaseOrderForStockReport` instances by calling `forStockReport()` on them. This means that the client still has access to that write model, even though our initial goal was to prevent that from happening:

```
public function __invoke(Request $request): Response
{
    // We still rely on `PurchaseOrder` instances here:
    $allPurchaseOrders = $this->repository->findAll();

    $forStockReport = array_map(
        function (PurchaseOrder $purchaseOrder) {
            return $purchaseOrder->forStockReport();
        },
        $allPurchaseOrders
    );

    // ...
}
```

There is another aspect of the design that isn't quite right: even though we now have `PurchaseOrderForStockReport` objects, we still need to loop over them and build up yet another data structure, before we can present the data to the user. What if we had an object whose structure completely matched the way we intend to use it? Concerning the name of this object, there's already a hint in the name of the read model (`ForStockReport`). So let's call this new object `StockReport`, and assume it already exists. The controller would become much simpler now:

```
final class StockReportController
{
    private StockReportRepository $repository;

    public function __construct(StockReportRepository $repository)
    {
        $this->repository = $repository;
    }

    public function __invoke(Request $request): Response
    {
        $stockReport = $this->repository->getStockReport();

        /*
         * `asArray()` is expected to return an array like we the
         * one we created manually before:
         */
        return new JsonResponse($stockReport->asArray());
    }
}
```

Besides `StockReport` we may create any number of read models which correspond to each of the application's specific use cases. For instance, we could create a read model that's used for listing purchase orders only. It would expose just the ID and the date on which it was created. We could then have a separate read model that provides all the details needed to render a form that allows the user to update some of its information; and so on.

Behind the scenes, the `StockReportRepository` could still create the `StockReport` object based on `PurchaseOrderForStock` objects provided by the write model entities. But there are much better and more efficient alternatives to do it. We'll cover some of them in the following sections.

Create read models directly from their data source

Instead of creating a `StockReport` model from `PurchaseOrderForStock` objects, we could go directly to the source of the data, that is, the database where the

application stores its purchase orders. If this is a relational database, there might be a table called `purchase_orders`, with columns for `purchase_order_id`, `product_id`, `ordered_quantity`, and `was_received`. If that's the case, then `StockReportRepository` wouldn't have to load any other object before it could build a `StockReport` object; it could make a single SQL query and use it to create the `StockReport`:

```
final class StockReportSqlRepository implements StockReportRepository
{
    public function getStockReport(): StockReport
    {
        $result = $this->connection->execute(
            'SELECT ' .
            ' product_id, ' .
            ' SUM(ordered_quantity) as quantity_in_stock ' .
            'FROM purchase_orders ' .
            'WHERE was_received = 1 ' .
            'GROUP BY product_id'
        );

        $data = $result->fetchAll();

        return new StockReport($data);
    }
}
```

Creating read models directly from the write model's data source is usually pretty efficient in terms of runtime performance. It's also an efficient solution in terms of development and maintenance costs. This solution will be less efficient if the write model changes often, or if the raw data can't easily be used as-is, because it needs to be interpreted first.

Build read models from domain events

One disadvantage of creating the `StockReport` read model directly from the write model's data is that the application will make the calculations again and again, every time the user requests a stock report. Although the SQL query won't take too long to

execute (until the table grows very large), in some cases it'll be necessary to use another approach for creating read models.

First, let's take another look at the result of the SQL query we used in the previous example:

product_id	quantity_in_stock
123	10
124	5

What would be another way in which we can come up with the numbers in the second column, that wouldn't involve looking up all the records in the `purchase_orders` table and summing their `ordered_quantity` values?

What if we could sit next to the user with a piece of paper, and whenever they mark a purchase order as "received", we'd write down the ID of the product and how many items of it were received. The resulting list would look like this:

product_id	received
123	2
124	4
124	1
123	8

Now, instead of having multiple rows for the same product, we could also look up the row with the product that was just received, and add the quantity we received to the number that's already in the `received` column:

product_id	received
123	2 + 8
124	4 + 1

Doing the calculations, this amounts to the exact same result as when we used the `SUM` query to find out how much of a product we have in stock.

Instead of sitting next to the user with a piece of paper, we should listen in our `PurchaseOrder` entity to find out when a user marks it as received. We can do this by recording and dispatching *Domain events*; a technique we already saw in a previous chapter. First, we need to let `PurchaseOrder` record a domain event, indicating that the ordered items were received:

```
// This is the new domain event:

final class PurchaseOrderReceived
{
    private int $purchaseOrderId;
    private int $productId;
    private int $receivedQuantity;

    public function __construct(
        int $purchaseOrderId,
        int $productId,
        int $receivedQuantity
    ) {
        $this->purchaseOrderId = $purchaseOrderId;
        $this->productId = $productId;
        $this->receivedQuantity = $receivedQuantity;
    }

    public function productId(): int
    {
        return $this->productId;
    }

    public function receivedQuantity(): int
    {
        return $this->receivedQuantity;
    }
}

// We record the domain event inside `PurchaseOrder`:

final class PurchaseOrder
{
    private array $events = [];

    // ...
```

```
    public function markAsReceived(): void
    {
        $this->wasReceived = true;

        $this->events[] = new PurchaseOrderReceived(
            $this->purchaseOrderId,
            $this->productId,
            $this->orderedQuantity
        );
    }

    public function recordedEvents(): array
    {
        return $this->events;
    }
}
```

Calling `markAsReceived()` will from now on add a `PurchaseOrderReceived` event object to the list of internally recorded events. These events can be taken out and handed over to an event dispatcher, for example in the `ReceiveItems` service:

```
final class ReceiveItems
{
    // ...

    public function receiveItems(int $purchaseOrderId): void
    {
        // ...

        $this->repository->save($purchaseOrder);

        $this->eventDispatcher->dispatchAll(
            $purchaseOrder->recordedEvents()
        );
```

```
    }
}
```

An event listener that has been registered for this particular event can take the relevant data from the event object, and update its own private list of products and quantities in stock. For instance, it could build up the stock report, by maintaining its own table `stock_report` with rows for every product. It has to process the incoming `PurchaseOrderReceived` events and create new rows or updating existing ones in this `stock_report` table:

```
final class UpdateStockReport
{
    public function whenPurchaseOrderReceived(
        PurchaseOrderReceived $event
    ): void {
        $this->connection->transactional(function () {
            // Find out if we have an existing row
            $existingRow = $this->connection
                ->prepare(
                    'SELECT quantity_in_stock ' .
                    'FROM stock_report ' .
                    'WHERE product_id = :productId FOR UPDATE'
                )
                ->bindValue('productId', $event->productId())
                ->execute()
                ->fetch();

            if ($existingRow === false) {
                /*
                 * If no row exists for this product, create a new
                 * one and set an initial value for
                 * quantity-in-stock
                 */
                $this->connection
                    ->prepare(
                        'INSERT INTO stock_report ' .
```

```
                        ' (product_id, quantity_in_stock) ' .
                        'VALUES (:productId, :quantityInStock)'
                    )
                    ->bindValue(
                        'productId',
                        $event->productId()
                    )
                    ->bindValue(
                        'quantityInStock',
                        $event->quantityReceived()
                    )
                    ->execute();
            } else {
                /*
                 * Otherwise, update the existing one; increase the
                 * quantity-in-stock
                 */
                $this->connection
                    ->prepare(
                        'UPDATE stock_report ' .
                        'SET quantity_in_stock = ' .
                        ' quantity_in_stock + :quantityReceived ' .
                        'WHERE product_id = :productId'
                    )
                    ->bindValue(
                        'productId',
                        $event->productId()
                    )
                    ->bindValue(
                        'quantityReceived',
                        $event->quantityReceived()
                    )
                    ->execute();
            }
        });
    }
```

```
}
```

Once we have a separate data source for the stock report, we can make `StockReportSqlReposito` even simpler, because all the information is already in the `stock_reports` table:

```
final class StockReportSqlRepository implements StockReportRepository
{
    public function getStockReport(): StockReport
    {
        $result = $this->connection->execute(
            'SELECT * FROM stock_report'
        );

        $data = $result->fetchAll();

        return new StockReport($data);
    }
}
```

This kind of simplification may offer you a way to make your read model queries more efficient. However, in terms of development and maintenance costs, using domain events to build up read models is more expensive. As you can see by looking at the examples in this section, there are more moving parts involved. If something changes about a domain event, it will be more work to adapt the other parts that depend on it. If one of the event listeners fails, you need to be able to fix the error, and run it again, which requires some extra effort in terms of tooling and operations as well.

What about event sourcing?

Things will be even more complex if besides using events for building up read models, you also use events for reconstructing write models. This technique is called *Event sourcing* and fits very well with the idea of separating write models from read models. However, as demonstrated in this chapter, you don't need to apply event sourcing if you're only looking for a better way to divide responsibilities between objects. Using any of the techniques described here, you can already provide clients that only want to retrieve information from an entity with a separate read model.

7.3 Summary

For your domain objects, make sure to separate write models from read models. Clients that are only interested in an entity because they need data from it should use a dedicated object, instead of the same entity that exposes methods for changing its state.

Read models can be created directly from the write model. A more efficient way would be to create it from the data source used by the write model. If that is impossible, or the read model can't be created in an efficient way, consider using domain events to build up the read model over time.

Changing the behavior of objects

You can design your services to be created, and used in certain ways. But the nature of a software project is that it will change over time. You'll often modify the class in such a way that, when it's used, it will behave the way you want it to. However, modifying a class comes with a cost: the danger of breaking it in some way. A common alternative for changing a class is to override some of its methods. However, this comes with even more trouble. That's why in general it's preferable to modify the structure of an object graph instead of the code in a class. It's better to replace parts than to change them.

8. Changing the behavior of services

8.1 Introduce constructor arguments to make behavior configurable

We've discussed earlier how services have to be created in one go, with all their dependencies and configuration values provided as constructor arguments. When it comes to changing the behavior of a service object, the constructor is again the place to be. Always prefer using a replaceable constructor argument to influence the behavior of a service.

Take for example the following `FileLogger` class which logs messages to a file:

```
final class FileLogger
{
    public function log($message): void
    {
        file_put_contents(
            '/var/log/app.log',
            $message,
            FILE_APPEND
        );
    }
}
```

To reconfigure the logger to log messages to another file, promote the log file path to a constructor argument which gets copied into a property:

```
final class FileLogger
{
    private string $filePath;

    public function __construct(string $filePath)
    {
        $this->filePath = $filePath;
    }

    public function log($message): void
    {
        file_put_contents($this->filePath, $message, FILE_APPEND);
    }
}

$logger = new FileLogger('/var/log/app.log');
```

8.2 Introduce constructor arguments to make behavior replaceable

We saw earlier how every dependency of a service should be injected as a constructor argument. Just like configuration values can be changed, these dependencies can also be replaced.

Consider the following `ParameterLoader`, which can be used to load a list of keys and values ("parameters") from a JSON file:

```
final class ParameterLoader
{
    public function load($filePath): array
    {
        /*
         * Load parameters from the file and merge add them
         * to the already loaded ones.
         */
        $rawParameters = json_decode(
            file_get_contents($filePath),
            true
        );

        $parameters = [];

        foreach ($rawParameters as $key => $value) {
            $parameters[] = new Parameter($key, $value);
        }

        return $parameters;
    }
}

$loader = new ServiceConfigurationLoader(
    __DIR__ . '/parameters.json'
);
```

What part of this class should be replaced in order to support loading an XML or maybe even a Yaml file instead? Most of the `ParameterLoader` is pretty generic, except for the call to `json_decode()`. To make this piece replaceable, we need to introduce an abstraction. This means, finding a more abstract concept than "decoding a JSON file", and introducing an interface which can represent the abstraction.

What we're basically looking for is a way to replace the "loading of a file" with a different implementation, which in concrete terms, will support (for now) "loading a JSON file". So we define an interface, and a class.

```
interface FileLoader
{
    /**
     * Load an array of key/value pairs representing parameters
     * stored in a file at the given location.
     */
    public function loadFile(string $filePath): array
}

final class JsonFileLoader implements FileLoader
{
    public function loadFile(string $filePath): array
    {
        Assertion::isFile($filePath);

        $result = json_decode(
            file_get_contents($filePath),
            true
        );

        if (!is_array($result)) {
            throw new RuntimeException(sprintf(
                'Decoding "%s" did not result in an array',
                $filePath
            ));
        }

        return $result;
    }
}
```

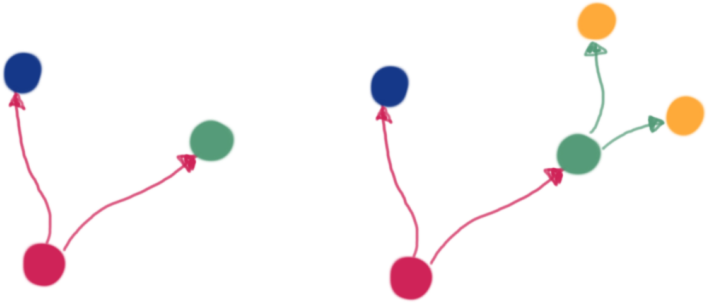
We've used the opportunity to add some pre-condition and even post-condition checks to make the JSON-specific implementation more reliable.

Now we make sure that `ParameterLoader` gets an instance of a `FileLoader` injected as a constructor argument, and we replace the existing file-loading code in `ParameterLoader` with a call to `FileLoader::loadFile()`:

```
final class ParameterLoader
{
    private FileLoader $fileLoader;

    public function __construct(FileLoader $fileLoader)
    {
        $this->fileLoader = $fileLoader;
    }

    public function load($filePath): array
    {
        // ...

        foreach (...) {
            if (...) {
                $rawParameters = $this->fileLoader->loadFile(
                    $filePath
                );
            }
        }

        // ...
    }
}

$parameterLoader = new ParameterLoader(new JsonFileLoader());
$parameterLoader->load(__DIR__ . '/parameters.json');
```

With part of the behavior of the `ParameterLoader` abstracted, we can replace it with any other *concrete* implementation, like an XML or Yaml file loader:

```
final class XmlFileLoader implements FileLoader
{
    // ...
}

$parameterLoader = new ParameterLoader(new XmlFileLoader());
$parameterLoader->load(__DIR__ . '/parameters.xml');
```



As long as the `FileLoader` dependency of `ParameterLoader` follows the contract defined by the interface, it doesn't matter what goes on behind the scenes of the real `FileLoader` that gets injected.

8.3 Compose abstractions to achieve more complicated behavior

With the proper abstraction in place, it'll be easy to compose multiple concrete instances into more complicated behavior. For instance, what if you want to support multiple formats based on the file name's extension? You could do something like this:

```php
/*
 * First we add an annotation the interface, indicating that loading
 * a file may throw a `CouldNotLoadFile` exception:
 */

interface FileLoader
{
    /**
     * ...
     *
     * @throws CouldNotLoadFile
     */
    public function loadFile(string $filePath): array
}

/*
 * Then we introduce a new `FileLoader` that is composed of multiple
 * other `FileLoader` instances. When asked to load a file, it will
 * delegate the call to all of the provided loaders, until one of
 * them _doesn't_ throw a `CouldNotLoadFile`.
 */

final class MultipleLoaders implements FileLoader
{
    private array $loaders;

    public function __construct(array $loaders)
    {
        Assertion::allIsInstanceOf($loaders, FileLoader::class);
        $this->loaders = $loaders;
    }

    public function loadFile(string $filePath): array
    {
        $lastException = null;
```

```
        foreach ($this->loaders as $loader) {
            try {
                return $loader->loadFile($filePath);
            } catch (CouldNotLoadFile $exception) {
                $lastException = $exception;
            }
        }

        throw new CouldNotLoadFile(sprintf(
            'None of the file loaders was able to load file "%s"',
            $filePath
        ), 0, $lastException ?? null);
    }
}
```

Note that the new logic is placed outside of `ParameterLoader` itself, which has no idea what's going on behind the `FileLoader` interface it uses.

Instead of trying different loaders, you may try other setups, for instance one where every loader can be registered for a particular file extension:

```
final class MultipleLoaders implements FileLoader
{
    private array $loaders;

    public function __construct(array $loaders)
    {
        Assertion::allIsInstanceOf($loaders, FileLoader::class);
        Assertion::allIsString(array_keys($loaders));
        $this->loaders = $loaders;
    }

    public function loadFile(string $filePath): array
    {
        /*
         * $this->loaders is supposed to be a map of keys and
         * values where the key is a file extension, and the
```

```
             * value is the `FileLoader` which should be used for
             * loading a file with that extension.
             */
            $extension = pathinfo($filePath, PATHINFO_EXTENSION);
            if (!isset($this->loaders[$extension])) {
                throw new CouldNotLoadFile(sprintf(
                    'There is no loader for file extension "%s"',
                    $extension
                ));
            }

            return $this->loaders[$extension]->loadFile($filePath);
        }
    }

    $parameterLoader = new ParameterLoader(
        new MultipleLoaders([
            'json' => new JsonFileLoader(),
            'xml' => new XmlFileLoader()
        ]);
    );
    $parameterLoader->load('parameters.json');
    $parameterLoader->load('parameters.xml');

    // This will throw a `CouldNotLoadFile` exception:
    $parameterLoader->load('parameters.yml');
```

As you can see, this setup is now very dynamic. However, always keep in mind that, if you're writing this code for your project, you don't usually have to support all these different configuration file formats. So introducing the `FileLoader` abstraction will be a smart thing to do, but writing all these different loader implementations should be considered generalization-before-it's-needed. Until it's needed...

8.4 Decorate existing behavior

Keep in mind the above example where we have multiple file loaders for JSON, XML, etc., which all return an array of raw parameters (key/value pairs). What if we want to allow the user to use environment variables as the values for these parameters? We wouldn't want to copy this replacement logic into all the `FileLoader` implementations that we have. Instead, we want to add the behavior on top of any existing behavior. We can do this using a particular style of composition—*Decoration*:

```
final class ReplaceParametersWithEnvironmentVariables
    implements FileLoader
{
    private FileLoader $fileLoader;
    private array $envVariables;

    public function __construct(
        FileLoader $fileLoader,
        array $envVariables
    ) {
        /*
         * We get the "real" file loader injected as a constructor
         * argument:
         */
        $this->fileLoader = $fileLoader;
        $this->envVariables = $envVariables;
    }

    public function loadFile(string $filePath)
    {
        // We use the real file loader to load the file:
        $parameters = $this->fileLoader->loadFile($filePath);

        /*
         * Before returning the parameters, we replace any
         * environment variable used as parameter value:
         */
```

```
        foreach ($parameters as $key => $value) {
            $parameters[$key] = $this->replaceWithEnvVariable(
                $value
            );
        }

        return $parameters;
    }

    private function replaceWithEnvVariable(string $value): string
    {
        if (isset($this->envVariables[$value])) {
            return $this->envVariables[$value];
        }

        return $value;
    }
}

$parameterLoader = new ParameterLoader(
    new ReplaceParametersWithEnvironmentVariables(
        new MultipleLoaders([
            'json' => new JsonFileLoader(),
            'xml' => new XmlFileLoader()
        ]),
        [
            'APP_ENV' => 'dev',
        ]
    )
);
```

Another common scenario for decoration is when the cost of using the real service is somewhat high. For instance, if the application has to load and parse the `parameters.json` file many times, it may be smart to wrap the original service, and remember the last result it returned:

```
final class CachedFileLoader implements FileLoader
{
    private FileLoader $realLoader;

    private $cache = [];

    public function __construct(FileLoader $realLoader)
    {
        $this->realLoader = $realLoader;
    }

    public function loadFile(string $filePath): array
    {
        if (isset($this->cache[$filePath])) {
            /*
             * We've loaded this file before, and we can just return
             * the cached result:
             */
            return $this->cache[$filePath];
        }

        // We didn't load this file before, so we do it now:
        $result = $this->realLoader->loadFile($filePath);

        /*
         * We keep the result in our "cache", so we don't have to
         * load the file again next time:
         */
        $this->cache[$filePath] = $result;

        return $result;
    }
}

$loader = new CachedFileLoader(new JsonFileLoader());
```

```
// This will forward the call to `JsonFileLoader`
$loader->load('parameters.json');

// The second time we won't hit the filesystem:
$loader->load('parameters.json');
```

The advantage of using composition in this scenario is that the caching logic doesn't have to be duplicated across the different file loader implementations. In fact, the logic in `CachedFileLoader` is agnostic of the actual `FileLoader` implementation that's being used. This means you can test it separately, and you can also develop it separately; if you want to make the caching logic more advanced, you only have to change the code of this single class that's dedicated to caching.

8.5 Use notification objects or event listeners for additional behavior

We already looked at using event listeners as a technique for separating the main job of a command method from its secondary tasks. If you want to reconfigure services to do other things than they did before, you could use the same technique.

As an example, take the `ChangeUserPassword` service:

```
final class ChangeUserPassword
{
    private PasswordEncoder $passwordEncoder;

    public function __construct(
        PasswordEncoder $passwordEncoder,
        // ...
    ) {
        // ...
    }

    public function changeUserPassword(
        UserId $userId,
```

```
        string $plainTextPassword
    ): void {
        $encodedPassword = $this->passwordEncoder->encode(
            $plainTextPassword
        );

        // Store the new password...
    }
}
```

A new requirement for this service is that it should also send an email to the user afterwards, to tell them that their password has changed (just in case it was a "hacker" who did it). Instead of adding more code to the existing class and method, this could be a nice opportunity for dispatching an event and set up a listener for it, that will actually send that email.

```
// First we define a new event type:

final class UserPasswordChanged
{
    private UserId $userId;

    public function __construct(UserId $userId)
    {
        $this->userId = $userId;
    }
}

// Then we define a listener for this event:

final class SendUserPasswordChangedNotification
{
    // ...

    public function whenUserPasswordChanged(
        UserPasswordChanged $event
```

```
    ): void {
        // Send the email...
    }
}
```

Finally, we have to rewrite the `ChangeUserPassword` service to dispatch our newly defined `UserPasswordChanged` event:

```
final class ChangeUserPassword
{
    private EventDispatcher $eventDispatcher;

    public function __construct(
        // ...,
        EventDispatcher $eventDispatcher
    ) {
        // ...
    }

    public function changeUserPassword(
        UserId $userId,
        string $plainTextPassword
    ): void {
        $encodedPassword = $this->passwordEncoder->encode(
            $newPassword
        );

        // Store the new password

        $this->eventDispatcher->dispatch(
            new UserPasswordChanged($userId)
        );
    }
}

/*
```

```
 * We have to make sure that the listener is registered in
 * the correct way:
 */

$listener = new SendUserPasswordChangedNotification(...);
$eventDispatcher = new EventDispatcher([
    UserPasswordChanged::class => [
        $listener,
        'whenUserPasswordChanged'
    ]
]);

$service = new ChangeUserPassword(..., $eventDispatcher);

/*
 * This will cause a `UserPasswordChanged` event to be dispatched to
 * the `SendUserPasswordChangedNotification` listener:
 */
$service->changeUserPassword(new UserId(...), 'Test123');
```

The advantage of using an event dispatcher is that it enables you to add new behavior to a service without modifying its existing logic. Once it's in place, an event dispatcher will keep offering the option to add new behavior. You can always register another listener for an existing event.

A disadvantage of using an event dispatcher is that it has a very generic name. When reading the code, it's not very clear what's going on behind that call to `dispatch()`. It can also be a bit difficult to figure out which listeners will respond to a certain event. An alternative solution in that case is to introduce your own abstraction.

As an example, take the following class that imports CSV files from a given directory and dispatches events to allow other services to listen in on the import process:

```
final class Importer
{
    private EventDispatcher $dispatcher;

    public function __construct(EventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function import(string $csvDirectory): void
    {
        foreach (Finder::in($csvDirectory)->files() as $file) {
            // Read the file
            $lines = ...;

            foreach ($lines as $index => $line) {
                if ($index === 0) {
                    // Parse the header
                    $header = ...;

                    $this->dispatcher->dispatch(
                        new HeaderImported($file, $header)
                    );
                }
                else {
                    $data = ...;

                    $this->dispatcher->dispatch(
                        new LineImported($file, $index)
                    );
                }
            }

            $this->dispatcher->dispatch(
                new FileImported($file)
            );
```

```
        }
    }
}
```

It turns out that every one of these events has just one listener, one that will write some debug information about the event to a log file. Although this is a very simple task, we have to maintain lots of code for it: we have all these event and event listener classes that we have to write, and then we have to remember to register the listeners in the correct way too. As we know now, most of these listeners do the same kind of job anyway, so instead of spreading this behavior across many classes, we might as well combine it in a single class, and introduce our own abstraction for it:

```
interface ImportNotifications
{
    public function whenHeaderImported(
        string $file,
        array $header
    ): void;

    public function whenLineImported(
        string $file,
        int $index
    ): void;

    public function whenFileImported(
        string $file
    ): void;
}

final class ImportLogging implements ImportNotifications
{
    private Logger $logger;

    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
```

```
    }

    public function whenHeaderImported(
        string $file,
        array $header
    ): void {
        $this->logger->debug('Imported header ...');
    }

    // and so on
}
```

Instead of injecting the event dispatcher into the `Importer` class, we can now inject an instance of `ImportNotifications`. And instead of calling `dispatch()`, we should now make a call to the dedicated "event method" on the injected `ImportNotifications` instance:

```
final class Importer
{
    private ImportNotifications $notify;

    public function __construct(ImportNotifications $notify)
    {
        $this->notify = $notify;
    }

    public function import(string $csvDirectory): void
    {
        foreach (Finder::in($csvDirectory)->files() as $file) {
            // Read the file
            $lines = ...;

            foreach ($lines as $index => $line) {
                if ($index === 0) {
                    // Parse the header
                    $header = ...;
```

```
                    $this->notify->whenHeaderImported(
                        $file,
                        $header
                    )
                }
                else {
                    $data = ...;

                    $this->notify->whenLineImported($file, $index);
                }
            }

            $this->notify->whenFileImported($file);
        }
    }
}
```

If besides logging you also want to output the debug information to the screen, you can easily do that in the same class. Or you can add another class, and use object composition again, to invoke both behaviors instead of just one.

8.6 Don't use inheritance to change an object's behavior

Taking another look at the `ParameterLoader` example we discussed earlier; what if the original class would've looked like this:

```
class ParameterLoader
{
    public function load($filePath): array
    {
        // ...

        $rawParameters = $this->loadFile($filePath);

        // ...

        return $parameters;
    }

    protected function loadFile(string $filePath): array
    {
        return json_decode(
            file_get_contents($filePath),
            true
        );
    }
}
```

The key differences are:

1. The `ParameterLoader` class isn't marked as `final`, meaning that it's possible to define a subclass, which extends `ParameterLoader`.
2. There is now a dedicated method for loading the file, and this method is `protected`, meaning it can be overridden by such a subclass.

With the class internals fully exposed, it's now a possibility to extend the class, inherit the core logic, and override the file loading part, to make it deal with XML:

```
final class XmlFileParameterLoader extends ParameterLoader
{
    protected function loadFile(string $filePath): array
    {
        $rawXml = file_get_contents($filePath);

        // convert to array somehow

        return ...;
    }
}
```

As you can imagine, this solution doesn't come with all the benefits of the previous one, like the "file loader" abstraction itself, which came with very clean options for composition, supporting multiple file loaders at once, etc. The alternative solution where we extend from the existing `ParameterLoader` class itself, doesn't come with any of this flexibility and reconfigurability.

In fact, class inheritance as a means for changing the behavior of an existing object actually comes with many downsides:

- Subclass and parent class become tied together. Changing implementation details that would normally be hidden behind the public interface of the class could now break the implementation of a subclass. Consider what would happen if that `protected` method's name was changed, or if it got an extra required parameter.
- Subclasses can override `protected` but also `public` methods. They gain access to `protected` properties and their data types, which have so far been internal information. In other words: a lot of the internals of the object are now exposed.

What if, instead, the parent class offered a so-called *Template* method, and allowed the implementer to only provide that method, not exposing any more internals than needed?

```php
abstract class ParameterLoader
{
    /*
     * Mark all properties "private" to keep them private to the
     * parent class.
     *
     * Mark all methods "final" to make it impossible to override
     * them.
     */

    final public function load($filePath): array
    {
        $parameters = [];

        foreach (...) {
            // ...
            if (...) {
                $rawParameters = $this->loadFile($filePath);
                // ...
            }
        }

        return $parameters;
    }

    /*
     * Then only allow one method to be implemented (not
     * overridden):
     */

    abstract protected function loadFile(string $filePath): array;
}
```

This is better, but still not optimal. We may not have the downsides of inheritance anymore, as listed earlier, but we don't have the endless possibilities of using composition either. Besides, looking at what's going on in this example, we can generalize and

claim that: everything that can be done with the *Template* method pattern can also be achieved with composition. The only thing you need to do is promote the `abstract protected` method to a regular `public` method on an injected object. Then we can also make the class itself `final` again. In the case of our `ParameterLoader` we already did that earlier:

```
final class ParameterLoader
{
    private FileLoader $fileLoader;

    public function __construct(FileLoader $fileLoader)
    {
        $this->fileLoader = $fileLoader;
    }

    final public function load($filePath): array
    {
        $parameters = [];

        foreach (...) {
            // ...
            if (...) {
                /*
                 * Use the public `loadFile()` method of the
                 * injected `FileLoader` here, instead of the
                 * protected `loadFile()` method we had earlier:
                 */
                $rawParameters = $this->fileLoader->loadFile(
                    $filePath
                );

                // ...
            }
        }

        return $parameters;
```

```
    }
}
```

Since many projects still don't mark their classes as "final" by default, you will encounter many frameworks and libraries that allow the behavior of their objects to be modified by extending their classes. Please refrain from doing so. Always choose a solution that uses only `public` methods, preferably those that are part of the published interface of a class. Don't rely on class internals by inheriting them from another class. By doing so, you'd be making your solutions more fragile, because you're relying on things that are more likely to change the published, supported API offered by the framework or library.

When is it okay to use inheritance?

Broadly speaking, inheritance should only be used to define a strict hierarchy of types. Like, a content block can be either a paragraph, or an image. In that case you could write: `Paragraph extends ContentBlock` and `Image extends ContentBlock`. In practice I rarely find a good case for using inheritance. It usually turns out a bit awkward, "forced", and very soon it starts to get in the way.

Usually, inheritance is used for code reuse. In this case too, composition is a much more powerful form of code reuse. However, some object types like entities or value objects don't support dependency injection, so you can't really achieve code reuse via that road. In that case I recommend using traits. Traits aren't inheritance, because the name of the trait doesn't end up becoming part of the class' hierarchy, like a parent class or an interface would. A trait is plain code reuse; a compiler-level copy/paste of code.

If, for example, you'd want to record domain events in all your entities. You could define the following interface for entities, to make sure that they all have methods for retrieving those events, and for clearing them after you've dispatched them:

```
interface RecordsEvents
{
    public function releaseEvents(): array;

    public function clearEvents(): void;
}
```

Because all entities will have the same implementation for these methods, and you don't want to manually copy/paste that implementation into all of the entity classes, you could use a `trait`:

```
trait EventRecordingCapabilities
{
    private array $events;

    private function recordThat(object $event): void
    {
        $this->events[] = $event;
    }

    public function releaseEvents(): array
    {
        return $this->events;
    }
```

8.7 Mark classes as final by default

For services we already made the case: changing behavior by using object composition, instead of inheritance, is the better, more flexible way. If you go this way, there's no need to allow a class to be extended at all. These objects can keep their internals to themselves, and only allow clients to use behavior that is part of their public interface. That means, every class can, and should be marked `final`. This will make it clear to the client that the class isn't meant to be extended, its methods aren't meant to be overridden. And it will force its users to look for better ways to change its behavior.

For other types of objects, like entities and value objects, the question should be asked too: do they have to be `final` too? Yes, they do. These objects represent domain concepts and the knowledge you have gained about them. It would be weird to override part of the behavior of these classes by extending from them. If you've learned something about the domain that makes you want to change the behavior of an entity, you shouldn't create a subclass to change its behavior, but change the entity itself.

The only exception to the rule is when you want to declare a hierarchy of objects. In that case, extending from a parent class can indicate the relation between these objects: the subclass should be considered a special case of the parent class. Then the parent class won't be `final`, because the subclass has to be able to extend it.

8.8 Mark methods and properties private by default

So far all the examples in this book have shown `final` classes with `private` properties. As soon as you mark your classes `final`, you may notice that there is no need to have `protected` properties anymore. Classes generally won't be used to extend from, so they can keep all of their internals really to themselves. The only way in which clients can interact with an object is by constructing it and by calling public methods on it. By closing down the class definition itself, you can design some really strong objects. The freedom to change anything about the object's internals, as long as it doesn't break the contract defined by its published interface, is a big win.

8.9 Summary

When you feel the need to change the behavior of a service, look for ways to make this behavior configurable through constructor arguments. If this isn't an option because

you want to replace a larger piece of logic, look for ways to swap out dependencies, which are also passed in as constructor arguments.

If the behavior you want to change isn't represented by a dependency yet, extract one by introducing an abstraction: a higher level concept, and an interface. You will then have a part you can replace, instead of modify. Abstraction offers the ability to compose and decorate behaviors, so they can become more complicated, without the initial service knowing about it (or being modified for it).

Don't use inheritance for changing the behavior of a service by overriding its methods. Always look for solutions that use object composition. In fact, completely close all your classes down for inheritance: mark them as `final` and make all properties and methods `private`, unless they are part of the public interface of the class.

A field guide to objects

So far we've discussed style guidelines for object design. These are meant to be general-purpose rules, that can be applied everywhere. This doesn't mean that all the objects in your application will look the same. Some objects will have lots of query methods, some will have only command methods. Some will have a mix of both, but with a certain ratio of them. You may find that different types of objects often share certain characteristics, which results in "pattern names" to be invented for them. For instance, developer will talk about "entities", "value objects", or "application services", to indicate the *nature* of the object they're talking about.

The last part of this book is a discussion of some of these common types of objects that you may find in an application, and how you can recognize them in their natural habitat. In this sense, the following chapters form a "field guide" for objects. If you find an object that doesn't really fit into a certain category, this guide may help you decide whether or not the object should be redesigned to fit in better with the rest of their species. On the other hand, if you encounter an object that doesn't look like any of the objects described in this chapter, don't worry. As long as it abides by the guidelines for object design in this book, it's perfectly fine.

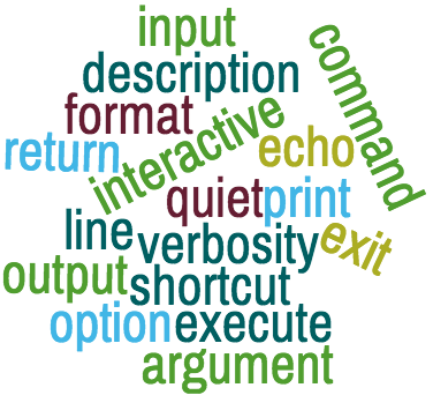
9. Controllers

In an application there's always some sort of *front controller*. For web applications this is where all the requests come in (e.g. `index.php`, `app.php`, etc.). Based on the request URI, its method, and headers, etc. the call will be forwarded to a *controller*, where the application can do anything it needs to do before it can return a proper response. For command-line (CLI) applications, the "front controller" would be the executable you'd call, e.g. `bin/console`, `artisan`, etc. Based on the arguments that the user provides, the call will be forwarded to something like a *command* object, where the application can perform the task requested by the user.

Though they are technically quite different, console commands are conceptually quite similar to web controllers. They both do work that was requested from outside the application, by a person, or some other application, that sent a web request, or ran the console application. So let's call both console commands and web controllers "controllers".

Controllers typically have code that reveals where the call came from. You'll find mentions of a `Request` object, or request parameters, forms, HTML templates, a *session* maybe, or *cookies*. All of these are web concepts. The classes used here, often originate from the web framework that your application uses.

Other controllers mention command-line arguments, options or flags, and contain code for outputting lines of text to the terminal, and formatting them in ways that the terminal can understand. These are all signs for the reader that this class takes input from and produces output for the command-line.



Because controllers talk about the particular delivery mechanism that initiated a call to them (the web, the terminal), controllers should be considered *infrastructure* code. They facilitate the connection between the client—who lives in *the world outside*—and the *core* of the application.

When a controller has examined the input provided to it, it will take whatever information it needs, and then call either an *application service* or a *read model repository*. An application service will be called when the controller is supposed to produce some kind of effect. For instance, when it's supposed to make a change to the application's state, or to send out an email, etc. A read model repository will be used if the controller is supposed to give back some information that the client requested.

An object is a controller if...

- A front controller calls it, and it's therefore one of the entry points for the graph of services and their dependencies.
- It contains infrastructure code that reveals what the delivery mechanism is.
- It makes calls to an application service or a read model repository (or both).

A typical web controller would look something like this (the framework used in these examples is Symfony[1]):

```
namespace Infrastructure\UserInterface\Web;

use Infrastructure\Web\Form\ScheduleMeetupType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

final class MeetupController extends AbstractController
{
    public function scheduleMeetupAction(Request $request): Response
    {
        $form = $this->createForm(ScheduleMeetupType::class);

        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            // ...

            return new RedirectResponse(
                '/meetup-details/' . $meetup->meetupId()
            );
        }

        return $this->render(
            'scheduleMeetup.html.twig',
            [
                'form' => $form->createView()
            ]
        );
    }
}
```

[1] https://symfony.com/

The alternative for the command line might look something like this:

```
namespace Infrastructure\UserInterface\Cli;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

final class ScheduleMeetupCommand extends Command
{
    protected function configure()
    {
        $this
            ->addArgument('title', InputArgument::REQUIRED)
            ->addArgument('date', InputArgument::REQUIRED)
            // ...
        ;
    }

    public function execute(
        InputInterface $input,
        OutputInterface $output
    ) {
        $title = $input->getArgument('title');
        $date = $input->getArgument('date');

        // ...

        $output->writeln('Meetup scheduled');
    }
}
```

10. Application services

An application service represents the task to be performed. It gets any dependency injected as a constructor argument. All the relevant data that's needed to perform the task, including contextual information like the logged in user ID, or the current time, will be provided as method arguments. When the data originates from the client itself, it will be primitive-type data. That way, the controller can provide the application service with the data as it was sent by the client, without converting it first.

The code of an application service should read like a recipe, with all the steps required to do the job. For instance: take out an object from this write model repository, call a method on it, and save it again. Or: collect some information from this read model repository, and send a report to a certain user.

An object is an application service if...

- It performs a single task.
- It contains no infrastructure code, that is, it doesn't deal with the web request itself, or SQL queries, or the filesystem, etc.
- It describes a single use case that the application should have. It will often correspond one-to-one with a feature request from a stakeholder. E.g. it should be possible to add a product to the catalog, to cancel an order, to send a delivery note to a customer, etc.

The web controller and console handler we saw earlier will take the data from the request (via a form), or from the command-line arguments, and provide it to the application service, which looks something like this:

```php
namespace Application\ScheduleMeetup;

use Domain\Model\Meetup\Meetup;
use Domain\Model\Meetup\MeetupRepository;
use Domain\Model\Meetup\ScheduleDate;
use Domain\Model\Meetup\Title;

final class ScheduleMeetupService
{
    private MeetupRepository $meetupRepository;

    public function __construct(MeetupRepository $meetupRepository)
    {
        $this->meetupRepository = $meetupRepository;
    }

    /*
     * The application service receives primitive-type arguments:
     */

    public function schedule(
        string $title,
        string $date,
        UserId $currentUserId
    ): MeetupId {
        /*
         * It converts these primitive-type values to value objects,
         * and instantiates a new `Meetup` entity using these
         * objects:
         */

        $meetup = Meetup::schedule(
            $this->meetupRepository->nextIdentity(),
            Title::fromString($title),
            ScheduledDate::fromString($date),
            $currentUserId
```

```
        );

        /*
         * It then saves the meetup to the write model repository:
         */

        $this->meetupRepository->save($meetup);

        /*
         * Finally, it returns the identifier of the new `Meetup`:
         */

        return $meetup->meetupId();
    }
}
```

Sometimes application services are called “command handlers”. They will still be application services though. Instead of invoking an application service using primitive-type arguments, you can also call it by providing a “command” object, which represents the client’s request in a single object. Such an object is called a *data transfer object* (DTO), because it can be used to carry the data provided by the client, and transfer it as one thing from controller to application service. This should be a simple, easy to construct object, and contain only primitive-type values, simple lists, and optionally other DTOs if some sort of hierarchy is required.

```
namespace Application\ScheduleMeetup;

/*
 * This is the command that contains the data needed to perform the
 * task of "scheduling a meetup:
 */

final class ScheduleMeetup
{
    public string $title;
    public string $date;
```

```
}

final class ScheduleMeetupService
{
    // ...

    /*
     * The application service could then take the data from the
     * command object:
     */

    public function schedule(
        ScheduleMeetup $command,
        UserId $currentUserId
    ): MeetupId {
        $meetup = Meetup::schedule(
            $this->meetupRepository->nextIdentity(),
            Title::fromString($command->title),
            ScheduledDate::fromString($command->date),
            $currentUserId
        );

        // ...
    }
}
```

The advantage of using a dedicated command object is that it's easy to map for instance a JSON request body onto such an object. It also works well with form libraries, which can map submitted data directly onto command DTO properties.

11. Write model repositories

Often an application service makes a change to the application's state, and most often this means that a domain object has to be modified and persisted. It uses an abstraction for this: a *repository*. To be more specific: a *write model repository*, because it's only concerned with retrieving an entity and storing the changes that are made to it.

The abstraction itself will be an interface that the application service gets injected as a dependency. This interface doesn't expose any details about *how* exactly the object is going to be persisted. It just offers some general-purpose methods like `getById()`, `save()`, or `add()` and `update()` if you wish. A corresponding implementation will fill in the details, like: which SQL queries will be issued, or which ORM will be used to map the object to a row in the database.

An object is a write model repository if...

- It offers methods for retrieving an object from storage, and for saving it,
- Its interface hides the underlying technology that's been used.

As an example, this is the `MeetupRepository` that the application service we saw earlier relies on:

```
namespace Domain\Model\Meetup;

interface MeetupRepository
{
    public function save(Meetup $meetup): void;

    public function nextIdentity(): MeetupId;

    /**
```

```
     * @throws MeetupNotFound
     */
    public function getById(MeetupId $meetupId): Meetup
}

namespace Infrastructure\Persistence\DoctrineOrm;

/*
 * The default implementation of `MeetupRepository` uses Doctrine
 * ORM:
 */

use Doctrine\ORM\EntityManager;
use Domain\Model\Meetup\Meetup;
use Domain\Model\Meetup\MeetupId;
use Ramsey\Uuid\UuidFactoryInterface;

final class DoctrineOrmMeetupRepository implements MeetupRepository
{
    private EntityManager $entityManager;
    private UuidFactoryInterface $uuidFactory;

    public function __construct(
        EntityManager $entityManager,
        UuidFactoryInterface $uuidFactory
    ) {
        $this->entityManager = $entityManager;
        $this->uuidFactory = $uuidFactory;
    }

    public function save(Meetup $meetup): void
    {
        $this->entityManager->persist($meetup);
        $this->entityManager->flush($meetup);
    }
```

```
    public function nextIdentity(): MeetupId
    {
        return MeetupId::fromString(
            $this->uuidFactory->uuid4()->toString()
        );
    }

    // ...
}
```

12. Entities

The objects that are being persisted will be the ones the user cares about, the ones that should be remembered, even when the application has to be restarted. These are the application's *entities*.

Entities represent the domain concepts of the application. They contain relevant data and offer useful behavior on this data. In terms of object design, they will often have named constructors, because this allows you to use domain-specific names for "creating" this particular kind of entity. They will also have modifier methods, which are command methods that change the entity's state. Entities will have only a few, if any, query methods. Retrieving information is usually delegated to a particular kind of object, called a query object. We'll get back to this.

Proper entities

Just like any object, an entity protects itself fiercely against ending up in an invalid state. This means that many entities seen in the wild shouldn't be considered proper entities according to this definition.

When a state change is allowed, an entity usually produces a domain event, representing the change. These events can be used to find out what exactly has changed, and to announce this change to other parts of the application that want to respond to it.

An object is an entity if...

- It has a unique identifier.
- It has a life-cycle.
- It will be persisted by a write model repository and can later be retrieved from it.
- It uses named constructors and command methods to provide the user with ways to instantiate it, and manipulate its state.
- It produces domain events when it gets instantiated or modified.

13. Value objects

Value objects are wrappers for primitive-type values, adding meaning, and useful behavior to these values. We've discussed them in detail earlier. In the context of the journey from controller to application service to repository, it should be noted that it's often the application service that instantiates them, and then passes them as arguments to the constructor or a modifier method of an entity. Therefore they end up being used or stored inside the entity.

However, it's good to remember that value objects aren't only meant to be used in combination with entities. They can be used in any place, and a value object is in fact a preferred way of passing around values.

An object is a value object...

- If it's immutable.
- If it wraps primitive-type data.
- If it adds meaning by using domain-specific terms (e.g. this isn't just an `int`, it's a `Year`).
- If it imposes limitations by means of validation (e.g. this isn't just any string, it's a string with an `'@'` in it).
- It acts as an attractor of useful behavior related to the concept (e.g. `Position::toTheLeft(int $steps)`).

The `Meetup` entity we saw earlier, and its related value objects and domain events look something like this:

```
namespace Domain\Model\Meetup;

final class Meetup
{
    private array $events = [];

    private MeetupId $meetupId;
    private Title $title;
    private ScheduledDate $scheduledDate;
    private UserId $userId;

    private function __construct()
    {
    }

    public static function schedule(
        MeetupId $meetupId,
        Title $title,
        ScheduledDate $scheduledDate,
        UserId $userId
    ): Meetup {
        $meetup = new self();

        $meetup->meetupId = $meetupId;
        $meetup->title = $title;
        $meetup->scheduledDate = $scheduledDate;
        $meetup->userId = $userId;

        $meetup->recordThat(
            new MeetupScheduled(
                $meetupId,
                $title,
                $scheduledDate,
                $userId
            );
        );
```

```
        return $meetup;
    }

    /*
     * The following methods are examples of other behavior that
     * this `Meetup` entity could offer:
     */

    public function reschedule(ScheduledDate $scheduledDate): void
    {
        // ...

        $this->recordThat(
            new MeetupRescheduled($this->meetupId, $scheduledDate)
        );
    }

    public function cancel(): void
    {
        // ...
    }

    // ...

    private function recordThat(object $event): void
    {
        $this->events[] = $event;
    }

    public function releaseEvents(): array
    {
        return $this->events;
    }

    public function clearEvents(): void
```

```
    {
        $this->events = [];
    }
}

final class Title
{
    private string $title;

    private function __construct(string $title)
    {
        Assertion::notEmpty($title);
        $this->title = $title;
    }

    public static function fromString(string $title)
    {
        /*
         * We might as well have used a regular public constructor
         * here...
         */

        return new self($title);
    }

    /*
     * This is an example of a piece of useful behavior which value
     * objects tend to "attract":
     */

    public function abbreviated(string $ellipsis = '...'): string
    {
        // ...
    }
}
```

```php
final class MeetupId
{
    private string $meetupId;

    private function __construct(string $meetupId)
    {
        Assertion::uuid($meetupId);
        $this->meetupId = $meetupId;
    }

    public static function fromString(string $meetupId)
    {
        return new self($meetupId);
    }
}
```

14. Event listeners

We already encountered domain events. They can be used to notify services about things that have happened inside the write model. These other services could then perform secondary actions, after the primary work has been done. Since application services are the ones that perform these primary tasks, domain events can be used to notify other services *after* the application service is done. They can also do it at the last moment, just before returning. At that point, an application service could fetch the recorded events from the entity it has modified, and hand them over to the *event dispatcher*:

```
final class RescheduleMeetupService
{
    private EventDispatcher $dispatcher;

    public function __construct(
        // ...
        EventDispatcher $dispatcher
    ) {
        $this->dispatcher = $dispatcher
    }
    public function reschedule(MeetupId $meetupId, ...): void
    {
        $meetup = ...;

        $meetup->reschedule(...);

        /*
         * Dispatch any event that has been recorded inside the
         * `Meetup` entity:
         */

        $dispatcher->dispatch($meetup->recordedEvents());
```

```
    }
}
```

Internally, the dispatcher will forward all events to services called “listeners” or “subscribers”, which have been registered for particular types of events.

An event listener could then perform secondary actions, for which it may even call another application service. It can actually use any other service it needs, for instance to send notification emails about the domain event that has just occurred. Take for example the `NotifyGroupMembers` listener, which will notify group members when a meetup has been rescheduled:

```
/*
 * A convenient naming standard for event listeners is the name of
 * the thing you're going to do (e.g. "notify group members"). The
 * methods will then point out the reasons for doing so (e.g. "when
 * meetup rescheduled").
 */

final class NotifyGroupMembers
{
    public function whenMeetupRescheduled(
        MeetupRescheduled $event
    ): void {
        /*
         * Send an email to group members using the information from
         * the event object.
         */

        // ...
    }
}
```

15. Read models and read model repositories

As mentioned earlier, a controller could invoke an application service to perform a task, but it may also invoke a *read model repository* to retrieve information from. Such a repository will return objects. These objects aren't meant to be manipulated, but to read information from. Earlier we called them "query objects"; they have only query methods, meaning that their state can't be influenced by its users.

When the call to the read model repository happens inside the controller, the read model that's returned could be passed on to the template renderer, which generates an HTML response using it. Or it could just as easily be used to generate a JSON-encoded response to an API call. In all these cases, the read model is specifically designed to fit the response that is going to be generated. All the data required for a particular use case should be available inside the read model, and no extra queries should have to be made. Such a read model is a DTO, because it's going to be used to *transfer* data from the core of the application back to the world outside. So the values that can be retrieved from such a read model should have primitive types.

As an example, take the following read model repository which returns a list of upcoming meetups. It serves a specific use case, and only contains the data required to render a simple list:

```
namespace Application\UpcomingMeetups;

/*
 * `UpcomingMeetup` is a read model (or "view model") - a DTO which
 * carries relevant data about upcoming meetups, to be shown in a
 * list on a web page:
 */

final class UpcomingMeetup
{
```

```
    public string $title;
    public string $date;
}

/*
 * It comes with a repository which returns instances of
 * `UpcomingMeetup` and can be used by, for instance, a web
 * controller. Once it has retrieved these `UpcomingMeetup` objects,
 * it can pass them along to a template renderer, and the template
 * can just render the data that's been provided to it:
 */

interface UpcomingMeetupRepository
{
    /**
     * @return UpcomingMeetup[]
     */
    public function upcomingMeetups(DateTimeImmutable $today): array
}

namespace Infrastructure\ReadModel;

use Application\UpcomingMeetups\UpcomingMeetupRepository;
use Doctrine\DBAL\Connection;

final class UpcomingMeetupDoctrineDbalRepository implements
    UpcomingMeetupRepository
{
    private Connection $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function upcomingMeetups(DateTimeImmutable $today): array
```

```
    {
        /*
         * This particular implementation of
         * `UpcomingMeetupRepository` uses the database directly to
         * fetch the data from. It then creates instances of the
         * `UpcomingMeetup` read model:
         */

        $rows = $this->connection->...;

        return array_map(
            function (array $row) {
                $upcomingMeetup = new UpcomingMeetup();
                $upcomingMeetup->title = $row['title'];
                $upcomingMeetup->date = $row['date'];

                return $upcomingMeetups;
            },
            $rows
        );
    }
}
```

An application service itself can also use a read model repository to retrieve some information. It then uses the information to make decisions based on it, or take further actions. A read model that's used by an application service is often a "smarter" read model than the one that's used to generate a response. It uses proper value objects for its return values, instead of primitive-type values, so the application service doesn't have to worry about the validity of the read model. It often feels like such a read model is itself a write model, except, there's no way to make changes to it; it's a query object after all.

As for the read model repositories themselves, they should be separated into an abstraction and a concrete implementation. Just like with write model repositories, there will be an interface which offers one or more query methods that can be used to retrieve the read models. The interface doesn't give a hint about the underlying storage mechanism for these models.

An object is a read model...

- If it has only query methods, i.e. it's a query object (and is therefore immutable).
- If it's designed specifically for a certain use case.
- If all the data needed (and no more) becomes available the moment you retrieve the object.

An object is a read model repository...

- If it has query methods that conform to a specific use case and will return read models, which are also specific for that use case.

Note that the distinction between a read model repository and a regular service which returns a piece of information isn't that clear. For example, consider the situation where an application service needs an exchange rate to convert some monetary value to a foreign currency. You might say that a service that can provide such information is basically a repository, from which you can get the exchange rate for a given currency conversion—such a service has access to a "collection" of exchange rates that's defined in some place we "don't care about". Still, this service which can provide you with an exchange rate, could also be considered a regular service, and may just as well be called `ExchangeRateProvider` or something like that.

The main idea is that for all these services, you need an abstraction and a concrete implementation, because the abstraction describes what you're looking for, and the implementation describes how you can get it.

```
namespace Application\ExchangeRates;

/*
 * The abstraction is the interface which represents the question
 * we're asking:
 */

interface ExchangeRateProvider
{
    public function getRateFor(
        Currency $from,
        Currency $to
    ): ExchangeRate
}

/*
 * The types of the return values used by the interface are also
 * part of the abstraction, because we care about how we can use
 * these values, but not about how their data ends up in them:
 */

final ExchangeRate
{
    // ...
}
```

In terms of their design, some objects aren't very different from others. For example, domain events look a lot like value objects. They are immutable objects holding data that belongs together. The difference between a domain event and a value object is how and where it's used: a domain event will be created and recorded inside an entity and later dispatched; a value object models an aspect of the entity.

16. Abstractions, concretions, layers, and dependencies

So far we've encountered different types of objects that you can find in your average web or console application. Besides certain characteristics, like the types of methods these objects have, what kind of information they expose, or what kind of behavior they offer, we should also consider whether they are *abstract* or *concrete*, and in which ways these objects are *dependent* on each other.

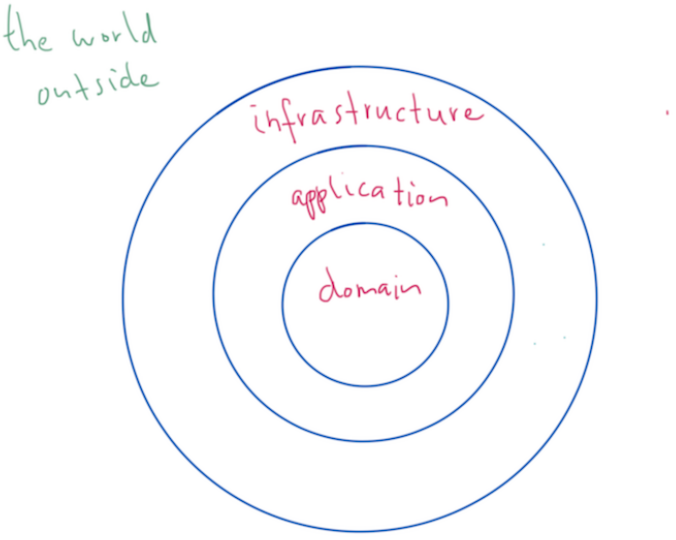
In terms of abstraction, we can define the following character traits for the object types we've discussed so far:

- Controllers are *concrete*. They are often coupled to a particular framework and are specific for the delivery mechanism. They don't have, nor need an interface. The only time you'd want to offer an alternative implementation is when you switch frameworks. In that case you'd want to rewrite these controllers instead of creating a second implementation for them.
- Application services are *concrete*. They represent a very specific use case of your application. If the story of a use case changes, the application service itself changes. So they don't have an interface.
- Entities and value objects are *concrete*. They are the specific result of the developer's understanding of the domain. These types of objects *evolve* over time. We don't provide an interface for them. The same goes for read model objects. We define and use them as they are, never through an interface.
- Repositories (for read models *and* write models) are services that will reach out and connect to something outside of the application, like a database, the filesystem, or some remote service. That's why they need an abstraction, which can represent what the service will do and what it will return. The implementation will then provide all the low-level details about how it should do that. The same goes for other service objects that will reach out to some service that's outside the application. These services will also need an interface and a concrete implementation.

The services for which we have abstractions according to the list above should be *injected* as abstract dependencies. If we do this, we can form three useful groups, or *layers*, of objects. When you'd stack them, you would encounter, from top to bottom:

1. The *Infrastructure* layer, which contains:
    - Controllers
    - Write and read model repository *implementations*
2. The *Application* layer, which contains:
    - Application services
    - Command objects
    - Read models
    - Read model repository *interfaces*
    - Event listeners
3. The *Domain* layer, which contains:
    - Entities
    - Value objects
    - Write model repository *interfaces*

In the examples you saw earlier in this chapter, the layer names have been used in the namespaces of the classes.



Layers can conveniently be visualized as concentric circles.

By injecting abstract dependencies, we can ensure that objects will only depend in one direction: from top to bottom. For instance, an application service that needs a write model repository will depend on that repository's interface, not its concrete implementation. This has two major advantages.

First, we can test the application service code, without an actual repository implementation that would need for instance a database that's up and running, with the correct schema, etc. We now have interfaces for all these services, and we can easily create test doubles for them.

Second, we can easily switch infrastructure implementations. Our application layer would survive a switch between frameworks (or an upgrade to the framework's next major version), and it would also survive a switch of databases (when you realize you're better off with a graph database than a relational database for instance) and remove services (when you no longer want to fetch exchange rates from an external service, but from your own local database).

17. Summary

An application's front controller will forward an incoming request to one of its controllers. These controllers are part of the application's *infrastructure layer*, and know how to translate incoming data into a call to an application service or a read model repository, which are both part of the *application layer*.

An application service will be agnostic of the delivery mechanism, and can be used just as easily in web or console applications. It performs a single task that could be considered one of the application's use cases. Along the way it may take an entity from a write model repository, call a method on it, and save its modified state. The entity itself, including its value objects, are part of the *domain layer*

A read model repository is a service which can be used to retrieve information. It returns read models that are specific to a use case, and provide all the information that's needed, and nothing more.

The types of objects described in this chapter naturally belong to layers. A layering system where code only depends on code in lower layers offers a way to decouple domain and application code from the infrastructural aspects of your application.

Epilogue

This book aims to be a style guide. It provides basic rules for object design that will be reflected in the declarations of your classes and methods. For many of these rules you could build a static analysis tool which emits warnings when you do something that doesn't follow the rules. Such a tool could, for instance, warn you about methods that make state changes *and* return something. Or about services with methods that change their behavior after construction time.

There are two comments to be made here. First, I think it's important to follow the rules, but also to allow yourself to bend them in some special cases. Like, when quality doesn't really matter, when you don't have to maintain the code for a long time. Or when, in certain cases, it will be a lot of work to apply *all* the rules, and the benefits don't outweigh the required effort. However, don't be too soon to judge; I'd estimate that in 95% of real world scenarios there really isn't a case for taking shortcuts.

Second, these rules aren't all there is to object design. They don't tell you exactly what objects you'll need, what their responsibilities should be, etc. For me, the rules in this book are rules I live by, almost without thinking. And because of this, there's more room for trying things out, for spending "mental energy" on different things.

In this chapter I'd like to point out one other topic that helps relieve part of the cognitive burden of application development: architectural patterns. And I'd also like to provide two possible topics to dive into after reading this book: Test-Driven Development (TDD), and Domain-Driven Design (DDD). Both "fields" can help you discover more about object design.

18. Architectural patterns

In the previous chapter, we already discussed how certain types of objects form a natural set of layers. Besides using layers to structure the application as a whole (which should be considered "an act of architecture"), it's important to be aware of the ways in which your application is connected to "the world outside". Recognizing the ways in which it communicates and can be communicated with results in a clean separation between code that supports this communication, and code that is "core" to your application. This approach to architecture is called "Hexagonal architecture", or sometimes "Ports & Adapters".

On this topic, I recommend taking a look at Vaughn Vernon's "Implementing Domain-Driven Design" (Addison-Wesley Professional, 2013), Chapter 4: Architecture. There are also a couple of my own articles that are relevant to this topic:

- "Layers, ports & adapters - Part 2, Layers"[1]
- "Layers, ports & adapters - Part 3, Ports & Adapters"[2]
- "When to add an interface to a class"[3]

[1] https://matthiasnoback.nl/2017/08/layers-ports-and-adapters-part-2-layers/
[2] https://matthiasnoback.nl/2017/08/layers-ports-and-adapters-part-3-ports-and-adapters/
[3] https://matthiasnoback.nl/2018/08/when-to-add-an-interface-to-a-class/

19. Testing

In this book we've discussed object design, and we've been looking at a few testing techniques as well. It's very convenient to design objects while testing them. When you adopt a test-first approach, you will find that you write only the code you actually need to implement the desired behaviors. The tests prove that the objects you designed can be used in the ways you imagined. And whenever you think of possible edge cases, or encounter bugs in the code behind your objects, you can describe the situation in a test case, see it fail, and then fix it.

19.1 Class testing versus object testing

Note that I speak about testing *objects*. I find that developers, myself included, often tend to test *classes*, not objects. This may seem to be a subtle difference, but it has some pretty big consequences. If you test *classes*, you usually test one method, of one class, with all its dependencies swapped out by test doubles. Such a test ends up being *too close to the implementation*. You'll be verifying that method calls are made, you'll be adding getters to get data out of the object, etc.

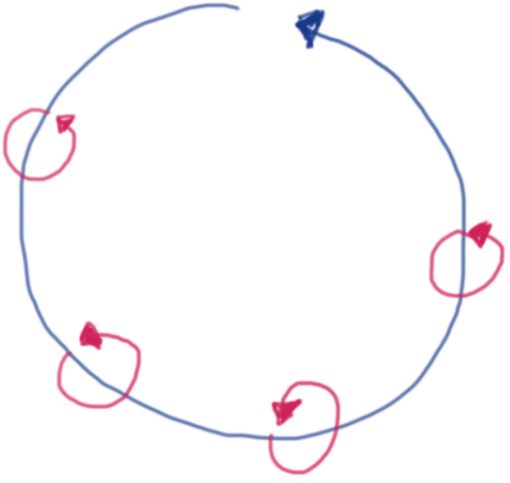
You could consider these tests that test *classes* to be *white box* tests, as opposed to *black box* tests, which are definitely more desirable. A black box test will test an object's behavior as perceived from the outside, with no knowledge about the class' internals. It will instantiate the object with only test doubles for objects that reach across a system boundary. Otherwise, everything is real. Such tests will show that not just a single class, but a larger unit of code works well as a whole.

Class tests will change all the time, alongside the changes made to the classes themselves. Object tests are more decoupled from the implementation of the object that's being tested. So object tests will be more useful in the long run. In fact, this is a rule for testing that you can follow: write your tests in such a way that as many implementation details as possible could be changed, before a change to the test code itself would be required.

19.2 Top down feature development

Another thing to be aware of when testing software is the level of detail you're working at. I find that developers, myself included, often prefer working on the smaller parts; building blocks, that can later be used to complete the feature. You'll often think about everything you're going to need for the full feature and start collecting all the ingredients. Create a repository, a database table, an entity, etc. Once you're trying to connect all the parts, you'll often find that you have to revisit them, because you made a few wrong assumptions, and the building blocks don't work well together in the end. This part of your development effort is basically waste.

I recommend working the other way around: first start with the bigger picture. Define how the feature will be used. First, describe user scenarios, make sketches of the interaction, etc. In other words: specify the high-level behavior of the application, as you expect it to be after you've finished your work. Don't dive into the low-level details too quickly. Once you know what the application, when treated as a black box, should be capable off, you can descend to deeper layers and write code for everything that's needed.



A higher-level TDD cycle comes to a close after successfully closing several lower-level TDD cycles.

A great book that demonstrates this approach is "Growing Object-Oriented Software, Guided by Tests" by Nat Pryce and Steve Freeman (Addison-Wesley Professional,

2009).

If you align this “top down” approach to software development with your approach to testing, you can define automatable acceptance criteria, that will tell you when you’re ready, and that help you prove that what you have built is what was actually needed. To find more about this fascinating topic, take a look at books by Gojko Adzic (“Specification by example”, “Bridging the communication gap”) and “Discovery - Explore behaviour using examples” by G□□sp□□r Nagy and Seb Rose (which is part of a series in the making).

20. Domain-Driven Design

If you're looking for more clues about what types of objects you should have in your application, I find Domain-Driven Design (DDD) an excellent area to look into. The idea behind it is to learn about your problem domain, and then reflect this knowledge in your application's domain model. A *domain-first approach* leads to a focus on design, taking it away from infrastructural details, like database tables and columns.

Although the *strategic* aspect of DDD is quite fascinating, in terms of object design you'll find the most useful suggestions from its tactical advice. Take a look at the books "Domain-Driven Design" by Eric Evans (Addison-Wesley Professional, 2003), and "Implementing Domain-Driven Design" by Vaughn Vernon (Addison-Wesley Professional, 2013). They will contain many practical suggestions for designing *Entities* and *Value objects*, as well as other related types of objects.

21. Conclusion

Of course, there's much more to discover about object design, and about software development and architecture in general. When it comes to object design, I hope this book has been able to provide you with a good foundation, and some useful pointers for learning more. You'll find that there's more to learn every day, so keep experimenting. Best of luck with that!

Changelog

Below is a list of changes I've been making to the manuscript, starting with the book release of December 2018.

6/3/2019

- Foreword
  - Added the foreword by Ross Tuck.
- Introduction
  - Cleaned up the design rules (removed some technical details about using PHPUnit).
- The lifecycle of an object
  - Rewrote the explanation about the two types of objects. What really defines these types is how they are related to each other. The first type uses the second type (services versus materials).
- Creating services
  - Moved the section "Inject what you need, not where you can get it from" up, and added a bridge sentence to the end of the "Inject dependencies and configuration values as constructor arguments" section so that it's clear that this advice also applies to configuration values: don't inject entire global configuration objects, but only the values that you need from them.
  - Added a subsection: "Keeping together configuration values that belong together", which introduces a way to keep together configuration values that belong together.
  - Added an aside: "What if I need the service and the service I retrieve from it?", which answers a common question that was asked by the technical reviewer.
  - Changed the example about wrapping `json_encode()`: it contained a call to `file_put_contents()` which was quite confusing, because the next section is about system calls. I enhanced the `json_encode()` example by adding a few lines about handling errors in your own preferred way.

  - Enhance the aside "Should all functions be promoted to object dependencies?", providing some more guiding questions to decide whether or not a function call should be turned into an object dependency.
  - Add an extra step in the redesign process of the `FileLogger` ("Do nothing inside a constructor, only assign properties"), showing how you can also push the `is_writable()` check out of the constructor.
- Creating other objects
  - Added a new section: "Don't use custom exception classes for invalid argument exceptions", explaining why you don't usually need custom exception classes for invalid argument exceptions.
  - Added an aside: "Adding more object types also leads to more typing, is that really necessary?", explaining some of the benefits of using object types instead of primitive types, just in case people are wondering if all that extra typing is really necessary.
  - Added another example to the section "Don't inject dependencies, optionally pass them as method arguments", explaining how you could rewrite the currency conversion logic using a simple services. Added a comment about the design trade-offs you have to make in this type of situation.
  - Added an aside about PHP's class-based scoping, explaining how it's possible that a named constructor can manipulate private properties directly.
  - Added a subsection ["Optionally use the private constructor to enforce constraints"[(#optionally-use-the-private-constructor-to-enforce-constraints) with an example showing how you can use the private constructor when you have multiple named constructors.
  - Turn "Don't use property fillers into its own section.
  - Be more explicit in the section "Don't test constructors" to make sure that people understand the point.
  - Finish the chapter with a new section: "The exception to the rule: Data transfer objects" about *Data transfer objects*, a type of object with less strict rules, which was not yet discussed in detail.
- Manipulating objects
  - Added a new introduction, explaining the different types of "other" objects and what their design characteristics are in the area of object manipulation, covering Entities (#manipulating-entities), Value objects and DTOs.

  - Elaborate on the meaning of == for object comparison and that it should be avoided.
  - Move the discussion about command methods on mutable objects before the discussion about modifier methods on immutable objects.
  - Add an alternative implementation example to "Don't implement fluent interfaces on mutable objects".
  - Add an aside: "A third-party library has some object design issues, what do I do?"
  - End the section "Use internally recorded events to verify changes on mutable objects with a pointer to a later chapter about retrieving information from objects (because we say: remove all getters, but then don't offer a good alternative).
- Retrieving information
  - Added an aside: "How do you handle ambiguous naming?", explaining how you can deal with query method names that could be read as verbs (e.g. "name").
  - Extract the section about separating query and command objects (CQRS) into its own chapter: "Dividing responsibilities".
  - Extract a class, not just a method in "Define specific methods and return types for the queries you want to make".
  - Rewrote the section "Define an abstraction for queries that cross system boundaries" to use a more concrete example of an actual exchange rates API (Fixer).
  - Added an aside: "Naming test methods", explaining why we use `it_...` for test method names.
- "Performing tasks"
  - Add a full alternative implementation example to the section "Make services immutable from the outside as well as on the inside".
- "Dividing responsibilities"
  - Added this new chapter, which elaborates on how you can make some objects be responsible for changing state, and others for providing information. It has a new, more detailed example, which is also more realistic than the current one about a player and its position. ****