

Advanced Web Application Architecture

Matthias Noback

Advanced Web Application Architecture

Matthias Noback

Advanced Web Application Architecture

Matthias Noback

©2020 Matthias Noback

ISBN 978-90-821201-6-5

Cover drawings by Julia Noback

Other books by Matthias Noback:

- A Year with Symfony (Leanpub, 2014)
- Principles of Package Design (Apress, 2018)
- Microservices for Everyone (Leanpub, 2018)
- Object Design Style Guide (Manning Publications, 2019)
- PHP for the Web (Leanpub, 2020)

In memory of my grandfather, H.A.J. Noback

Contents

Introduction

- 1 [Preface](#)
- 2 [Why is decoupling from infrastructure so important?](#)
- 3 [Who is this book for?](#)
- 4 [Overview of the contents](#)
- 5 [The accompanying demo project](#)
- 6 [About the author](#)
- 7 [Acknowledgements](#)
- 8 [Changelog](#)
 - 8.1 [April 15th, 2020 \(initial version\)](#)
 - 8.2 [April 23rd, 2020](#)
 - 8.3 [May 1st, 2020](#)
 - 8.4 [May 6th, 2020](#)
 - 8.5 [May 29th, 2020](#)
 - 8.6 [June 4th, 2020](#)
 - 8.7 [June 12th, 2020](#)
 - 8.8 [June 17th, 2020](#)
 - 8.9 [June 22nd, 2020](#)
 - 8.10 [June 26th, 2020](#)
 - 8.11 [July 1st, 2020](#)
 - 8.12 [July 6th, 2020](#)
 - 8.13 [July 8th, 2020](#)

I Decoupling from infrastructure

1 Introduction

- 1.1 [Rule no 1: No dependencies on external systems](#)
- 1.2 [Abstraction](#)
- 1.3 [Rule no 2: No special context needed](#)
- 1.4 [Summary](#)

2 The domain model

- 2.1 [SQL statements all over the place](#)

- 2.2 [Trying to fix it with a table data gateway](#)
- 2.3 [Designing an entity](#)
- 2.4 [Introducing a repository](#)
- 2.5 [Mapping entity data to table columns](#)
 - 2.5.1 [Using an ORM](#)
 - 2.5.2 [Manual mapping](#)
- 2.6 [Generating the identifier earlier](#)
 - 2.6.1 [Using UUIDs instead of \(auto-\)incrementing integer IDs](#)
- 2.7 [Using a value object for the identifier](#)
- 2.8 [Active Record versus Data Mapper](#)
- 2.9 [Summary](#)

3 Read models and view models

- 3.1 [Reusing the write model](#)
- 3.2 [Creating a separate read model](#)
- 3.3 [Read model repository implementations](#)
 - 3.3.1 [Sharing the underlying data source](#)
 - 3.3.2 [Using write model domain events](#)
- 3.4 [Using value objects with internal read models](#)
- 3.5 [A specific type of read model: the view model](#)
- 3.6 [Using view models for APIs](#)
- 3.7 [Summary](#)

4 Application services

- 4.1 [Considering other infrastructures](#)
- 4.2 [Designing a use case to be reusable](#)
- 4.3 [Extracting an application service](#)
- 4.4 [Introducing a parameter object](#)
- 4.5 [Dealing with multiple steps](#)
- 4.6 [Summary](#)

5 Service locators

- 5.1 [From service location to explicit dependencies](#)
- 5.2 [Depending on global state](#)
- 5.3 [Injecting dependencies](#)
- 5.4 [Injecting configuration values](#)
- 5.5 [Using method arguments for job-specific data](#)
- 5.6 [Clients of reusable services](#)

- 5.7 [Testing](#)
- 5.8 [Effective testing](#)
- 5.9 [The Composition root is near the entry point](#)
- 5.10 [Summary](#)

6 External services

- 6.1 [Connecting to the external service](#)
- 6.2 [Introducing an abstraction](#)
- 6.3 [Architectural advantages](#)
- 6.4 [Testing](#)
- 6.5 [Summary](#)

7 Time and randomness

- 7.1 [Passing current time and random data as method arguments](#)
- 7.2 [Introducing factories](#)
- 7.3 [Introducing value objects](#)
- 7.4 [Improving the factories](#)
- 7.5 [Manipulating the current time](#)
- 7.6 [Integration tests again](#)
- 7.7 [Summary](#)

8 Validation

- 8.1 [Protecting entity state](#)
- 8.2 [Using value objects to validate separate values](#)
- 8.3 [Form validation](#)
- 8.4 [Using exceptions to talk to users](#)
- 8.5 [When validation is not the answer](#)
- 8.6 [Creating and validating command objects](#)
- 8.7 [Summary](#)

9 Conclusion

- 9.1 [Core code and infrastructure code](#)
- 9.2 [A summary of the strategy](#)
 - 9.2.1 [Use dependency injection and inversion everywhere](#)
 - 9.2.2 [Make use cases universally invokable](#)
- 9.3 [Focus on the domain](#)
- 9.4 [Focus on testability](#)
- 9.5 [Pure object-oriented code](#)

9.6 Summary

II Organizing principles

10 Introduction

11 Key design patterns

11.1 Framework-inspired structural elements

11.2 Entities

11.2.1 Protect invariants

11.2.2 Constrain updates

11.2.3 Model state changes as actions with state transitions

11.2.4 Don't think too much about tables

11.2.5 Record domain events

11.3 Repositories

11.4 Application services

11.4.1 Return the identifier of a new entity

11.4.2 Input should be defined as primitive-type data

11.4.3 Wrap input inside command objects

11.4.4 Translate primitive input to domain objects

11.4.5 Add contextual information as extra arguments

11.4.6 Save only one entity per application service call

11.4.7 Move secondary tasks to a domain event subscriber

11.5 Event subscribers

11.5.1 Move subscribers to the module where they produce their effect

11.5.2 Delegate to an application service

11.6 Read models

11.6.1 Use internal read models when you need information

11.6.2 Choose a standard implementation for the repository

11.6.3 For view models, prepare the data for rendering

11.7 Process modelling

11.8 Summary

12 Architectural layers

12.1 MVC

12.2 A standard set of layers

12.2.1 The infrastructure layer

12.2.2 The application layer

- 12.2.3 [The domain layer](#)
- 12.2.4 [Up and down the layer stack](#)
- 12.3 [The Dependency rule](#)
- 12.4 [Making layers tangible](#)
- 12.4.1 [Documenting the architecture](#)
- 12.4.2 [Using namespaces for layering](#)
- 12.4.3 [Automated verification of design decisions](#)
- 12.5 [Summary](#)

[13 Ports and adapters](#)

- 13.1 [Hexagonal architecture](#)
- 13.2 [Ports](#)
- 13.3 [Adapters for outgoing ports](#)
- 13.4 [Adapters for incoming ports](#)
- 13.5 [The application as an interface](#)
- 13.6 [Combining ports and adapters with layers](#)
- 13.7 [Structuring the Infrastructure layer](#)
- 13.8 [Summary](#)

[14 A testing strategy for decoupled applications](#)

- 14.1 [Unit tests](#)
- 14.2 [Adapter tests](#)
- 14.3 [Contract tests for outgoing port adapters](#)
- 14.4 [Driving tests for incoming port adapters](#)
- 14.5 [Use case tests](#)
- 14.6 [End-to-end tests](#)
- 14.7 [Development workflow](#)
- 14.8 [Summary](#)

[15 Conclusion](#)

- 15.1 [Is a decoupled architecture the right choice for all projects?](#)
- 15.2 [My application is not supposed to live longer than two years](#)
- 15.3 [My application offers only CRUD functionality](#)
- 15.4 [My application is a legacy application](#)
- 15.5 [I can never make my entire application decoupled](#)
- 15.6 [Isn't this over-engineering?](#)

Introduction

1 Preface

My last book, the *Object Design Style Guide*, ends with chapter 10 - “A field guide to objects”, showing the characteristics of some common types of objects like *controllers*, *entities*, *value objects*, *repositories*, *event subscribers*, etc. The chapter finishes with an overview of how these different types of objects find their natural place in a set of architectural layers. Some readers pointed out that the field guide itself was not detailed enough to help them use these types of objects in their own projects. And some people objected that the architectural concepts briefly described in this chapter could not easily be applied to real-world projects either. They are totally right; that last chapter turned out to be more of a teaser than a treatise. Unfortunately I couldn’t think of an alternative resource that I could provide to those readers. There are some good articles and books on this topic, but they cover only *some* of the patterns and architectural concepts. As far as I know, there is no comprehensive guide about all of these patterns combined. So I decided to write it myself: a showcase of design patterns, like entities and application services, explaining how they all work together in a “well-architected” application. However, a plain description of existing patterns isn’t nearly as useful as showing how you could have invented them by yourself, simply by trying to decouple your application code from its surrounding infrastructure. That’s how this book became a guide to decoupling your domain model and your application’s use cases from the framework, the database, and so on.

2 Why is decoupling from infrastructure so important?

Separating infrastructure concerns from your core application logic leads to a domain model that can be developed in a *domain-driven* way. It also leads to application code that is very easy to test, and to develop in a *test-driven* way. Finally, tests tend to be more stable and run faster than your average framework-inspired functional test.

Supporting both Domain-Driven Design (DDD) and Test-Driven Development (TDD) is already a great attribute of any software system. But separating infrastructure from domain concerns by applying these design patterns gives you two more advantages. Without a lot of extra work you can start using a standard set of layers (which we'll call *Domain*, *Application*, and *Infrastructure*). On top of that, you can easily mark your decoupled use cases as *Ports* and the supporting implementation code as *Adapters*.

Using layers, ports, and adapters is a great way of standardizing your high-level architecture, making it easier for everybody to understand the code, to take care of it, and to continue developing it. And the big surprise that I'll spoil to you now is that by decoupling core code from infrastructure code you get all of this for free.

If your application is supposed to live longer than, say, two years, then decoupling from infrastructure is a safe bet. Surrounding infrastructure like frameworks, remote web services, storage systems, etc. are likely to change at a different rate than your domain model and use cases. Whatever happens in the world of the technology surrounding your application, your precious core code won't be disturbed by it. Both can evolve at their own speed. Upgrading to the next version of your framework, migrating to a different storage backend, or switching to a different payment provider won't cost as much as it would if core and infrastructure code were still mixed together. Dependencies on external code or systems will always be isolated and if a change has to be made, you'll know immediately where to make it.

If on the other hand your application is not supposed to live longer than two years, that might be a good reason not to care about the approach presented in this book. That said, I have only seen such an application once or twice in my life.

3 Who is this book for?

This book is for you:

- If you have some experience with “framework-inspired” development, that is, following a framework’s documentation to structure a web application, or
- If you have seen some legacy code, with every part of the code base knowing about every other part, and different concerns being completely mixed together, or
- If you’ve seen both, which is quite likely since these things are often related.

I imagine you’re reading this book because you’re looking for better ways to structure things and escape the mess that a software project inevitably becomes. Here’s my theory: software always becomes a mess, even if you follow all the known best practices for software design. But I’m convinced that if you follow the practices explained in this book it will take more time to become a mess, and this is already a huge competitive advantage.

4 Overview of the contents

This book is divided into three parts. In Part [I](#) (“Decoupling from infrastructure”) we look at different code samples from a legacy application where core and infrastructure code are mixed together. We find out how to:

- Extract a domain model from code that mixes SQL queries with business decisions ([Chapter 2](#))
- Extract a reusable application service from a controller that mixes form handling, business logic, and database queries ([Chapter 4](#)).
- Separate a read model from its underlying data storage ([Chapter 3](#))
- Rewrite classes that use service location to classes that rely on dependency injection ([Chapter 5](#))
- Separate *what* we need from external services from *how* we get it ([Chapter 6](#))
- Work with current time and random data independently from how the running application will retrieve this information ([Chapter 7](#))

Along the way we find out the common refactoring techniques for separating these concerns. We notice how these refactoring techniques result in possibly already familiar design patterns, like entities, value

objects, and application services. We finish this part with an elaborate discussion of validation, and where and how it should or can happen (Chapter 8).

Part II (“Organizing principles”) provides an overview of the organizational principles that can be applied to an application’s design at the architectural scale. Chapter 11 is a catalog of the design patterns that we derived in Part I. We cover them in more detail and add some relevant nuances and suggestions for implementation. Chapter 12 shows how separating core from infrastructure code using all of these design patterns allows you to group the resulting classes into a standardized set of *layers*. Chapter 13 then continues to explain how you can use the architectural style called *Ports and adapters* as a kind of overlay for this layered architecture. In Chapter 14 we look at a possible testing strategy for decoupled applications. With Chapter 15 we reach the book’s conclusion.

5 The accompanying demo project

Of course all the design techniques and principles discussed in this book are illustrated with many code samples. However, these samples are always abbreviated, idealized, and they show only the most essential aspects. To get a full understanding of how all the different parts of an application work together, we need a demo project. Again, not an idealized or simplified one, but a real-world project that is running in production. People have often asked for such a project and I’ve always answered: I’d love to work on that, now I need to find some time. This time I decided to make it happen. The demo project is the source code for the new *Read with the Author* platform. This software runs in production. In fact, you may have used the software already if you bought a ticket for it on Leanpub. But the most important quality of the project is that it shows the design techniques and principles from this book in practice. For \$5 you can explore the source code, including any of its future updates. Go to <https://advwebapparch.com/repository> to get access right away.

6 About the author



Matthias Noback is a professional web developer since 2003. He lives in Zeist, The Netherlands, with his girlfriend, son, and daughter.

Matthias has his own web development, training and consultancy company called Noback's Office. He has a strong focus on backend development and architecture, always looking for better ways to design software.

Since 2011 he's been writing about all sorts of programming-related topics on his blog^{[1](#)}. Other books by Matthias are *Principles of Package Design*

(Apress, 2018), and *Object Design Style Guide* (Manning, 2019).

You can reach Matthias:

- By email: info@matthiasnoback.nl
- On Twitter: [@matthiasnoback](https://twitter.com/@matthiasnoback)

7 Acknowledgements

This has been the sixth book I published using the Leanpub² platform. It has always been a great experience, so thanks again Peter Armstrong and Lenn Ep.

Thank you, 307 interested readers, for signing up for this book before it was written. Thank you, 440 readers, for buying this book before it was finished.

To the readers who joined the read-with-the-author sessions for this book: thank you for your insightful questions and heartwarming comments. And thank you to everyone who has shared feedback, comments, and suggestions through different channels. Let's hope I'm not missing anyone: Christopher L Bray, Ondřej Bouda, Samir Bouilil, Iosif Chiriluta, Biczó Dezsö, Nicola Fornaciari, Ramon de la Fuente, Raúl Fraile, Alex Gemmell, Gary Jones, Luis-Ramón López, Hazem Noor, Thomas Nunninger, Nikola Paunovic, José María Valera Reales, Gildas Quéméner, Onno Schmidt, Daniel Martín Spiridione, Harm van Tilborg, Stijn Vergote, Tom de Wit.

8 Changelog

8.1 April 15th, 2020 (initial version)

1. Released: Front matter, Part I: Decoupling from infrastructure, and Chapter 1: The domain model.

8.2 April 23rd, 2020

Thank you for your suggestions, Christopher L. Bray, Iosif Chiriluta, Biczó Dezsö, Luis Ramon Lopez, and Thomas Nunninger!

1. Released: Chapter [3](#): Read models and view models
2. Fixed some spelling issues
3. Chapter 1: Added an aside (“Wait, is UUID the best we can get?”)
4. Fixed the caption of listing 2.29.
5. Chapter 1: Added subsection 2.5.1 (Using an ORM)

8.3 May 1st, 2020

Thank you for your suggestions, Christopher L. Bray, José María Valera Reales, and Raúl Fraile!

1. Released: Chapter [4](#): Application services
2. Added two exercises to Chapter [2](#)
3. Used PHP arrow functions (`fn () => /* ... */`) where possible
4. Added a paragraph to Section [1.3](#) explaining why I’m using the words “core” and “infrastructure”
5. Added a diagram to show how the application’s infrastructure is around its core, connecting it to external systems and users

8.4 May 6th, 2020

Thank you for your suggestions, Christopher L Bray, Thomas Nunninger, Iosif Chiriluta, Nikola Paunovic, Gildas Quéméner, Samir Boulil, Harm van Tilborg!

1. Released: Chapter [5](#): Service locators
2. Fixed a mistake in the last example of the aside “Are getters on entities forbidden?”
3. Fixed some typos and inconsistencies in the code samples of Chapter [4](#)
4. Fixed indentation issue in exercises of Chapter [4](#)
5. Changed the font to something more readable

8.5 May 29th, 2020

Thank you for your suggestions, Christopher L Bray, Harm van Tilborg, Daniel Martín Spiridione, Gary Jones, Thomas Nunninger!

1. Released: Chapter [6](#): External services
2. Fixed invalid return statement in Listing [4.12](#)
3. Fixed an inconsistency related to \$ebookPrice in Section [3.4](#)
4. Added the UserRepository dependency as an injected constructor argument in Listing [5.7](#)
5. Rewrote Section [5.9](#) because it was hard to follow

8.6 June 4th, 2020

Thank you for your suggestions Ramon de la Fuente!

1. Released: Chapter [7](#): Time and randomness.
2. Fix the answer to Exercise 2 of Chapter [4](#).
3. Replaced hard-coded 'NL' with usage of \$countryCode variable in Listing [6.14](#).

8.7 June 12th, 2020

Thank you for your suggestions, Christopher L Bray and Thomas Nunninger!

1. Released: Chapter [8](#): Validation.
2. Fixed reference to Listing [3.10](#).
3. Fixed reference to Figure [7.1](#).
4. Fixed reference to Listing [7.8](#).
5. Added some missing listing captions.

8.8 June 17th, 2020

1. Released: Chapter [9](#): Conclusion

8.9 June 22nd, 2020

1. Released: Part [II](#), Chapter [10](#): Introduction
2. Released: Chapter [11](#): Key design patterns

8.10 June 26th, 2020

1. Released: Chapter [12](#): Architectural layers

8.11 July 1st, 2020

1. Released: Chapter [13](#): Ports and adapters
2. Changed PDF fonts to Droid, thanks to Harm van Tilborg

8.12 July 6th, 2020

1. Released: Chapter [14](#): Testing strategy
2. Improved the EPUB and MOBI files (endnotes instead of footnotes, fixed font encoding issues)
3. Added a paragraph about the asymmetry between incoming and outgoing ports and adapters (at the end of Section [13.5](#))

8.13 July 8th, 2020

Thank you for your suggestions Hazem Noor, Ramon de la Fuente, and Biczó Dezsö!

1. Released last chapter: Chapter [15](#)
2. Fixed invalid listing references in EPUB and MOBI files
3. Fixed an inconsistency related to \$ebookPrice in Chapter [4](#)
4. Exercise 2 of Chapter [13](#): changed “Port” into “Adapter”
5. Added an example of loading a view model in the controller based on an entity ID returned from an application service (Section [11.4.1](#)).
6. Added a short discussion about adding value object getters to command DTOs in Section [11.4.4](#).
7. Added an aside about repository implementation requirements (“That’s too easy!”) in Section [11.3](#).
- 8.

Rewrote Section [11.5.2](#) to make the distinction between core and infrastructure event subscribers more clear.

Part I

Decoupling from infrastructure

1 Introduction

This part covers:

- Decoupling your domain model from the database
 - Decoupling the read model from the write model (and from the database)
 - Extracting an application service from a controller
 - Rewriting calls to service locators
 - Splitting a call to external systems into the “what” and “how” of the call
 - Inverting dependencies on system devices for retrieving the current time, and randomness
-

The main goal of the architectural style put forward in this book is to make a clear distinction between the core code of your application and the infrastructure code that supports it. This so-called infrastructure code connects the core logic of your application to its surrounding systems, like the database, the web server, the file system, and so on. Both types of code are equally important, but they shouldn’t live together in the same classes. The reasons for doing so will be discussed in detail in the conclusion of this part, but the quick summary is that separating core from infrastructure...

- provides a strong technical foundation for doing domain-first development, and
- enables a rich and effective set of testing possibilities, making test-first development easier

To help you develop an eye for the distinction between core and infrastructure concerns, each of the following chapters starts with some common examples of “mixed” code in a legacy web application. After pointing out the problems with this kind of code we take a number of refactoring steps to separate the core part from the infrastructure part. After six of these iterations you will have seen all the programming techniques that can save you from having mixed code in your classes.

But before we start refactoring and improving the code samples, let’s establish a definition of the terms “core” and “infrastructure” code. We’ll define core code by introducing two rules for it. Any other code that doesn’t follow the rules for *core* code should be considered *infrastructure code*.

1.1 Rule no 1: No dependencies on external systems

Let’s start with the first rule:

Core code doesn’t directly depend on external systems, nor does it depend on code written for interacting with a specific type of external system.

An external system is something that lives outside your application, like a database, some remote web service, the system’s clock, the file system, and so on. Core code should be able to run without these external dependencies. Listing 1.1 shows a number of class methods that don’t follow this first rule, and should therefore be considered infrastructure code. You can’t call any of these methods without their external dependencies being actually available.

Listing 1.1: Examples of code that needs external dependencies to run.

```
final class NeedsExternalDependencies
{
    public function callARemoteService(): void
    {
        /*
         * To run this code, we need an internet connection,
         * and the API of remoteservice.com should be responsive.
        */
    }
}
```

```

    $ch = curl_init('https://remoteservice.com/api');

    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec($ch);

    // ...
}

public function useTheDatabase(): void
{
    /*
     * To run this code, the database that we connect to
     * using 'new PDO('...')' should be up and running, and
     *
     * it should contain a table called 'orders'.
     */
    $pdo = new PDO('...');
    $statement = $pdo->prepare('INSERT INTO orders ...');

    $statement->execute();
}

public function loadAFile(): string
{
    /*
     * To run this code, the 'settings.xml' file should exist
     * in the correct location.
     */
    return file_get_contents(
        __DIR__ . '/app/config/settings.xml'
    );
}
}

```

When code follows the first rule, it means you can run it in complete isolation. Isolation is great for testability. When you want to write an automated test for core code, it will be very easy. You won't need to set up a database, create tables, load fixtures, etc. You won't need an internet connection, or a hard disk with files on it in specific locations. All you need is to be able to run the code, and have some computer memory available.

1.2 Abstraction

What about the `registerUser()` method in Listing [1.2](#)? Is it also infrastructure code?

Listing 1.2: Depending on an interface.

```
interface Connection
{
    public function insert(string $table, array $data): void;
}

final class UserRegistration
{
    /**
     * @var Connection
     */
    private Connection $connection;

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

    public function registerUser(
        string $username,
        string $plainTextPassword
    ): void {
        $this->connection->insert(
            'users',
            [
                'username' => $username,
                'password' => $plainTextPassword
            ]
        );
    }
}
```

The `registerUser()` method doesn't use PDO³ directly to connect to a database and start running queries against it. Instead, it uses an *abstraction* for database connections (the `Connection` interface). This means that the `Connection` object that gets injected as a constructor argument could be replaced by a simpler implementation of that same interface which doesn't actually need a database (see Listing [1.3](#)).

Listing 1.3: An implementation of `Connection` that doesn't need a database.

```
final class ConnectionDummy implements Connection
{
    /**
     * @var array<array<string,mixed>>
     */
    private array $records;

    /**
     * @param array<string,mixed> $data
     */
    public function insert(string $table, array $data): void
    {
        $this->records[$table][] = $data;
    }
}
```

This makes it possible to run the code in that `registerUser()` method, without the need for the actual database to be up and running. Does that make this code *core* code? No, because the `Connection` interface is specifically designed to communicate with relational databases, as the `insert()` method signature itself reveals. So although the `registerUser()` method doesn't directly depend on an external system, it does depend on code written for interacting with a specific type of external system. This means that the code in Listing 1.2 is not core code, but infrastructure code.

In general though, abstraction is the go-to solution to get rid of dependencies on external systems. We'll discuss several examples of abstraction in the next chapters, but it might be useful to give you the summary here. Creating a complete abstraction for services that rely on external systems consists of two steps:

1. Introduce an interface
2. Communicate *purpose* instead of implementation details

As an example: instead of a `Connection` interface and an `insert()` method, which only makes sense in the context of dealing with relational databases,

we could define a Repository interface, with a `save()` method instead. Such an interface communicates purpose (saving objects) instead of implementation details (storing data in tables). We'll discuss the details of this type of refactoring in Chapter [2](#).

1.3 Rule no 2: No special context needed

The second rule for core code is:

Core code doesn't need a specific environment to run in, nor does it have dependencies that are designed to run in a specific context only.

[Listing 1.4](#) shows some examples of code that requires special context before you can run it. It assumes certain things have been set up, or that it runs inside a specific type of application, like a web or a command-line (CLI) application.

Listing 1.4: Examples of code that needs a special context to run in.

```
final class RequiresASpecialContext
{
    public function usesGlobalState(): void
    {
        /*
         * Here we rely on global state, and we assume this
         * method gets executed as part of an HTTP request.
         */

        $host = $_SERVER['HTTP_HOST'];

        // ...
    }

    public function usesAStaticServiceLocator(): void
    {
        /*
         * Here we rely on 'Zend_Registry' to have been
         * configured before calling this method.
         */

        $translator = Zend_Registry::get('Zend_Translator');
    }
}
```

```

        // ...
    }

    public function onlyWorksAtTheCommandLine(): void
    {
        /*
         * Here we rely on 'php_sapi_name()' to return a specific
         * value. Only when this application has been started from
         * the command line will this function return 'cli'.
        */

        if (php_sapi_name() !== 'cli') {
            return;
        }

        // ...
    }
}

```

Some code could in theory run in any environment, but in practice it will be awkward to do so. Consider the example in Listing 1.5. The `OrderController` could be instantiated in any context, and it would be relatively easy to call the action method and pass it an instance of `RequestInterface`. However, it's clear that this code has been designed to run in a very specific environment only, namely a web application.

Listing 1.5: Code that is designed to run in a web application.

```

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

final class OrderController
{
    public function createOrderAction(
        RequestInterface $request
    ): ResponseInterface {
        // ...
    }
}

```

Only if code doesn't require a special context, and also hasn't been designed to run in a special context or has dependencies for which this is the case, can it be considered core code.

[Listing 1.6](#) shows several examples of core code. These classes can be instantiated anywhere, and any client should be able to call any of the available methods. None of these methods depend on anything outside the application itself.

Listing 1.6: Some examples of core code.

```
/*
 * This is a proper abstraction for an object that talks to the
 database:
 */
interface MemberRepository
{
    public function save(Member $member): void;
}

final class MemberService
{
    private MemberRepository $memberRepository;

    public function requestAccess(
        string $emailAddress,
        string $purchaseId
    ): void {
        $member = Member::requestAccess(
            EmailAddress::fromString($emailAddress),
            PurchaseId::fromString($purchaseId)
        );

        $this->memberRepository->save($member);
    }
}

final class EmailAddress
{
    private string $emailAddress;

    private function __construct(string $emailAddress)
    {
        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {
```

```

        throw new InvalidArgumentException('...');
    }

    $this->emailAddress = $emailAddress;
}

public static function fromString(string $emailAddress): self
{
    return new self($emailAddress);
}

final class Member
{
    public static function requestAccess(
        EmailAddress $emailAddress,
        PurchaseId $purchaseId
    ): self {
        // ...
    }
}

```

Not having to create a special context for code to run in is, again, great for testability. The only thing you have to do in a test scenario is instantiate the class and call a method on it. But following the rules for core code isn't just great for testing. It also helps keeping your core code protected against all kinds of external changes, like a major framework upgrade, a switch to a different database vendor, etc.

It's not a coincidence that the classes in this example are domain-oriented. In Chapter [12](#) we will discuss architectural layering and define rules for the *Domain* and *Application* layers which naturally align with the rules for core and infrastructure code. In short: all of the domain code and the application's use cases should be core code, and not rely on or be coupled to surrounding infrastructure.

This also explains why I'm using the words "core" and "infrastructure". Infrastructure is a common term used to describe the technical aspects of an interaction. In a web application, infrastructure supports the communication between your application and the outside world. The core is the center of

your application, the infrastructure is around it, both protecting the core and connecting it to external systems and users (Figure 1.1).

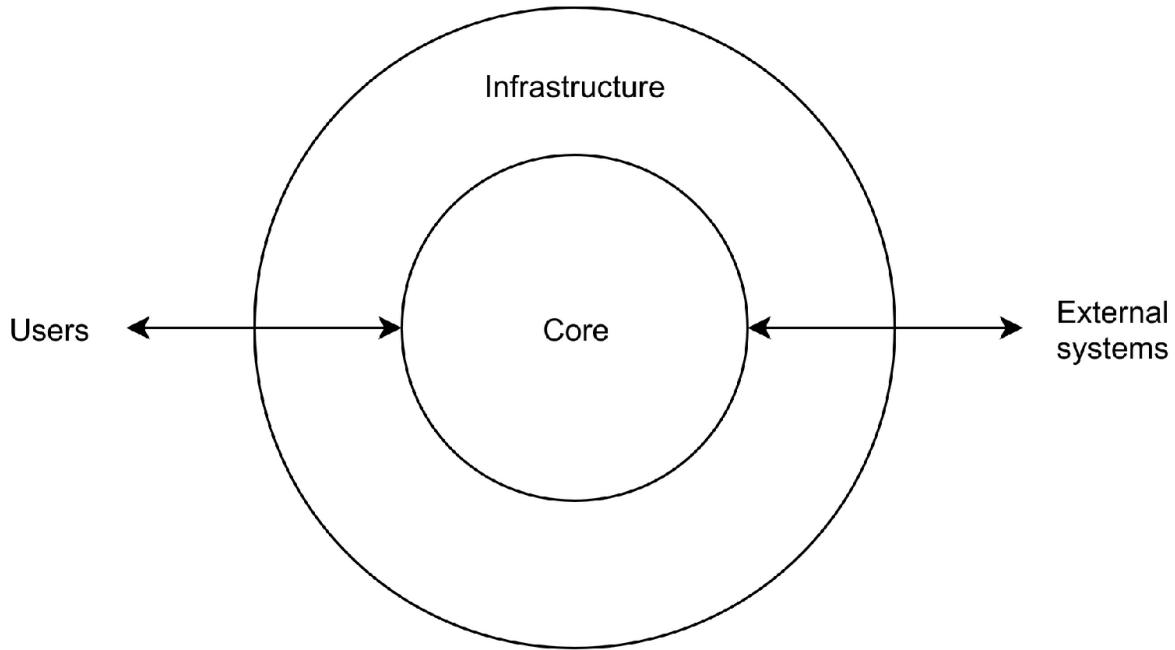


Figure 1.1: Connecting the core to external systems and users through infrastructure

“Is all code in my vendor directory infrastructure code?”

Great question. In `/vendor` you’ll find your web framework, which facilitates communication with browsers and external systems using HTTP. You’ll also find the ORM, which facilitates communication with the database, and helps you save your objects in tables. All of this code doesn’t comply with the definition of core code provided in this chapter. To run this code, you usually need external systems like the database or the web server to be available. The code has been designed to run in a specific context, like the terminal, or as part of a web request/response cycle. So *most* of the code in `/vendor` should be considered infrastructure code.

However, being in a particular directory doesn't determine whether or not something is infrastructure code. The rules don't say anything about that. What matters is what the code does, and what it needs to do that. This means that some, or maybe a lot of the code in `/vendor` could be considered core code after all, even though it's not written by you or for your application specifically.

1.4 Summary

Throughout this book we make a distinction between core and infrastructure code, which will be the foundation of some architectural decisions later on. Core code is code that can be executed in any context, without any special setup, or external systems that need to be available. For infrastructure code the opposite is the case: it needs external systems, special setup, or is designed to run in a specific context only.

In the next chapters we'll look at how to refactor mixed code into properly separated core and infrastructure code which follows the rules provided in this chapter.

Exercises

1. Should the code below be considered *infrastructure* code?⁴

```
$now = new DateTimeImmutable('now');
$expirationDate = $now->modify('+2 days');

$membershipRequest = new MembershipRequest($expirationDate);
```

2. Should the code below be considered *infrastructure* code?⁵

```

namespace Symfony\Component\EventDispatcher;

class EventDispatcher implements EventDispatcherInterface
{
    // ...

    public function dispatch(
        object $event,
        string $eventName = null
    ): object {
        $eventName = $eventName ?? get_class($event);

        // ...

        if ($listeners) {
            $this->callListeners($listeners, $eventName, $event)
        }
    }

    return $event;
}

// ...
}

```

3. Should the code below be considered *core* code?⁶

```

interface HttpClient
{
    public function get(string $url): Response;
}

final class Importer
{
    private HttpClient $httpClient;

    public function __construct(HttpClient $httpClient)
    {
        $this->httpClient = $httpClient;
    }

    public function importPurchasesFromLeanpub(): void
    {
        $response = $this->httpClient->get(
            'https://leanpub.com/api/individual-purchases'
        );
    }
}

```

```
 );  
 // ...  
 }  
 }
```

2 The domain model

This chapter covers:

- Extracting an entity and a repository from database interaction code
 - Using an entity to protect the model against inconsistent data
 - Mapping the entity to a database table inside a repository implementation
 - Providing an entity with identity before saving it
-

2.1 SQL statements all over the place

In this chapter we work with an imaginary web application that allows users to browse through a catalog of e-books, select one, and order it. It's a legacy application that's becoming hard to maintain. It's difficult to find out what the code does, what concepts it deals with, and what business rules are applicable. Also, nobody wrote tests because it's hard to write them for this kind of code.

Let's take a look at the code that our application runs when the user selects an e-book from our catalog and orders it (see Listing 2.1).

Listing 2.1: The original `orderEbookAction()`.

```
public function orderEbookAction(Request $request): Response
{
    $connection = $this->container->get('connection');

    $ebookPrice = $connection->execute(
        'SELECT price FROM ebooks WHERE id = :id',
        [
            'id' => $request->request->get('ebook_id')
        ]
    )->fetchColumn(0);

    $orderAmount = (int)$request->get('quantity')
        * (int)$ebookPrice;

    $record = [
        'email' => $request->get('email_address'),
        'quantity' => (int)$request->get('quantity'),
        'amount' => $orderAmount,
    ];

    $columns = array_keys($record);
    $values = array_map(
        fn ($value) => $connection->escape($value),
        array_values($record)
    );
    $sql = 'INSERT INTO orders (
        . implode(', ', $columns)
```

```

        . ') VALUES (' . implode(', ', $values) . ')';
$connection->execute($sql);

$lastInsertedId = $connection->execute(
    'SELECT LAST_INSERT_ID()'
)->fetchColumn(0);

$this->container->get('session')->set(
    'currentOrderId',
    $lastInsertedId
);
// ...

return new Response(/* ... */);
}

```

If you are able to read between the lines (and see through the SQL statements), you'll learn a few things. As a user you can order an e-book by its ID. You have to provide your email address, and your order will be persisted as a record in the `orders` table. The `orders` table itself has an auto-incrementing identifier column. After inserting a new record into the `orders` table, the controller fetches the automatically assigned ID for it and stores it in the session.

It may already be obvious to you that this is bad code. Nonetheless, I'll briefly mention the disadvantages here:

1. It's really hard to find out what the story, or *scenario* of this action is. What does the use case of ordering an e-book entail? Which steps are involved? What is the outcome?
2. Implementation details obscure the view on the higher level steps of the scenario. For instance, one step would be to save the new order. The code doesn't say "save order" though. It simply shows how that saving is done. We have to analyze the SQL statement and notice that it's an `INSERT` statement, from which we can derive that this is the code for saving a new order.
3. Mixing high-level steps with low-level implementation details directly couples the use case itself to any related technological decisions. This makes it very difficult to change directions later on. For instance, we would have a hard time migrating to a different database. We would also have a hard time replacing the web form with a JSON API for this use case.

In this chapter we'll focus on the second problem. We want to save the order in one step, and hide the details of how exactly that's done. In Chapter 3 and Chapter 4 we'll work on the remaining issues.

2.2 Trying to fix it with a table data gateway

A traditional solution for pushing SQL statements outside of regular code is to use the *Table Data Gateway* (or table gateway for short) design pattern⁷. It hides the SQL statements and other implementation details behind a single interface per database table. Listing 2.2 shows what the controller action looks like when we'd start using table gateways.

Listing 2.2: `orderEbookAction` uses *Table gateways* for interacting with the database.

```
public function orderEbookAction(Request $request): Response
{
    $ebooksGateway = $this->container->get('ebooks_gateway');

    $ebookPrice = $ebooksGateway->select(
        [
            'id' => $request->request->get('ebook_id')
        ]
    )[0]['price'];

    $orderAmount = (int)$request->get('quantity')
        * (int)$ebookPrice;

    $ordersGateway = $this->container->get('orders_gateway');

    $lastInsertedId = $ordersGateway->insert(
        [
            'email' => $request->get('email_address'),
            'quantity' => (int)$request->get('quantity'),
            'amount' => $orderAmount
        ]
    );

    $this->container->get('session')->set(
        'currentOrderId',
        $lastInsertedId
    );

    // ...

    return new Response(/* ... */);
}
```

The method is now a lot shorter. We no longer have SQL statements in our regular code. When reading it, we don't have to switch contexts as often, so it's definitely easier to understand what's going on. Still, we only managed to fix half of the original problem. We could hide only some implementation details (the table names, the SQL queries), but left other implementation details inside the controller action (column names and types). This leaves us still very much coupled to the technological decisions we made for this piece of code: the code remains very table-oriented code, so we're stuck using a relational database.

Offering generic table-oriented manipulations like a table gateway does, exposes another problem that we didn't notice yet, but has been there from the start. There's nothing preventing us from inserting a string into the `email` column that doesn't even look like an email address. The table gateway also doesn't prevent us from incorrectly calculating an order amount and inserting it directly into the database. In other words: we aren't able to protect the *internal consistency* of an e-book order.

2.3 Designing an entity

We want to take a customer's order of an e-book, and remember it, so we can later process it. So before saving an order, we should make sure that it's complete and correct. If we don't, we may even have to contact our customers afterwards and ask them to correct the data we received from them. We also want to make sure that what ends up in the database, can safely be used by other parts of the application, for instance by the payment module, or the fulfillment module, which will send the e-book to the customer.

We're lucky to be doing object-oriented programming, because with *objects* we can achieve all of our goals. When an object gets created, it can accept data as constructor arguments, analyze that data, and throw exceptions when any part of it doesn't look right, or leads to the object ending up in an inconsistent or incomplete state. In our case we would define an `Order` class (see Listing 2.3), which by the presence of its constructor parameters can force its creator to provide all the necessary data at once.

Listing 2.3: The initial version of the `Order` entity.

```
final class Order
{
    private int $ebookId;
    private string $emailAddress;
    private int $quantityOrdered;
    private int $pricePerUnitInCents;
    private int $orderAmountInCents;

    public function __construct(
        int $ebookId,
        string $emailAddress,
        int $quantityOrdered,
        int $pricePerUnitInCents,
        int $orderAmountInCents
    ) {
        $this->ebookId = $ebookId;
        $this->emailAddress = $emailAddress;
        $this->quantityOrdered = $quantityOrdered;
        $this->pricePerUnitInCents = $pricePerUnitInCents;
        $this->orderAmountInCents = $orderAmountInCents;
    }
}

// A client has to supply all the required arguments:
$order = new Order(/* ... */);
```

Forcing clients to supply a number of arguments when instantiating the `Order` class is not enough to guarantee consistency. The information that clients provide could still be invalid or meaningless. For example, the current implementation won't stop you from instantiating an `Order` in the following, obviously invalid way: `new Order(-10, 'foobar', 0, 1000000, 25)`.

We can improve the constructor by doing some basic checks inside of it, using assertions from one of the available assertion libraries (e.g. `beberlei/assert`⁸), or using native assertions⁹ if you like. See Listing 2.4 which shows how predefined assertion functions can be used to prevent bad data from being assigned to properties of an `Order` instance.

Listing 2.4: The `Order` entity validates its constructor arguments using assertions.

```
use Assert\Assertion;

final class Order
{
    // ...

    public function __construct(
        int $ebookId,
        string $emailAddress,
        int $quantityOrdered,
        int $pricePerUnitInCents,
        int $orderAmountInCents
    ) {
        Assertion::greaterThan($ebookId, 0);
        Assertion::email($emailAddress);
        Assertion::greaterThan($quantityOrdered, 0);
        Assertion::greaterThan($pricePerUnitInCents, 0);
        Assertion::greaterThan($orderAmountInCents, 0);

        $this->ebookId = $ebookId;
        $this->emailAddress = $emailAddress;
        $this->quantityOrdered = $quantityOrdered;
        $this->pricePerUnitInCents = $pricePerUnitInCents;
        $this->orderAmountInCents = $orderAmountInCents;
    }
}
```

The code in `Assertion::greaterThan()` will throw an exception if the `$ebookId` is 0 or less. Likewise, if `$emailAddress` is a string, but doesn't look like an email address, it will throw an exception. These assertions will thereby prevent an `Order` object from being instantiated with invalid data. With these assertions in place, the `Order` object can provide basic consistency for the data it holds.

“Can we use these assertion functions for validating user input?”

If the user fills in an invalid looking email address, we'd likely want to show a nice and friendly error message next to the email form field where the user provided it. With the current version of our `Order` entity, we wouldn't be able to do that, because calling `Assertion::email()` with a string that does not look like an email address will throw an exception. If you don't catch that exception somewhere, it will just show the application's default error page with some generic message like “Oops, an error occurred”. In short: assertions won't be very useful when we need to validate user input. Instead, they should be used by objects as a way to protect themselves against incomplete, inconsistent, or meaningless data. When it comes to talking back to a user, informing them about their mistakes, you should look for alternatives. We'll discuss several of those in Chapter 8.

A stateful object that guarantees its own consistency, and is going to be persisted somehow, is often called an *Entity*¹⁰. Entities by definition have an identity, which we can use to save it and get it back from storage again. Even though our `Order` doesn't have an identity (ID) yet, we are going to give it an identity in Section 2.6, so let's consider `Order` to be an entity already.

We're almost ready to use the new `Order` instance in the controller action. There's one thing that prevents us from doing so: the table gateway for `orders` has an `insert()` method which accepts an array of `columns => values` (see Listing 2.5). But now that we pass the form data to the constructor of `Order`, we no longer have such an array inside the controller. We could add it back in, but getting rid of actual column names inside the controller action was already on our list of improvements, so we shouldn't do that.

Listing 2.5: To save an `Order` entity using a table gateway, we still need a map of columns to values.

```
public function orderEbookAction(Request $request): Response
{
    // ...

    // We'd like to use the new 'Order' entity...

    $order = new Order(
        $request->get('ebook_id'),
        $request->get('email_address'),
        (int)$request->get('quantity'),
        (int)$pricePerUnitInCents,
        (int)$orderAmount
    );

    // But how to save an 'Order' object to the database?

    $ordersGateway = $this->container->get('orders_gateway');
    $lastInsertedId = $ordersGateway->insert(
        [
            // We need a map of columns => values
        ]
    );
    // ...
}
```

The thing we want to save (the `Order` object), and the thing that can save it (the `OrdersGateway`) turn out to be incompatible. But we still want to save the `Order` object to the database somehow, so we need to find a different design for the thing that can do this.

2.4 Introducing a repository

If a certain facility is not yet available in a project, you can apply a programming trick: act as if it was already available. For instance, if you’re looking for a thing that can save an Order to the database, just imagine that the thing already exists, and start using it (see Listing 2.6).

Listing 2.6: An imagined object for saving orders.

```
$order = new Order(/* ... */);  
$lastInsertedId = $orderSaver->save($order);
```

To be useful for us in this spot, the thing that can save an order only needs a `save()` method with a single parameter of type `Order`. Since the database determines the ID of a new order using an auto-incremented integer column, we could give this method an `int` return type, so a client of the method can later use the newly assigned ID. Let’s formalize all of this by defining an interface for our “order saver” (see Listing 2.7).

Listing 2.7: The `OrderSaver` interface.

```
interface OrderSaver  
{  
    /**  
     * @return int The ID of the saved 'Order'  
     */  
    public function save(Order $order): int;  
}
```

“Object savers” are usually called *repositories*. Repository is the name of a design pattern which provides a solution to a common problem: the need to save a domain object, and later reconstitute it. To make it clear that we intend to use the repository design pattern¹¹ here, let’s rename `OrderSaver` to `OrderRepository` (see Listing 2.8).

Listing 2.8: The `OrderRepository` interface.

```
interface OrderRepository  
{  
    public function save(Order $order): int;  
}
```

“Shouldn’t we also have a `getById()` method?”

Besides saving an entity, a repository usually offers a way to retrieve a previously saved entity from the database. Normally a repository has a `getById()` method that allows clients to do so:

```

interface OrderRepository
{
    public function save(Order $order): void;

    /**
     * @throws CouldNotFindOrder
     */
    public function getById(int $orderId): Order;
}

```

A client provides the ID of the entity that it wants to retrieve, and the repository takes the data for the corresponding order from the database, and finally reconstitutes the entire entity object using this data. If the repository can't find an order with the provided ID, it will throw a custom exception (e.g. `CouldNotFindOrder`) which extends from `RuntimeException`.

Since we're refactoring existing code for only one controller, I didn't want to propose adding a `getById()` method to the `OrderRepository` right away, but when the time comes, it's good to know that `save()` has a symmetrical counterpart called `getById()`.

In the controller, things are becoming much cleaner now (see Listing 2.9). We can instantiate a new `Order` object and hand it over to the `OrderRepository`, which will then save it to the database. Assuming that we can somehow get a working `OrderRepository` instance from our service container, that is.

Listing 2.9: `orderEbookAction()` now uses the `OrderRepository` and the `Order` entity.

```

public function orderEbookAction(Request $request): Response
{
    // ...

    $order = new Order(/* ... */);

    $orderRepository = $this->container->get('order_repository');
    $lastInsertedId = $orderRepository->save($order);

    // ...
}

```

2.5 Mapping entity data to table columns

So far we've been working with the `OrderRepository` interface, which currently has no implementation. Writing an implementation that actually saves an `Order` to the database may not be as straightforward as we'd hope. As you can see in Listing 2.10, at some point we'll still need that array of `columns => values` that we no longer have.

Listing 2.10: SqlOrderRepository needs a map of columns and values.

```
final class SqlOrderRepository implements OrderRepository
{
    private Connection $connection;

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

    public function save(Order $order): int
    {
        $data = [
            // Again, we need an array of columns => values
        ];

        $columns = array_keys($data);
        $values = array_map(
            fn ($value) => $this->connection->escape($value),
            array_values($data)
        );
        $sql = 'INSERT INTO orders (
            . implode(', ', $columns)
            . ') VALUES (' . implode(', ', $values) . ')';

        $this->connection->execute($sql);

        $lastInsertedId = $this->connection->execute(
            'SELECT LAST_INSERT_ID();'
        )->fetchColumn(0);

        return $lastInsertedId;
    }
}
```

There are different options here. The most common one is to install an ORM in your project, which can do the mapping from object properties to table columns for you.

2.5.1 Using an ORM

In Section [2.8](#) we'll talk about what type of ORM works best in this scenario. For now, let's look at an example using the popular Doctrine ORM^{[12](#)}. Once we have installed Doctrine ORM in our project and have set up the database connection we first need to add mapping configuration to the entity class and its properties. Listing [2.11](#) shows how to do that using annotations.

Listing 2.11: Using annotations for mapping configuration.

```
use Doctrine\ORM\Mapping as ORM;

/**
```

```

 * @ORM\Entity
 * @ORM\Table(name="orders")
 */
final class Order
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private int $id;

    /**
     * @ORM\Column(type="string")
     */
    private string $emailAddress;

    /**
     * @ORM\Column(type="int")
     */
    private int $quantityOrdered;

    /**
     * @ORM\Column(type="int")
     */
    private int $pricePerUnitInCents;

    // ...
}

```

Based on the annotations Doctrine should be able to save the object's data in the right table and columns. We can now write a very straightforward implementation of the `OrderRepository` interface which uses Doctrine's `EntityManager` to persist `Order` objects (see Listing 2.12).

Listing 2.12: An implementation of `OrderRepository` using Doctrine ORM.

```

use Doctrine\ORM\EntityManagerInterface;

final class OrderRepositoryUsingDoctrineOrm implements OrderRepository
{
    private EntityManagerInterface $entityManager;

    public function __construct(EntityManagerInterface $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function save(Order $order): void
    {
        $this->entityManager->persist($order);
        $this->entityManager->flush();
    }
}

```

As you can see in these code samples, it doesn't take a lot of work to install and use an ORM to save an entity to the database. And yes, it might save you some time, and potentially many lines of code, but it may also get you into trouble. Personally I've spent a lot of time trying to figure out how or why something didn't work, or why Doctrine suddenly had to do so many queries. It also happened more than once that I found out something was broken when it was already running in production. The problem is not Doctrine ORM itself, but using generic abstractions. Hiding away so many implementation details and so much "magic" behind a single abstract `EntityManagerInterface` means you'll run into trouble sooner or later. Having said that, there are also several advantages to using a popular ORM, like:

1. Extensive documentation, online examples, blog posts, questions and answers on Stack Overflow, etc.
2. Automated solutions for common problems like database migrations, fixture loading, etc.

In my experience, it's okay to use an ORM if you can stick to the following rules:

1. Only use simple mapping configuration; no table inheritance, "embeddables", custom types, etc.¹³
2. Stick to one-to-many associations.
3. Reference entities by their ID.
4. Don't jump from entity to entity using association fields.

It's not a coincidence that these rules have much in common with the rules for "effective aggregate design" as described by Vaughn Vernon¹⁴. We'll get back to this topic in Section [11.2](#).

How is Doctrine able to get the data out of the entity? It uses *reflection*¹⁵ to reach into the object, copy the data from the object's private properties, and prepare the desired array using this data. Listing [2.13](#) shows what it would look like if we'd inline the mapping code in our own `save()` method.

Listing 2.13: An implementation of `save()` that uses reflection.

```
public function save(Order $order): int
{
    // ...
    $data = [];
    $object = new ReflectionObject($order);
    $emailProperty = $object->getProperty('emailAddress');
    // Make the private property accessible:
    $emailProperty->setAccessible(true);
    // Get the current value of the emailAddress property:
    $data['email'] = $emailProperty->getValue($order);

    // And so on, for all the properties of 'Order'...
    // ...
}
```

2.5.2 Manual mapping

Though it's great that Doctrine can do all this work for us, we don't see how it does all of it. And this is the reason that it'll be hard to find out what the problem is when anything doesn't work as expected. In the past few years I've come to the conclusion more than once that doing the mapping manually, that is, writing the code for this myself, can be a pretty good solution. In that case we can, but don't have to use reflection, and we don't need separate mapping configuration (using annotations, or XML, etc.).

There are two implementation options: either you let the entity prepare the `columns => values` array, or you let it expose its internal data as an array and do the mapping inside the repository. Listing [2.14](#) shows an example of the first option.

Listing 2.14: `save()` retrieves the mapped data from the `Order` entity.

```
final class Order
{
    // ...

    public function mappedData(): array
    {
        return [
            'email' => $this->emailAddress,
            'quantity' => $this->quantityOrdered,
            // ...
        ];
    }
}

final class SqlOrderRepository implements OrderRepository
{
    // ...

    public function save(Order $order): int
    {
        // ...

        $data = $order->mappedData();

        // $data is an array of columns => values

        // ...
    }
}
```

The downside of this approach is that the `Order` entity has knowledge about the names and types of the database columns. Whenever a column gets renamed, you would also have to update the `Order` class.

Listing [2.15](#) shows the alternative, where the entity only exposes its internal data, and the repository performs the mapping itself.

Listing 2.15: save() performs the mapping to database columns.

```
final class Order
{
    // ...

    public function internalData(): array
    {
        return get_object_vars($this);
    }
}

final class SqlOrderRepository implements OrderRepository
{
    // ...

    public function save(Order $order): int
    {
        // ...

        $internalData = $order->internalData();

        $data = [
            'email' => $internalData['emailAddress'],
            'quantity' => $internalData['quantityOrdered'],
            // ...
        ];
        // ...
    }
}
```

The downside of this approach is that it reduces the level of encapsulation of `Order`. The `SqlOrderRepository` has to know about `Order`'s private properties: how many properties there are, and what their names and types are. Whenever you rename a property, change its type, add or remove properties, you'd also have to update the corresponding mapping code in `SqlOrderRepository`.

I prefer `Order` to keep the names and the types of its internal properties to itself. It will be very hard to refactor the `Order` object if its internals are no longer private. The ability to freely change the internal structure of an object is what enables you to improve its design. So for me exposing private property names is too high a price to pay, and we should go with the first option: letting the entity do the mapping itself.^{[16](#)}

“But now we have column names inside the entity...Doesn’t that cause an unhealthy mix of infrastructure and core code?”

A great question, with a subtle answer.

To demonstrate that having column names inside an entity class does not automatically turn it into infrastructure code, let's check the rules once more.

1. Core code doesn't directly depend on external systems, nor does it depend on code written for interacting with a specific type of external system.
2. Core code doesn't need a specific environment to run in, nor does it have dependencies that are designed to run in a specific context only.

Take a look at the `mappedData()` method:

```
final class Order
{
    // ...

    public function mappedData(): array
    {
        return [
            'email' => $this->emailAddress,
            'quantity' => $this->quantityOrdered,
            // ...
        ];
    }
}
```

When calling `mappedData()`, external systems like the database, don't have to be available. In fact, the code in this method has no dependencies at all. The `mappedData()` method can run in any context, without any special setup. It doesn't need any special setup: you can instantiate an `Order` object in the normal way and call this `mappedData()` method. This code has no dependencies that are designed to run in a specific context either. So `mappedData()` complies with both rules for core code. How could it not be core code; it only does some simple transformations on values in memory.

How about adding Doctrine mapping annotations to your entity, like `@Entity`, `@Table`, and `@Column`? Does that result in mixed code? Well, instantiating an entity with mapping annotations doesn't require any special setup. And calling any method on it doesn't require external dependencies to be available. So if your entity is ready to be persisted using Doctrine, it should still be considered core code, not infrastructure code.

However, an entity with Doctrine annotations or a `mappedData()` method does contain technical implementation details (like table and column names and column types). So when you get to the point where you want to switch databases after all, you will still have to modify this code. For me this is no reason to move the mapping code outside of the entity. In particular because it's so convenient to keep the entity's properties and the mapping code closely together.

2.6 Generating the identifier earlier

Let's take another look at the controller code as it was after we started using the `Order` entity and the `OrderRepository` (see Listing 2.16).

Listing 2.16: The current state of the `orderEbookAction()`.

```
public function orderEbookAction(Request $request): Response
{
    // ...

    $order = new Order(/* ... */);

    $orderRepository = $this->container->get('order_repository');
    $lastInsertedId = $orderRepository->save($order);

    // ...
}
```

By introducing the `OrderRepository` interface we managed to hide most of the implementation details related to saving orders. There is only one left. The ID of the `Order` that we create is an integer. Its value will only be known to us when the `OrderRepository` has saved the order, which is why that integer is the return value of its `save()` method. This still reveals to the reader of the code that the mechanism used to persist an `Order` uses an auto-incrementing integer column for the primary ID of an order.

It's not necessarily bad if an object reveals part of its inner workings, although we generally tend to avoid it. The real issue here is that we're still not in the position of completely replacing the underlying storage technology. Not every database will support auto-incrementing ID columns or fields. Not every database will be able to generate an ID and return it. And in the most extreme case: some persistence mechanisms might not even be able to synchronously return an identifier to the client.

Another problem is that the `Order` entity is supposed to be complete from its beginning. It should hold the minimum set of data, in order to be useful, and consistent in its behavior. Given that an `Order` doesn't have an ID until it has been saved, we should come to the opposite conclusion: `Order` isn't consistent, until the database has finished saving it.

What we'd like instead is a way to provide an `Order` with an ID the moment we instantiate it. Adding it as a required constructor argument would accomplish this. By doing so, we can make sure that an `Order` object always has an identifier (see Listing 2.25).

Listing 2.17: `Order` now has an identity from the start.

```
final class Order
{
    private int $id;

    // ...

    public function __construct(
```

```

        int $id
        /* ... */
    ) {
    $this->id = $id;
    // ...
}

```

Clients will then have to supply an ID upfront when they want to create a new Order. But how could a client find out what the next available ID is? Given that the `OrderRepository` is close to the source of this knowledge, namely the `orders` table itself, let's give it a new method which can answer this question, and call it `nextIdentity()` (see Listing 2.18).

Listing 2.18: `nextIdentity()` returns the next available ID.

```

interface OrderRepository
{
    public function nextIdentity(): int;
    // ...
}

```

A possible implementation of the `nextIdentity()` method could be the one shown in Listing 2.19. It selects the highest ID currently in use, and returns the next number, which will therefore be the next available ID.

Listing 2.19: A naive implementation of `nextIdentity()`.

```

final class SqlOrderRepository implements OrderRepository
{
    // ...

    public function nextIdentity(): int
    {
        return (int)$this->connection->execute(
            'SELECT MAX(id) AS highestId FROM orders'
            )->fetchColumn(0) + 1;
    }
}

```

You may have already spotted the problem: if the application has many concurrent users, chances are that two clients end up trying to insert a record with the same ID. If concurrency is usually not an issue for you, the naive implementation can be the right implementation. If you start having concurrency issues or if you already have them, you could switch to a more robust implementation, for example one that uses a sequence at database-level. If your database doesn't support sequences, it wouldn't be a lot of work to write the code yourself. See Listing 2.20 for a sample implementation.

Listing 2.20: This version of `nextIdentity()` uses a sequence table.

```
public function nextIdentity(): int
{
    return $this->connection->transactional(function () {
        $nextId = (int)$this->connection->execute(
            'SELECT last_id FROM order_id_sequence'
        )->fetchColumn(0) + 1;

        $this->connection->execute(
            'UPDATE order_id_sequence SET last_id = :last_id',
            [
                'last_id' => $nextId
            ]
        );
        return $nextId;
    });
}
```

Once we have added a suitable implementation of `nextIdentity()` we no longer have to wait for the database to return the auto-incremented ID value to us. Instead, now that the `Order` entity already contains its ID from the start, we should use that ID when mapping the entity's data to the columns in the database (see Listing [2.21](#)).

Listing 2.21: `mappedData()` also returns a value for the `id` column.

```
final class Order
{
    // ...

    public function mappedData(): array
    {
        return [
            'id' => $this->id,
            'email' => $this->emailAddress,
            // ...
        ];
    }
}
```

In the controller, things are looking much better now: we first get the next order ID, provide it to `Order` as a constructor argument, then save the `Order` using the repository (see Listing [2.22](#)).

Listing 2.22: `orderEbookAction()` uses `nextIdentity()` to determine the order ID upfront.

```
public function orderEbookAction(Request $request): Response
{
    // ...

    $orderId = $orderRepository->nextIdentity();
```

```

$order = new Order(
    $orderId,
    // ...
);

$orderRepository->save($order);

// ...
}

```

Since we no longer have to rely on `save()` to return the new order's ID, we should remove that `int` return type from the repository's `save()` method (see Listing 2.23).

Listing 2.23: `save()` should return `void` now.

```

interface OrderRepository
{
    public function save(Order $order): void;
}

```

As a bonus, this will make the `save()` method conform to the *Command Query Separation principle*.¹⁷

2.6.1 Using UUIDs instead of (auto-)incrementing integer IDs

A good alternative to using incrementing integers would be to use a so-called Universally Unique Identifier (UUID). A UUID is often represented as a string, but it can be converted to a big integer, and back again. When you encounter it as a string, it will look like this: `eb13b0b9-d320-4a45-84f1-62adfc5e0a8e`. A UUID is based on two elements: the current time, and a random number generated by the system's random device. Listing 2.24 shows an implementation of `OrderRepository::nextIdentity()` which uses the `ramsey/uuid`¹⁸ library for generating a random UUID.

Listing 2.24: `nextIdentity()` returns a UUID.

```

use Ramsey\Uuid\Uuid;
use Ramsey\Uuid\UuidInterface;

final class SqlOrderRepository implements OrderRepository
{
    // ...

    public function nextIdentity(): UuidInterface
    {
        return Uuid::uuid4();
    }
}

```

Now we only have to change the type of the `Order`'s `$id` constructor parameter to accept a `UuidInterface` instance, instead of an `int` (see Listing 2.25).

Listing 2.25: The ID of an `Order` is an instance of `UuidInterface` now.

```
final class Order
{
    private UuidInterface $id;

    // ...

    public function __construct(
        UuidInterface $id
        /* ... */
    ) {
        $this->id = $id;

        // ...
    }
}
```

“Wait, is UUID the best we can get?”

According to several readers who have emailed me, there are some aspects of UUID (version 4) that we should be aware of. For instance, Thomas Nunninger writes that “As far as I understood, innodb reorders the records of a table by the primary key when you insert new records. So if you have a random UUID it has to reorganize the database pages all the time.” Also, there may be more modern solutions available when you’re reading this, as Luis Ramon Lopez writes: “Some weeks ago I heard about ULIDs...I think they may be a nice addition to the chapter as it’s a good alternative to UUIDs in some cases.”

In the context of this book I think there are two important messages here:

1. Technology changes. There will always be new solutions for old problems like ID generation.
2. Technical decisions have an influence on more than just the design qualities of our code.

As developer-architects we should be aware of the consequences of picking a technology, like UUID version 4, or ULID. We need to answer the question how our choice influences database performance, how it influences usability, and so on. At the same time, we should keep an eye on technological developments. In order to keep making the best decisions, we should know what's going on, and be ready to make a better decision tomorrow. At the same time, and this is the entire point of this book, we shouldn't be forced to update our entire code base when some external technology changes.

2.7 Using a value object for the identifier

Now that we're changing the type of an Order's identifier, let's wrap the identifier inside a *Value object*¹⁹. That way, we can fully encapsulate the actual data type of the identifier, allowing it to be an internal implementation detail of the entity. Listing 2.26 shows the new OrderId value object class, and the changes we have to make to the Order entity and the OrderRepository to use this new type instead of int or UuidInterface.

Listing 2.26: OrderId hides the underlying ID type.

```
final class OrderId
{
    private UuidInterface $id;

    private function __construct(UuidInterface $id)
    {
        $this->id = $id;
    }

    public static function fromUuid(UuidInterface $id): self
    {
        return new self($id);
    }
}

final class Order
{
    private OrderId $id;

    // ...

    public function __construct(
        OrderId $id
        /* ... */
    ) {
        $this->id = $id;

        // ...
    }
}

final class SqlOrderRepository implements OrderRepository
{
    // ...

    public function nextIdentity(): OrderId
    {
        return OrderId::fromUuid(
            Uuid::uuid4()
        );
    }
}
```

The code in the controller is looking great now (see Listing 2.27). It shows all the steps involved in creating and saving an order, and none of it is directly tied to the particular database that we use.

Listing 2.27: `orderEbookAction()` now uses `nextIdentity()` to determine the order ID upfront.

```
public function orderEbookAction(Request $request): Response
{
    // ...

    $orderRepository = $this->container->get('order_repository');
    $orderId = $orderRepository->nextIdentity();

    $order = new Order(
        $orderId,
        $request->get('ebook_id'),
        $request->get('email_address'),
        (int)$request->get('quantity'),
        (int)$pricePerUnitInCents,
        (int)$orderAmount
    );

    $orderRepository->save($order);

    $this->container->get('session')->set(
        'currentOrderId',
        $orderId
    );
    // ...
}
```

We didn't break anything, except the code that saves the current order ID in the session. Since `$orderId` is now an `OrderId` instance, this will no longer work; you can only save things in a session that are serializable. In this case, we could add a simple `asString()` method to the `OrderId` class, and call it when saving the current order ID in the session, as shown in Listing 2.28.

Listing 2.28: `OrderId` supports serialization using its `asString()` method.

```
final class OrderId
{
    // ...

    public function asString(): string
    {
        // 'Ramsey\Uuid' conveniently has a 'toString()' method
    }
}
```

```

        return $this->id->toString();
    }
}

$this->container->get('session')->set(
    'currentOrderId',
    $orderId->asString()
);

```

2.8 Active Record versus Data Mapper

So far, without mentioning it, we followed the Data Mapper design pattern^{[20](#)} for storing entities. This means that we have an object, and when we want to store it, we give it to a repository, which then takes out the data and stores it in the database. A common alternative for storing entities is the *Active Record* design pattern^{[21](#)}. If you use this pattern, the entity will be able to load itself from the database, and it can save and delete itself as well. Bringing in this extra functionality is usually achieved by extending the entity class from a class provided by the framework (see Listing [2.29](#)).

Listing 2.29: An Active Record entity fulfills repository tasks as well.

```

final class Order extends ActiveRecordEntity
{
    // ...
}

// We can create an order
$order1 = new Order();

// Save it to the database
$order1->save();

// Delete it, if we like
$order1->delete();

// Or load one from the database
$order2 = Order::getById(2);

```

This may look very convenient, but there are downsides to it from a design perspective:

- By inheriting a lot of infrastructure code, we lose the isolation we need for proper unit testing an `Order`.
- Active Record frameworks usually require a lot of custom code inside your entities to make everything work well. This code is specific to the framework, making your domain model directly coupled to, and only functional in the presence of that framework.
- Clients of `Order` can do many more things with the object than they most likely should be allowed to do.

We don't have any of these downsides when we apply the data mapper pattern, like we have done earlier in this chapter.

The biggest downside I can think of when using an entity and a repository is that you have several extra elements: the repository interface and at least one repository implementation. In the context of this book, that shouldn't be considered a downside; by introducing an abstraction you achieve decoupling from surrounding infrastructure. We have seen one other downside in this chapter already, when we realized that we had to make a decision: should we do the mapping to database columns inside or outside the entity? Although we'd rather not make this kind of decision, it does not compare to the issues that Active Record has. Comparing these design patterns, it's clear that data mapper allows for a better separation between core and infrastructure code.

If Active Record is omnipresent in your project and the team is very effective with it, you may still want to stick to your framework's Active Record (AR) approach. In that case, my advice is to mitigate some of the downsides by keeping yourself to the following rules:

- Design your AR entities like real entities. Take a look at Section [11.2](#) to learn more about this topic.
- Don't use the same AR entity for changing state and retrieving state. Separate your model into a write and a read model. Take a look at Chapter [3](#) for more information about read models.
- Don't use your AR entity to navigate from one AR entity to other AR entities. If you want to make a change to a different AR entity, fetch it by its ID from the corresponding repository.
- Ignore the fact that an AR entity provides typical service facilities like saving and deleting. Use a repository and *double dispatch* to perform these tasks.

As an example, you should consider the `save()` method of the `Order` entity to be unavailable for regular clients. Instead, whenever you want to save an `Order`, save it using the repository's `save()` method (see Listing [2.30](#)).

Listing 2.30: Saving an AR entity through the repository.

```
final class ActiveRecordOrderRepository implements OrderRepository
{
    public function save(Order $order): void
    {
        $order->save();
    }

    // ...
}
```

Likewise, when you want to retrieve an `Order` by its ID, use the repository as well (see Listing [2.31](#)).

Listing 2.31: Retrieving an AR entity through the repository.

```
final class ActiveRecordOrderRepository implements OrderRepository
{
```

```

// ...

public function getById(OrderId $orderId): Order
{
    return Order::find($orderId->asString());
}

```

A problem that doesn't go away with this approach is that there will be code inside your AR entity that is framework or ORM-specific. This isn't necessarily a problem. You will always need some code to satisfy a framework or ORM, whether it's based on the Active Record or Data Mapper pattern. Just keep as many of these details inside the object. And design your entities in such a way that you can instantiate them and call methods on them without the need for any special setup. This will allow you to write actual unit tests for your entities, instead of the more expensive (harder to write, harder to maintain) integration tests which need an actual database to run.

Is all this extra work really necessary? It seems like this would create extra layers around out-of-the-box framework functionality that was supposed to give you the highest development speed possible. Well, the extra work is needed if you want to separate core from infrastructure code, which I assume you do since it's the premise of this book. On the other hand, if your expert intuition tells you that this may indeed be too much work for not enough gains, you should not ignore that feeling. In Chapter 15 we'll take a closer look at the risk of over-engineering and how to decide if the approach described in this book is a good choice for your particular situation.

2.9 Summary

In this chapter, we started with a controller action with mixed domain logic and SQL statements. Trying to separate core code from infrastructure code, we took an intermediate refactoring step, introducing a table gateway. This left us with code that was still tied to a specific technology – a relational database. Finally we refactored the code using two known design patterns: *Entity* and *Repository*. The initial implementation could be improved by providing the entity with an ID upfront. The repository turned out to be a good place for generating that ID. By introducing a value object for the entity's ID we were able to encapsulate the underlying data type of the ID, leaving that implementation to the repository.

Now that we have an entity class, a repository class, and a repository interface, we can tick all the boxes and mark this code as core code:

1. The `Order` entity and `OrderRepository` interface don't reveal anything about the client that's going to call or use it.
2. The `OrderRepository` interface is itself an abstraction, using which we can later run other core code without actual databases, or other external dependencies.
3. The `Order` entity can indeed be used without an actual web server, database, etc. It's a plain old PHP class that you can run without any special setup.

We started with mixed code and extracted an entity and a repository interface, including a default implementation. In Chapter 4, we'll extract even more core code from the controller and fix the remaining problems:

1. Scenario steps for creating an e-book order are still pretty unclear and mixed with low-level implementation details
2. The controller action is only useful in a web application where input is provided by filling in a web form.

First, let's take a look at read models.

Exercises

1. What's wrong about the following abstraction for saving Order entities?[22](#)

```
interface OrderRepository
{
    public function insert(string $tableName, array $data): void;
}
```

2. Is the following class an entity or a value object?[23](#)

```
final class OrderId
{
    // ...

    public static function fromString(string $orderId): self
    {
        return new self($orderId);
    }
}
```

3. Is the following class an entity or a value object?[24](#)

```
final class Order
{
    // ...

    public static function create(OrderId $orderId): self
    {
        return new self($orderId);
    }
}
```

4. Why should we generate an identity value for an entity before instantiating it for the first time?[25](#)

1. Because the entity wouldn't be complete, nor behave consistently, without an identity.
2. Because otherwise the same identity might be used in another process.
3. Because by letting the database determine it, we are implicitly relying on its ability to do so, which might not be true for alternative persistence mechanisms.

5. Should the following code be considered mixed code (combining core and infrastructure code)?[26](#)

```
use Doctrine\ORM\Mapping as ORM;

/**
 * @Entity
 * @Table(name="todo_items")
 */
final class ToDoItem
{
    /**
     * @Id
     * @Column(type="integer", name="id")
     * @GeneratedValue
     */
    private int $id;

    /**
     * @Column(type="string", name="desc")
     */
    private string $description;

    ...
}
```

1. Yes, it mentions database columns and their types.
2. No, you can instantiate and use this class in any context, and it doesn't rely on external systems.

6. Should the following code be considered core or infrastructure code?[27](#)

```
final class ToDoItemUsingDoctrineRepository
    implements ToDoItemRepository
{
    private EntityManagerInterface $em;

    public function __construct(EntityManagerInterface $em)
    {
        $this->em = $em
    }

    public function save(ToDoItem $ToDoItem): void
```

```
{  
    $this->em->persist($ToDoItem);  
    $this->em->flush();  
}  
}
```

1. Core code, because a repository is part of the domain model, which consists solely of core code.
 2. Infrastructure code, because this is a repository implementation that uses an external system (the database) to perform its task.
-

3 Read models and view models

This chapter covers:

- Creating a new model just for retrieving information
 - Different solutions for implementing a read model repository
 - Hiding query complexity behind view models (a special kind of read models)
-

In the previous chapter we got rid of two of the three SQL queries that were originally in the controller action:

- We moved the `INSERT INTO orders` query to the `SqlOrderRepository`.
- We got rid of the `SELECT LAST_INSERT_ID()` query by using our own mechanism to generate identity.

The remaining SQL query does a lookup in the `ebooks` table to find out the price of the e-book the user has selected to buy (Listing 3.1).

Listing 3.1: One remaining SQL query inside the controller.

```
public function orderEbookAction(Request $request): Response
{
    $connection = $this->container->get('connection');

    // Retrieve the price of the e-book

    $ebookPrice = $connection->execute(
        'SELECT price FROM ebooks WHERE id = :id',
        [
            'id' => $request->request->get('ebook_id')
        ]
    )->fetchColumn(0);

    $orderAmount = (int)$request->get('quantity')
        * (int)$ebookPrice;

    // Save the order (Chapter 1)

    // ...

    return new Response(/* ... */);
}
```

For the same reasons as we discussed in the previous chapter, we'd like to have a better way to represent this crucial step of the scenario; retrieving the price of an e-book. We want to hide the low-level implementation details (relational database, `ebooks` table, `price` column, etc.) behind a high-level interface, which represents *what* information we're interested in, instead of

how the system retrieves that information. Maybe we can apply the same kind of refactoring as we did in Chapter 2? That is, introduce an *Entity* to represent the e-book and retrieve it from its repository.

3.1 Reusing the write model

Let's assume we already have such an `Ebook` entity and an `EbookRepository` interface in our project, as shown in Listing 3.2. So far these classes have only been used inside the `EbookController` to add a new e-book to the catalog, or to change its details.

Listing 3.2: The `Ebook` entity and `EbookRepository` interface.

```
final class Ebook
{
    private EbookId $ebookId;
    private int $price;

    // ...

    public function __construct(
        EbookId $ebookId,
        int $price
        // ...
    ) {
        $this->ebookId = $ebookId;
        $this->price = $price;

        // ...
    }

    public function changePrice(int $newPrice): void
    {
        $this->price = $newPrice;
    }

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

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

    // More actions...
}

interface EbookRepository
{
    /**
     * @throws CouldNotFindEbook
     */
    public function getById(EbookId $ebookId): Ebook;
```

```

    /**
     * @throws CouldNotSaveEbook
     */
    public function save(Ebook $ebook): void;
}

```

Now let's find out if we can use this `Ebook` entity during the order process when we need to know the price of an e-book. The simplest thing we could do is add a `getPrice()` method to the entity. This makes it immediately usable inside `orderEbookAction()`. Listing 3.3 shows how the controller fetches the right `Ebook` entity from the `EbookRepository` based on the ID that the user provided by submitting the HTML form.

Listing 3.3: Using the `Ebook` entity inside the `OrderController`.

```

public function orderEbookAction(Request $request): Response
{
    $ebook = $this->ebookRepository->getById(
        EbookId::fromString($request->request->get('ebook_id'))
    );

    $ebookPrice = $ebook->getPrice();

    $orderAmount = (int)$request->get('quantity')
        * (int)$ebookPrice;

    // Save the order (Chapter 1)

    // ...

    return new Response(/* ... */);
}

```

This solution looks good. The existing `Ebook` entity is a convenient object to quickly get the information from, since it already carries that information inside its `$price` field. However, there are a couple of issues with reusing an existing entity in a different context.

First, the existing object is not designed to retrieve information from. We use it to add new e-books to our catalog. When we want to temporarily remove it from the catalog, but don't want to actually delete it, we call its `hide()` method and save it again. Anything else we do with this `Ebook` entity is related to state changes, and anywhere we load the entity we do so with the intention to manipulate it and save it. But now we've started using `Ebook` in the `createOrderAction()` where we don't want to change anything about it at all. We just want to get a bit of information from it, and for this reason we had to add the `getPrice()` method. However, by retrieving the full `Ebook` object, we gain access to all these methods that can change the object's state, like `changePrice()` and `hide()`. It's generally a smart idea to limit the number of methods that a client of an object has access to. Even more so, if those methods have side-effects like state changes. In this situation too, we should probably not use the `Ebook` entity when we only need information but never want to change it.

The other issue is related to reusing objects in general, not just entities. If you start reusing an object in different locations and for different reasons, the object starts to play too many roles at the same time. The more roles an object has to play, the more methods and therefore lines of code it will contain. Soon it becomes too big to read the code and understand what it does, let alone to make changes to it. When the methods are calling each other, or when they rely on the same object properties, it will be really difficult to change anything about it. Since many clients are now using the object, they rely on its behavior to stay the same. It will be difficult to assess whether a change is safe to make, or if it will break one of its clients which is still relying on some undocumented existing behavior. Such an object becomes *resistant* to change, which is a bad quality for objects in general. You probably recognize this chain of events: it's how legacy code is created.

Of course, without any reuse, it would be really hard to accomplish anything at all as a software developer. But at least keep track of the intended use of objects, and watch for tension in the design. When two clients use an object, soon it will attract behavior that's only relevant to one of its clients. Or a client may gain access to knowledge it shouldn't have. In my experience, you can prevent a lot of this design tension by introducing separate objects for changing information and retrieving information. Or as this is traditionally called: separating your write model from your read model. A client that needs an object for getting information from (reading) should not retrieve the same object as clients that want to make changes to it (writing).

Although the current version of our controller isn't in big trouble yet, we are now combining writing and reading in the same object, so we might as well go ahead and prevent problems by using separate objects for changing state and retrieving information. This means we leave the Ebook entity as it is, and we create a new object, an Ebook read model that serves the local need to know the price of the e-book. This Ebook read model is going to be a read-only object (also known as an *immutable* object). Clients that have access to the read model won't be able to (accidentally) change its state.

“Are getters on entities forbidden?”

I've been trying to make it clear that an entity shouldn't be used in a place where information is needed. Adding a getter to an entity is often a sign that you've loaded the entity just to get data from it. You should consider introducing a read model in such a scenario, just like we're going to do in the next section. It doesn't mean you can never add a getter to an entity. Usually I need at least a getter for the entity's ID and a getter for retrieving the internally recorded events (we'll talk about that in Section [3.3.2](#)). Depending on your situation, there might be other information that an entity has to expose. But always consider alternative designs too. Here is an example from a previous project where an entity was given an extra getter to expose its “booking period”:

```
// $vatReturn is the entity, $bookingPeriods a service
/*
 * 
```

```

 * We need to verify that the booking period of the VAT return wasn't
 * closed yet, before we try to roll it back:
 */
if ($bookingPeriods->isClosed($vatReturn->bookingPeriod())) {
    throw CouldNotRollBack::becauseBookingPeriodIsClosed();
}

$vatReturn->rollBack();

```

This is an alternative implementation where the entity wouldn't need to have that getter:

```

$vatReturn->rollBack($bookingPeriods);

// Inside the entity:
public function rollBack(BookingPeriods $bookingPeriods): void
{
    if ($bookingPeriods->isClosed($this->bookingPeriod)) {
        throw CouldNotRollBack::becauseBookingPeriodIsClosed();
    }

    // ...
}

```

Regardless of design alternatives and rules of thumb, there's no need to be afraid of getters. And they certainly aren't forbidden.

3.2 Creating a separate read model

In any situation where you need information you can introduce a read model. You frame the question in such a way that it's easy for you to ask, and you design the type of answer you want to retrieve. In our case, the question would be: give me the price of an e-book with ID [...]. The answer will be an object that represents the price of the e-book.

Generally there are two options for modeling the question with code. You could use the repository pattern once more, and create classes with the same or a similar name as the entity classes themselves (see Listing 3.4). In that case, you have to put the code in a different namespace to make it easy to distinguish between the read and the write model.

Listing 3.4: The Ebook read model and read model repository interface.

```

interface EbookRepository
{
    /**
     * @throws CouldNotFindEbook
     */
    public function getById(EbookId $ebookId): Ebook;

```

```

}

final class Ebook
{
    private EbookId $ebookId;
    private int $price;

    /**
     * @internal Only to be used by implementations
     *          of 'EbookRepository'
     */
    public function __construct(
        EbookId $ebookId,
        int $price
    ) {
        $this->ebookId = $ebookId;
        $this->price = $price;
    }

    public function price(): int
    {
        return $this->price;
    }
}

// usage in the controller:
$ebook = $this->ebookRepository->getById(
    EbookId::fromString($request->request->get('ebook_id'))
);
$ebookPrice = $ebook->price();

```

This approach is useful if you want to retrieve more than one piece of information, or you're pretty sure that you'll want to do that in the near future. For instance, let's say you also need the title of the e-book to save it on the `Order` itself. Then it makes sense to reuse that same `Ebook` read model and add a `title()` method to it, so we can get both the title and the price from the same object.

If on the other hand you only need one piece of information, you could let go of the pseudo-entity. The interface could have a method that directly returns the data you need. Listing 3.5 shows how this could result in a read model class (`Price`) and an interface with a method for retrieving a single price (currently represented as an integer) based on an `EbookId`.

Listing 3.5: The `Ebook` read model and repository interface.

```

interface GetPrice
{
    /**
     * @throws CouldNotFindEbook
     */
    public function ofEbook(EbookId $ebookId): int;
}

// usage in the controller

```

```

$ebookPrice = $this->getPrice->ofEbook(
    EbookId::fromString($request->request->get('ebook_id'))
);

```

As you can see in the usage example, the second approach could lead to code that is nicer to read (or more exotic, depending on your taste). But whether or not you should take the first or second approach depends on your situation.

3.3 Read model repository implementations

For now let's stick with the first option: an `Ebook` read model and an `EbookRepository` interface. Whenever a new `Ebook` entity gets created by an administrator, there should also be a corresponding `Ebook` read model object that can expose the e-book's price to the clients that need this information. Whenever the entity changes, the corresponding read model should also be updated to reflect those changes.

3.3.1 Sharing the underlying data source

The simplest solution to align the write with the read model would be to let the read model use the underlying data source of the entity. In our case, the data of an `Ebook` entity will be saved in the `ebooks` table. We could provide an implementation of the read model's repository interface that gets its data from the very same table (see Listing 3.6).

Listing 3.6: Creating an `Ebook` read model object from the entity's data source.

```

final class SqlEbookRepository implements EbookRepository
{
    private Connection $connection;

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

    public function getById(EbookId $ebookId): Ebook
    {
        $record = $this->connection->execute(
            'SELECT price FROM ebooks WHERE id = ?',
            [
                $ebookId->asString()
            ]
        )->fetchAssoc();

        if ($record === false) {
            throw CouldNotFindEbook::withId($ebookId);
        }

        return new Ebook(
            $ebookId,
            (int)$record['price']
        );
    }
}

```

```
    }
}
```

Although a convenient solution, when the write and read model share the same data source, this can lead to new problems. It could happen that the read model repository interprets the data in a different way than the write model does. Even in the example above, the assumption is that the `price` column contains the price of the e-book in cents. What if at some point the write model switches to a native decimal representation. The read model would start to provide bad prices, because 1.50 euro in the database when cast to an integer will become 1 cent in the application. One way to reduce that risk is to write integration tests for your read model repositories. Another thing you could do to improve the situation is to have one class implement both the write and the read model repository interface. That way, the knowledge about the database table and the meaning of its columns is at least in a single place, making it less likely that you'll run into this kind of problem.

3.3.2 Using write model domain events

Another approach to keep the read model in sync with the write model would be to dispatch a *domain event* for every important change inside the entity. The read model is then able to update its own state based on the information contained in those events. These are the ingredients you'll need for this approach:

1. An entity.
2. A domain event for every state change that is relevant to the read model.
3. A service that subscribes to these domain events and updates the read model according to the changes indicated by the events.

Listing 3.7 shows how the entity can record domain events internally when its state changes in a way that might be relevant to others.

Listing 3.7: An entity records domain events internally.

```
final class Ebook
{
    /**
     * @var array<object>
     */
    private array $events;

    private int $price;

    // ...

    public function changePrice(int $newPrice): void
    {
        $this->price = $newPrice;

        // When state changes, we additionally "record" a domain event
    }
}
```

```

        $this->events[] = new PriceChanged($this->ebookId, $newPrice);
    }

    public function recordedEvents(): array
    {
        // Clients can find out what happened by calling this method
        return $this->events;
    }
}

/*
 * A 'PriceChanged' domain event is an object that holds the ID of the
 * e-book whose price changed, as well as the new price.
 */
final class PriceChanged
{
    private EbookId $ebookId;

    private int $newPrice;

    public function __construct(EbookId $ebookId, int $newPrice)
    {
        $this->ebookId = $ebookId;
        $this->newPrice = $newPrice;
    }

    public function ebookId(): EbookId
    {
        return $this->ebookId;
    }

    public function newPrice(): int
    {
        return $this->newPrice;
    }
}

```

In order for the events to be useful outside of the entity, they have to be extracted using the entity's `recordedEvents()` method. We then need to notify any interested service of these events, so they can take further action. We'll look at the details of event dispatching and event subscribing in Section [11.5](#), so here we only look at the big picture. Listing [3.8](#) shows how a service makes a change to an `Ebook` entity, saves it using its repository, and then dispatches the internally recorded events using an event dispatcher service. Eventually this will trigger a call to the `UpdateEbookReadModel` event subscriber, which fetches the corresponding read model from the repository and updates its price field using the data from the `PriceChanged` domain event.

Listing 3.8: Using domain events to update a read model.

```

/*
 * Whenever a service changes the price of an e-book, it will
 * internally record a 'PriceChanged' event. We broadcast this
 * event by sending it (and other recorded events) to the
 * event dispatcher.
*/

```

```

$ebook->changePrice(150);
$this->ebookRepository->save($ebook);
$this->eventDispatcher->dispatchAll($ebook->recordedEvents());

/*
 * If we register 'UpdateEbookReadModel' as an event subscriber
 * for 'PriceChanged' events, the event dispatcher will
 * call it whenever such an event occurs.
 *
 * The listener then updates the read model using the data from
 * the event object.
 */

final class UpdateEbookReadModel
{
    // ...

    public function whenPriceChanged(PriceChanged $event): void
    {
        $readModel = $this->readModelRepository->getById(
            $event->ebookId()
        );
        $readModel->setPrice($event->newPrice());
        $this->readModelRepository->save($readModel);
    }
}

```

Figure 3.1 shows how all the moving parts are working together in this scenario.

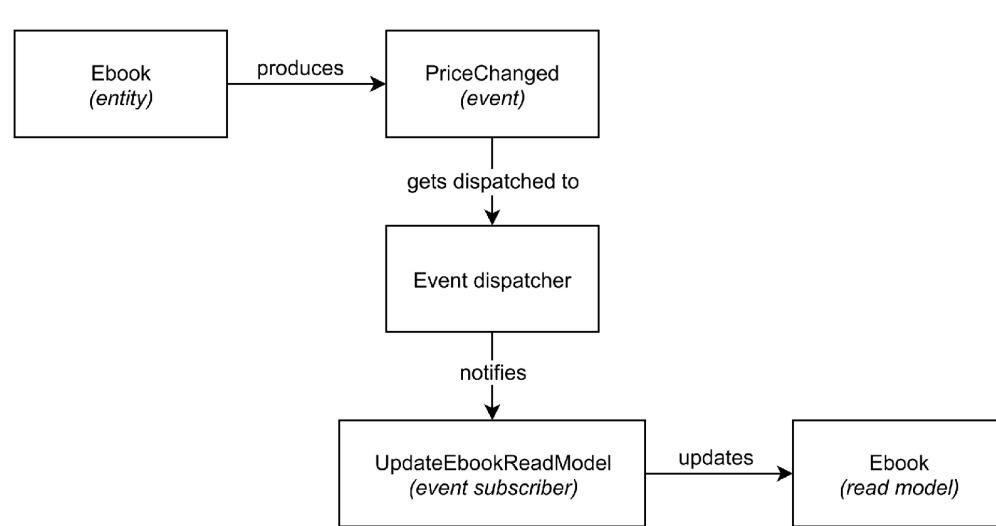


Figure 3.1: Using write model events to update a read model.

Note that setting the price on the read model isn't the same thing as changing the price on the entity. The change on the entity is the real change. When we update the read model, we're merely *reflecting* that original change onto our read model object.

The remaining question is: what happens inside the `save()` method of the read model repository? This is an infrastructural concern: do we save it to the same database where we save our entities to? Do we use a different database? Maybe we want to store the e-book read model as a document in our Elasticsearch database, so we can make it easily searchable.

None of this really matters for the core of the application, since we already have a read model repository interface, which indicates: “I have a particular need, but I don’t care how you’ll fulfill this need. I also don’t mind if you need to talk to something outside the application for it.”

Whether you implement your read model repository to use domain events as the source of the data, or the write model’s database, or something else entirely, what matters is: any client can use the repository interface to get the information it needs. In our controller, we only needed the price of an e-book, which we can now retrieve from the e-book read model repository (Listing 3.9).

Listing 3.9: Using the read model inside the controller.

```
public function orderEbookAction(Request $request): Response
{
    $ebook = $this->ebookRepository->getById(
        EbookId::fromString($request->request->get('ebook_id'))
    );
    $ebookPrice = $ebook->price();
    // ...
    return new Response(/* ... */);
}
```

“I want to know more about this event-based approach!”

Read more about this approach in Greg Young’s “CQRS Documents”²⁸.

Mathias Verraes, “Patterns for Decoupling in Distributed Systems: Summary Event”²⁹.

Using events for “synchronization” between write and read models can be a complicated business. For advice on the topic, I’d recommend Vaughn Vernon’s “Implementing Domain-Driven Design” (Addison-Wesley Professional, 2013; look for “Integrating bounded contexts”) and Udi Dahan’s blog³⁰.

3.4 Using value objects with internal read models

A read model is often designed to support a specific use case for one of its clients. That's why it's important to look at how the data from the read model is used by each specific client. In our example, the `Ebook` read model is used to get the e-book's price from. When creating the order we calculate the order amount based on the quantity ordered and the price per unit of the e-book:

```
$ebookPrice = $ebook->price();  
$orderAmountInCents = (int)$request->get('quantity') * $ebookPrice;
```

The assumption that the client is making here, is that both quantity and price are integers. Indeed, it makes sense to assume that the ordered quantity will always be a whole number because you can't order half a book. But what does it mean for the price to be an integer as well? The client correctly assumes that the price is provided in cents. We have already briefly considered what happens if the underlying data type changes from cents (integer, e.g. 150 cents equals 1.50 euro) to euros (using a string with decimal notation, e.g. '1.50'). What happens if `$ebook->price()` returns a string instead of an integer?

Surely, changing the underlying data type of a read model's property will affect the clients of that read model. And this is where we should use an old trick again. The one we also used in Section [2.7](#) of the previous chapter, where we wanted to hide the underlying type of an entity identifier: we introduce a *value object*. By using a value object we can ensure that clients won't rely on raw data, nor on the particular primitive type of that data.

A first refactoring step would be to introduce a simple wrapper object for the e-book price; let's call it `Price`. Since the underlying type of the price is currently an integer, we need to add a way to get the integer into the object. We also need a way to get the integer out again, because that's what the client is currently able to work with. Listing [3.10](#) shows the basic setup we need. I wouldn't say this improves the design a lot, but it's a great starting point.

Listing 3.10: Introducing a very basic `Price` value object.

```
final class Price  
{  
    private int $priceInCents;  
  
    private function __construct(int $priceInCents)  
    {  
        $this->priceInCents = $priceInCents;  
    }  
  
    public static function fromInt(int $priceInCents): self  
    {  
        return new self($priceInCents);  
    }  
  
    public function asInt(): int  
    {  
        return $this->priceInCents;  
    }  
}
```

```

}

final class Ebook
{
    // ...

    private Price $price;

    public function __construct(
        /* ... */
        Price $price
    ) {
        // ...
        $this->price = $price;
    }

    // ...

    public function price(): Price
    {
        return $this->price;
    }
}

// Inside the controller:

/*
 * 'price()' returns a 'Price' object now,
 * so we have to convert it to an integer before multiplying it:
*/
$orderAmountInCents = (int)$request->get('quantity')
    * $ebook->price()->asInt();

```

Getting data in and out of a value object is the least interesting feature of value objects. If we don't even validate the raw value, we might as well not use a value object in the first place. So the next step should be to make the `Price` object really useful for its clients. In this case, it would be helpful if the client didn't need to take the integer out of the value object, but could multiply the price with a given quantity directly. Listing 3.11 shows how to do that by adding a `multiply()` method to the `Price` class.

Listing 3.11: Adding multiplication behavior to `Price`.

```

final class Price
{
    // ...

    public function multipliedBy(int $quantity): int
    {
        return $this->priceInCents * $quantity;
    }
}

$orderAmountInCents = $ebook->price()->multipliedBy(
    (int)$request->get('quantity')
)

```

```
);
```

The order amount is still an integer and would definitely benefit from a value object called `Amount`, which represents a quantity multiplied by a price. Maybe you could even use a generic `Money` value object. But for now, we'll leave it at this. We've seen how you can improve a read model by adjusting it to the needs of the client that uses it. An added benefit is that using value objects allows you to decouple from the infrastructure. Clients can keep using value objects even if the underlying data type changes.

3.5 A specific type of read model: the view model

In the previous section we introduced a dedicated read model and read model repository because the `OrderController` needed the price of an e-book. You could classify the resulting model as an *internal* read model, since the data it provides is only used internally, by the application itself. The information is never directly shared with or displayed to the user.

On the other hand, there are places in our application where we fetch data from the database in order to show it to the user. One such place is the page where the user can browse through the list of e-books that are available for purchase. Listing 3.12 shows how that's currently done. The controller action uses the database connection directly to find e-books that are not "hidden", and while it's in the database it also collects some sales statistics.

Listing 3.12: Fetching a list of available e-books from the database.

```
final class EbookController
{
    // ...

    public function listEbooksAction(): Response
    {
        $connection = $this->container->get(Connection::class);

        $query = <<<EOD
SELECT
    e.*,
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.ebook_id = e.ebook_id
    ) AS number_of_times_sold
    FROM ebooks e
    WHERE e.is_hidden = 0
    ORDER BY number_of_times_sold DESC
EOD;
        $records = $connection->executeQuery($query)->fetchAllAssoc();

        return new Response(
            $this->render(
                'list.html.twig', [
                    'ebooks' => $records
                ]
            )
        );
    }
}
```

```
        );
    }
}
```

By now we quickly recognize the issue here: this code is coupled to the infrastructure. Being able to produce a list of available e-books is a crucial use case for e-book shops. Yet, this use case isn't recognizable in our code as something separable from the application's infrastructure.

I usually do a little thought experiment to find out if we even need to decouple a particular functionality from its underlying infrastructure. It goes as follows:

What if we would still be running the same business; we still want to sell e-books to our customers. Except, from now on people will have to use the command-line to order their e-books (I know, it's silly, but hang on). The question is: should our application still provide the functionality we're considering to decouple? If not, the functionality was in fact justifiably tied to the application's infrastructure. Switching from a web frontend to a CLI frontend makes the need for such a functionality disappear completely. If, however, people would still need to be able to use that functionality, even from the command line, it means we have to *decouple* it.

In our situation the more specific question is: should the user be able to look at a list of available e-books before buying one? Of course they do. How would they figure out which e-book to buy if they don't even know which e-books we sell? In other words, listing available e-books deserves to be more than just a controller with a database query. It needs to be represented in core code.

We need to do several things:

1. Model our query as a method on an interface.
2. Model the result of that query as an object that gives us the exact answer we need.
3. Provide an implementation of the interface that can be used in production.

Modeling the query might be the easiest thing to do. We only have to find a good description of our question. We could have an `Ebooks` interface with a `listAvailableEbooks()` method (see Listing 3.13). That method would return an array of `Ebook` read model objects.

Listing 3.13: An interface for retrieving a list of available e-books.

```
interface Ebooks
{
    /**
     * @return array<Ebook>
     */
    public function listAvailableEbooks(): array;
}
```

Again, this `Ebook` class is not the same as the entity class. Objects of this type will serve as read models for displaying data on an HTML page. Such read models can be called “view models”,

since they are used to display data to users or external systems. This is different from the purpose that the `Ebook` read model from Section 3.2 served. That one was only used internally, to calculate the correct order amount. The data from the `Ebook` view model that we're going to create will travel across our application's boundaries, to the outside world, to actual users of our application.

What does this particular `Ebook` view model look like? What data does it contain, what data types should be used?

To find an answer to these questions, we should look at the place where the view model is going to be used. In this case that's the `list.html.twig` file (Listing 3.14), which is the Twig³¹ template that produces the HTML response for `listEbooksAction()`.

Listing 3.14: The Twig template that renders the list of available e-books.

```
{% extends base.html.twig %}

{% block body %}
<h1>Available e-books</h1>



| Title             | Number of readers             | Price             | Actions                                                                         |
|-------------------|-------------------------------|-------------------|---------------------------------------------------------------------------------|
| {{ ebook.title }} | {{ ebook.numberoftimesSold }} | {{ ebook.price }} | <a href="{{ path('order_ebook', { ebookId: ebook.ebookId }) }}"> Order now </a> |


```

Based on the intended usage of the `Ebook` view model inside the template, we can derive the required public getter methods and their types. Listing 3.15 shows the outline of the class.

Listing 3.15: An outline of the `Ebook` view model class

```
final class Ebook
{
    public function ebookId(): string
    {
        // ...
    }

    public function title(): string
    {
        // ...
    }

    public function numberoftimesSold(): int
    {
        // ...
    }

    public function price(): string
    {
        // ...
    }
}
```

Most of the data exposed by the view model should be of type `string` because rendering the template itself is basically an exercise in string concatenation. In some cases it makes sense to use other primitive return types like `int` for the `numberoftimesSold()` method.

Note that `price()` doesn't return a `Price` value object, like the `price()` method we saw earlier in this chapter. It returns a `string`, which is supposed to be a properly formatted amount of money, including the currency sign, and with the correct decimal precision. If we'd do all this formatting work in the template itself, the view model wouldn't be as portable as it can be. Consider again the example of running a command-line-based e-book shop; we'd still want to show the e-book's price, and we wouldn't want to rewrite the logic for price formatting in a place where we can't use HTML templates. We should make it as easy as possible for any client to show the data to actual users. Templates in particular shouldn't need to know anything about the domain objects that our application uses internally. This means our view model should only return primitive-type values.

We know what our view model object should look like, and we have defined an interface method `Ebooks::listAvailableEbooks()` which will return an array of these view model objects. Now we only need an implementation of that `Ebooks` interface which can query the database and use actual data to return the list of available e-books. Just like with the read model in Section 3.3 there are different options, but let's go with querying the database that's used by the write model. We can reuse the existing SQL query.

Listing 3.16: An implementation for the `Ebooks` interface.

```
final class EbooksUsingSql implements Ebooks
```

```

{
    private Connection $connection;

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

    public function listAvailableEbooks(): array
    {
        $query = <<<EOD
SELECT
    e.*,
    (
        SELECT COUNT(*)
        FROM orders o
        WHERE o.ebook_id = ebooks.ebook_id
    ) AS number_of_times_sold
FROM ebooks e
WHERE e.is_hidden = 0
ORDER BY number_of_times_sold DESC
EOD;

        $records = $this->connection
            ->execute($query)
            ->fetchAllAssoc();

        // Instantiate an Ebook view model for every record

        return array_map(
            fn (array $record) => new Ebook(
                // ...
            ),
            $records
        );
    }
}

```

Now we need to connect the dots and make sure the data from the database record ends up in the correct properties of the `Ebook` instance. Also, we have to make sure each getter returns the right information, with the correct type. `ebookId()`, `title()` and `numberOfTimesSold()` are straight-forward, but `price()` needs some additional work. The value that comes from the database is an `int`, but the return value of `price()` should be a formatted price. Inside the `price()` method we can easily make that conversion though, and we can even use the `Price` value object as an intermediate data type if we like (see Listing 3.17). It will never escape the view model object, so it remains an implementation detail.

Listing 3.17: Converting between types inside the view model.

```

final class Ebook
{
    // ...

    private int $price;

```

```

public function __construct(
    // ...
    int $price
) {
    // ...
    $this->price = $price;
}

// ...

public function price(): string
{
    return Price::fromInt($this->price)
        ->asFormattedAmount();
}
}

```

There are many alternatives here. Maybe you don't like to have a method for string formatting directly on your value object. In that case create a separate number formatter object or utility class. Maybe you want to change the `$price` constructor argument to be a `Price` value object already. In that case, let the repository implementation do the conversion from `int` to `Price`.

“I’m afraid we’ll end up with too many classes...”

Good point. You will definitely have more classes and interfaces when you separate core from infrastructure code. We can even see some kind of pattern emerge here. When we want to rewrite a query (like `listAvailableEbooks()` or `getById()`), we start with a single “mixed” class, but end up with a new interface and two new classes (Figure 3.2). The interface represents the query, one of the classes represents the answer, and the other class is an implementation of the query interface.

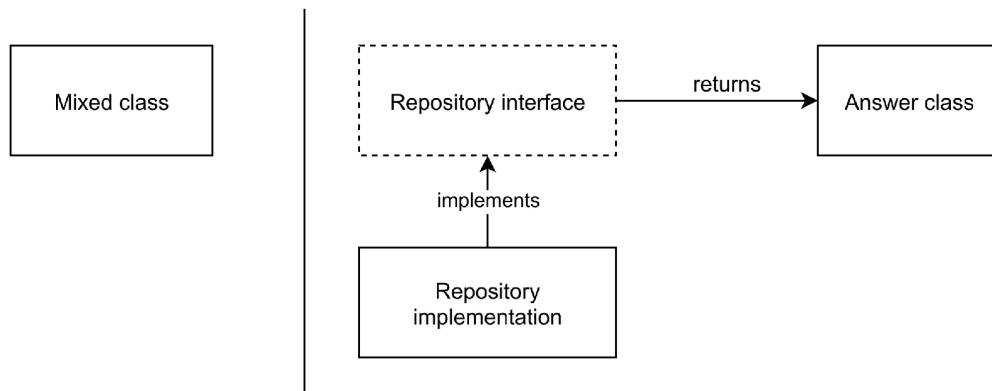


Figure 3.2: After decoupling a query from underlying infrastructure, you will have three elements.

If you want to decouple from infrastructure, this is the way to go. But there are several ways too keep the number of elements in your system manageable.

1. Only introduce an interface for objects that actually communicate with something outside your application. This might save you a couple of interfaces.
2. Combine multiple interface methods in a single interface. This might also save you a couple of interfaces.
3. Let a single class implement multiple interfaces. This certainly saves you some classes.
4. Reuse the “answer” class for different queries. This also saves you some classes.

However, always keep an eye on tension in the design. The downsides of reducing the number of elements in your system are, respectively:

1. With fewer interfaces, it can be harder to replace an actual service implementation with a test double. In my experience this is rarely a problem though. When this happens, it should be easy to re-introduce the interface after all.
2. An interface with multiple methods may give clients access to many unrelated methods, which might confuse their own purpose, make them dependent on too many things, and make it harder to change the interface.
3. A class that implements too many interfaces will become hard to maintain when each method may have its own set of dependencies and imports knowledge inside that class that gets entangled with other similar knowledge. One way to fix this is to have a class implement only interfaces with methods that are truly related.
4. If a class is used in one place, it will be easy to change it, because there is only one client that may need to be updated. If it’s used in many places it will be harder to change it, since there are many clients that need to be updated and may potentially break.

A great example of a single class that could implement multiple related interfaces is the SQL repository class which could implement the write model’s `EbookRepository`, the read model’s `EbookRepository` interface, and the view model’s `Ebooks` interface. Each method has a common set of dependencies anyway (the database connection object), and performs similar work. However, in practice I usually separate at least the write model repository from the read model repository implementations.

3.6 Using view models for APIs

In the previous section we’ve been discussing the use of view models in HTML templates. In that case there will be some code looping over and echoing the return values of some of the view model’s getters. This assumes that your application’s end user is an actual person using a web browser to visit your web application’s pages. Other types of users need the data to be in different forms. A command-line user needs plain text information, some remote system might want to speak SOAP with your application.

Let’s find out how that works in our application. Say we want to expose the list of available ebooks as a JSON-encoded array of e-book objects. Inside the controller we could loop over the array of `Ebook` view model objects returned by `listAvailableEbooks()` and turn it into a JSON-serializable data structure, like an array of associative arrays (see Listing 3.18).

Listing 3.18: Returning a list of available e-books as JSON.

```
final class EbookApiController
{
    private Ebooks $ebooks;

    public function __construct(Ebooks $ebooks)
    {
        $this->ebooks = $ebooks;
    }

    public function listAvailableEbooksAction(): Response
    {
        $data = [];

        foreach ($this->ebooks->listAvailableEbooks() as $ebook) {
            $data[] = [
                'ebookId' => $ebook->ebookId(),
                'title' => $ebook->title(),
                // ...
            ];
        }

        return new Response(json_encode($data));
    }
}
```

Read models are supposed to be as user-friendly for its clients as possible. But this controller still needs to do a relatively large amount of work before it can show the data to the user. We can make it a lot easier for this client if the view model objects could be serialized to JSON in one step. There are many ways to accomplish this. We could let the view model convert itself to an associative array (see Listing 3.19).

Listing 3.19: A view model with an `asArray()` method.

```
final class Ebook
{
    // ...

    /**
     * @return array<string,mixed>
     */
    public function asArray(): array
    {
        return [
            'ebookId' => $this->ebookId(),
            'title' => $this->title(),
            // ...
        ];
    }
}

// Inside the controller:
```

```

        return new Response(
            json_encode(
                array_map(
                    fn (Ebook $ebook) => $ebook->asArray(),
                    $this->ebooks->listAvailableEbooks()
                )
            )
        );
    )
);

```

You could also make the view model object immediately serializable by making its properties `public` (see Listing 3.20). The only thing missing here is a way to enforce immutability on this object after we make its properties `public`. PHP itself has no built-in options to do this at runtime, but it can be accomplished at “compile”-time using a static analyzer like Psalm.

Listing 3.20: A view model with public properties.

```

final class Ebook
{
    public string $ebookId;
    public string $title;
    // ...
}

// Inside the controller:

return new Response(
    json_encode(
        $this->ebooks->listAvailableEbooks()
    )
);

```

3.7 Summary

In the beginning of this chapter we recognized the need to get information about a related entity. Instead of reusing the entity itself we decided not to combine write and read responsibilities in the same object. We introduced an immutable `Ebook` read model object, specialized in providing information. Such a read model object comes with its own read model repository interface. This interface needs an implementation, which fetches the data and instantiates the read model objects using that data. We discussed several alternatives for repository implementations: you can use the data source of the write model, or you can update the read model based on events from the write model.

In another situation we needed to show some data to the user. Again, we didn’t reuse the write model for this purpose, but created a dedicated view model. The view model consists of a view model object, a repository interface and a repository implementation. The view model object contains all the required data. In a regular web application a view model has getters that make it

easy to get the data out and render it inside an HTML template. When a view model is going to be returned as an API response, a requirement is that it's serializable in one step.

Exercises

1. Is it smart to reuse an entity for the purpose of querying data?[32](#)
 2. A read model consists of three class/interface elements. What are they?[33](#)
 3. What are two common ways of keeping a read and a write model synchronized?[34](#)
 4. Some models return value objects from their methods. Which ones?[35](#)
 1. Entities
 2. Read models
 3. View models
 5. What is a true requirement for read models?[36](#)
 1. They should be synchronized with the write model based on domain events.
 2. They should support the use of case of its clients instead of serving some generic purpose.
 3. They should share the same read model repository interface.
-

4 Application services

This chapter covers:

- Extracting an application service from a web controller
 - Making a use case independent from its surrounding infrastructure
 - Introducing a parameter object to present the input data for the application service
 - Separating use case scenario steps into multiple services
-

After all the work we've done in the first two chapters, the `OrderController`'s `orderEbookAction()` is already in a great shape (see Listing 4.1).

Listing 4.1: The current state of `orderEbookAction()`.

```
public function orderEbookAction(Request $request): Response
{
    $ebook = $this->container->get('ebook_repository')
        ->getById((int)$request->get('ebook_id'));

    $orderAmount = $ebook->price()->multipliedBy(
        (int)$request->get('quantity'))
    );

    $orderRepository = $this->container->get('order_repository')
;

    $orderId = $orderRepository->nextIdentity();
    $order = new Order(
        $orderId,
        $ebook->id(),
        $request->get('email_address'),
        (int)$request->get('quantity'),
        $ebook->price(),
        $orderAmount
    );

    $orderRepository->save($order);
```

```

    $this->container->get('session')->set(
        'currentOrderId',
        $orderId->asString()
    );
    // ...
    return new Response(/* ... */);
}

```

We've pushed domain logic into an `Order` entity and introduced an abstraction for saving orders, which we called `OrderRepository`. We've also created an `Ebook` read model, which answers the question: what's the price of this e-book? The e-book price is represented as a `Price` value object. It has a convenient `multipliedBy()` method to calculate the total order amount. We've spent a lot of effort on making the domain concepts and behaviors more clear, and this has paid off. The story of ordering an e-book is getting easier to follow with every refactoring step. There's still one problem: the controller action requires the context of a web application to run in. So everything inside `orderEbookAction()` should still be considered infrastructure code.

4.1 Considering other infrastructures

The code for the use case of “ordering an e-book” reveals that it’s supposed to be called over HTTP. There is a web page with a form that makes this use case available to its users. There is no other way to order an e-book. Let’s repeat the thought experiment that we did in Chapter 3. What if our business would make a major infrastructure switch: from the web to the command line. Would our application still need to implement this use case? If so, and I’m sure this is the case, it should be decoupled from the infrastructure, so it would survive such a change.

Now let’s find out what would be needed to make the “order an e-book” use case work in a command-line application. Listing 4.2 shows the basic structure of the command class that could be used to order e-books from the command-line instead of a web browser. As you can see, the code sample uses the Symfony Console component³⁷, but other tools have similar base

classes or interfaces, so I'm sure you can translate the example to a framework you're familiar with.

Listing 4.2: The basic structure of a Symfony console command.

```
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

final class CreateOrderCommand extends Command
{
    protected function configure(): void
    {
        $this
            ->setName('create-order')
            ->addArgument('ebook_id', InputArgument::REQUIRED)
            ->addArgument('quantity', InputArgument::REQUIRED)
            ->addArgument('email_address', InputArgument::REQUIRED);
    }

    protected function execute(
        InputInterface $input,
        OutputInterface $output
    ): int {
        // How to create an order?

        return 0; // success
    }
}
```

Once Symfony knows about this command, you can run it like this:

`bin/console create-order "21" "2" "info@matthiasnoback.nl".` But how do we do the actual work, creating an order based on the provided command-line arguments? Of course, we could simply copy all the code from the controller, and I know applications where this has happened. What's the result of copying use case code?

- If we actually copy the code there are now two places that contain the code for this use case. As you probably know, this gets very confusing

and quickly becomes a maintenance nightmare. A developer changes something in one location, and forgets to update the other location too. Or they change it in a special undocumented way and nobody understands what's going on for what reason anymore.

- If we don't actually copy the code but move it to the new location and we remove the old code, we're still looking at the same problem that we started with: the core use case of ordering an e-book is now tied to other infrastructure and can only run in the context of a command-line application.

So we should neither move the code to the command class, nor copy it. Could we somehow run the same controller code from the command-line instead? Since a command line application can't deal with HTTP requests, responses, or sessions, we'd have to forge a `Request` object and define a stand-in session service in the container. Finally you would be able to call the controller, as shown in Listing 4.3.

Listing 4.3: The command calls the controller action.

```
final class CreateOrderCommand extends Command
{
    // ...

    protected function execute(
        InputInterface $input,
        OutputInterface $output
    ): int {
        $request = new Request([], [
            'ebook_id' => $input->getArgument('ebook_id'),
            'quantity' => $input->getArgument('quantity'),
            'email_address' => $input->getArgument('email_address')
        ]);
        $response = $this->container->get('order_controller')
            ->orderEbookAction($request);

        // Somehow extract the order ID from the Response...

        return 0;
}
```

}

Of course, this is an ugly workaround; I think most people would consider it a hack.

Let's take a step back. We are introducing hacks to make a web application work from the command line. Maybe this is getting out of hand; maybe the whole exercise is too far-fetched? It doesn't even make sense to order e-books from the command-line.

Well, I know people who are building a web application but don't want to wait until the user interface and user interactions have been designed before they can use it. They just want to exercise the application's use cases without the web frontend that the application will eventually have. They are familiar with the command-line so why not build a few commands that lets them do some exploratory testing?

But there are more compelling cases where it's important that a single use case can be used by different types of clients. Here are a few not so far-fetched examples:

1. The application has to offer an API endpoint for an external system that accepts e-book orders through their own frontend and sends them to us as a JSON string.
2. The application should have an import function, where an administrator could upload a CSV file with manually entered orders from customers who visited our booth at a conference.
3. Some use cases need to be invoked on a regular basis, and now we have to set up a cron job that can do this (well, maybe not ordering an e-book, but synchronizing our local e-books catalog with some remote service).

In each example, the client is of a different type. In the first case we'll need another controller that parses the JSON we get from the external system and somehow converts it into an actual order. In the second case we have to parse a CSV file and loop over its lines, creating a new order for every line. The third example is more like the command-line example we discussed

before: a cron job also needs a command-line entry point to particular use cases of our application.

How can you make a use case usable by all these different clients?

4.2 Designing a use case to be reusable

To make a use case reusable we should find a common ground that allows different types of clients to invoke the same use case. If you start with an existing controller like the one we are working on right now, you should look for the things that tie the code to specific infrastructure and then decouple it from these elements. For instance, the code currently relies on a `Request` object, which makes it only useful in a web context, and the same goes for passing data to the next request by storing something in a `Session` object. An API controller should not keep state between requests, and a CLI client wouldn't even know about sessions. So to make the use case code reusable we should remove its dependencies on `Request` and `Session`.

We should also analyze the input and output of the use case. In our example, the input is the information needed to create the order: the e-book ID, the order quantity, and the buyer's email address. What goes out is the ID of the order that was created. Currently the input data is taken from the `Request` object. In order to decouple from it and make this code reusable, we have to redefine the input without mentioning any infrastructure-specific concept like "request". The output is just an ID, which isn't tied to a particular infrastructure, so we won't have additional work there.

Providing the input data in a form that's decoupled from the mechanism that delivered the data (that is, HTTP), isn't too hard. We should *only use primitive-type data*, and that would do the trick. In our example we'll use a string for the email address of the buyer and integers for the e-book ID and the order quantity. Listing 4.4 shows what happens if we push usages of the `Request` object towards the top of the method and stop using it in the middle. The refactoring technique we use here is called *Extract Variable*³⁸. Your IDE may help you with it.

Listing 4.4: Push usages of the `Request` object to the top of the controller action.

```

public function orderEbookAction(Request $request): Response
{
    // We extract
    $ebookId = (int)$request->get('ebook_id');
    $orderQuantity = (int)$request->get('quantity');
    $emailAddress = $request->get('email_address');

    // Below this point, we don't need the 'Request' anymore:

    $ebook = $this->container->get('ebook_repository')
        ->getById($ebookId);

    $orderAmount = $ebook->price()
        ->multipliedBy($orderQuantity);

    // ...

    $orderRepository->save($order);

    // We make sure that we only use the 'Session' below this point:
    $this->container->get('session')->set(
        'currentOrderId',
        $orderId->asString()
    );
    // ...

    return new Response(/* ... */);
}

```

Except for a few usages of the service locator (`$this->container->get()`), a large part of the controller action is starting to look like core code; independent of the infrastructure that connects it to its users, and independent of underlying storage, etc. It isn't officially core code yet, because the method itself is still coupled to its web context.

4.3 Extracting an application service

The next step will be to move all the code that is not web-specific and move it to its own class. This is another standard refactoring step called *Extract Class*³⁹. To accomplish this we need to:

- Create a new class. Let's call it `EbookOrderService`.
- Add a public method to it. Let's call it `create()`.
- Move the infrastructure-independent code from the controller to the new method.
- Inject the necessary dependencies (`EbookRepository` and `OrderRepository`) as constructor arguments.
- Use the new service inside the controller.
- Provide the data we extracted into variables as method arguments of the service's `create()` method.

The resulting service class and its usage in the controller are shown in Listing 4.5. Such a service is often called an *Application service*. It's used to model a use case that has side-effects (like saving a new entity) in a reusable way. We'll discuss the pattern in more detail in Section 11.4.

Listing 4.5: The extracted `EbookOrderService`.

```
final class EbookOrderService
{
    private EbookRepository $ebookRepository;
    private OrderRepository $orderRepository;

    public function __construct(
        EbookRepository $ebookRepository,
        OrderRepository $orderRepository
    ) {
        $this->ebookRepository = $ebookRepository;
        $this->orderRepository = $orderRepository;
    }

    public function create(
        int $ebookId,
        int $orderQuantity,
        string $emailAddress
    ): OrderId {
        $ebook = $this->ebookRepository->getById($ebookId);

        $orderAmount = $ebook->price()
            ->multipliedBy($orderQuantity);

        $orderId = $this->orderRepository->nextIdentity();

        $order = new Order(
```

```

        $orderId,
        $ebook->id(),
        $emailAddress,
        $orderQuantity,
        $ebook->price(),
        $orderAmount
    );
    $this->orderRepository->save($order);
    return $orderId;
}
}

```

Listing 4.6 shows how the new service can be used inside the controller.

Listing 4.6: orderEbookAction() uses the EbookOrderService.

```

public function orderEbookAction(Request $request): Response
{
    $ebookId = (int)$request->get('ebook_id');
    $orderQuantity = (int)$request->get('quantity');
    $emailAddress = $request->get('email_address');

    $orderId = $this->container->get('ebook_order_service')
        ->create(
            $ebookId,
            $orderQuantity,
            $emailAddress
        );
    // ...
}

```

Since we don't really need the input data to be in a variable anymore, we should do a quick *Inline Variable*⁴⁰ refactoring:

```

$orderId = $this->container->get('ebook_order_service')
->create(
    (int)$request->get('ebook_id'),
    (int)$request->get('quantity'),
    $request->get('email_address')
)

```

```
);
```

We've reached the point where the `EbookOrderService` can be used by any client that is able to provide the required strings and integers. You could now build an API endpoint that accepts a JSON request body, decodes it, and calls `EbookOrderService` using its data. Or you could just as easily create a command-line tool that accepts a number of arguments and passes it to the `EbookOrderService`, as shown in Listing 4.7 (Figure 4.1). As long as the client can provide primitive data it can invoke the use case.

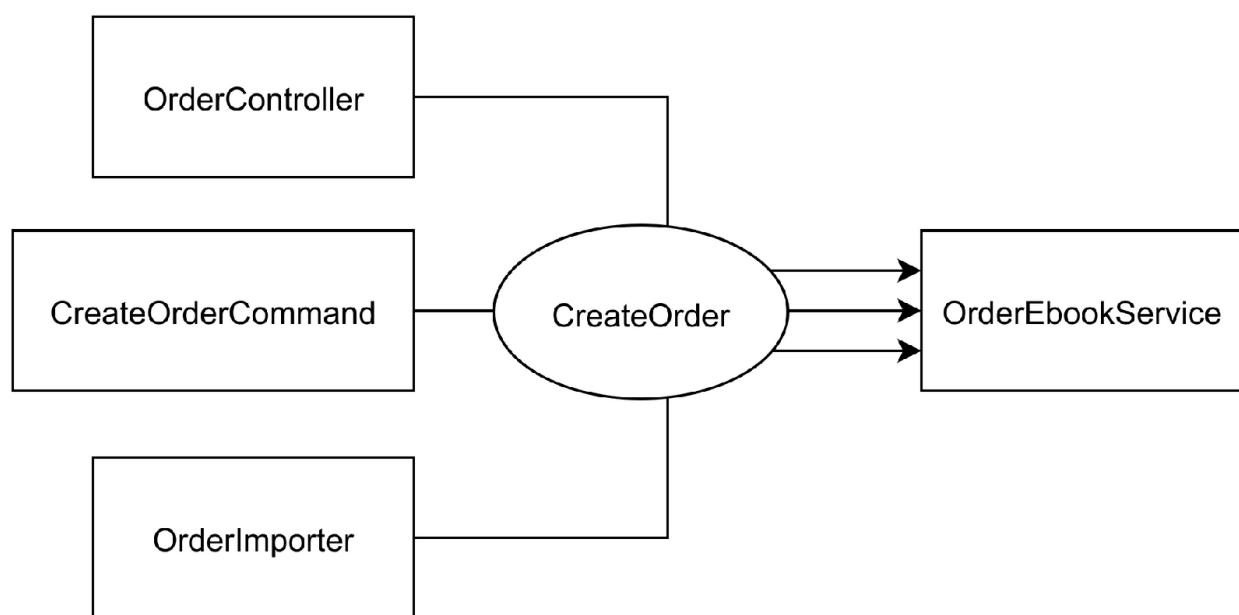


Figure 4.1: Different types of clients can invoke the same use case.

Listing 4.7: It's easy to reuse the same `EbookOrderService` in a command-line application.

```
final class CreateOrderCommand extends Command
{
    // ...

    protected function execute(
        InputInterface $input,
        OutputInterface $output
    ): int {
```

```

    $orderId = $this->container->get('ebook_order_service')
    ->create(
        (int)$input->getArgument('ebook_id'),
        (int)$input->getArgument('quantity'),
        $input->getArgument('email_address')
    );

    $output->writeln(
        sprintf(
            '<success>Created a new order with ID %s</succes
s>',
            $orderId->asString()
        )
    );
}

return 0;
}
}

```

4.4 Introducing a parameter object

We've decoupled a nice chunk of core code from the infrastructure code it was originally mixed with, but there's one thing left to do. The number of method parameters of a service like the one we extracted earlier, may get quickly out of hand. Besides an email address, you may also want to record the name of the buyer, or you may want to take orders in a different currency. To keep the method signature of an application service as simple as possible, there's another nice refactoring we can do, called *Introduce Parameter Object*⁴¹. The result will be an object that holds all the input data for this use case as a single value. It makes sense to name this object after its intention. You want to “create an order” with it, so let's call it `CreateOrder` (see Listing 4.8). Such an object is often called a *Command* object. This has nothing to do with command-line applications. It's called a command because the object represents a user intention: something the user wants the application to do.

Listing 4.8: The `CreateOrder` object holds all the data needed to create an order.

```
final class CreateOrder
```

```

{
    private int $ebookId;
    private int $orderQuantity;
    private string $emailAddress;

    public function __construct(
        int $ebookId,
        int $orderQuantity,
        string $emailAddress
    ) {
        $this->ebookId = $ebookId;
        $this->orderQuantity = $orderQuantity;
        $this->emailAddress = $emailAddress;
    }

    public function ebookId(): int
    {
        return $this->ebookId;
    }

    public function orderQuantity(): int
    {
        return $this->orderQuantity;
    }

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

```

The service should be modified to accept an instance of the new `CreateOrder` class (see Listing 4.9). Whenever the service needs input data, it has to fetch it from the `CreateOrder` object.

Listing 4.9: A `CreateOrder` object has to be passed to the `EbookOrderService` as a method argument.

```

final class EbookOrderService
{
    // ...

    public function create(CreateOrder $createOrder): OrderId
    {

```

```

        $ebook = $this->ebookRepository->getById(
            $createOrder->ebookId()
        );

        $orderAmount = $ebook->price()
            ->multipliedBy($createOrder->orderQuantity());

        // ...

        $order = new Order(
            $orderId,
            $ebook->id(),
            $createOrder->emailAddress(),
            $createOrder->orderQuantity(),
            $ebook->price(),
            $orderAmount
        );

        // ...
    }
}

```

Finally, any client that uses the `EbookOrderService` has to be updated too. From now on, they need to prepare a `CreateOrder` instance and pass it to the service. See Listing [4.10](#) for what this looks like in the controller action.

Listing 4.10: `orderEbookAction()` instantiates and populates a `CreateOrder` object and passes it to `EbookOrderService`.

```

public function orderEbookAction(Request $request): Response
{
    $createOrder = new CreateOrder(
        (int)$request->get('ebook_id'),
        (int)$request->get('quantity'),
        $request->get('email_address')
    );

    $orderId = $this->container->get('ebook_order_service')
        ->create($createOrder);

    // ...
}

```

4.5 Dealing with multiple steps

We've successfully moved a chunk of decoupled code from the controller to its own application service. But usually it's not so easy to do this. In legacy projects you'll find controller actions that consist of several hundreds (if not thousands) of lines. If you're lucky, these methods read like scripts: first do this, then do that, then do this other thing, and so on. If that's the case then you might be able to extract these steps and put them in their own classes so they become maintainable and maybe even testable. Sometimes it's not so easy to recognize distinct steps in these methods, and you'll have to rearrange and rewrite parts of the code to make the steps more clear.

In the case of the `orderEbookAction()` the steps are fairly recognizable. We've extracted the first step (saving the order), now let's take a look at the second step: sending an order confirmation email (see Listing [4.11](#)).

Listing 4.11: After creating a new order, we also send a confirmation email.

```
public function orderEbookAction(Request $request): Response
{
    // ...

    $orderId = $this->container->get('ebook_order_service')
        ->create($createOrder);

    $this->container->get('session')->set(
        'currentOrderId',
        $orderId->asString()
    );

    $message = (new Swift_Message('Your Order'))
        ->setFrom($this->container->getParameter('system_email_address'))
        ->setTo($request->request->get('email_address'))
        ->setBody(
            $this->container->get('twig')
                ->render('email/order_confirmation.html.twig')
        );

    $this->container->get('mailer')->send($message);

    return new Response(/* ... */);
```

```
}
```

Should we keep this code in the controller? No, because *whenever* somebody creates an order, they should receive a confirmation email too. The one and only way to create an order is by calling `EbookOrderService::create()`. But the code for sending the email is in `orderEbookAction()`. So if there will ever be a second place where `EbookOrderService::create()` gets called, the customer won't receive that email.

Knowing that the code can't stay inside the controller, should we move it to the `EbookOrderService` instead? That way, the service could create the order first, and then send the email. Listing 4.12 shows the result of moving the code to the application service.

Listing 4.12: Sending the email inside the application service.

```
final class EbookOrderService
{
    private EbookRepository $ebookRepository;
    private OrderRepository $orderRepository;
    private string $systemEmailAddress;
    private Environment $twig;
    private Swift_Mailer $mailer;

    public function __construct(
        EbookRepository $ebookRepository,
        OrderRepository $orderRepository,
        string $systemEmailAddress,
        Environment $twig,
        Swift_Mailer $mailer
    ) {
        $this->ebookRepository = $ebookRepository;
        $this->orderRepository = $orderRepository;
        $this->systemEmailAddress = $systemEmailAddress;
        $this->twig = $twig;
        $this->mailer = $mailer;
    }

    public function create(CreateOrder $createOrder): OrderId
    {
        // Step 1: Create the Order entity, save it
    }
}
```

```

    // ...

    // Step 2: Send the confirmation email

    $message = (new Swift_Message('Order ' . $orderId->asString()))
        ->setFrom($this->systemEmailAddress)
        ->setTo($createOrder->emailAddress())
        ->setBody(
            $this->twig->render('email/order_confirmation.html.twig')
        );
    $this->mailer->send($message);

    // ...
}

}

```

One advantage of moving the code to a service is that the dependencies are more clear now because the service uses constructor injection (more on this in Chapter 5). But at the same time seeing all these dependencies worries me. I wouldn't want my application service to depend on all this specific technology. The Twig Environment class and template, the Swift_Mailer and Swift_Message classes, the system's email address. These things don't belong inside the more domain-oriented code of the application service.

But we still want to send that email. What can we do to reduce the number of dependencies, and also reduce their specificity? We should use the power of abstraction once more. Instead of using Twig and the SwiftMailer directly inside the application service we should let the service talk to an interface, like the `SendOrderConfirmationEmail` interface in Listing 4.13.

Listing 4.13: An abstraction for sending order confirmation emails.

```

interface SendOrderConfirmationEmail
{
    public function send(OrderId $orderId, string $emailAddress)
: void;
}

```

The interface is like a guarantee that at runtime, it will be possible to send an actual email. And if any service wants to do that, it should depend on this interface, so it doesn't have to worry about the details of sending that email. Now let's modify the application service to use this new interface instead of the concrete dependencies it used before. Listing 4.14 shows how this simplifies the code in the application service a lot. An added benefit is that it's now easy to recognize that sending the confirmation email is the second step in the process of creating an order. You no longer have to read all the lines of code dealing with Twig and SwiftMailer to figure out what's happening.

Listing 4.14: The application service depends on an abstraction.

```
final class EbookOrderService
{
    // ...
    private SendOrderConfirmationEmail $sendConfirmationEmail;

    public function __construct(
        // ...
        SendOrderConfirmationEmail $sendConfirmationEmail
    ) {
        // ...
        $this->sendConfirmationEmail = $sendConfirmationEmail;
    }

    public function create(CreateOrder $createOrder): OrderId
    {
        // Create the Order entity, save it

        $this->sendConfirmationEmail->send(
            $orderId,
            $createOrder->emailAddress()
        );
        // ...
    }
}
```

Often when I introduce an interface to move lower-level implementation details out of sight, I forget to provide an actual implementation of that interface. Knowing that the `SendOrderConfirmationEmail` interface exists,

I trust it can be used. For me this is really stress-reducing. I can postpone all my worries about the correct usage of the mailer API, and I can move these messy details out of sight. I know everything will be fine the moment this code runs in production.

However, at some point you do have to send actual emails, so you can't postpone this work indefinitely. Let's get it over with and provide an implementation for the interface (see Listing 4.15).

Listing 4.15: An implementation for the `SendOrderConfirmationEmail` interface.

```
final class SendOrderConfirmationEmailWithSwiftMailer
    implements SendOrderConfirmationEmail
{
    private string $systemEmailAddress;
    private Environment $twig;
    private Swift_Mailer $mailer;

    public function __construct(
        string $systemEmailAddress,
        Environment $twig,
        Swift_Mailer $mailer
    ) {
        $this->systemEmailAddress = $systemEmailAddress;
        $this->twig = $twig;
        $this->mailer = $mailer;
    }

    public function send(OrderId $orderId, string $emailAddress)
: void
    {
        $message = (new Swift_Message('Order ' . $orderId->asString()))
            ->setFrom($this->systemEmailAddress)
            ->setTo($emailAddress)
            ->setBody(
                $this->twig->render('email/order_confirmation.html.twig')
            );
        $this->mailer->send($message);
    }
}
```

```
}
```

In practice most services that send emails will need more information than just an ID and the recipient email address. In that case you should introduce a read model that fetches the required information in one go. The read model should prepare the data in a format that will be useful when rendering the email body. Given that such a read model is going to be used for presentation purposes, it's actually a view model.

Now we have two steps inside the application service:

1. Creating the order and saving it
2. Sending a confirmation email

You could say that the first step actually consists of two more steps, but creating the order isn't useful without also saving it, you can't save an order without creating it first. So this really is a single step. It's not uncommon for a use case scenario to consist of multiple steps, but I also find that it works best if the application service only performs the first step, and leaves the other steps to other services. In our case, the application service will only create the order and save it. It then produces an event that indicates that the order was created. Other services can respond to that event, for instance by sending the order confirmation email. We've seen the setup already in Section [3.3.2](#), and we'll discuss the details of the pattern in Section [11.5](#). As a sneak preview, Listing [4.16](#) will give you an idea of how this would work. Figure [4.2](#) shows the relations between these objects in a single view. Note that the `EbookOrderService` is no longer in charge of sending the confirmation email. The event subscriber takes care of this, and the only thing that the `EbookOrderService` has to do is dispatch the recorded domain events to the `EventDispatcher` service.

Listing 4.16: Sending the email after receiving a `OrderWasCreated` event.

```
// This is a domain event:  
final class OrderWasCreated  
{
```

```

private OrderId $orderId;
private string $emailAddress;

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

public function orderId(): OrderId
{
    return $this->orderId;
}

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

final class Order
{
    /**
     * @var object[]
     */
    private array $events;

    public function __construct(
        OrderId $orderId,
        string $emailAddress
        // ...
    ) {
        // ...

        // When we create an order, a domain event will be "recorded"
        $this->events[] = new OrderWasCreated($orderId, $emailAddress);
    }

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

final class EbookOrderService

```

```

{
    // ...
    private EventDispatcher $eventDispatcher;

    public function __construct(
        // ...
        EventDispatcher $eventDispatcher
    ) {
        // ...
        $this->eventDispatcher = $eventDispatcher;
    }

    public function create(CreateOrder $command): OrderId
    {
        // ...

        $order = new Order(/* ... */);

        // First, save the Order

        // Then dispatch the events that were recorded inside the event:
        $this->eventDispatcher->dispatchAll($order->releaseEvents());

        return $orderId;
    }
}

/*
 * This is an event subscriber, which should be registered as a
 * subscriber for the 'OrderWasCreated' event:
 */

final class SendEmail
{
    private SendOrderConfirmationEmail $sendConfirmationEmail;

    public function __construct(
        SendOrderConfirmationEmail $sendOrderConfirmationEmail
    ) {
        $this->sendConfirmationEmail = $sendOrderConfirmationEmail;
    }

    /*
     * When the event dispatcher receives an 'OrderWasCreated'

```

```

event
    * it will call this method:
    */

    public function whenOrderWasCreated(OrderWasCreated $event):
void
{
    $this->sendConfirmationEmail
        ->send($event->orderId(), $event->emailAddress());
}
}

```

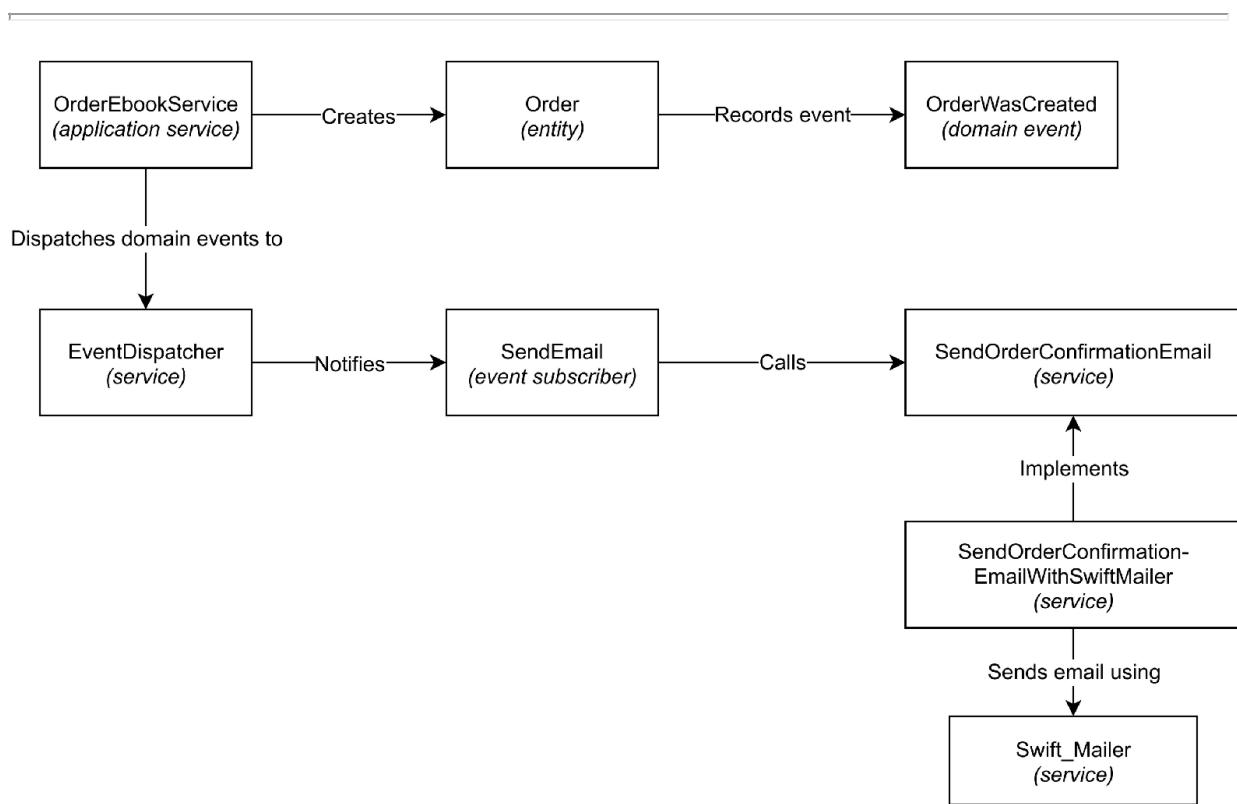


Figure 4.2: The resulting set of objects and their interactions.

4.6 Summary

We started this chapter with a controller that represented the use case of ordering an e-book. The only way this use case could be invoked was through a web form request. To make the use case reusable, we first removed web-specific dependencies like Request, Response and the

`session` object. After that, the code only relied on primitive-type input data and a few service dependencies. We moved the infrastructure-independent code to its own class, making the use case effectively reusable. By introducing this class, which is called an *Application service*, we made it possible for any type of client to provide the necessary input data and exercise this use case. Finally, we used the *Introduce Parameter Object* refactoring to combine all the primitive-type parameters of the application service into a single class. Such a class is called a command class.

After decoupling the primary step of the use case, we also wanted to move the second step of the use case to the application service. Instead of directly moving the code for sending the order confirmation email, we introduced a proper abstraction. This allowed us to move the implementation details out of sight and have a clear representation of this step inside the application service. We also looked at a more generic solution for dealing with multiple use case steps which relies on domain events and dispatching them to event subscribers by means of an event dispatcher service.

Exercises

1. What makes an application service reusable by different clients?⁴²
 1. Providing input data in a way that doesn't tie it to a particular delivery mechanism
 2. Combining all input data in a single *Parameter object*
2. Put the refactoring steps for decoupling a use case in the correct order:⁴³
 1. *Extract Class*
 2. *Introduce Parameter Object*
 3. *Extract Variable*
 4. *Inline Variable*
3. Add at least four more examples of clients that don't want to invoke an application use case by submitting a web form.⁴⁴

1. An HTTP-JSON API endpoint
 2. A console command
 3. ...
-
-

5 Service locators

This chapter covers:

- Rewriting service and configuration locator calls to dependencies injected as constructor arguments
 - Defining “pure” object-oriented services, which require nothing more than construction and method invocation
 - Pushing the composition root closer to the application’s entry point
-

5.1 From service location to explicit dependencies

In Chapter 4 we extracted an application service from a web controller. When we created the class, we also made its dependencies more explicit. When the code was still inside the controller we had to fetch the `OrderRepository` from the service container using `$this->container->get('order_repository')`. Once we moved the core logic to the `EbookOrderService` class, `OrderRepository` became a proper constructor argument:

```
// Inside the controller:  
$orderRepository = $this->container->get('order_repository');  
  
// In the application service:  
final class EbookOrderService  
{  
    private OrderRepository $orderRepository;  
  
    public function __construct(  
        // ...  
        OrderRepository $orderRepository  
    ) {  
        // ...  
        $this->orderRepository = $orderRepository;  
    }  
}
```

We could have injected the service container itself, which would have made moving the code even easier. If we had done that, the application service would depend on the framework-specific `ContainerInterface`:

```
use Symfony\Component\DependencyInjection\ContainerInterface;

final class EbookOrderService
{
    private ContainerInterface $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }

    public function createOrder(/* ... */): OrderId
    {
        // ...

        $orderRepository = $this->container->get('order_repository');

        // ...
    }
}
```

It's not a very nice thing to introduce a dependency on framework code inside core code. It doesn't break the first rule for core code though, because there is no runtime dependency on external systems. However, it does conflict somewhat with the second rule. By injecting the generic `ContainerInterface` we now require a special context for our service to run in. The container has to know about all the dependencies that the application service might need, and know how to instantiate any one of them at any point in time. If we inject actual services instead of the service container the application service will only be coupled to our own core interfaces and classes. Any client that can provide implementations for the required dependencies can instantiate the service and call methods on it. So injecting actual dependencies is an important part of making the service usable in different contexts, by different clients.

5.2 Depending on global state

Let's take a look at a similar situation where the service container doesn't get injected but where dependencies can be instantiated upon request, anywhere in the code. Listing 5.1 shows an example using *helper functions*⁴⁵ from the Laravel framework. Can this code run in any context? How is it tied to its surrounding infrastructure?

Listing 5.1: Code that uses static calls to load dependencies and configuration.

```
final class SendIpConfirmationEmail
{
    public function send(): void
    {
        $message = EmailMessage::create()
            ->to(Auth::user()->email())
            ->from(
                config('mail.default_sender')
            )
            ->text(
                trans(
                    'Add your :ip to the whitelist',
                    [
                        'ip' => request()->ip()
                    ]
                )
            );
        resolve(Mailer::class)->send($message);
    }
}
```

Before you can run this code, you have to make sure that the `resolve()` function can in fact resolve, i.e. instantiate, a `Mailer` service. You also need to set up a configuration value for `mail.default_sender`, or else the call to `config()` will fail. A bigger problem may be that this code assumes there's a session with a logged in user, and that this code runs as part of an HTTP request which has a known client IP address. Normally, the framework will take care of all of this by preparing the service container and loading the configuration before the `SendIpConfirmationEmail` service is used. But the

fact that these preparations have to happen *before* that moment, means that this code isn't as portable as we would like it to be. We can't just instantiate this service and start calling its methods. Besides a *temporal* dependency on the framework, the code itself is also coupled to global functions provided by the framework. `resolve()` and `config()` are functions defined by Laravel, so we need this particular framework to run this code.^{[46](#)}

Laravel isn't the only framework that offers this kind of tooling. Most of the frameworks I've worked with so far provided some global method which you could use to retrieve a service or configuration value. Famous examples are symfony 1's `sfContext`^{[47](#)} and Zend Framework 1's `Zend_Registry`^{[48](#)}. You could use `Zend_Registry` to store any value or object and make it globally accessible using its static `get()` method. `sfContext` allowed you to retrieve configuration values or built-in services, so you could basically send emails from your domain model if you wanted.

If you write reusable code like a library that you publish on Packagist^{[49](#)}, framework coupling is generally a bad thing. It limits the potential audience for your library to the people who use the same framework as you do. But your application code is not intended for reuse outside your current project, so do you still have to worry about framework coupling? In short: yes. There are several reasons for that. Your code will be more future-proof if you don't rely on framework-specific helpers or "syntactic sugar", which are subject to fashion. Your code will also be easier to test, and will remain easier to test, even when you have migrated to a different framework. In fact, if you depend on framework-specific classes and functions, it will be very hard to migrate to another framework because you will have to make changes everywhere.

In this chapter, we'll rewrite code that relies on framework tools for retrieving services or configuration values. We'll make sure that dependencies and configuration values are provided as constructor arguments, so objects will no longer rely on a special context to run. This is a very effective way to make sure that switching frameworks, or upgrading to the next major BC-breaking version, doesn't cause any trouble in the biggest part of your code base.

5.3 Injecting dependencies

Instead of retrieving services from some globally available function, we should simply inject all of them as required constructor arguments. Listing 5.2 shows what this looks like.

Listing 5.2: Injecting the `Mailer` as a dependency.

```
final class SendIpConfirmationEmail
{
    private Mailer $mailer;

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

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

        $this->mailer->send($message);
    }
}
```

An added advantage of promoting the `Mailer` service to a constructor argument, is that it makes this dependency explicit. I like to read a service's constructor as a list of things that the service needs. In this case: "The `SendIpConfirmationEmail` service needs the `Mailer` service to send an IP confirmation email to the currently logged in user." Compare this to having to read the entire method just to find out which dependencies will be fetched at runtime.

5.4 Injecting configuration values

Besides dependencies that are themselves services, a service sometimes needs configuration values too. In the case of the `SendIpConfirmationEmail` service, it needs to know what the default sender of system emails is. Configuration values should also be injected as constructor arguments, as shown in Listing 5.3. I'm adding an assertion as well, because one day that will save us some debugging time.

Listing 5.3: Injecting a configuration value as a constructor argument.

```
use Assert\Assertion;

final class SendIpConfirmationEmail
{
    private Mailer $mailer;
    private string $defaultSender;

    public function __construct(
        Mailer $mailer,
        string $defaultSender
    ) {
        Assertion::email($defaultSender);

        $this->mailer = $mailer;
        $this->defaultSender = $defaultSender;
    }

    public function send(): void
    {
        $message = EmailMessage::create()
            ->to(Auth::user()->email())
            ->from($this->defaultSender)
            // ...
            ;

        $this->mailer->send($message);
    }
}
```

With two constructor arguments, we can now be even more explicit: “The `SendIpConfirmationEmail` service needs the `Mailer` service and a default sender address to send an IP confirmation email to the currently logged in user.”

5.5 Using method arguments for job-specific data

Looking at the signature of the `send()` method, we can’t be certain to whom the system will send the email, or which IP address they will be asked to confirm. This leaves the client of the method with absolutely no control over the outcome.

There are two pieces of information that the `send()` method needs to do its job: the IP address from which the current HTTP request originates, and the email address of the currently logged in user. The use of the word “current” hints at the fact that we’re talking about contextual data.⁵⁰ This data is different for every request. Examples of contextual data are the request time, the client’s IP address, the ID of the logged in user, the size of the terminal the user uses to run our application, whether or not it supports color, etc.

In our example, the `SendIpConfirmationEmail` service reaches out to globally available methods to find out more about the context in which the service was invoked. It inspects the current request object after retrieving it from the `request()` helper function, and it retrieves the logged in user’s email address through the `Auth::user()` method. The current user and IP address are neither services nor configuration values. So we can’t inject them as constructor arguments. Instead, we need to pass them to the service as arguments of its `send()` method. Listing 5.4 shows the result of this change.

Listing 5.4: Passing contextual data as method arguments.

```
final class SendIpConfirmationEmail
{
    // ...

    public function send(
        User $user,
        string $ipAddress
    ): void {
        $message = EmailMessage::create()
            ->to($user->email())
            ->from($this->defaultSender)
            ->text(
                trans(
                    'Add your :ip to the whitelist',
                    [
                        'ip' => $ipAddress
                    ]
                )
            );
        $this->mailer->send($message);
    }
}
```

```
}
```

There's one last thing we have to make explicit: the call to `trans()`, which is supposed to translate the email message to the user's language. In practice, `trans()` will use the locale that is configured for the current request. Even though we don't explicitly fetch this information from the request, like we did with the client's IP address, we still rely on it to be configured before this service gets invoked. This makes us once more dependent on the framework's infrastructure. To make this dependency explicit, we have to do two things: pass the user's locale explicitly to the translator, and make the translator an explicit dependency of the service. Listing 5.5 shows the result.

Listing 5.5: Injecting an extra dependency and providing an explicit locale.

```
final class SendIpConfirmationEmail
{
    // ...
    private Translator $translator;

    public function __construct(
        /* ... */
        Translator $translator
    ) {
        // ...
        $this->translator = $translator;
    }

    public function send(/* ... */): void {
        $message = EmailMessage::create()
            // ...
            ->text(
                $this->translator->trans(
                    'Add your :ip to the whitelist',
                    [
                        'ip' => $ipAddress
                    ],
                    $user->locale()
                )
            );
        // ...
    }
}
```

```
}
```

5.6 Clients of reusable services

Adding more required constructor arguments to a service requires some changes to the place where services are instantiated. If you use auto-wiring for services it'll be no effort, but if you have a hand-written service container⁵¹ or one based on configuration files, you need to make some manual changes. Clients that use the `send()` method also have to be updated. They can no longer call that method without any argument, but have to provide the logged-in `User` and the client IP address from the current HTTP request.

Listing 5.6 shows one of the clients of the `SendIpConfirmationEmail` service: a web controller action. A controller naturally has access to the authenticated user and the current `Request` object, so it's easy to update the code to call `send()` with the right arguments.

Listing 5.6: `UserController` has access to the required contextual information and passes it to the `SendIpConfirmationEmail` service.

```
final class UserController
{
    private SendIpConfirmationEmail $confirmationEmail;

    public function __construct(
        SendIpConfirmationEmail $confirmationEmail
    ) {
        $this->confirmationEmail = $confirmationEmail;
    }

    public function sendIpConfirmationEmail(
        Request $request
    ): Response {
        $this->confirmationEmail->send(
            Auth::user(),
            $request->ip()
        );
    }
}
```

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

Since the `SendIpConfirmationEmail` service itself no longer takes the IP address from the current `Request` object, and it no longer relies on an active session with an authenticated user, we have successfully decoupled this code from its surrounding infrastructure. It doesn't rely on a special context to have been prepared for it. And it doesn't rely on external systems to be available.

We've already done something similar in the previous chapter, where we extracted the order-an-e-book use case from the web controller. Doing this for the `SendIpConfirmationEmail` service too, makes it effectively reusable in a different context. For example, your company's support website could offer a form, allowing an employee to manually send the confirmation email, in case something went wrong. Such a controller action gets the user's information not from the session, but from the form. Because the refactored `SendIpConfirmationEmail` service doesn't rely on global state for its input, it can easily be reused in this new context, as is shown in Listing 5.7.

Listing 5.7: Reusing the `SendIpConfirmationEmail` service in a different context.

```
final class SupportController
{
    private SendIpConfirmationEmail $confirmationEmail;
    private UserRepository $userRepository;

    public function __construct(
        UserRepository $userRepository,
        SendIpConfirmationEmail $confirmationEmail
    ) {
        $this->userRepository = $userRepository;
        $this->confirmationEmail = $confirmationEmail;
    }

    public function sendIpConfirmationEmail(
        Request $request
    ): Response {
        $user = $this->userRepository->getById(
            $request->get('id')
        );
        $confirmationEmail = $this->confirmationEmail;
        $confirmationEmail->send($user);
        return $this->responseFactory->createResponse();
    }
}
```

```

        // Take the user ID from the form:
        $request->request->get('user_id')
    );

    $this->confirmationEmail->send(
        $user,
        // Take the user's IP address from the form:
        $request->request->get('ip')
    );

    // ...
}

}

```

5.7 Testing

I already mentioned testability as an important reason to reduce framework coupling. In theory, you should be able to write a unit test for every class. In practice, you may not want to do this, since different areas of a code base require different types of tests (we'll cover this topic in Chapter [14](#)). For now, let's say we want to write a unit test for `SendIpConfirmationEmail`. A unit test is a test that doesn't use any IO (it doesn't use the file system, a database, an external service, etc.) and usually but not necessarily covers a smaller unit of code, like a single class.

Let's write a unit test for the initial version of the `SendIpConfirmationEmail`. Listing [5.8](#) shows what this test would look like.

Listing 5.8: A test for the initial version of the `SendIpConfirmationEmail` service.

```

public function it_sends_an_ip_confirmation_email(): void
{
    $user = new User(/* ... */);
    $ip = '123.234.123.234';

    // Fake the currently logged in user
    Auth::shouldReceive('user')
        ->once()
        ->andReturn($user);

```

```

// Set the user's locale for 'trans()'
App::setLocale($user->locale());

// Configure the default sender email address
$sender = 'info@mycompany.com';
Config::set('mail.default_sender', $sender);

// Fake the current request
Request::shouldReceive('ip')
    ->once()
    ->andReturn($ip);

// Set up the Mailer mock
$mailer = $this->createMock(Mailer::class);
$mailer->expects($this->once())
    ->method('send')
    ->willReturn(
        function (Message $message) use ($user, $ip, $sender)
) {
    $this->assertEquals(
        $user->emailAddress(),
        $message->to()
    );
    $this->assertEquals($sender, $message->from());
    $this->assertContains($message->text(), $ip);
}
);
// Make sure the container will return our mock
Container::getInstance()->instance(
    Mailer::class, $mailer
);

$service = new SendIpConfirmationEmail();
$service->send();
}

```

The framework we've used in this example offers convenient methods for replacing the requested services, configuration values, and contextual information with test doubles.⁵² The don't-use-any-IO rule isn't actually guaranteed though. If we would forget to set up a correct test double for the `Mailer` service, this test might start sending actual emails. But the `Mailer` isn't the only problem. Since the class could fetch any service using `resolve()`, and any of those services could use IO, we can't guarantee that this test is a unit test, or remains a unit test.

Now compare the test for the initial version with the test for the refactored service, shown in Listing 5.9.

Listing 5.9: A test for the refactored service.

```
public function it_sends_an_ip_confirmation_email(): void
{
    $user = new User(/* ... */);
    $ip = '123.234.123.234';
    $sender = 'info@mycompany.com';

    $mailer = $this->createMock(Mailer::class);
    $mailer->expects($this->once())
        ->method('send')
        ->willReturn(
            function (Message $message) use ($user, $ip, $sender)
) {
            $this->assertEquals(
                $user->emailAddress(),
                $message->to()
            );
            $this->assertEquals($sender, $message->from());
            $this->assertContains($message->text(), $ip);
        }
    );

    $translator = $this->createMock(Translator::class);
    $translator->expects($this->once())
        ->method('trans')
        ->willReturn(
            function ($message, $replacements, $locale) use ($us
er) {
                $this->assertEquals($locale, $user->locale());
                return $message . ' (translated)';
            }
        );
}

$service = new SendIpConfirmationEmail(
    $mailer,
    $sender,
    $translator
);
$service->send($user, $ip);
}
```

The number of lines of code is roughly the same in both tests. But the latter has a few advantages.

First of all, just like the refactored service itself, it's completely decoupled from framework-specific tooling like:

1. `Auth::shouldReceive()`
2. `Container::getInstance()`
3. `App::setLocale()`
4. `Config::set()`
5. `Request::shouldReceive()`

This means that the unit test itself would survive a framework migration, just like the service itself.

Second, there's no global state that the service relies on, so there's no global state that needs to be prepared before testing or running it. Everything it needs is provided as constructor or method arguments. You can test various branches of the code by introducing a bit of variation to the constructor or method arguments. There are no other, hidden ways in which the behavior of the code may change.

The test proves that the service is fully portable. It can be used in other areas of the application that use a different framework, or no framework at all. The framework doesn't even have to be booted or anything, you only need to set up the service with the correct dependencies.

Knowing that predictability and the ability to run in isolation are desirable things when writing a unit test, let's formulate this as a characteristic of unit-testable code:

The result of calling a method on an object should be determined by its own implementation logic, and optionally by the behavior of one of its constructor arguments, or the method arguments provided to it; and nothing more.

As an example: the behavior of the original version of the `SendIpConfirmationEmail` service can't be explained by looking at its constructor arguments or method arguments only. In fact, it has no constructor or method arguments. So we would expect that its own

implementation is able to explain its behavior. That's not true either. A large part of the behavior of this service is defined by the dependencies that are retrieved using the static helper functions, and the service also relies on the correct behavior of those helper functions themselves. This is very inconvenient when you're testing code. You'd want to be able to treat the unit as a black box, call methods on it, and assert that it has the desired behavior. What goes on inside, should not be a concern of the unit test. However, when testing the initial version, the resulting test code is very much concerned with the internals of the subject under test. It has to ensure that everything is ready to call the `send()` method. Of course, for the rewritten service we also need to set up a few things. But this can be a “compiler-assisted” process, and when the compiler says you're done, you will actually be done. See Listing 5.10 and Figure 5.1 for a demonstration of such a process.

Listing 5.10: The compiler assists us when instantiating and using the service.

```
new SendIpConfirmationEmail();
// Error: Missing constructor argument of type 'Mailer'

$mailer = $this->createMock(Mailer::class);
new SendIpConfirmationEmail($mailer);
// Error: Missing constructor argument of type 'string'

$sender = 'a string';
new SendIpConfirmationEmail($mailer, $sender);

// Error: Missing constructor argument of type 'Translator'
$translator = $this->createMock(Translator::class);
 getService = new SendIpConfirmationEmail(
    $mailer,
    $sender,
    $translator
);

// Error: Value "a string" was expected to be a valid e-mail address.

$sender = 'info@matthiasnoback.nl';
getService = new SendIpConfirmationEmail(
    $mailer,
```

```

    $sender,
    $translator
);
// OK

$service->send();
// Error: Missing method argument of type 'User'

$user = new User(/* ... */);
$service->send($user);
// Error: Missing method argument of type 'string'

$ip = '127.0.0.1';
$service->send($user, $ip);
// OK

```

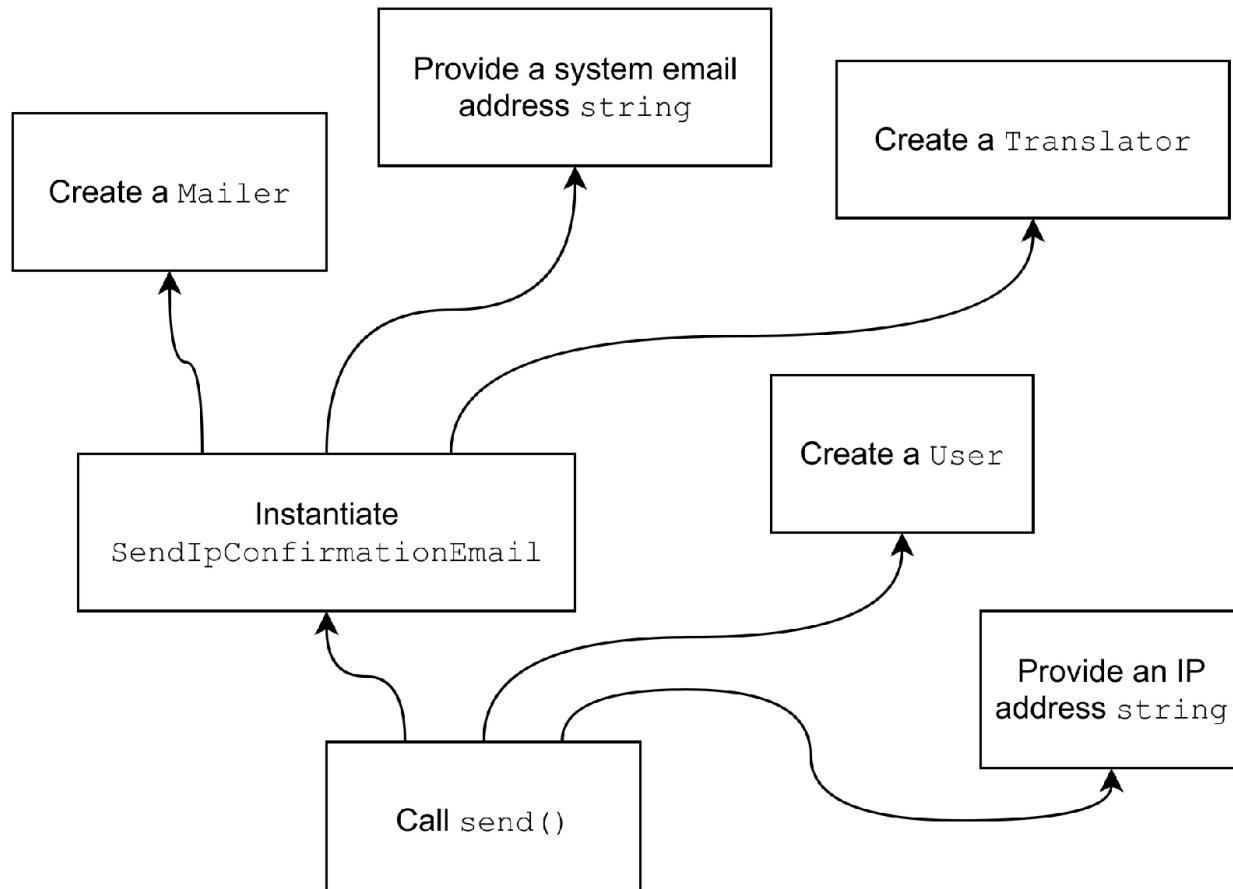


Figure 5.1: Before you can call `send()`, the compiler will help you discover everything you need to provide.

If you try to do the same thing with the original version of the service, the result would be completely different. It will be very easy to instantiate the service since the constructor doesn't have any required argument (there isn't even an official constructor), but once you start calling `send()` to verify correct behavior, you'll be in trouble ([Listing 5.11](#)).

Listing 5.11: With implicit dependencies, instantiation is easy, usage is not.

```
new SendIpConfirmationEmail();
// OK

$service->send();
// Error: No logged in user

// Or maybe: Fatal error: Call to a member function email() on null

// Or maybe: Error: Undefined config key "mail.default_sender"
```

In order to fix the problems you have to take a closer look at the code of the `send()` method. You'll have to set up all its implicit dependencies first. Unfortunately, you won't even know when you're done. Even if you look carefully through the code to find out what it needs, there's still a chance that you missed something. For instance, the `trans()` call secretly relies on a `locale` to be provided by a call to `App::setLocale()`. The code in `send()` doesn't reveal this information; you have to *know it*. Figure [5.2](#) shows how the prerequisites of calling `send()` are disconnected from the structure of the class (its constructor and method arguments).

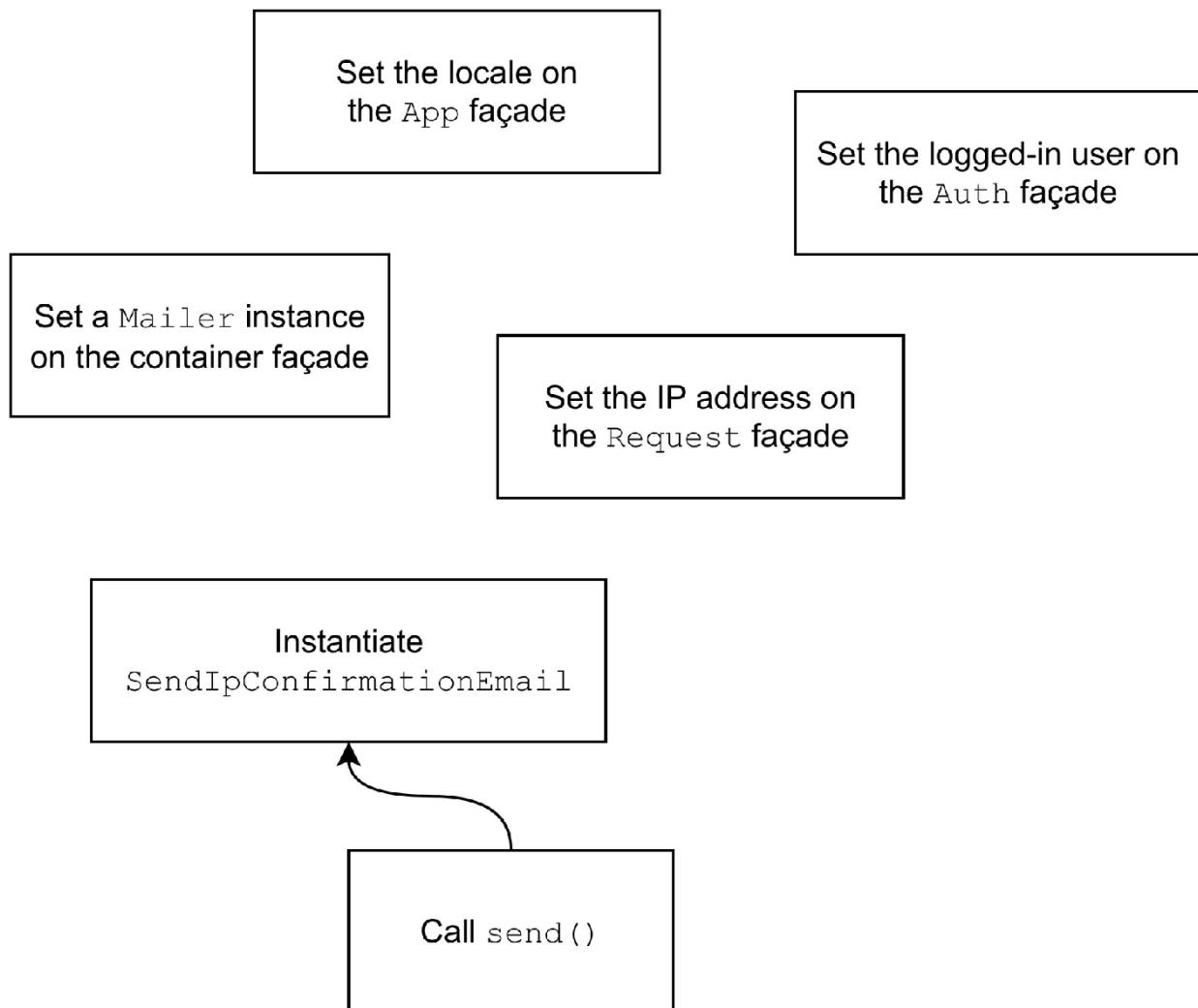


Figure 5.2: The compiler can't help you figure out what's needed to call `send()`. The prerequisites can't be derived from the constructor or method signatures.

5.8 Effective testing

Is it bad that you have to read the code of an object, in order to write a unit test for it? Well, I think it is. You'll have to go back and forth between the production code and the test code, and redo all the work that the framework normally does for you. This makes you feel like you're doing twice the amount of work that you would do without writing a test. At the same time, if you change the production code in such a way that an extra setup step is

required, you'll certainly break another test which doesn't have that extra step yet.

The problem here is that this kind of test is too close to the production code. It doesn't help you to work faster, or be more effective. After a few of these tests you will be tempted to stop writing tests at all.

Part of the solution is to design your services in a different way. Make things more explicit: both dependencies and method arguments help with that. Another part of the solution is not to write a unit test for every single class. Unit tests only fit a certain type of objects. Mailer services, controllers, repositories, they all need integration tests; tests that use the real thing (a mail server, a web server, a database server). Testing this kind of code as a unit test doesn't prove anything about its effectiveness.

But it always pays to find a way to separate the logic from the infrastructural concerns. If you can do this, you will end up with two islands of code: for the one you can write unit tests, for the other integration tests. We'll see an example of an effective separation of these two types of code in Chapter [6](#). The bigger the first island is, the better. You will have more code that can be unit tested. This code will be easier to work with, since all you need to do is instantiate an object and call methods on it. The test suite will be very fast, since running a unit test takes almost no time. This encourages you to write more tests, showing how the code deals with variations of the input data. And it tightens the feedback loop between writing the code, and finding out if you did the right thing.

“Dependency injection, passing contextual information; seems like a lot of work!”

A common objection to using dependency injection, explicitly passing contextual information, etc. is that it's a lot of work to write all that code. Comparing the refactored version of the `SendIpConfirmationEmail` with the original version of the class you could indeed reach this conclusion: too much work. Maybe there aren't a lot more statements in the final version,

but there are indeed more symbols in it, e.g. constructor and method parameters and properties. I hope that I've been able to demonstrate the advantages of adding these extra elements, but that still doesn't answer the question: isn't this a lot of work? Of course, if you're using global static helper functions everywhere, refactoring all of your code to this new style will be quite the effort. But this is exactly the point: just make sure you'll never have to do that. Don't use any framework-assisted global state-based tool for retrieving dependencies, configuration values, etc. You can't easily fix old code, but you can do the right thing from now on.

Unfortunately, this chain of reasoning is lacking. "You should do A now, because if you did B and you want to go to A, that'll be a lot of work!" So why not stay at B if you're happy there? Because if your application lives longer than 2 or 3 years, you still want to go to A. Anecdotal evidence for this is that my team and I had a lot of trouble getting rid of the global static service and configuration locator called `Zend_Registry` (and I'm sure I'm not the only one with this experience). So again, all I can say is: just don't start using anything like it.

5.9 The Composition root is near the entry point

In the initial version of `SendIpConfirmationEmail` we used `resolve()` to let the service container instantiate the `Mailer` service for us. Step by step we were able to transform all those dynamically resolved dependencies and configuration values into constructor arguments. But we still rely on the service container to resolve all those things for us, and inject them as constructor arguments. The difference is that in the beginning we used the service container to *locate* services and configuration values for us, whereas now we rely on the container to *provide* these things to us. The `SendIpConfirmationEmail` class no longer has to worry about where its dependencies should come from, or how to make them. It only declares a number of required constructor arguments and their types. This is called *inversion of control*. A general rule is to never use the service container as a *locator*, only as an *injector*, sometimes called an *inversion of control container*.

But even if we use proper constructor injection everywhere, there has to be at least a single call to the service container to instantiate the first service so we can use it in a controller. However, we just said that a container should never be used as a service locator.

To solve this paradox, we should turn to Mark Seemann, who wrote about this specific problem⁵³. He introduces the concept of an *entry point*, which is the first user code that the framework calls. User code is code that is not part of the framework, but written by you or your team members. Often such an entry point is the controller which the framework decides to call based on the current request URL. In this case the framework may be able to instantiate the controller by doing something like `new $controllerClass()`. But once inside the controller, where do you get your services from? You have to start somewhere. Mark suggests that the container plays a very specific role in these situations: the role of *Composition root*.

A Composition Root is a (preferably) unique location in an application where modules are composed together.

It's where the object graph, consisting of the first service including all its dependencies, and their dependencies, and so on, gets composed. In our example the controller is the *Entry point* of the application:

The *entry point* is the user code that the framework calls first.

Seemann suggests that at or near the entry point of the application, it seems as if we're using the container as a service locator, but in fact, we use it as a composition root, which is a different role. We can therefore be more specific and say that not all calls to `resolve()`, `Container::get()`, etc. should be forbidden. As long as the container plays the role of composition root, we can use it near the entry point of our application:

A Composition Root is located at, or near, the entry point. An entry point is where user code is first executed by a framework.⁵⁴

In practice this means that inside a controller it's perfectly okay to retrieve a service from the container. Once *inside* that service though, the use of the service container is no longer allowed. If your framework offers the ability

to define controllers as services too, then you can even move the composition root one level up. The framework would then ask the container to instantiate your controller, and all of its dependencies will be injected as constructor arguments. Note that this last step isn't required. The big win is that we're not using a service locator anywhere in our code other than near the entry points.

5.10 Summary

In this chapter we started with a service that used globally available helper functions to resolve dependencies, configuration values, and contextual information. We refactored the code to get its dependencies and configuration values injected as constructor argument. We also made sure that any relevant contextual information was provided as method arguments. Changing the structure of the service made it a “pure” service; its behavior is fully determined by its own implementation and the behavior of the constructor and method arguments. This makes the service a much better candidate for unit testing. Finally, we discussed how using dependency injection everywhere can push the composition root of your service object graph closer to the application’s entry point.

Exercises

1. What are the advantages of using constructor injection over service location?⁵⁵
2. Why is core code not allowed to rely on global state for dependencies and contextual information?⁵⁶
 1. Because doing so means that external systems have to be available at runtime.
 2. Because doing so means that a special context needs to be provided for this code to run in.

3. Categorize these things as job-specific data, dependency, or contextual information.^{[57](#)}

1. The `EbookRepository`
2. The hostname of the current HTTP request
3. An `EbookId`
4. A `CreateOrder` command object
5. The `EventDispatcher`

4. How should these things be provided to a service, as constructor arguments or as method arguments?^{[58](#)}

1. Dependencies
2. Contextual information
3. Job-specific data
4. Configuration values

5. Is your code ever allowed to fetch a service directly from a service container?^{[59](#)}

1. No, never. Always use constructor injection.
 2. Yes, but only very close to where user code is first executed by the framework.
-
-

6 External services

This chapter covers:

- Creating an object-oriented layer around calls to external services
 - Introducing proper abstractions for calls to external services
 - Creating integration tests for code that uses external services
-

We ended Chapter 4 with an extracted use case for ordering an e-book. An important aspect of that use case is still missing: the correct calculation of VAT (value-added tax). VAT calculation is a huge topic, with lots of rules, and lots of exceptions too. Trying to get it right for all the possible situations and all the countries in the world, would be a tremendous effort. Standing in front of a large amount of work, we should consider our options. Instead of spending an awful lot of time getting every detail right, we could reduce the scope by limiting the number of possible situations the application needs to care about. Or we could outsource the solution. That doesn't mean we should let a team of external developers work on it. Instead, we should consider using an already existing solution. In the case of VAT calculation there are multiple web services that can give us the answers we need. For now, let's use `vatapi.com` (for no specific reason; I'm also not affiliated to them in any way).

The domain model has already been prepared to deal with VAT (Listing 6.1). There's a `VatRate` value object which can be created from an `int`, representing the VAT percentage (ignoring for now the possibility that a VAT rate could actually be a decimal number). The `Order` class has an extra constructor argument which accepts a `VatRate` object. `order` will use this `VatRate` internally to calculate the amount of VAT that should be charged.

Listing 6.1: The `VatRate` class and how it's used in `Order`'s constructor.

```
final class VatRate
{
    public static function fromInt(int $percentage): self
```

```

    {
        // ...
    }

final class EbookOrderService
{
    // ...

    public function create(/* ... */): OrderId
    {
        // ...

        $vatRate = // ...

        $order = new Order(
            $orderId,
            $ebook->id(),
            $emailAddress,
            $orderQuantity,
            $ebook->priceInCents(),
            $vatRate
        );

        // ...
    }
}

```

6.1 Connecting to the external service

Now, how to determine that VAT rate? Looking at the documentation of [vatapi.com⁶⁰](#) we notice that there is a “VAT Rate Check” API endpoint. They even offer some examples written in PHP. Copying the code into the service gives us a more-or-less working feature, as shown in Listing [6.4](#).

Listing 6.2: EbookOrderService uses curl to fetch a VAT rate from [vatapi.com](#).

```

public function create(/* ... */): OrderId
{
    // ...

    $curl = curl_init();

```

```

/*
 * Search for TBE services (Telecommunications,
 * broadcasting & electronic services).
 * Filter on "ebooks".
 * For now, let's use a hard-coded country code.
 */
$url = 'https://eu.vatapi.com/v2/vat-rate-check?' .
    http_build_query(
        [
            'rate_type' => 'TBE',
            'country_code' => 'NL',
            'filter' => 'ebooks'
        ]
    );
// @todo Inject the API key as a configuration value
$apiKey = 'THE_SECRET_API_KEY';

curl_setopt_array(
    $curl,
    [
        CURLOPT_URL => $url,
        CURLOPT_RETURNTRANSFER => true,
        CURLOPT_ENCODING => "",
        CURLOPT_MAXREDIRS => 10,
        CURLOPT_TIMEOUT => 0,
        CURLOPT_FOLLOWLOCATION => false,
        CURLOPT_HTTP_VERSION => CURL_HTTP_VERSION_1_1,
        CURLOPT_CUSTOMREQUEST => 'GET',
        CURLOPT_HTTPHEADER => [
            'x-api-key: ' . $apiKey
        ]
    ]
);
$response = curl_exec($curl);
$curlError = curl_error($curl);
curl_close($curl);

if ($curlError) {
    throw new RuntimeException('cURL Error #' . $curlError)
;
}
$responseData = json_decode($response, true);
if ($responseData['status'] === 200) {

```

```

        if ($responseData['filter_match']) {
            $vatRate = VatRate::fromInt(
                // The "ebooks" filter returned a match
                $responseData['rate']
            );
        } else {
            /*
             * The "ebooks" filter didn't return a match.
             * We should take the rate from the generic rates o
bject.
            */
            $vatRate = VatRate::fromInt(
                $responseData['rates']['electronic']['rate']
            );
        }
    } else {
        throw new RuntimeException(
            'Could not determine the VAT rate'
        );
    }

    // ...
}

```

It's a lot of code, and the code itself doesn't look great, but it works. This is fine, as long you're just “spiking” – fiddling with the code and the design, until you know what to do. But it would be really bad for the long term maintainability of the code to leave it like this. One design issue is that we have polluted the service with all kinds of low-level details. The `EbookOrderService` was supposed to give a high-level overview of the use case scenario it implements. Now it's mostly about connecting to an external service. All this code really represents one step in the process: “Determine the VAT rate that should be applied to the order”. We should do our best in this chapter to simplify the code until you can clearly recognize this step by reading the code.

Another issue is that by making a call to an external service, we now have extremely limited testing options for the `EbookOrderService` class. We can no longer write an isolated test for it, since running the code will produce an actual network call. We also don't have a way to test *only* the call to the external service, meaning that our integration with `vatapi.com` is tied to the use case of creating an order. This implies that we can't switch to a different

service, or upgrade the code in case the API response format changes, without touching the `EbookOrderService` as well.

In the following sections we'll gradually improve the situation until we've reached the following objectives:

- `EbookOrderService` should be concerned with the use case of ordering an e-book, and should only have code for the high-level steps of that use case. Every step that makes a connection to some external system, like the database, or the VAT API, should be replaceable to make the service testable in isolation.
- The low-level implementation details of connecting to an external service for determining a VAT rate should be extracted to a different class, which the `EbookOrderService` can use. We can test this class separately, so we know that it works correctly with the actual VAT API remote service.

The thing that bothers me about all those `curl_*` calls and `json_decode()` stuff is that it's old-school procedural code which leaves a lot of possibilities open, allowing all kinds of bugs to slip by unnoticed. I'd normally want to use objects and let these objects do all kinds of checks to make sure that the data is good and safe to use. Basically what I want is an object-oriented wrapper around the remote API. I would start developing such a layer by creating a single object that represents the API, like the `VatApi` class in Listing 6.3. It would allow clients to use the web service without worrying about what those HTTP requests should look like. Such a `VatApi` class could have a method that has the same name as the API's endpoint. In our case, the class gets a method called `vatRateCheck()` which corresponds to the `/vat-rate-check` endpoint. The `vatRateCheck()` method performs the actual HTTP request, and takes care of the response decoding, including basic error handling. The query parameters `rate_type`, `country_code`, and `filter` are promoted to method parameters. Note that the API key is provided as a constructor argument, not as a method argument, because it's a *configuration value*.

Listing 6.3: `VatApi` makes the actual call to `vatapi.com`.

```
final class VatApi
{
    private string $apiKey;

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

    public function vatRateCheck(
        string $rateType,
        string $countryCode,
        ?string $filter
    ): array {
        $curl = curl_init();

        $url = 'https://eu.vatapi.com/v2/vat-rate-check?' .
            http_build_query(
                [
                    'rate_type' => $rateType,
                    'country_code' => $countryCode,
                    'filter' => $filter
                ]
            );
        curl_setopt_array(
            $curl,
            [
                // ...
                CURLOPT_HTTPHEADER => [
                    'x-api-key: ' . $this->apiKey
                ]
            ]
        );
        // ...

        $responseData = json_decode($response, true);
        // ...

        return $responseData;
    }
}
```

We could call this type of object a *Façade*⁶¹. It “defines a higher-level interface that makes the subsystem easier to use”. You will often find classes like `VatApi` in a so-called SDK (software development kit). An API vendor sometimes offers these as programming language-specific libraries. They allow developers to use an API as if it were a local service.

The new `vatRateCheck()` method is generic enough to be usable in our `EbookOrderService` class, but it isn’t very user-friendly at the moment. Clients get the raw response data as an array and have to manually extract the VAT rate from this array. The problem with arrays is that their shape is undefined. Clients wouldn’t know how they can extract from it the information they need. In our application we’d find code like this:

```
$responseData = $this->vatApi->vatRateCheck(  
    'TBE',  
    'NL',  
    'ebooks'  
)  
  
if ($responseData['filter_match']) {  
    // use $responseData['rate']  
} else {  
    // use $responseData['rates']['electronic']['rate']  
}
```

This code only works for the particular remote API that we’re using now. A bigger problem is that this code might produce PHP errors. There’s nothing about the `array` return type that guarantees us that these keys are actually defined. So what we should do is define a clear outline of the response and return an object that represents this response. Objects are easier to use than an array of data because they have a predefined set of methods that can be used without fear. Objects can also validate their input upon construction, which will make them safer to use.

So let’s improve the `vatRateCheck()` method by returning for instance a `VatRateCheckResult` object. This object receives the raw response data as a constructor argument, and offers getters which know how to extract the requested information from that response data (see Listing 6.4).

Listing 6.4: vatRateCheck() returns a specialized VatRateCheckResult object.

```
final class VatRateCheckResult
{
    private array $responseData;

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

    public function rate(string $fallbackType): int
    {
        if ($this->responseData['filter_match']) {
            return (int)$this->responseData['rate'];
        }

        return (int)$this->responseData['rates']
            [$fallbackType]['rate'];
    }
}

final class VatApi
{
    // ...

    public function vatRateCheck(
        string $rateType,
        string $countryCode,
        ?string $filter
    ): VatRateCheckResult {
        // ...

        return new VatRateCheckResult($responseData);
    }
}
```

I still wouldn't consider this excellent code. The issue is that it makes abundant use of primitive types (you could say this code has *Primitive obsession*⁶²). This makes it really hard to use. Clients of `VatApi::vatRateCheck()` would have to guess what a valid value for `$fallbackType`, `$rateType`, `$countryCode`, and `$filter` could be. This is a classic scenario where introducing some *value objects* would be very

helpful. They will assist the client while discovering the possible values that they could provide. For instance, `$rateType` could be promoted to a `RateType` value object, with two named constructors, which make it an *enum-like* data type (Listing 6.5).

Listing 6.5: A value object for representing “rate types”.

```
final class RateType
{
    private const GOODS = 'GOODS';
    private const TBE = 'TBE';

    private string $rateType;

    private function __construct(string $rateType)
    {
        $this->rateType = $rateType;
    }

    public static function goods(): self
    {
        return new self(self::GOODS);
    }

    /**
     * TBE stands for "Telecommunications, broadcasting,
     * and electronic services"
     */
    public static function tbe(): self
    {
        return new self(self::TBE);
    }
}
```

But before we accidentally develop an entire SDK for `vatapi.com`, we should take a step back and look at the overall design as it is now. We have created a `VatApi` class which provides a more-or-less object-oriented wrapper for the remote `vatapi.com`. Indeed, we could now use `VatApi` as a dependency in `EbookOrderService` and leave most of the low-level details to the `VatApi` class (see Listing 6.6).

Listing 6.6: The `EbookOrderService` could use the `VatApi` service.

```

final class EbookOrderService
{
    // ...
    private VatApi $vatApi;

    public function __construct(
        // ...
        VatApi $vatApi
    ) {
        // ...
        $this->vatApi = $vatApi;
    }

    public function create(/* ... */): OrderId
    {
        // ...

        $rateCheckResult = $this->vatApi->vatRateCheck(
            RateType::tbe(),
            'NL',
            'ebooks'
        );
        $vatRate = VatRate::fromInt(
            $rateCheckResult->rate('electronics')
        );

        // ...
    }
}

```

We no longer see `curl_*`() calls littered around the code. But we still have words like `vatRateCheck()`, `rateCheckResult` and a string constant `'electronics'` in our code, which are alien to our own business domain. All we actually care about is a VAT rate for our e-book order. So the current version of the code doesn't communicate in the clearest way possible that what we're doing here is determine the correct VAT rate for the order.

Furthermore, the `EbookOrderService` depends directly on the `VatApi` class (Figure [6.1](#)).

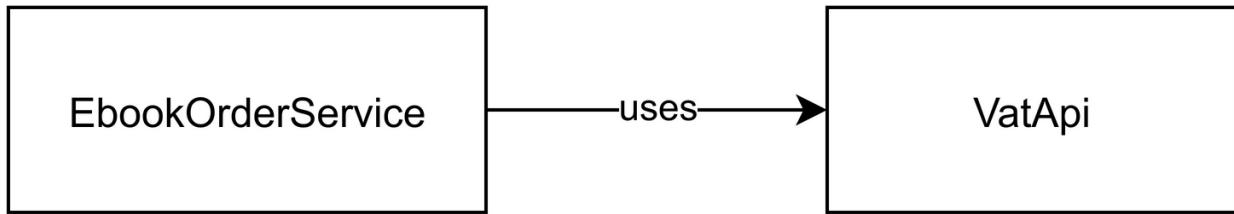


Figure 6.1: EbookOrderService depends on VatApi.

Even though we've hidden most implementation details behind this class, the `vatApi` class itself is still very specific to the particular remote service we use here. Imagine switching to a different VAT API service. That would be impossible without touching the code in `EbookOrderService`. Most likely it has different endpoints which accept different types of query parameters with different names.

This situation is pretty common in applications that talk to external systems. Yes, implementation details are nicely hidden inside other objects. But these objects don't have the proper *level of abstraction*, making it impossible to switch implementations, whether that be for testing purposes, or when you actually want to switch to a different remote API.

6.2 Introducing an abstraction

A very powerful technique to make sure that levels of abstraction are correct, is to replace the dependency to an actual class with a dependency on an interface. However, the interface should not simply be extracted from the class. If the only thing we did was define all methods of `vatApi` in an interface, the situation wouldn't be much better. In fact, *nothing* would change about the `EbookOrderService` and how it uses the injected `VatApi` instance (see Listing 6.7).

Listing 6.7: Extracting an interface from the `VatApi` class is not the solution.

```
interface VatApi
{
    public function vatRateCheck()
```

```

        string $rateType,
        string $countryCode,
        ?string $filter
    ): VatRateCheckResult;
}

final class ActualVatApi implements VatApi
{
    public function vatRateCheck(
        string $rateType,
        string $countryCode,
        ?string $filter
    ): VatRateCheckResult {
        // Connect to the remote API...
    }
}

final class EbookOrderService
{
    // Although VatApi is an interface now...
    private VatApi $vatApi;

    // ...

    public function create(/* ... */): OrderId
    {
        // ... Nothing changes here:

        $rateCheckResult = $this->vatApi->vatRateCheck(
            RateType::tbe(),
            'NL',
            'ebooks'
        );
        $vatRate = VatRate::fromInt(
            $rateCheckResult->rate('electronics')
        );
        // ...
    }
}

```

A better solution would be to rephrase our need and show what we are really looking for at this point. Basically, we're asking for a service to establish the correct VAT rate for an e-book. There should be some other object that can provide us with that information. Let's call that object a "VAT rate provider".

See Listing 6.8 for an example of how `EbookOrderService` would use such an object.

Listing 6.8: How `EbookOrderService` would use a “VAT rate provider”.

```
$vatRate = $this->vatRateProvider  
->vatRateForSellingEbooksInCountry('NL');
```

By imagining the object and the exact behavior we want from it, we've reached the right level of abstraction for the `EbookOrderService`. It shows what this service needs, in its own terms, nothing more, and nothing less. Now that we have defined the intended usage of this `vatRateProvider` object, we can define an interface for it. And since we already have an implementation which uses the `vatapi.com` service, we can also provide a standard implementation for the interface (see Listing 6.9). That implementation will use the already existing `VatApi` façade, which it receives as a constructor argument (see also Figure 6.2).

Listing 6.9: The `VatRateProvider` interface and its standard implementation.

```
interface VatRateProvider  
{  
    public function vatRateForSellingEbooksInCountry(  
        string $countryCode  
    ): VatRate;  
}  
  
final class VatRateProviderUsingVatApiDotCom  
    implements VatRateProvider  
{  
    private VatApi $vatApi;  
  
    public function __construct(VatApi $vatApi)  
    {  
        $this->vatApi = $vatApi;  
    }  
  
    public function vatRateForSellingEbooksInCountry(  
        string $countryCode
```

```

): VatRate {
    $rateCheckResult = $this->vatApi->vatRateCheck(
        RateType::tbe(),
        $countryCode,
        'ebooks'
    );

    return VatRate::fromInt(
        $rateCheckResult->rate('electronics')
    );
}
}

```

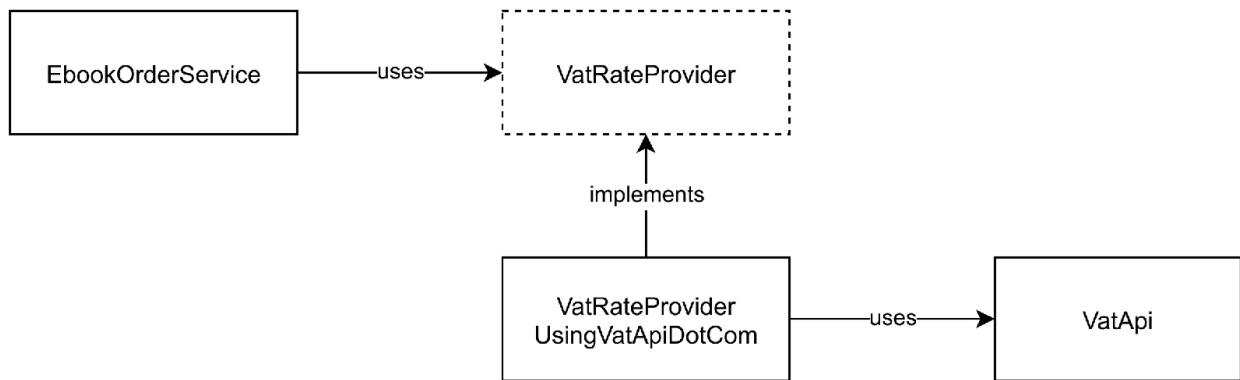


Figure 6.2: EbookOrderService depends on VatRateProvider.

The big difference between `VatApi::vatRateCheck()` and `VatRateProvider::vatRateForSellingEbooksInCountry()` is the return value. The former returns a `VatRateCheckResult` object, the latter a `VatRate` object. It has always been an object of type `VatRate` that we were really looking for, and so far, we were only able to create it after jumping through many hoops. Now we can get the appropriate `VatRate` instance in a single call, with a single argument.

Instead of depending on the `VatApi` class directly, `EbookOrderService` now depends on “anything that can provide us with the correct VAT rate to apply when selling e-books in country X”. As long as the dependency implements the `VatRateProvider` interface, the service should be happy. `EbookOrderService` only cares about “what” it needs, not about “how” the

real dependency gets it. This means that we have successfully introduced an abstraction.

6.3 Architectural advantages

Introducing an abstraction is a very helpful technique when you want to move low-level implementation details out of sight, and focus instead on the higher-level steps of a scenario. It's a technique that you'll often use when you want to decouple core code from infrastructure. How does introducing abstractions benefit the resulting application architecture?

In the first place, depending on a dependency for which we have defined our own specific interface, allows us to replace its default implementation with a minimum amount of work. If we want to switch to an alternative VAT rate API, maybe `taxtools.io`, we can easily do so. We just include their SDK in our project (or write our own little API client), and we provide a new implementation of a `VatRateProvider` which uses that new client (see Listing [6.10](#) and Figure [6.3](#)).

Listing 6.10: Switching to `taxtools.io` is easy.

```
final class TaxTools
{
    public function rates(string $countryCode): Rates
    {
        // ...
    }
}

final class VatRateProviderUsingTaxToolsIO
    implements VatRateProvider
{
    private TaxTools $taxTools;

    public function __construct(TaxTools $taxTools)
    {
        $this->taxTools = $taxTools;
    }

    public function vatRateForSellingEbooksInCountry(
        string $countryCode
    )
    {
        return $this->taxTools->rates($countryCode);
    }
}
```

```

    ): VatRate {
        return $this->taxTools->rates($countryCode)
            ->rateFor('ebooks');
    }
}

```

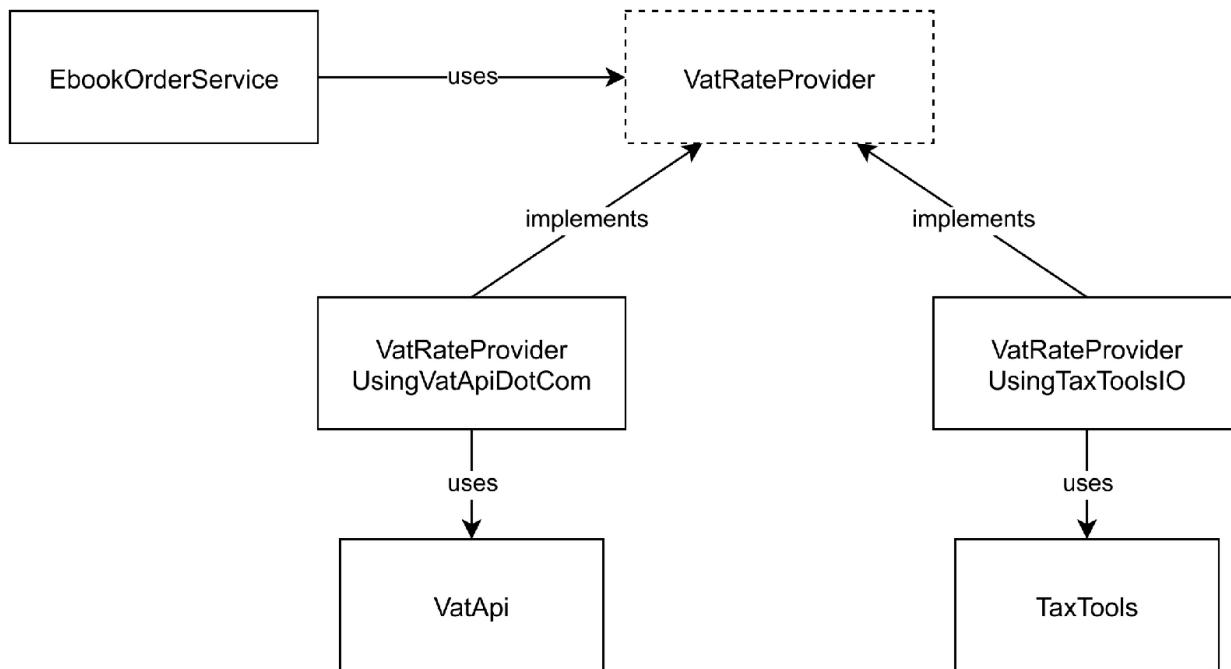


Figure 6.3: Adding an alternative implementation for `VatRateProvider`.

We don't even need to touch `EbookOrderService`. It wouldn't notice anything about the change in the background, since it depends on the interface, and the interface is still the same ([Listing 6.11](#)).

Listing 6.11: Nothing changes inside `EbookOrderService` if we switch to a different VAT rate provider.

```

final class EbookOrderService
{
    // ...
    private VatRateProvider $vatRateProvider;

    public function __construct(
        // ...

```

```

        VatRateProvider $vatRateProvider
    ) {
        // ...
        $this->vatRateProvider = $vatRateProvider;
    }

    public function create(/* ... */): OrderId
    {
        // ...

        $vatRate = $this->vatRateProvider
            ->vatRateForSellingEbooksInCountry('NL');

        // ...
    }
}

```

This may come in handy the moment we realize that the service we currently use is actually not that good. Or when it turns out it can't give us all the answers we need. Or maybe, you'll realize that calling an external service is a real danger for the responsiveness of your own application, and you want to keep a local list of VAT rates. In this sense, introducing abstractions is an *architectural* tool. Using it, you can build in flexibility for years to come. You can start with a cheap-to-implement solution, whether that be an external service that does the trick for now, or a locally stored JSON file. You can always "upgrade" when the time is there.

Think about other possible implementations

When you are introducing abstractions for your service dependencies, make sure to actually consider possible alternative implementations. Sometimes the interface you have in mind is still too much tied to the implementation you already have for it. Consider if the interface would still be useful if you switch to a different web service, or when you'd start using a database table instead of a remote service. Often little details about the underlying implementation sneak into an interface, making it a less powerful abstraction. Consider for instance the alternative `VatRateProvider` interface in Listing [6.12](#).

Listing 6.12: This interface offers a leaky abstraction.

```
interface VatRateProvider
{
    public function vatRateForCountry(
        string $countryCode,
        string $filter
    ): VatRate;
}
```

This is a more generic, possibly more reusable method than the one we proposed before. It removes the specific “e-book” aspect from the original method: `vatRateForSellingEbooksInCountry()`. But it introduces an extra parameter which ties it to our `vatapi.com`-based implementation. It’s most likely that the `$filter` parameter will have no use once we move away from `vatapi.com` and start using another API. Maybe we should even conclude that the e-book-specificness was a benefit of the interface, since it encoded the filtering aspect in the method name itself. This would allow the implementing class to take care of the actual filtering itself, without the client having to worry about it.

6.4 Testing

So far we’ve seen several examples of splitting core code and infrastructure code. We’ve also seen how doing this influences the tests you create for your code. Core code is code that doesn’t use IO (network, file system, etc.). Therefore, code can be tested in complete isolation. You don’t need any special setup. It’s enough to instantiate the object, call a method on it, and compare the outcome with what you expect. These tests will give you quick feedback while you’re developing the code. They won’t fail for weird reasons, like concurrency problems, or a remote service that is down or responds in unexpected ways. You’re only doing manipulations on data in memory, which is a *deterministic* process.

These properties of isolated tests are very desirable. You'd want to maximize the number of classes that can be tested like this. Separating core code from infrastructure code is the best way to accomplish this. Always look for that part of the logic that you can test in isolation and extract it into its own class.

We started this chapter by ruining our options for testing, because we had all those `curl_*` calls right inside the `EbookOrderService`. Extracting those calls to a separate class, the `VatApi` service, was a good idea, but still didn't give us more things to test in isolation. Only when we introduced an abstraction for providing a VAT rate, we were back at a testable `EbookOrderService`. When testing this class we can now easily provide a test double for the `VatRateProvider` interface. Listing 6.13 shows what this would look like. This prevents an actual network connection from being created while testing the behavior of `EbookOrderService`.

Listing 6.13: When testing `EbookOrderService` we can easily define a test double for `VatRateProvider`.

```
final class EbookOrderServiceTest extends TestCase
{
    /**
     * @test
     */
    public function itCreatesAnEbookOrder(): void
    {
        $vatRateProvider = $this->createMock(VatRateProvider::class);
        $vatRateProvider
            ->expects($this->any())
            ->method('vatRateForSellingEbooksInCountry')
            ->with('NL')
            ->willReturn(VatRate::fromInt(21));

        $service = new EbookOrderService(
            /* ... */
            $vatRateProvider
        );

        $service->create(/* ... */);

        // ...
    }
}
```

}

That's great, but now all the code in the `VatRateProviderUsingVatApiDotCom` and the `VatApi` classes remains untested (Figure 6.4).

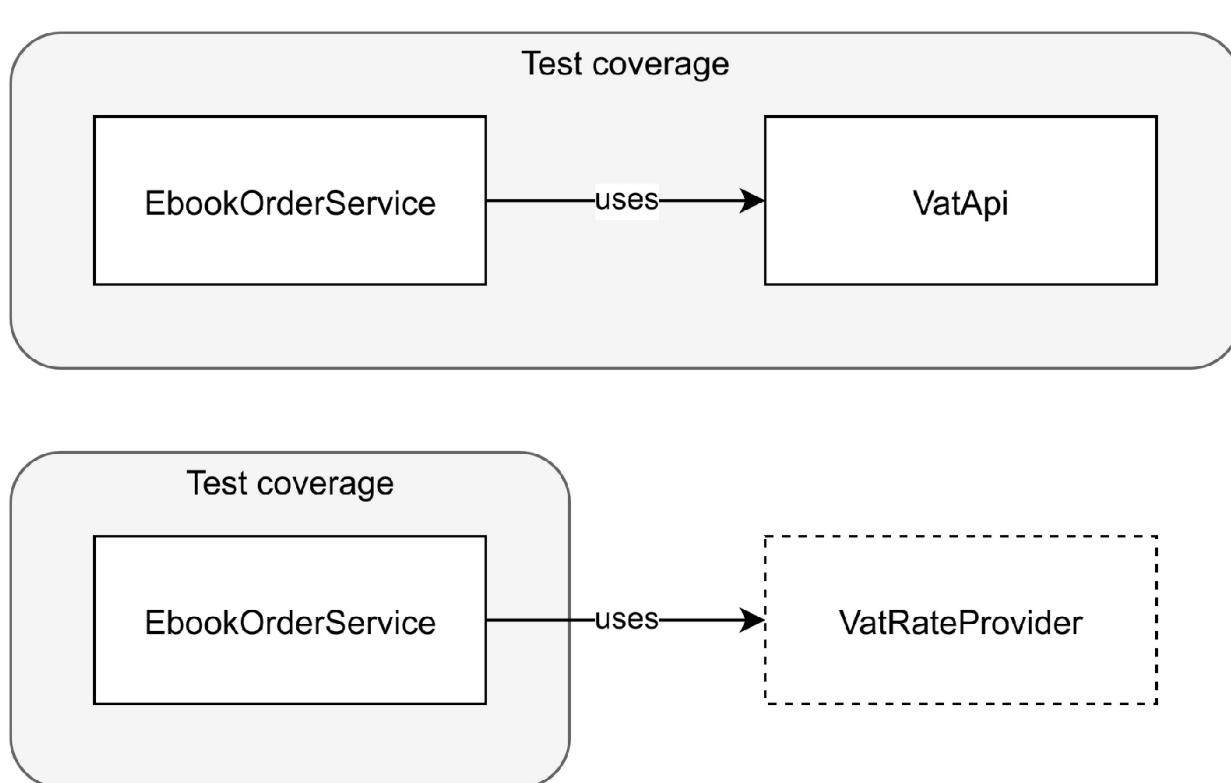


Figure 6.4: Introducing an abstraction makes `EbookOrderService` testable in isolation, but leaves `VatApi` untested.

This is a bad situation, because we can no longer be sure that the code is correct, and cooperates nicely with the `vatapi.com`'s actual API endpoint. We can't write a unit test for the `vatapi.com` integration though, because we need the internet, which means, the test uses IO and can therefore no longer be called a unit test. Let's call the kind of test we'd write for the `VatRateProviderUsingVatApiDotCom` class an *integration test*. Such an integration test can prove that we've written code that:

1. Is a correct and complete implementation of the interface we've defined for it
2. Correctly integrates with the external service

Once we can prove that, we can be sure that any client that needs a `VatRateProvider` can safely use our `VatRateProviderUsingVatApiDotCom` class (Figure [6.5](#)).

[Listing 6.14](#) shows a test that demonstrates that a `VatRateProviderUsingVatApiDotCom` object is able to provide the correct VAT rate for e-books sold in the Netherlands.

Listing 6.14: An integration test for `VatRateProviderUsingVatApiDotCom`.

```
final class VatRateProviderUsingVatApiDotComTest extends TestCase
{
    /**
     * @test
     */
    public function it_provides_the_dutch_vat_rate(): void
    {
        $provider = new VatRateProviderUsingVatApiDotCom(
            new VatApi('TEST_API_KEY')
        );

        self::assertEquals(
            VatRate::fromInt(21),
            $provider->vatRateForSellingEbooksInCountry('NL')
        );
    }
}
```

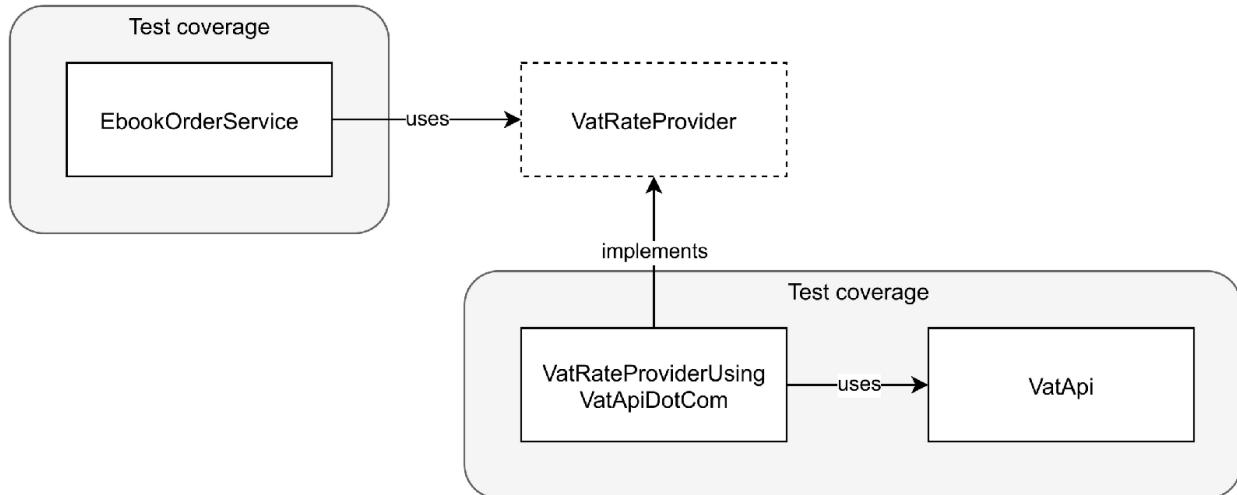


Figure 6.5: Testing both the `EbookOrderService` and the implementation of the `VatRateProvider` interface.

It still looks like a unit test because it's the same test framework that runs this test. But because of the kind of code it executes, it definitely is an integration test. I recommend marking the test as such, either by adding an annotation to the test class (e.g. `@group integration`) or by defining a dedicated test suite for integration tests (see Listing 6.15).

Listing 6.15: Define a separate test suite for integration tests in `phpunit.xml`.

```

<?xml version="1.0" encoding="UTF-8"?>
<phpunit>
    <testsuites>
        <testsuite name="Unit tests">
            <directory>./test/unit</directory>
        </testsuite>
        <testsuite name="Integration tests">
            <directory>./test/integration</directory>
        </testsuite>
    </testsuites>
</phpunit>
    
```

The test for `VatRateProviderUsingVatApiDotCom` shows that it functions as a proper implementation of the `VatProvider` interface. It will return a call to

that function with the correct type of answer. There may be other aspects you'd like to verify in an integration test like this one. Maybe the interface defines a specific type of exception that will be thrown in case of failure. In that case, you should add an extra test which simulates that failure, and verifies that the expected exception will indeed be thrown.

As you may have noticed, we don't have a dedicated integration test for the `VatApi` class itself. We're only testing its behavior as a dependency of the `VatRateProviderUsingVatApiDotCom` class. Whether or not you should write an integration test for `VatApi` alone, depends on:

- Whether or not the `VatApi` class will be used by other clients too.
- Whether or not the `VatApi` class has other methods or execution paths that are troublesome to test only indirectly through testing `VatRateProviderUsingVatApiDotCom`.

These things are usually interrelated: if `VatApi` is more like a library class, used by different clients, it will have other behaviors that should not only be tested through `VatRateProviderUsingVatApiDotCom`. In our case, the `VatApi` class is likely to have special logic for dealing with connection failures, API response validation, etc. It's probably inconvenient to test the edge cases and more exotic execution branches through another class than `VatApi` itself. Which is why it makes sense to write a separate integration test for `VatApi` alone. In that case, the integration test for `VatRateProviderUsingVatApiDotCom` can be quite small, with just one or two test methods, and the integration test for `VatApi` can be more elaborate. See Listing 6.16 for an example.

Listing 6.16: An integration test for the `VatApi` class.

```
final class VatApiTest extends TestCase
{
    /**
     * @test
     */
    public function it_does_a_vat_check_for_TBE_sold_in_NL(): void
    {
        $vatApi = new VatApi('TEST_API_KEY');
```

```

        $result = $vatApi->vatRateCheck(
            RateType::tbe(),
            'NL',
            null
        );

        self::assertEquals(
            VatRate::fromInt(21),
            $result->rate('telecom')
        );
    }
}

```

The integration test for `VatApi` is more low-level, since it's closer to the actual API call itself. This allows you to test more and different variations. For instance, here you can try different arguments for `$countryCode`, or `$rateType`. You can test the mechanism for the “fallback rate type”. And you can make `VatApi` more robust, by preventing invalid arguments, like an unknown country code, or a bad API key.

“Won’t these tests be slow, and brittle?”

The problem with integration tests is that they fail for reasons that are beyond your control. The network may not be reliable, the `vatapi.com` servers will be offline sometimes, someone may one day deploy a broken version of the API, and so on. Once again this proves the need for as many stable (unit) tests as possible. But you still have to have these integration tests that sometimes fail. There are different options, some of which may be to replace the actual external service with a look-alike service that runs locally and more reliably. Another option would be to automatically retry these tests, or allow them to fail without failing the entire build. You can accomplish this by tagging or grouping them, or keeping them in a separate test suite. Make sure the code itself gives detailed reports about the failures, so you won’t accidentally ignore important issues, like when an external service has introduced a breaking change in their API. You will find more suggestions for improving unreliable tests in “xUnit Test Patterns:

Refactoring Test Code” by Gerard Meszaros, Addison-Wesley Professional (2007).

The advice to extract as much unit-testable code as possible holds here too. For instance, take the `VatApi::vatRateCheck()` method. Its return value is a `VatRateCheckResult` object. It contains all the logic for interpreting the response data and returning the requested values from it (see Listing 6.17)

Listing 6.17: `VatRateCheckResult` again.

```
final class VatRateCheckResult
{
    private array $responseData;

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

    public function rate(string $fallbackType): int
    {
        if ($this->responseData['filter_match']) {
            return (int)$this->responseData['rate'];
        }

        return (int)$this->responseData['rates']
            [$fallbackType]['rate'];
    }
}
```

The different execution paths in `rate()` shouldn’t be tested through an integration test. You don’t need a network connection to verify that this logic works. Instead, a unit test would suffice (see Listing 6.18).

Listing 6.18: A unit test for `VatRateCheckResult`.

```
final class VatRateCheckResultTest extends TestCase
```

```

{
    /**
     * @test
     */
    public function it_uses_the_filter_rate(): void
    {
        $result = new VatRateCheckResult(
            [
                'filter_match' => true,
                'rate' => 21.0,
                'rates' => [
                    'telecom' => [
                        'rate' => 6.0
                    ]
                ]
            ]
        );
        self::assertEquals(
            21.0,
            $result->rate('telecom')
        );
    }

    /**
     * @test
     */
    public function it_uses_the_fallback(): void
    {
        $result = new VatRateCheckResult(
            [
                'filter_match' => false,
                'rates' => [
                    'telecom' => [
                        'rate' => 6.0
                    ]
                ]
            ]
        );
        self::assertEquals(
            6.0,
            $result->rate('telecom')
        );
    }
}

```

This, again, introduces a more fine-grained test that allows you to test many variations of constructor and method arguments. The more coarse-grained tests like the test for `VatApi` itself will only have to test one or two of the most common execution paths. The details will be verified in a unit test. Make sure to always look for opportunities like this, where you can extract part of the infrastructural code and write unit tests for it.

“What if `VatApi` were indeed part of the official API SDK, or some other library?”

Do you still need to write your own integration tests for it? To find the answer, you need to look at their code. Is it well written? Is it well tested? Then, of course, you don’t need to write your own tests for their classes. If you rely on certain special cases that are not covered by their tests, or if they don’t have any tests, write them yourself. That way, you can verify your assumptions about their code. And whenever they change their code, you will know if they didn’t break things *for you*.

So you may or may not have to write your own integration tests for API client code like `VatApi`. But for classes that are implementations of your own abstractions, you should always provide integration tests. Whether or not such an implementation uses a third-party dependency, you should always demonstrate that your implementation is a proper implementation for the interface you defined.

6.5 Summary

In this chapter we looked at integrating our use case with an external service for determining VAT rates. Pasting the integration code directly into the `EbookOrderService` made it work, but left us with a bad design. We moved the low-level communication details to a separate service class. This left us with an API client, or *Façade*, called `VatApi`, which could be injected as a

dependency, but could not be replaced with anything else. Directly depending on a class endangers the future of the service, but also degrades its testability. We introduced a proper abstraction to overcome these problems, leaving us with a `VatRateProvider` interface and a standard implementation, which uses the `VatApi` client that we already had. The testing situation is much better, since clients of the `VatRateProvider` can now easily set up a test double for it. This means that the `EbookOrderService` is once more testable in isolation. Any implementation of the interface still needs its own *integration test*. As part of running such a test, the implementation makes an actual call to the remote service to verify that its assumptions about the external service are correct, and that the class is a correct implementation of the interface.

Exercises

1. What is the right way to abstract calls to external services?^{[63](#)}
 1. Use an HTTP client abstraction instead of `curl_*` functions.
 2. Introduce an abstract class that you can depend on. The subclass can do the real work.
 3. Introduce an interface and an implementation which makes the actual call.
2. What's a good test to find out if you have achieved the right level of abstraction for a service dependency?^{[64](#)}
 1. It's a good abstraction if you can create a test double for it.
 2. It's a good abstraction if it is still useful when the implementation details change radically.
3. If an object communicates with an external service, what type of test should you write for it?^{[65](#)}
 1. A unit test
 2. An integration test

7 Time and randomness

This chapter covers:

- Providing current time and random data as method arguments
 - Introducing abstractions for creating current time and random data
 - Setting up a central place for establishing the current time
-

So far we've been looking at obvious ways in which an application reaches out to something in the outside world. Connecting to a database, the file system, or some external web service; the code that does all this work is without a doubt infrastructure code. But there are some more subtle ways of "reaching out" that we'll cover in this chapter: retrieving the current time, and producing a random value.

Take a look at Listing 7.1. Is this core code?

Listing 7.1: The Order class.

```
final class Order
{
    private Uuid $id;
    private DateTimeImmutable $orderDate;

    private function __construct()
    {
    }

    public static function create(): self
    {
        $order = new self();

        $order->id = Uuid::uuid4();
        $order->orderDate = new DateTimeImmutable('now');

        $order->recordThat(
            new OrderWasCreated($order->id, $order->orderDate)
        );
    }
}
```

```

        return $order;
    }

    // ...
}

```

The code is part of an entity, which is supposed to be core code. However, this can't be considered core code. When you instantiate an `Order` by using its named constructor `create()`, it will create two more objects: a `Uuid` object and a `DateTimeImmutable` object. Both objects will *reach out to a system device* during their construction phase. `Uuid::uuid4()` talks to the system's random device to retrieve random bytes for creating a unique identifier. `new DateTimeImmutable('now')` will ask the system's clock what the current date and time is. So running this code means that these external dependencies need to be available, which makes this infrastructure code (Figure 7.1).

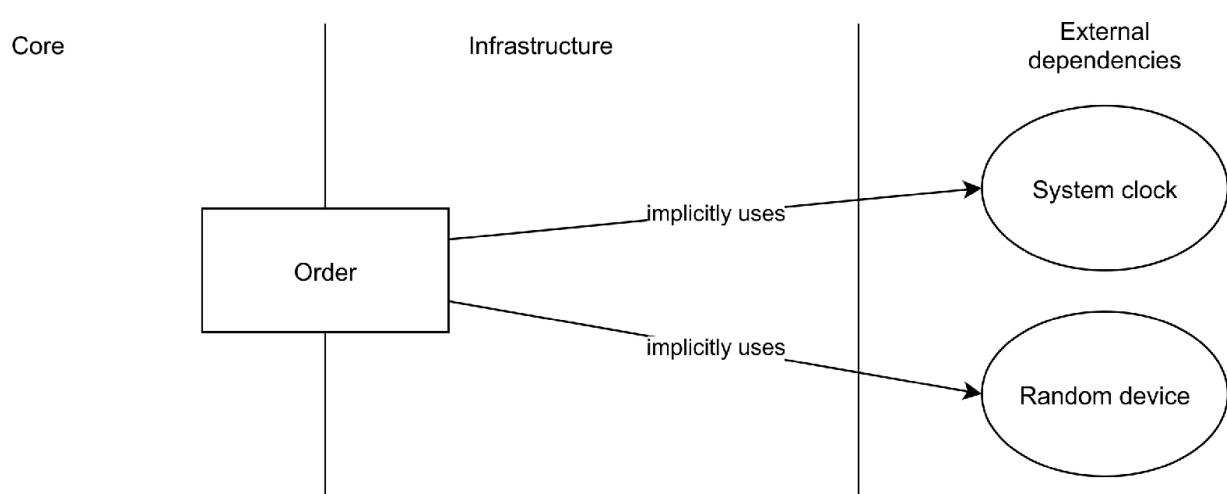


Figure 7.1: `order` and its implicit external dependencies.

When you write a unit test for `Order`, like the one in Listing 14.1, you'll experience some difficulties caused by the mix of core and infrastructure code.

Listing 7.2: A test for `Order`.

```

final class OrderTest extends TestCase
{
    /**
     * @test
     */
    public function it_can_be_created(): void
    {
        $order = Order::create();

        self::assertEquals(
            [
                new OrderWasCreated(
                    Uuid::uuid4(),
                    new DateTimeImmutable('now')
                )
            ],
            $order->releaseEvents();
        );
    }
}

```

The problem is that this test will never pass. When `Order::create()` instantiates a `Uuid` and a `DateTimeImmutable` instance, it will definitely not be equal to the one we create in the test method. The data that is involved in this unit test will be different every time you run the test. That's very inconvenient and makes testing very hard. We want things to be deterministic, predictable.

Yes, we could ignore the variation, and only verify that the array of events contains an *instance* of `OrderWasCreated`. That would make the test pass, but it would still leave us with an unpredictable entity. Also, since `Order::create()` uses IO, it shouldn't really be considered a unit test. It still relies on external dependencies to run, although they are used implicitly, hidden inside the call to `Uuid::uuid()` and somewhere in `DateTimeImmutable`'s constructor. This means that the test we just wrote is actually an integration test. An integration test still tests a single unit of code, but since it uses IO, it can show that your code integrates well with third-party or standard libraries and is able to connect to the relevant system devices. For domain objects we'd want unit tests, which are fast,

deterministic, don't rely on external dependencies, and don't require a special context to run.

How can we turn this test back into a unit test? By making `Order` a “pure” object again. We have to make sure that its behavior doesn't depend on anything else than its constructor arguments, its own implementation, and the method arguments provided to it. We should make the object no longer use the *actual* current time, or a *truly* random number. Instead, we should allow clients to *pass* the current time or random data to it as method arguments.

7.1 Passing current time and random data as method arguments

The example in Listing 7.3 shows the result of passing randomness and current time as constructor arguments, instead of letting `order` retrieve this information itself.

Listing 7.3: Time and random data are passed to `order` as constructor arguments.

```
final class Order
{
    private Uuid $id;
    private DateTimeImmutable $orderDate;

    private function __construct()
    {
    }

    public static function create(
        Uuid $id,
        DateTimeImmutable $orderDate
    ): self {
        $order = new self();

        $order->id = $id;
        $order->orderDate = $orderDate;

        $order->recordThat(
            new OrderWasCreated($order->id, $order->orderDate)
        );
    }
}
```

```

    );
    return $order;
}
// ...
}

```

This change allows us once again to write an actual unit test (see Listing [7.4](#)).

Listing 7.4: The test itself provides the current time and random data.

```

final class OrderTest extends TestCase
{
    /**
     * @test
     */
    public function it_can_be_created(): void
    {
        $id = Uuid::uuid4();
        $orderDate = new DateTimeImmutable('now');

        $order = Order::create($id, $orderDate);

        self::assertEquals(
            [
                new OrderWasCreated(
                    $id,
                    $orderDate
                )
            ],
            $order->releaseEvents()
        );
    }
}

```

The situation has improved a lot: taking control over the creation of these values will be helpful in making the subject-under-test (SUT) more predictable. `Order` itself no longer relies on the actual current time or actually random data. Figure [7.2](#) shows the new dependency relations.

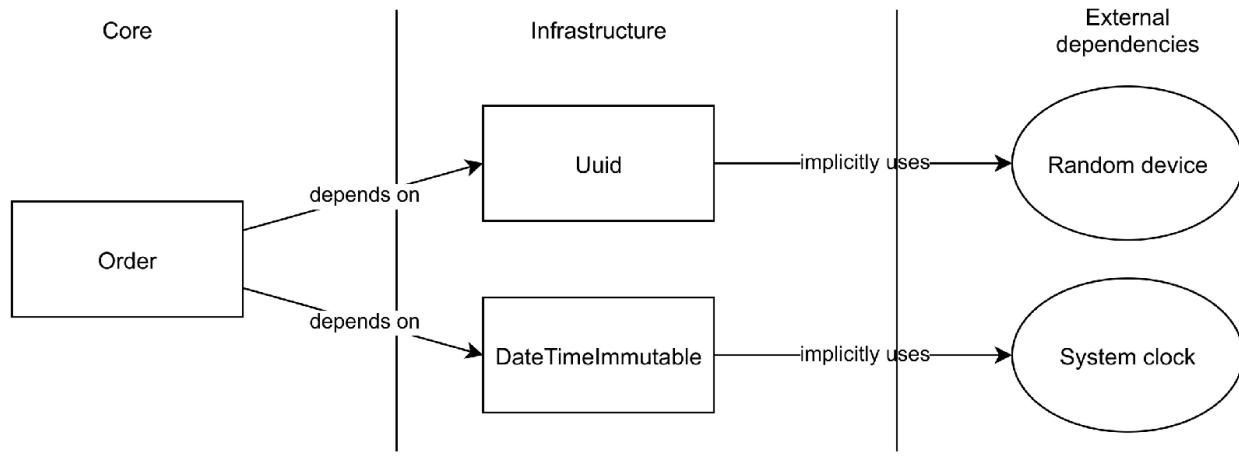


Figure 7.2: Order explicitly depends on Uuid and DateTimeImmutable, which still have implicit external dependencies.

7.2 Introducing factories

There are some remaining design issues with this code. One issue is that producing the current time or a random value are actually *service* responsibilities. Even though `Uuid` and `DateTimeImmutable` look like value objects, they really aren't. They could never come up with random data or the current time on their own. Imagine that you'd have to write the constructor of `DateTimeImmutable`, or the named constructor `Uuid::uuid4()`. What should go in there? At some point you'd have to "reach out" and ask the host system for some input. Your code can never come up with true randomness, or with the actual current time. This doesn't correspond with the nature of a value object, which is supposed to be a pure object, whose behavior is determined solely by the input provided it, and its own logic.

When an object talks to external systems this object should be a service so it can communicate this clearly. And when we create a service class for something that talks to the outside world, we should always provide an interface for it. This allows us to properly inject the service as a constructor dependency, instead of using it ad hoc whenever we need the current time or a unique ID.

For the creation of a UUID, an abstraction could be an interface named `UuidFactory`, like the one in Listing 7.5.

Listing 7.5: The abstract `UuidFactory` and a standard implementation.

```
use Ramsey\Uuid;

interface UuidFactory
{
    public function create(): Uuid;
}

final class UuidFactoryUsingRamseyUuid
    implements UuidFactory
{
    public function create(): Uuid
    {
        return Uuid::uuid4();
    }
}
```

For the creation of a `DateTimeImmutable` instance we might consider introducing a `TimeFactory`, but this is commonly known as a `Clock` anyway. Listing 7.6 shows an example of such an abstraction.

Listing 7.6: The abstract `Clock` and a standard implementation.

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

final class ClockUsingSystemClock
    implements Clock
{
    public function currentTime(): DateTimeImmutable
    {
        return new DateTimeImmutable('now');
    }
}
```

We're still using the original objects provided by the `ramsey/uuid`⁶⁶ library and the PHP standard library. But now we have interfaces that allow us to use those objects without relying on their service responsibilities. Whenever we need a `DateTimeImmutable` instance we use the `Clock`. When we need a `Uuid`, we use the `UuidFactory`. We never have to instantiate these objects ourselves and when we use the new service abstractions it will be totally clear that we expect their implementations to reach out to external systems. Figure 7.3 shows what the situation looks like now.

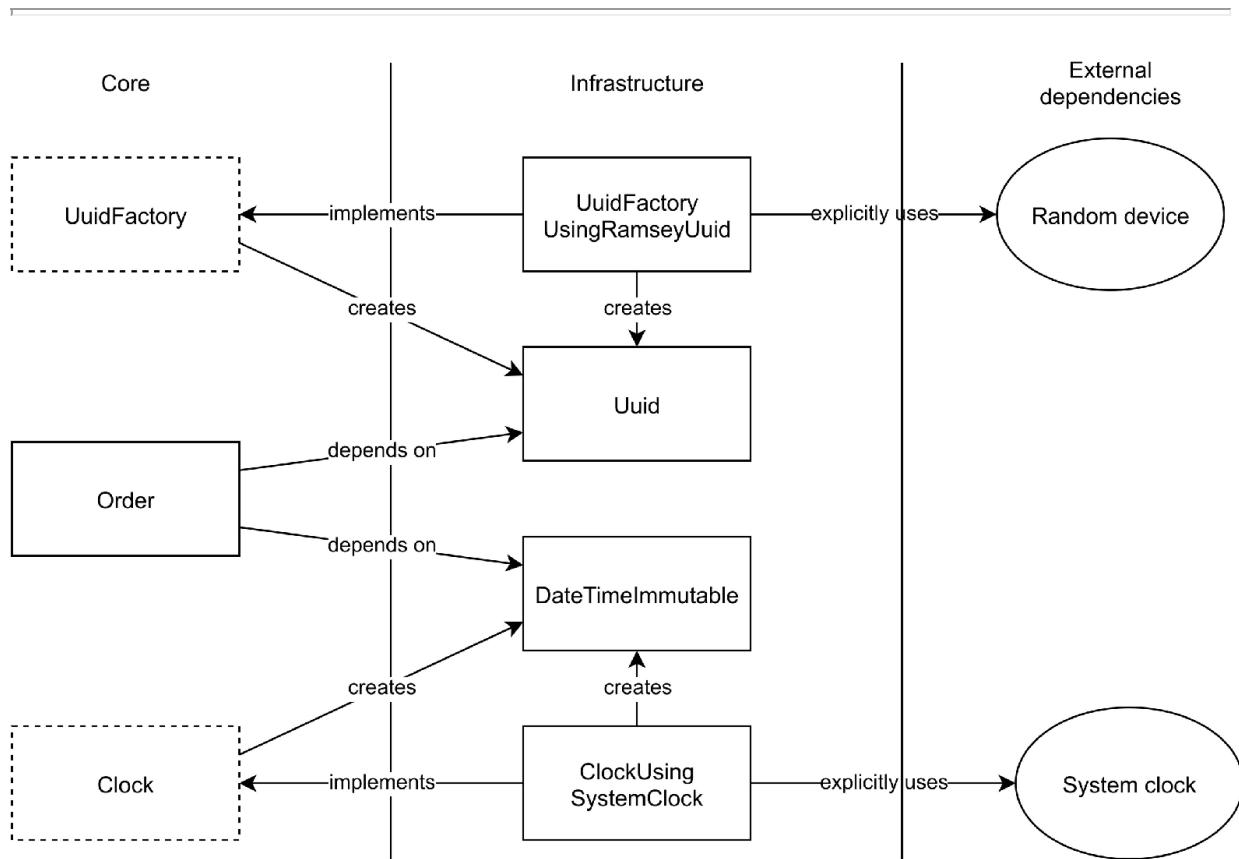


Figure 7.3: The dependency diagram after introducing service abstractions.

In our test we could now use the new services (see Listing 7.7).

Listing 7.7: Using the new services inside a test.

```
final class OrderTest extends TestCase
{
```

```

/**
 * @test
 */
public function it_can_be_created(): void
{
    $id = (new UuidFactoryUsingRamseyUuid())->create();
    $orderDate = (new ClockUsingSystemClock())
->currentTime();

    $order = Order::create($id, $orderDate);

    self::assertEquals(
        [
            new OrderWasCreated(
                $id,
                $orderDate
            )
        ],
        $order->releaseEvents();
    );
}
}

```

Although a bit more verbose, this is definitely a more honest way of showing where the current time and the unique ID come from.

7.3 Introducing value objects

Instead of using the services while testing, couldn't we just use hard-coded values, like in Listing 7.8? We could generate a UUID once and reuse it for every test run. The same for the timestamp: we should be able to set it to a fixed moment in time. And we don't even need some ugly hack like overriding PHP's built-in functions or classes. We can just provide a string argument to `DateTimeImmutable`'s constructor (Listing 7.8).

Listing 7.8: Using hard-coded values for `Uuid` and `DateTimeImmutable`

```

/**
 * @test
 */
public function it_can_be_created(): void
{
}
}

```

```

{
    $id = Uuid::fromString('77d69702-e3b4-4af5-b40a-c9981d483880
');
    $orderDate = new DateTimeImmutable('2020-06-03 09:53');

    // ...
}

```

Yes, I think this is the way forward. The advantage is that now the test itself doesn't use IO anymore. This gives us a small performance improvement, which in the long run will keep your test suite fast. We also gain full control over the state of `Order` by doing this. Its state will always be the same, and therefore the object will always behave in the same way.

We treat the `Uuid` and `DateTimeImmutable` classes as value objects instead of factory services. Their behavior will now be determined by the input arguments only. Or, maybe not... It turns out that `DateTimeImmutable` has some interesting behavior when you instantiate it with a string argument. If you don't provide some date parts it will copy the respective values from the actual current time. In our case, we don't provide seconds and microseconds, so the `DateTimeImmutable` instance will have the number of seconds and the number of microseconds from the current time. As a consequence, we still don't have full control over all the state involved in running our test. And that means the test isn't fully deterministic, and it can't be called a unit test.

Besides the fact that these objects aren't real value objects, and that they can behave in undeterministic ways that we might not know about, there's another design issue. Third-party classes like `DateTimeImmutable` and `Uuid` are designed to be useful in many projects and many different situations. They therefore have an API that is not designed with your specific use case in mind, but with *any* use case in mind. For instance, `DateTimeImmutable` isn't just a way to represent a timestamp as an object, it also has methods like `sub()`, `setDate()`, `getTimezone()`, etc. These methods may be totally irrelevant in your use case, but still, by introducing the class to your domain model, its behavior now becomes part of your domain. In some cases the behavior might actually get you in trouble, because the ideas of the author don't match with your ideas or expectations about the same concept. So a good rule of thumb is that the classes you use as dependencies (constructor

arguments, method arguments and return types) in your domain model should also be designed by you. It's a good thing to make every aspect of your domain model's design explicit, and suitable for your particular use case.

In the case of our use of `Uuid` and `DateTimeImmutable`, the solution will be to define our own value objects. This solves two issues at the same time:

1. We'll have complete control over the API of our domain objects.
2. We won't accidentally use IO in our domain objects, nor in the unit tests we create for them.

The two value objects we need should represent “the ID of an order” and “the creation date of an order”. Let's start with the order ID and introduce an `OrderId` value object that can represent this ID. Behind the scenes we'll still use a UUID. We'll create an `OrderId` instance from a string that contains a UUID, instead of from an already instantiated `Ramsey\Uuid` object. Listing 7.9 shows a simple `OrderId` value object that should replace existing usages of `Uuid` is a parameter type.

Listing 7.9: The `OrderId` value object class.

```
final class OrderId
{
    private string $id;

    private function __construct(string $id)
    {
        Assertion::uuid($id);
        $this->id = $id;
    }

    public static function fromString(string $id): self
    {
        return new self($id);
    }
}

final class Order
{
    private OrderId $orderId;
    // ...
}
```

```

public function __construct(
    OrderId $orderId,
    // ...
) {
    $this->orderId = $orderId;
    // ...
}
}

```

Upon construction, the `OrderId` accepts a string that looks like a UUID. It verifies this by using an assertion from the `beberlei/assert` package^{[67](#)}. The fact that neither `OrderId`, nor `Order` itself, ever creates a `Uuid` instance from scratch, allows us to stop relying on IO in the test class. It would be sufficient to generate an actual UUID once (e.g. using the command line tool `uuidgen`, or using a simple PHP script which shows the result of calling `Uuid::uuid4()`). We could then copy the generated UUID and use it in our unit test, as shown in Listing [7.10](#).

Listing 7.10: Providing a hard-coded string to `OrderId::fromString()`.

```

/**
 * @test
 */
public function it_can_be_created(): void
{
    $order = new Order(
        OrderId::fromString('77d69702-e3b4-4af5-b40a-c9981d48388
0'),
        // ...
    );
    // ...
}

```

`OrderId` is a simple class which only offers methods that we actually need in our domain code, as opposed to the `Uuid` class. For now, it's only a wrapper for the UUID string.

For the creation date we can do the same thing: accept a date string in a given format. Behind the scenes we can still rely on `DateTimeImmutable` to determine if the given date string represents an actual date and doesn't just look like one (e.g. 2019-02-29). Listing 7.11 shows how `Date` uses `DateTimeImmutable` in its constructor.

Listing 7.11: `Date` uses `DateTimeImmutable` to validate its constructor argument.

```
final class Date
{
    private const DATE_FORMAT = 'Y-m-d';
    private string $date;

    private function __construct(string $date)
    {
        if (! DateTimeImmutable::createFromFormat(
            self::DATE_FORMAT,
            $date
        )) {
            throw new InvalidArgumentException(
                sprintf(
                    'Invalid date provided: %s. Expected format:
%s',
                    $date,
                    self::DATE_FORMAT
                )
            );
        }

        $this->date = $date;
    }

    public static function fromString(string $date): self
    {
        return new self($date);
    }
}

final class Order
{
    private Date $orderDate;

    public function __construct(
        // ...
    )
}
```

```

        Date $orderDate,
    // ...
) {
// ...
$this->orderDate = $orderDate;
}
}

```

To validate the provided date string we could've also used `Assertion::date(self::FORMAT, $date)`. However, I wanted to point out in this example that you can still use supporting objects from third-party libraries to do the heavy lifting for you. It's perfectly fine to rely on other people's classes inside your value object, as long as they don't reintroduce some of the problems we've been trying to evade:

1. Value objects shouldn't contain infrastructure code. In other words:
2. Value objects should have no service responsibilities.
3. Value objects should offer no behavior that hasn't been explicitly enabled and designed for your use case.

This means you can use value objects from other libraries if they offer the (more or less) exact behaviors you need. If they don't, then you should adapt them by wrapping them inside your own value objects. Just a quick note: when using third-party tools like `DateTimeImmutable` you still have to check your own assumptions. An elaborate unit test would be useful for that. As an example, I was quite surprised to find out that instantiating an obviously meaningless date like `2019-02-29` doesn't produce an exception. `DateTimeImmutable`'s constructor simply turns it into `2019-03-01...`

Date/time quirks aside, introducing your own value objects for dates and times is a way of gaining design ownership over the types of objects that are involved, and the APIs they offer to their clients.^{[68](#)} These value objects will attract useful behaviors that are specific to the domain of your application.^{[69](#)}

7.4 Improving the factories

The result of introducing value objects with named constructors that accept a string and won't need IO is that the unit test and the subject-under-test both are no longer using IO, as shown in Listing [7.12](#).

Listing 7.12: Now that we have custom value objects, `OrderTest` no longer needs IO.

```
/**  
 * @test  
 */  
public function it_can_be_created(): void  
{  
    $order = new Order(  
        OrderId::fromString('77d69702-e3b4-4af5-b40a-c9981d48388  
0'),  
        Date::fromString('2019-07-09')  
    );  
  
    // ...  
}
```

However, when the `Order` entity is going to be used in production, the application service that creates it needs to provide a truly unique ID, and the actual current time. Since, again, determining the current time or producing randomness are service responsibilities, we need to have some services that can create `OrderId` and `Date` objects for the application service. In Chapter 2 we already saw how this can be solved for the `OrderId`: by adding a `nextIdentity()` method to the `OrderRepository` interface. Listing 7.13 shows once more how it's done.

Listing 7.13: `OrderRepository` can produce new `OrderId` instances.

```
interface OrderRepository  
{  
    // ...  
  
    public function nextIdentity(): OrderId;  
}  
  
final class OrderRepositoryUsingSql  
    implements OrderRepository  
{  
    private UuidFactory $uuidFactory;  
  
    public function __construct(UuidFactory $uuidFactory)
```

```

{
    $this->uuidFactory = $uuidFactory;
}

public function nextIdentity(): OrderId
{
    return OrderId::fromString(
        $this->uuidFactory->create()->toString()
    );
}
}

```

This offers a clean separation between core and infrastructure code. The infrastructure code is the code that creates a fresh UUID based on what data the system's random device provides. The core code is the `OrderId` class itself, and the interface from which you can retrieve a new `OrderId` instance.

For retrieving a `Date` instance we could do a similar thing. We'd define an interface, with a method that creates a `Date` instance for the caller. We could call it a `DateFactory` but maybe a `calendar` with a `today()` method would be more appropriate. This method returns a `Date` instance correctly representing today's actual date. The standard implementation of this interface can use the `Clock` abstraction we defined earlier to get the current time and create a `Date` instance based on it (see Listing 7.14).

Listing 7.14: The `Calendar` interface and its standard implementation.

```

interface Calendar
{
    public function today(): Date;
}

final class CalendarUsesClock
    implements Calendar
{
    private Clock $clock;

    public function __construct(Clock $clock)
    {
        $this->clock = $clock;
    }
}

```

```

public function today(): Date
{
    return Date::fromString(
        $this->clock->currentTime()
            ->format('Y-m-d')
    );
}
}

```

We should definitely introduce a method `Date::fromDateTimeImmutable()` to prevent all this type juggling, and to keep knowledge of `DateTimeImmutable` and the supported date string format inside `Date` itself. Listing 7.15 shows how you could implement this alternative constructor for `Date` objects.

Listing 7.15: Instantiating `Date` based on an already existing `DateTimeImmutable` instance.

```

final class Date
{
    private const DATE_FORMAT = 'Y-m-d';
    private string $date;

    private function __construct(string $date)
    {
        // ...
    }

    public static function fromDateTimeImmutable(
        DateTimeImmutable $dateTimeImmutable
    ): self {
        return new self(
            $dateTimeImmutable->format(self::DATE_FORMAT)
        );
    }

    // ...
}

```

Adding a named constructor like this exposes some of the object's internal implementation details. But the benefits of allowing this to happen (easier

conversion between different types of values) may be larger than the costs (keeping everything inside the object).

Listing 7.16 shows how the `Calender` and `OrderRepository` can now be used inside the `EbookOrderService`.

Listing 7.16: Using `Calendar` and `OrderRepository`.

```
final class EbookOrderService
{
    private Calendar $calendar;
    private OrderRepository $orderRepository;
    // ...

    public function __construct(
        Calendar $calendar,
        OrderRepository $orderRepository,
        // ...
    ) {
        $this->calendar = $calendar;
        $this->orderRepository = $orderRepository;
        // ...
    }

    public function create(/* ... */): OrderId
    {
        $order = Order::create(
            $this->orderRepository->nextIdentity(),
            $this->calendar->currentDate(),
            // ...
        );
        // ...

        return $order->orderId();
    }
}
```

7.5 Manipulating the current time

Now that we have the `Clock` interface and a standard implementation, we can rewrite any instantiation of a `DateTimeImmutable` in our application as a

call to `Clock::currentTime()`. This gives us a single place to determine the current time, which will be very helpful when we start writing acceptance tests later on (see Chapter [14](#)). Listing [7.17](#) shows what we can do in a test to influence the current time as our application will perceive it.

Listing 7.17: In a test, you can easily configure the “current” time.

```
final class CreateOrderTest extends TestCase
{
    public function itCreatesAnOrder(): void
    {
        $clock = new class implements Clock {
            public function currentTime(): DateTimeImmutable
            {
                return new DateTimeImmutable('2019-07-09');
            }
        };

        $service = new EbookOrderService(
            new CalendarUsesClock($clock),
            // ...
        );

        $service->create(/* ... */);

        // ...
    }
}
```

It will likely be more useful if instead of an anonymous class we’d define an actual class (e.g. `FakeClock`) which facilitates the process of providing the “current time” that we want. You may even want to make the `FakeClock` mutable, allowing time to “pass” while running a test (see Listing [7.18](#)).

Listing 7.18: A `FakeClock` and how to use it in a test.

```
final class FakeClock implements Clock
{
    private ?DateTimeImmutable $currentTime;

    public function setCurrentTime(

```

```

        DateTimeImmutable $currentTime
): void {
    $this->currentTime = $currentTime;
}

public function currentTime(): DateTimeImmutable
{
    if ($this->currentTime === null) {
        return new DateTimeImmutable('now');
    }

    return $this->currentTime;
}
}

$clock = new FakeClock();
$currentTime = new DateTimeImmutable('2020-06-03 09:53');
$clock->setCurrentTime($currentTime);

// Do something

$clock->setCurrentTime($currentTime->modify('+1 day'));

// Do something else, the code will think it's one day later mpw

```

Figure 7.4 is a diagram of the current situation. We have several “domain-specific” abstractions that we can now use. All the important concepts are represented as core code, and infrastructure code is really only there to support them.

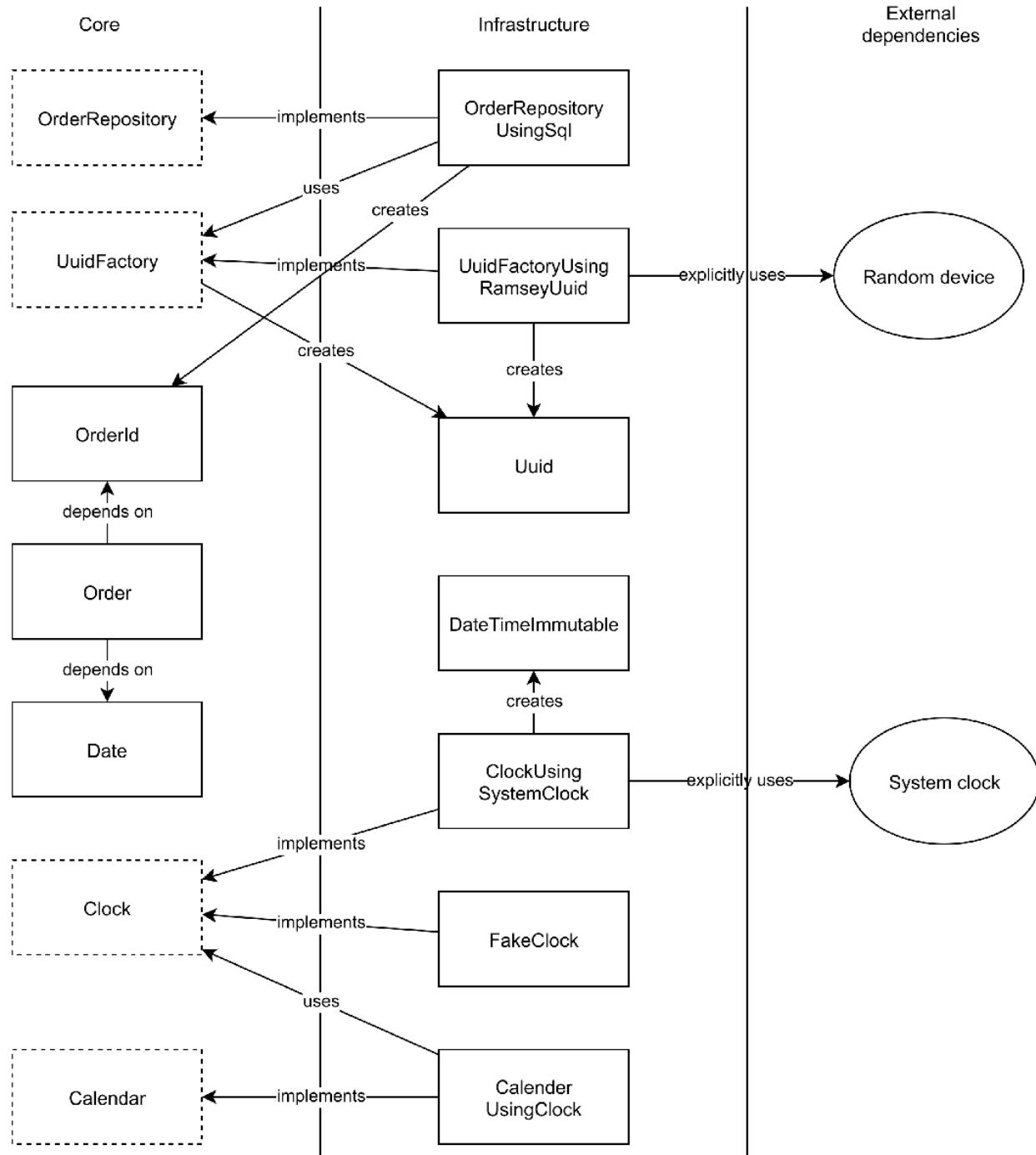


Figure 7.4: The dependency diagram for the current situation.

Even though our intentions were good, we now have a rather large number of additional elements. We started with three classes, now we have 9 extra classes and interfaces. As you know, introducing abstractions usually results

in two extra elements (an interface and a standard implementation), or three when you introduce a new return type too. So abstractions have a cost, but as we discussed before: they have many benefits. Both for the long-term maintainability of your application as for today’s testability of the code. However, it’s important that you keep trying to limit the number of elements or parts in your application.

Looking at Figure [7.4](#) an important clue that we can remove a part is that `OrderRepositoryUsingSql` uses the `UuidFactory` interface to get a `Uuid` instance. We introduced this interface because creating a UUID is a service responsibility that requires external dependencies to work. We didn’t want to create a UUID in core code. But since `OrderRepositoryUsingSql` is already infrastructure code it doesn’t have to jump back to core code by depending on `UuidFactory`. It can just use the `Uuid::uuid4()` method directly (see Listing [7.19](#)).

Listing 7.19: Skipping the `UuidFactory`.

```
final class OrderRepositoryUsingSql
    implements OrderRepository
{
    public function nextIdentity(): OrderId
    {
        return OrderId::fromString(
            Uuid::uuid4()->toString()
        );
    }
}
```

This allows us to remove both the `UuidFactory` interface and the `UuidFactoryUsingRamseyUuid` class.

A similar reasoning applies to the `Calendar` interface, but there it’s the other way around. The only thing that `CalenderUsingClock` does is talk to the `Clock` interface and use the current time to create a new `Date` instance. Using the `Calendar` gives us “jumpy” code, where we go from application service to abstraction, to concrete class, to abstraction, to concrete class. Instead, an application service could just use the `Clock` directly, and create a `Date` from it, using its `fromDateTimeImmutable()` factory method.

Listing 7.20 shows the application service after removing Calender (and CalenderUsingSystemClock).

Listing 7.20: Using Clock and OrderRepository.

```
final class EbookOrderService
{
    private Clock $clock;
    private OrderRepository $orderRepository;
    // ...

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

    public function create(/* ... */): OrderId
    {
        $order = Order::create(
            $this->orderRepository->nextIdentity(),
            Date::fromDateTimeImmutable($this->clock->currentTim
e()),
            // ...
        );
        // ...
        return $order->orderId();
    }
}
```

To improve the language used in this code, we should rename `Date::fromDateTimeImmutable()` to `Date::fromCurrentTime()`. This would take away the focus from the data type used, and bring back focus on the story: we're making a date based on the current time, which makes a lot of sense too.

Figure 7.5 shows the elements we have left after cleaning up.

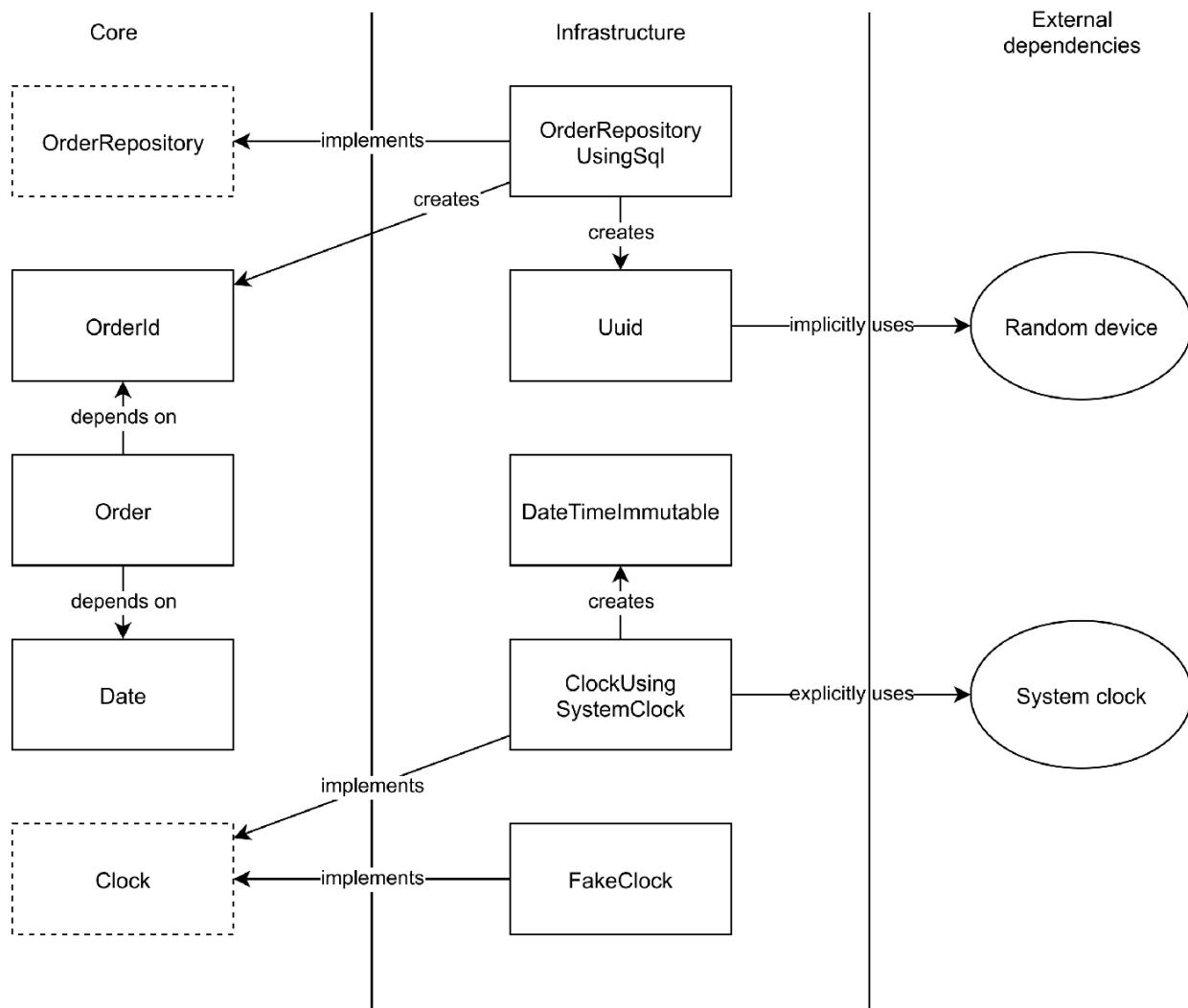


Figure 7.5: The result of cleaning up.

“Do you recommend starting with more elements than needed?”

Great question; that’s not exactly what I wanted to show in this chapter. I don’t think it should be your goal to create more elements than needed, but I know that it sometimes happens. That’s why I thought it would be useful to show what are signs of too many elements (“jumpy” code), and also how to reduce the number of elements.

7.6 Integration tests again

Testing core code is easy because core code by definition runs well in isolation. You need no special context, and no external dependencies. With infrastructure code it's a different story. They usually need special setup and external dependencies need to be actually available. As mentioned before, tests for infrastructure code are called integration tests. They show that the code works well with third-party libraries and the external dependencies that the code relies on. In Section 6.4 we saw how to test code that communicates with remote services. The system clock and the system's random device are in a sense remote services too, although we don't need an HTTP client to connect to these devices. Still, we'd want to show in a test that our application works correctly when it uses these external devices.

The thing is, we don't have a lot of code to test. We used to have a `UuidFactory-UsingRamseyUuid` but we removed it already. We still have a `ClockUsingSystemClock` class without tests. It's so very simple that I'd be tempted not to write a test for it. But what if your team has a rule that every class should have a test? Then get rid of this rule (no joke). As a fun experiment, let's find out what happens if we do write a test for `ClockUsingSystemClock`. Listing 7.21 shows a first try:

Listing 7.21: A test for `ClockUsingSystemClockTest`.

```
final class ClockUsingSystemClockTest extends TestCase
{
    /**
     * @test
     */
    public function it_returns_the_current_time(): void
    {
        $clock = new ClockUsingSystemClock();

        $actualCurrentTime = new DateTimeImmutable('now');

        self::assertEquals(
            $actualCurrentTime->format('c'),
            $clock->format('c')
        );
    }
}
```

```

        $actualCurrentTime,
        $clock->currentTime()
    );
}
}

```

Unfortunately, this test will never pass. That's because part of the timestamp created by `DateTimeImmutable` instance is the number of microseconds that have passed within the current second. And between creating an instance inside the test and creating another instance in the `ClockUsingSystemClock` class there will always be a small difference. So these two instances will never be equal. However, for the usefulness of the class it doesn't really matter what the microseconds part is. So instead of doing a full comparison, we might just "round" the timestamp to seconds, as is shown in Listing [7.22](#).

Listing 7.22: Comparing timestamps rounded to seconds

```

$clock = new ClockUsingSystemClock();

$actualCurrentTime = new DateTimeImmutable('now');

self::assertEquals(
    $actualCurrentTime->format('U'),
    $clock->currentTime()->format('U')
);

```

This test is *more likely* to pass, but might still fail when the test creates the `DateTimeImmutable` instance at the end of the current second, and the `Clock-UsingSystemClock` creates it in the next second. A common solution for these scenarios is to calculate the difference between the two timestamps and assert that the difference is smaller than a certain allowed value. This value represents the time difference in microseconds between the two instantiations of `DateTimeImmutable` that you think would be acceptable. Listing [7.23](#) shows how to do it.

Listing 7.23: Comparing the difference between timestamps.

```
final class ClockUsingSystemClockTest extends TestCase
```

```

{
    /**
     * @test
     */
    public function it_returns_the_current_time(): void
    {
        $clock = new ClockUsingSystemClock();

        $actualCurrentTime = new DateTimeImmutable('now');

        self::assertEquals(
            self::secondsWithMicroseconds($actualCurrentTime)),
            self::secondsWithMicroseconds($clock->currentTime())
        ,
            0.1
        );
    }

    private static function secondsWithMicroseconds(
        DateTimeImmutable $timestamp
    ): float {
        /*
         * We concatenate the number of seconds, a decimal separator,
         * and the number of microseconds within this second which
         * already has leading 0s. We then cast it to a float.
         */
        return (float) $timestamp->format('U.u');
    }
}

```

This should work, and might be a correct integration test, but I doubt if it's a very useful test.

7.7 Summary

We started this chapter with an entity that implicitly used the system's clock and random device to collect part of its relevant data. Instead of letting the object retrieve the information itself, we changed it to accept the information as constructor arguments. By wrapping the data inside custom value objects, we made it possible to further separate core from infrastructure code. We provided abstractions for creating instances of these value objects. Finally,

for producing value objects based on the current time, we set up a generic clock abstraction.

Exercises

1. By using certain built-in PHP functions your code would instantly become infrastructure code. For which of the following functions is this the case?⁷⁰

1. `time()`⁷¹
2. `date(string $format, int $timestamp = null)`⁷² providing only the first argument
3. `date(string $format, int $timestamp = null)` providing both arguments
4. `checkdate(int $month, int $day, int $year)`⁷³

2. An object can't provide truly random data all by itself. It would need to reach out to the system's random device. What about this value object, which uses `mt_rand()`⁷⁴. This function produces numbers from a predefined sequence. If you don't "seed" the function using `mt_srand()`⁷⁵, it will produce seemingly random numbers. If you do seed it, it will always produce the same sequence though. What do you think, should this code be considered infrastructure code too?⁷⁶

```
final class PseudoRandomNumber
{
    private int $number;

    private function __construct(int $number)
    {
        $this->number = $number;
    }

    public static function create(int $seed): self
    {
        mt_srand($seed);

        return new self(
            mt_rand()
        );
    }
}
```

```
    } ;
```

8 Validation

This chapter covers:

- Addressing validation concerns in code that separates core from infrastructure
 - Making a distinction between user-oriented validation and entity-level data protection
 - Using exceptions instead of validating up-front
 - Reducing the amount of validation code
-

I had been separating core from infrastructure code for quite some time. Yet, there was one thing that didn't have an obvious place in this new structure: validation. I first tried to decide if validation logic is something that should survive changes in the infrastructure. Doing the same thought experiment we did earlier: if I would migrate from a web application to a CLI application, would input data still have to be validated? Well, of course. There needs to be some kind of protective mechanism to ensure that the application doesn't accept bad or inconsistent data. Also, if I change the technical solution I use for saving my data (e.g. when I switch to a document database), would I still be able to protect the consistency of this data? Imagine not being able to rely on foreign key constraints or unique indexes; then there has to be some other way in which the application core can guarantee data consistency.

Then again, if my web application would become a CLI application, I guess the user interactions would become quite different. Instead of showing an HTML form with error messages for every bad input, it would probably be sufficient to show only the first error, let the user fix the command-line arguments and try again. If instead I would rewrite my web controllers to become API endpoints, I'd accept a JSON request, and return a list of JSON-encoded errors. Except, these errors would not be translated into the user's language, but they would be machine-readable. Maybe I'd only return some documented error code and mention the name of the field that produced the validation error.

So whether the user is a machine or a human being makes a big difference in how we tell them about validation errors. It also makes a difference what technology they use to communicate with our application. Regardless of who is talking to our application in what way, the input data that we accept and store inside our application has to be validated according to the same rules (see Figure 8.1).

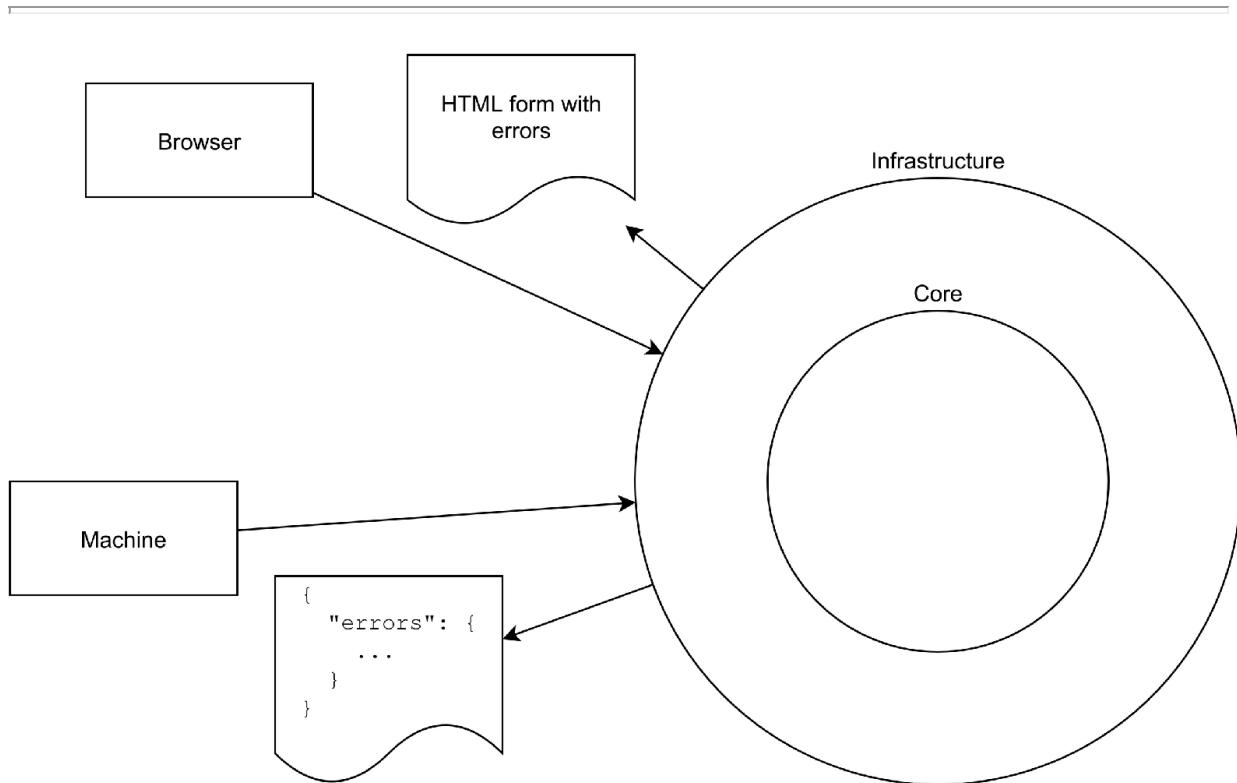


Figure 8.1: Different types of users will provide different types of input and receive different types of validation errors. However, in the core of the application the data needs to be valid, regardless of who provided it.

Whether it's a user who fills in a form in a web browser, or a machine sending a JSON POST request, when we accept an order for e-books, we always have to verify certain aspects of it:

1. The provided email address has to look like an actual email address.
2. The number of copies of an ordered e-book should be at least 1.
- 3.

The provided e-book IDs should refer to e-books that are actually available in our catalog.

This is a nice set of rules that will help me demonstrate different aspects of how a validation process should be implemented in a project that separates core from infrastructure code. In this chapter we'll go over each rule and find out how and where it should be verified.

8.1 Protecting entity state

Eventually, data provided by a user of our application will end up inside an entity. An application service will then save the entity to the database. As you probably know, you can get yourself into a lot of trouble if your database contains bad, incomplete, or inconsistent data. So we have to make sure nothing ends up in our entity that, when saved to the database, leads to a corrupt database.

However, a very common approach to validation, one that I've been using for years, does not prevent this at all. Listing 8.1 shows what this old-school style of validation looks like: we create the order object and gradually populate its fields with data from the request. We then call a validator to validate the object. If the validator does not return any validation error, everything is fine, and we can go ahead and save the order.

Listing 8.1: Validating an object after populating its fields

```
final class OrderController
{
    // ...

    public function createOrder(Request $request): Response
    {
        $order = new Order();
        $order->setEmailAddress($request->get('email'));
        $line = new Line();
        $line->setEbookId($request->get('ebook_id'));
        $line->setQuantity($request->get('quantity'));

        $errors = $this->validator->validate($order);
        if (count($errors) === 0) {
```

```
        $this->orderRepository->save($order);

        /*
         * Redirect to the "thank you" page
         */
    } else {
        /*
         * Render the form again including the validation e
rrors
        */
    }
}
```

The validator will have to reach into the object, inspect its data, and verify certain rules. This means that the validator will contain lots of knowledge about what a valid order is. It needs to validate the email address, the number of lines, the ordered quantity, whether the provided e-book ID refers to an actual e-book, etc. There are validator libraries that can make this easy for you by allowing you to use configuration instead of code. I doubt that's any better though, since it's so easy to make a mistake in validator configuration which would let bad data go through. Whether you write the code yourself or use a validator library, effectively it's the same thing. The validator would have to call getters on the `Order` object to get the data out of the object and analyze it. If anything is wrong, it will add a message to a list of validation errors. If you'd have to write this code yourself, it would look something like Listing 8.2. Note that I'm using translation keys (e.g. `'create_order.invalid_email_address'`) instead of full text messages, so I can later translate these messages into the user's language.

Listing 8.2: Validating an order by inspecting its data from the outside

```
final class OrderValidator
{
    // ...

    /**
     * @return array<string,array<string>>
     */
    public function validate(Order $order): array
```

```

{
    $errors = [];

    if (!filter_var($order->getEmail(), FILTER_VALIDATE_EMAIL)) {
        $errors['email']
    } = 'create_order.invalid_email_address';
    }

    foreach ($order->getLines() as $line) {
        try {
            $this->ebookRepository->getById($line->getEbookId());
        } catch (CouldNotFindEbook $exception) {
            $errors['ebook_id']
    } = 'create_order.could_not_find_ebook'
            . $line->getEbookId();
        }
    }

    // ...

    return $errors;
}
}

```

I don't like this code at all, because all those getters on the `Order` object are only there for the validator to get the data out again. We had already concluded in Chapter 3 that this is undesirable. A bigger problem is the fact that our `Order` object also has setters that you can use to set each value separately. The setter itself can only enforce the type of the data provided by adding types to its parameters. But the setter doesn't look any closer at the value provided; it doesn't validate the value, it doesn't throw an exception if somebody provides a bad email address. This means that bad data can end up inside the `Order` entity. And if nobody calls `OrderValidator::validate()` to validate the `Order`, its invalid data could be saved directly to the database. The relationship between these two objects, the validator and the entity, is not a very healthy one. They always have to be used together and they will evolve together, in other words: they are coupled to each other. Coupling isn't bad per se, but when there is a rule that one object (the validator) has to be used *before* the other one (the entity)

can be used, it's a problem. In this case it means the entity isn't safe to use at all, because who knows if it has been validated.

This problem can be solved quite easily. We can make the entity *self-validating*. Whenever a client tries to create a new instance of the entity, it can enforce a minimum of required data to be provided as constructor arguments. Whenever a client wants to provide some more data or modify existing data, the entity can again validate the provided arguments before assigning any new values to its properties. You could say that this isn't validation, it's more like *protection*. We don't assign a value to a property, then validate it; we protect the object from ending up in an incorrect state.

This relieves us from having to use a validator before we save our entity. Any method (including the constructor) that assigns values to properties will ensure that the resulting state of the entity is always valid. We can't make a mistake like calling these methods with missing or invalid arguments, because the entity will tell us loudly what we did wrong. Listing 8.3 shows how this works in the case of the `Order` entity. Every method that accepts data will validate this data and throw exceptions to indicate any problem as soon as possible.

Listing 8.3: The `Order` entity only accepts valid data.

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

    /**
     * @var array<Line>
     */
    private array $lines;

    public function __construct(string $emailAddress)
    {
        // Before assigning, we verify that the email address is
        valid
        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {

            // We don't return an error, but throw an exception
            throw new InvalidArgumentException(
                sprintf('Invalid email address: ' . $emailAddress
            );
        }
    }
}
```

```

        s)
    );
}

$this->emailAddress = $emailAddress;
}

public function addLine(EbookId $ebookId, int $quantity): void
{
    // We delegate the validation of $quantity to the Line constructor
    $this->lines[] = new Line($ebookId, $quantity);
}
}

final class Line
{
    private EbookId $ebookId;

    private int $quantity;

    public function __construct(EbookId $ebookId, int $quantity)
    {
        if ($quantity <= 0) {
            throw new InvalidArgumentException(
                sprintf('Line quantity should be at least 1')
            );
        }

        $this->ebookId = $ebookId;
        $this->quantity = $quantity;
    }
}

```

“Exceptions aren’t very user-friendly, right?”

You’re totally right, exceptions aren’t user-friendly. In fact, they should never show up on the user’s screen. The exception message and the accompanying stack trace should only be visible to developers. They should know when a validation exception happens, because it’s often an opportunity

for improving the user experience. We'll talk about this in Section [8.4](#) when we discuss how to use exceptions to talk back to the user after all. When it comes to user-friendliness, we may provide an additional kind of validation. One that doesn't throw exceptions but produces a nice list of friendly form errors in the user's language. We'll cover this approach in Section [8.3](#).

8.2 Using value objects to validate separate values

When talking about objects that protect themselves against bad data, the concept of a *Value object* may come to mind. A value object is an object that wraps one or more other values. These values could be primitive-type values like strings, integers, booleans or floats, or they could be value objects themselves. There are multiple benefits to wrapping a value inside a value object. In the context of validation, value objects are useful because they'll never need to be validated. The constructor of a value object will analyze the raw value provided to it, and throw an exception if it's invalid.

Rewriting the previous example using value objects, we could delegate the validation of the provided email address and order line quantity to the specialized value objects `EmailAddress` and `Quantity`. Listing [8.4](#) shows what the classes of these value objects might look like. We can basically move the existing `if` and `throw` statements from the `Order` entity to these new classes.

Listing 8.4: Value objects wrapping and validating primitive-type values.

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

    public function __construct(string $emailAddress)
    {
        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {

            // We don't return an error, but throw an exception
            throw new InvalidArgumentException(

```

```

        sprintf('Invalid email address: ' . $emailAddress
s)
    );
}

$this->emailAddress = $emailAddress;
}
}

final class Quantity
{
    private int $quantity;

    public function __construct(int $quantity)
    {
        if ($quantity <= 0) {
            throw new InvalidArgumentException(
                sprintf('Line quantity should be at least 1')
            );
        }

        $this->quantity = $quantity;
    }
}

```

Now whenever a method argument is of type `EmailAddress` or `Quantity`, you know that the value inside has been validated already. There is no need to validate the value again, so you can now safely accept this object. This simplifies the code in `Order` a lot, as you can see in Listing 8.5.

Listing 8.5: Value objects simplify validation inside the entity.

```

final class Order
{
    private EmailAddress $emailAddress;

    /**
     * @var array<Line>
     */
    private array $lines;

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

```

```

    }

    public function addLine(EbookId $ebookId, Quantity $quantity
): void
{
    $this->lines[] = new Line($ebookId, $quantity);
}
}

final class Line
{
    private EbookId $ebookId;

    private Quantity $quantity;

    public function __construct(EbookId $ebookId, Quantity $quantity)
    {
        $this->ebookId = $ebookId;
        $this->quantity = $quantity;
    }
}

```

There's a lot more to say about objects and how they should protect their own state. I've written about it in another book called *Object Design Style Guide* (Manning, 2019). For now let's summarize it as: an object should ensure that its data is never incomplete, invalid, or inconsistent.

- *Incomplete* means that data is missing that really should be there to perform even the most basic tasks. For instance data would be incomplete if we'd have an amount, but no currency.
- *Invalid* means that the data is of the right type, but its value does not have the desired qualities. For example: we have a currency (string), but it isn't a known currency (valid ones are 'EUR', 'USD', etc.).
- *Inconsistent* means there are two or more pieces of data that can't co-exist. For instance, the order is marked for payment by bank transfer, yet it also has a value for the credit card holder.

Besides protecting its current state, an object also has a responsibility to prevent bad state *transitions* to happen. If an order has been paid and delivered, it can't be cancelled anymore. Only a pending order can be

cancelled. In other situations you may first have to request a refund, or return the delivery somehow.

Once you've added all these measures to let objects protect their own state, they will be completely fool-proof. Any client could instantiate the object, call any method on it, and if they provide the wrong data, or the requested state transition is not allowed, it will fail and tell the client about what they did wrong. Compare this to the original example where we used a validator to validate the object. In that example we received a list of errors from the validator, but the entity would be in a bad state nonetheless. Now that the entity fails loudly at the first sign that something is wrong, we've accomplished what we wanted. We can now be sure that any data that is assigned to a property of the entity is correct and consistent. So if we (accidentally or on purpose) save it to the database, we won't end up with corrupt data.

Well, there's one scenario we didn't cover that might still give us corrupt data. An entity can validate most data that gets passed to it as constructor or method arguments. But it can't validate everything, because it can't look beyond itself and the data provided to it. An example is the `EbookId` parameter of the `addLine()` method ([Listing 8.6](#)): Order can't figure out all by itself if a given `EbookId` refers to an actual e-book in our catalog.

Listing 8.6: Does the provided `EbookId` refer to an actual e-book?

```
final class Order
{
    // ...

    public function addLine(EbookId $ebookId, int $quantity): void
    {
        /*
         * There's no way to verify that the provided ID refers
         * to an actual e-book from our catalog...
         */
    }
}
```

Even the fact that we use a value object to represent the e-book identifier doesn't mean it's a valid `EbookId`. The only thing `EbookId` itself can do is validate its own input data, verify that the string provided to it when instantiated matches the pattern of a UUID.

Because the `Order` entity itself can't validate the provided `EbookId`, the service that calls `addLine()` should do it instead. A nice way to do this is to use the entity repository for the related entity and fetch the entity from it. If it doesn't exist, the repository will throw an exception, stopping us from using the ID of an entity that doesn't exist.

In many cases the relation is there for a reason, and we actually need more information from the related entity than just the fact that it exists. In our current situation, we need to know that the `Ebook` exists, and we also need the e-book's price. So it makes sense to validate the relation between order line and e-book by fetching the `Ebook` read model from its read model repository, like we did in Chapter [3](#).

[Listing 8.7](#) shows how that works in an application service. By fetching the `Ebook` read model by the provided ID we verify its existence. If it doesn't exist, `getById()` will throw a `CouldNotFindEbook` exception.

Listing 8.7: Validating the e-book ID by loading a corresponding read model.

```
$ebook = $this->ebookRepository->getById(  
    EbookId::fromString($ebookId)  
);  
  
$order = new Order(/* ... */);  
  
$order->addLine(  
    $ebook->ebookId(),  
    $quantity,  
    $ebook->price()  
);
```

Couldn't we pass the entire `Ebook` object as an argument to the `addLine()` method? Yes, I think we can. It allows `addLine()` to be less passive. Instead

of providing only separate values (the `EbookId` and the price), we could give it the `Ebook` read model. This ensures that the price and the `EbookId` on the line actually belong to the same e-book. As a bonus, the code of the `addLine()` method now demonstrates where these values come from and that they belong together (Listing 8.8).

Listing 8.8: Passing the `Ebook` read model to the `Order` entity.

```
// ...  
  
$order->addLine($ebook, $quantity);  
  
// inside the 'Order' class:  
  
public function addLine(Ebook $ebook, int $quantity): void  
{  
    $this->lines[] = new Line(  
        $ebook->ebookId(),  
        $ebook->price(),  
        $quantity  
    );  
}
```

However, we should be aware that by doing so we may let `Order` in on too many details about `Ebooks`. This depends on how much data the `Ebook` read model exposes. If the read model is designed for this specific use case only, there's no real danger. In that case the read model only exposes what this particular client needs. If the read model is also used in other places, it may expose too much irrelevant information. That may still not be a big problem, but keep in mind that the fewer methods you expose to a client, the more flexible your design will be.

One thing you shouldn't do is pass an entire *entity* to another object. Relations should be established by ID, not by object reference. Sharing an entity exposes all the entity's modifier methods to clients that don't intend to (or shouldn't intend to) make changes to it.

8.3 Form validation

So far we've been looking at ways to prevent bad data from entering our domain objects. By now we can be pretty sure that all the data that we keep inside entities and value objects is going to be valid. We've also established the fact that related entities actually exist, so we no longer need to worry about that either. However, the way we've been protecting our domain model against bad input is by checking each input value at the last possible moment. Just before we assign the provided value to an object property, we throw an exception if anything looks wrong. Exceptions are great when you want to block an operation and stop further execution. For other purposes it's not as useful.

If we want to provide an error message in the user's language, an exception isn't very helpful. Exceptions are meant to be visible only to the maintainers of the application, so they can find out what went wrong. Exposing them directly to the user is often a security risk. So we can't really turn an exception into a user-friendly error message.

If instead of a single error message we want to provide a list of error messages, we can't do it either. We'll only know about the first thing that went wrong. When the user submits the form again, we can only tell them about the next problem. That's not user-friendly. The user shouldn't have to submit the form again and again until they finally got it right. They should be able to see what's wrong with any of the fields in the form the first time they submit it.

So for form validation we can't rely on entity-level exceptions. We want to collect all the problems and show them as a collection of form errors to the user. We also don't want to wait until the data has already ended up inside the entity. Should we then collect form errors inside our application service? We can't do that, because the application service isn't aware of the origin of the data. It can't provide a list of form errors since a form is a very web-specific thing. We need to do it before the data is used inside the application service, that is, in the web controller. An overly simplified implementation would look something like Listing 8.9.

Listing 8.9: Validating form data inside the controller.

```

public function createOrderAction(Request $request): Response
{
    $formErrors = [];

    if (!filter_var($request->get('email'), FILTER_VALIDATE_EMAIL)) {
        $formErrors['email']
    } = 'create_order.invalid_email_address';
    }

    // More validation...

    if (count($formErrors) === 0) {
        /*
         * Call the application service that creates the 'Order'
         *
         * Redirect to "thank you" page
         */
    } else {
        /*
         * Render the form again including the validation errors
         */
    }
}

```

It makes sense to validate input data inside the controller. The validation errors should be linked to the corresponding form fields, and this knowledge is available inside or close to the controller (actually, inside the template). At the same time, this approach of validating form data inside the controller doesn't fully make sense, because we are duplicating validation logic. We've seen that exact same call to `filter_var()` several times now. It's the same as the one in the `EmailAddress` value object we introduced in Section [8.2](#). To prevent duplication and keep the knowledge about what a valid email address is in one place, we should reuse the value object while validating the form input. Removing duplication and keeping the knowledge in one place has an important benefit. Whenever something changes about the validation logic, we don't have to update the code in two places.

[Listing 8.10](#) shows how you can reuse a value object when doing form validation. You try to create the value object in the normal way, and when this produces an exception, you know something was wrong.

Listing 8.10: Reusing the `EmailAddress` class for validation

```
public function createOrderAction(Request $request): Response
{
    $formErrors = [];

    try {
        new EmailAddress($request->get('email'));
    } catch (InvalidArgumentException $exception) {
        $formErrors['email']
[] = 'create_order.invalid_email_address';
    }

    // ...
}
```

What about validating the e-book ID that the user submitted? We could do a similar thing here, that is: try to fetch the `Ebook` from its repository and if that fails, add a form error (see [Listing 8.11](#)).

Listing 8.11: Validating an ID using the repository.

```
public function createOrderAction(Request $request): Response
{
    $formErrors = [];

    try {
        new EmailAddress($request->get('email'));
    } catch (InvalidArgumentException $exception) {
        $formErrors['email']
[] = 'create_order.invalid_email_address';
    }

    try {
        $this->ebookRepository->getById(
            EbookId::fromString($request->get('ebook_id'))
        );
    } catch (CouldNotFindEbook $exception) {
```

```
    $formErrors['ebook_id']
[] = 'create_order.could_not_find_ebook';
}

// ...
}
```

If I'd find this code in a real project, I would have two comments:

1. All of this validation code clutters the view on what this controller really does. Let's move it out of the controller.
2. We've removed the duplication by reusing validation logic, but we still duplicate the effort. Everything gets checked twice. Are you sure we can't skip one of these checks?

The first suggestion is a very common one. If a method becomes too large, move part of it out of the way, and enjoy the view. However, if we follow the second suggestion, we could maybe even get rid of *all* this validation code.

The question was: should we check everything twice? The answer is: no. Consider the email address validation. Can we remove it from the value object or from the controller? I think it's obvious that we should never remove the validation from the value object. We have already established that we need it to protect our objects from containing invalid state. So can we remove it from the controller? Yes, totally.

In fact, we should do everything we can to ensure that no well-behaving user would ever see a validation error. Just make sure you help the user provide the correct data when they are still filling out the form. Use frontend validation and assistance, in whatever way makes sense. While the user is entering their email address, the frontend can already provide a warning if the email address doesn't look right. Form submission could be disabled for as long as the email address field doesn't contain a correct value. All of this reduces the odds of the user providing bad data to the application. Even to the point where only a malevolent user who manually forges a POST request could submit invalid data. At that point, we really don't have to worry anymore about showing friendly form error messages in the user's language. A simple 400 Bad Request status message would suffice.

The same goes for validating the e-book ID. Do we really have to do that? The source of this value is whatever element in the user interface allows the user to select the e-book they want. This could be a `<select>` element, or some kind of button they press, which results in a hidden field getting populated with the correct value. If the value that ends up being submitted is incorrect, it's not the user's fault. It could be a programming mistake (maybe the hidden input value wasn't set correctly, or we made a mistake when rendering the select element). No validation error can help the user then, and you'd hope that you caught this issue before the code was released to production. Or it could be a malevolent user again, but they still don't need a friendly validation error.

So for the e-book ID validation I'd suggest the same strategy: don't even validate. Simply assume that the user has used the user interface in the correct way, and didn't tamper with the request data. If they did, everything should go fine once the data is being used to instantiate and manipulate domain objects. If they didn't, we still have our entity-level protection in place, and we simply show the generic error page.

Since throwing an entity-level exception is a just-in-case protection mechanism, you should keep track of them. If you find these exceptions in your server logs, it usually means that:

1. Somebody is abusing your application, or
2. You should improve the user interface so users can't make the same mistake again.

Looking at the data that was submitted, it should be easy to choose between these two options. Also, most users are good people, so we can assume most of the times it's just an issue with the user interface that enables users to submit bad data. Take action to make sure the exception never shows up again.

8.4 Using exceptions to talk to users

To be fair, not all entity-level exceptions fall in the previously mentioned categories. There's a third category. Some things can't be validated in the user interface, or at least, it would be very inefficient to do so. And in some

situations you only know late in the process if the request could be fully processed.

For instance, what if besides e-books, we'd also sell physical books. We only have a limited supply of each title, and we keep track of how many books we still have in stock. Since we only sell what we have and can't re-stock, the number of books still available is the number of books we can offer to our customers.

Should we do validation the moment the request comes in? Listing [8.12](#) shows how to do that, supported by a repository that can tell us how many books are available.

Listing 8.12: Validating the ordered quantity by checking stock levels.

```
public function orderPhysicalBookAction(Request $request): Response
{
    $formErrors = [];

    $quantityAvailable = $this->stockRepository
        ->numberOfAvailableBooks($request->get('book_id'));

    if ($quantityAvailable < $request->get('quantity')) {
        $formErrors['quantity'][] =
            'create_order.insufficient_quantity_in_stock';
    }

    // ...
}
```

Imagine once more that this application is no longer a web application. It's now a CLI application that system administrators can use to buy their books from. This means we would no longer use the existing web controller, so this validation code would not be called either. People could now order books that aren't even available. So again, we should conclude it's the application core that has to prevent this from happening. That way, our business rules will always be enforced, regardless of the surrounding infrastructure.

Given that an entity can only validate input data based on its current state, we can't expect it to validate the given quantity. So the only logical place to validate the order quantity is inside an application service. Listing [8.13](#) shows what it looks like if we move the validation logic to the application service that processes physical book orders.

Listing 8.13: Checking availability in the application service.

```
final class OrderPhysicalBookService
{
    private StockRepository $stockRepository;

    // ...

    public function order(
        string $bookId,
        int $quantity,
        /* ... */
    ): OrderId {
        $quantityAvailable = $this->stockRepository
            ->numberofAvailableBooks($bookId);

        if ($quantityAvailable < $quantity) {
            // What to do here?
        }

        // ...
    }
}
```

It's basically the same code, except, we don't really know what to do if we find out the available quantity is insufficient. We can't return a form error of some sorts; the application service is supposed to be useful even in a different kind of application.

We might consider returning some kind of infrastructure-agnostic “result” object. When there is any kind of a problem, we can return an `Error` object, and when everything is okay, we return an `OrderId`. Although in some programming languages you can easily do that, object-oriented languages aren't generally suited for that. The traditional way to indicate that a method

was successful is to return nothing (or a value object like `OrderId` in this example), and to throw an exception when something goes wrong.

We could catch the exception in the controller and manually map it to a form error. Or we could use the power of objects to do that in a semi-automatic way. First, we introduce a way to distinguish regular exceptions from exceptions we want to tell the user about: the `UserErrorMessage` interface. Whenever we want to talk to the user about something that went wrong, we create a custom exception class that implements the `UserErrorMessage` interface. We may also introduce a base class that can save us some boilerplate code (`BaseUserErrorMessage`). Listing 8.14 shows both the interface and the base class.

Listing 8.14: `UserErrorMessage` and `BaseUserErrorMessage`,

```
interface UserErrorMessage extends Throwable
{
    public function translationKey(): string;
}

abstract class BaseUserErrorMessage
    extends RuntimeException
    implements UserErrorMessage
{
    private string $translationKey;

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

        parent::__construct($translationKey);
    }

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

Now we introduce a custom exception class and let it extend from `BaseUserErrorMessage`. Using a named constructor we can create the

exception in an intention-revealing way. Listing [8.15](#) shows the custom exception class and how the application service can throw it.

Listing 8.15: The application service throws a `UserErrorMessage` exception.

```
final class CouldNotOrderPhysicalBook
    extends BaseUserErrorMessage
{
    public static function becauseInsufficientStockLevels(): self
{
    {
        return new self(
            'create_order.insufficient_quantity_in_stock'
        );
    }
}

final class OrderPhysicalBookService
{
    private StockRepository $stockRepository;

    public function order(
        string $bookId,
        int $quantity
        /* ... */
    ): OrderId {
        // ...

        if ($quantityAvailable < $quantity) {
            throw CouldNotOrderPhysicalBook
                ::becauseInsufficientStockLevels();
        }
        // ...
    }
}
```

Finally we'd want to show the error message to the user. In order to do that we'd have to catch it in a place where we know who the user is and how we want to talk to them. This means we should catch the exception inside the controller and turn the error message into a form error. Listing [8.16](#) shows how to do that.

Listing 8.16: Turning the exception into a form error.

```
public function orderPhysicalBookAction(Request $request): Response
{
    $formErrors = [];

    // ...

    try {
        $this->orderPhysicalBook->order(
            $request->get('book_id'),
            $request->get('quantity')
            /* ... */
        );
    } catch (UserErrorMessage $exception) {
        $formErrors['general']
    [] = $exception->translationKey();
    }

    // ...
}
```

Once you have the ability to give feedback to users using exceptions, you may start removing some validation code and throw a `UserErrorMessage` instead. Not every domain-level exception needs to be user-facing though. As we've discussed in the previous sections, most validation issues can be prevented by providing a good user experience. Assisting the user when they are working with the user interface is the best way to communicate with the user. Using an exception for that should be a last resort. As an example, when we ask the `StockRepository` for the number of available books, and it fails to find a record in the `stock_levels` table, should it throw a `UserErrorMessage`? To answer that question; what's a reason that this could fail? I can only think of reasons where either we made some kind of programming mistake, or the user tampered with the request data. In both these situations, it doesn't make sense to tell the user about the problem. We should be notified about the problem, and fix it. Or in the second case, ignore it, because we can't do anything about it.

“Shouldn’t exceptions only be used in exceptional situations?”

I’ve heard this catchphrase often, and although intuitively it may make sense, in practice, it’s not very useful advice. You’d first have to define “exceptional”. We could go with “it doesn’t usually happen”. But, how many times is “usually”? The `CouldNotOrderPhysicalBook` in the example above is actually a great example of an exception that we only throw in exceptional situations. Only when we run out of stock between the moment the user selects the book and submits the form should this exception be thrown. Depending on how many concurrent users the application has, the chance that this happens should be really small. If it turns out that an exception is no longer exceptional, that is, it gets thrown a lot, this is a strong sign that we need to improve the user experience somehow.

Another definition of “exceptional” would be: “when things are really wrong”. For instance, the network is down, we couldn’t write to a file, etc. Of course, these are situations where we should feel free to throw an exception. But I don’t think it’s the only case in which we should be allowed to use them. Exceptions are a way to stop the execution of a function, and we want to do that when we know we shouldn’t continue. Preventing bad things from happening to our domain objects is a great reason for throwing exceptions.

8.5 When validation is not the answer

Let’s stay with the insufficient-stock-level situation a little longer. The title above this section spoils the surprise: maybe validation is not the answer in this case. Of course, a rudimentary check that we’re not selling something we don’t have is a nice idea. But the answer we get from `StockRepository`’s `numberOfAvailableBooks()` may not be as accurate as we like it to be. We call this method at the last possible moment, but theoretically we may still get the wrong answer from it. Maybe between calling

`numberOfAvailableBooks()` and saving the order the system also saves another order for the same book but from a different customer. If there was only one remaining book from stock, the customer will never get the book they ordered, since the other customer gets it (they were first, after all!). Did `StockRepository` give us a wrong answer then? No, the answer it gave is always the correct answer *at the moment it calculates the answer*. It's just that the answer changes all the time.

In functional programming terms, `numberOfAvailableBooks()` is an impure function. Such a function is not referentially transparent, meaning that it can give different answers if you call the function again. But while the answer is continuously shifting, we still want to rely on it while validating the form data submitted to the application. Well, it's unfortunate, but we can't rely on an impure query function.

This may not be a problem for your system and their users. Maybe `numberOfAvailableBooks()` gives a useful answer in 99.9999% of all cases and it doesn't get in your way. If your system deals with many customers using the system at the same time, ordering the same thing from it, it may become a problem though. If that's the case, then exact comparison between two quantities like we have been doing so far is not be the right approach. There are better options though. Some systems implement a fuzzy comparison; they don't assume that a query function like `numberOfAvailableBooks()` will be able to provide an exact answer. Other systems accept any order that comes in and process it later. This of course influences the user experience. The user only gets a preliminary answer: "Yes, we're going to process your order, please wait until we have the definitive answer for you." If the odds are very small that an order fails because people order the same thing at the same time, the system might also give the customer a "yes" anyway, just to keep them happy. In the unlikely case that the product can't be delivered after all, they will talk to the customer and find out how to proceed. Maybe the customer gets a discount on their next purchase, maybe they find an alternative product, and so on. In these cases it's better to be friendly and helpful towards your users than to throw errors when the user wasn't expecting them.

Given that using impure query functions may not work very well in a validation process, I think a good rule of thumb would be: when validating

input data, only use pure functions. It means that validation itself, if seen as a function, becomes a pure function itself. If you validate the data now and it's valid, it should still be valid when you validate it again. Likewise, if it's invalid, the next time you validate the data, it should also be invalid.

In all other situations, you may either stick with validation and remain conscious of the fact that it's not ideal. Or you could redesign the system to deal with issues in other ways. Your strategy could be that instead of preventing certain situations, you aim to recover from them.

8.6 Creating and validating command objects

In the beginning of this chapter we've looked at various ways to validate input data by looking at it. We didn't yet consider the question: what if the data isn't there or isn't of the right type? When processing forms, a known issue is that empty input fields may not be submitted by the browser. So if we ask the `Request` object to give us the value of field X, we may find out that field X wasn't even in the request body. Another issue is that we can't distinguish between an empty string and `null` when it comes to serialized data like a request body. So before we know it, we're passing around an empty string, where it would make more sense to use `null`. If not the browser, then your JS framework may do weird things to form data, so we need to improve the situation either way.

What I suggest is to provide some shape to the input data first. Instead of taking out separate values from the `Request` we could bundle these values and put them in an object. We give the object a good class name, which tells the reader how the data inside it is going to be used. We've seen such an object before, in Section [4.4](#): this is a *Command object*. There we introduced it as a way to group the parameters of an application service call. A command object also serves as a recognizable shape for input data in your application. The object is a data holder and transfers data from a controller to an application service, which is why a command object is also a Data Transfer Object (DTO).

There are several advantages to accessing the properties or getters of a custom object over accessing request data key by key.

1. A custom object provides a complete list of all the available “keys” (i.e. properties or getters). A `Request` object can’t provide such a list and just does a lookup in the request data for any key you provide when calling `get()`.
2. A potential client of an application service can look at the command object and find out what data it needs to provide. It couldn’t find that out by looking at the structure of the request body, because it may be incomplete, or contain more fields than necessary, which will be ignored by the application.
3. A `Request` object does not inherently use specific data types. Values that are extracted from a `Request` object are either of type `string` or `undefined (null)`. Using a custom object enables us to explicitly define types and make it clear for clients whether or not a value is “nullable”.
4. Inside a custom object we can define default values for missing request data by setting default property values.

Since request data is of type `string`, we usually have to do some casting before we can use it. I find it convenient to do this work inside a named constructor on the command class itself. That way knowledge about the fields and what goes in them lives inside a single class. Listing 8.17 shows a simple example. The `OrderEbook` command object should contain an e-book ID, a quantity, an email address, and optionally the name of the buyer. `OrderEbook::fromrequestData()` relies on the complete array of request data, so the code can remain decoupled from the particular framework that we use.

Listing 8.17: A command gets instantiated based on request data.

```
final class OrderEbook
{
    private string $emailAddress;
    private int $ebookId;
    private int $quantity;
    private ?string $buyerName;

    public function __construct(
        string $emailAddress,
```

```

        int $ebookId,
        int $quantity,
        ?string $buyerName
    ) {
        $this->emailAddress = $emailAddress;
        $this->ebookId = $ebookId;
        $this->quantity = $quantity;
        $this->buyerName = $buyerName;
    }

    /**
     * @param array<string, string|null> $data
     */
    public static function fromrequestData(array $data): self
    {
        return new self(
            $data['email'],
            $data['ebook_id'],
            $data['quantity'],
            $data['buyer_name']
        );
    }
}

```

As you may have noticed the current version of `fromrequestData()` is not able to deal with missing request data. If certain keys are missing we'll get PHP notices for them. The method also doesn't do any type casting and although we don't want to ignore notices either, we certainly can't ignore type errors.

How to get from an array of request data (which is potentially even empty) to an `OrderEbook` object that contains values of the correct type? In our example, each value needs a different approach:

- The `$emailAddress` property requires a `string`, but the `email` key may not be defined in `$data`.
- The `$ebookId` property requires an `int`, but the `ebook_id` key may not be defined in `$data`.
- The `$quantity` property requires an `int`, but the `quantity` key may not be defined in `$data`.
- The `$buyerName` property requires a `string`, but is nullable. The `buyer_name` key may not be defined in `$data`, or it may be an empty

string, in which case we should convert it to null.

[Listing 8.18](#) shows how to enhance the `fromrequestData()` method to deal with each of these cases.

Listing 8.18: Dealing with undefined keys and type casting.

```
public static function fromrequestData(array $data): self
{
    return new self(
        isset($data['email']) ? (string)$data['email'] : '',
        isset($data['ebook_id']) ? (int)$data['ebook_id'] : 0,
        isset($data['quantity']) ? (int)$data['quantity'] : 0,
        isset($data['buyer_name']) && $data['buyer_name'] !== ''
            ? (string)$data['buyer_name'] : null
    );
}
```

This ensures that where our application expects an integer, there will be an integer (even if it's a meaningless one like 0). If there has to be a string, there will be a string, even if it's an empty one. [Listing 8.19](#) illustrates this for the email address. Clients can call the `emailAddress()` getter of the command object to get the email address from it. The return value will always be a string, but it could be an empty string. That doesn't really matter, because down the stream there will always be some domain object that looks inside the variable and validates it. At least we won't get generic type errors anymore; we'll get proper exceptions which indicate a specific problem with the content of a variable.

Listing 8.19: The type of input data will always be correct, but the value has to be validated.

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

    // ...

    public function emailAddress(): string
```

```

    {
        return $this->emailAddress;
    }
}

final class EmailAddress
{
    public static function fromString(string $emailAddress): self
    {
        // Validate the string
    }
}

$command = OrderEbook::fromrequestData($emptyrequestData = []);
/*
 * 'emailAddress()' returns an empty string.
 * 'fromString()' will throw an exception.
 */
$emailAddress = EmailAddress::fromString($command->emailAddress());

```

There are ways in which we could improve this code even more. For instance, methods like `fromrequestData()` are really hard to read, and it's really easy to make programming mistakes as well. That's why I like to introduce a few helper functions that can help us remove some of the duplication and adds some meaning to all those `isset()` calls and ternary operators. Listing 8.20 shows how you can use a trait for that, but you could also use a utility class with public static methods if you like.

Listing 8.20: A trait that helps with mapping.

```

trait Mapping
{
    private static function getString(array $data, string $key): string
    {
        if (!isset($data[$key])) {
            return '';
        }

        return (string)$data[$key];
    }
}

```

```

    private static function getInt(array $data, string $key): int
{
    if (!isset($data[$key])) {
        return 0;
    }

    return (int)$data[$key];
}

private static function getNonEmptyStringOrNull(
    array $data,
    string $key
): ?string {
    if (!isset($data[$key])) {
        return null;
    }

    if (isset($data[$key]) && $data[$key] === '') {
        return null;
    }

    return (string)$data[$key];
}
}

final class OrderEbook
{
    use Mapping;

    // ...

    public static function fromrequestData(array $data): self
    {
        return new self(
            self::getString($data, 'email'),
            self::getInt($data, 'ebook_id'),
            self::getInt($data, 'quantity'),
            self::getNonEmptyStringOrNull($data, 'buyer_name')
        );
    }
}

```

Should we write a unit test for `fromrequestData()`? I think we don't have to do that; `fromrequestData()` is a very simple method with a single

execution path. The complexity has been moved to the helper functions and those definitely should be unit-tested. Because many classes are going to rely on these helper functions, they should be safe to use and work in all possible cases.

Summarizing the approach in this section: we started out with shapeless request data, or at least, request data with an unclear shape. We defined a command object with an explicit list of all the fields, types, and whether or not certain values are optional (i.e. nullable). We then created a method that was able to copy request data into the object. Once the request data is inside the object, the object's getters and their return types ensure that any client inside our application can safely use the request data, knowing that each value is defined and is of the expected type. This is a major advantage of using command objects instead of separate values.

But there's more! In Section [8.3](#) we talked about form validation and how it's actually request data validation. We shouldn't put data into an entity and then validate the entity, we should validate the request data before giving it to the entity. An entity will throw an exception when input data is invalid. When we validate request data, no exception will be thrown, and this allows us to build up a list of validation errors.

The same is true for a command object. We can populate the command object and we won't get any exception. All we do is transform the request data array into an object, fill in missing data, and quietly cast data to the correct types. We might just as well first populate the command object, and then validate its values, instead of validating the request data. Listing [8.21](#) shows a custom validator class that doesn't validate the request or request data, but validates a command object.

Listing 8.21: Validating a command object.

```
final class OrderEbookValidator
{
    /**
     * @return array<string, array<string>>
     */
    public function validate(OrderEbook $command): array
    {
```

```

$formErrors = [];

try {
    EmailAddress::fromString($command->emailAddress());
} catch (InvalidArgumentException $exception) {
    $formErrors['email'][] = 'invalid_email_address';
}

// ...
}

// Inside the controller:

$createOrder = CreateOrder::fromRequestData(
    $request->request->all()
);

$errors = $this->validator->validate($createOrder);
if (count($errors) === 0) {
    // Create the order
} else {
    // Render the form again
}

```

If you use Symfony and its Validator component^{[77](#)} you could also add some configuration to the command object and its properties. Then you can use the generic validator service to create a list of validation errors for you. This is really useful if you use Symfony’s Form component^{[78](#)} as well, and don’t want to write the validator and the integration code with form rendering yourself. In that case you may have to do some extra work, like adding setters with nullable parameters, which may seem a little “impure” to you. But if you keep an eye on the relation between the time you invest and the value that it delivers, I think it’s something you should consider.

8.7 Summary

When separating core from infrastructure like we’ve been doing in previous chapters, the activity of validating user input also has to be divided in two parts. In the first place, domain objects like entities have to protect themselves against ending up in an invalid state. An entity will throw an exception as soon as any part of the input is invalid, incomplete, or leads to

an inconsistent state. Validating relations between entities can be done by retrieving the related entity from its (read model) repository.

Once we can be sure that an entity is always-valid, we still need to assist the user in providing the right data. We need to keep in mind who the user is, and how they communicate with our application. This is the infrastructure-specific part of validation, where we render HTML form errors in the user's language, or provide a JSON-encoded list of machine-readable validation errors.

Sometimes validation is not the answer, because the data involved in the validation process may be outdated, like when you check the stock levels of a certain product before you accept an order for it. When validation might not be able to give the right answer, look for other design options.

Exercises

1. Which of the following object types are allowed to throw exceptions in case the provided input is invalid?⁷⁹

1. Value objects
2. Entities
3. Application services
4. Controllers
5. Validators

2. Where should we collect a list of validation errors for submitted form data?⁸⁰

1. In the entity
2. In the application service
3. In the controller

3. Which of these sentences apply to command objects and command object validation?⁸¹

1. A command object should have only `string`-type properties and they should all be allowed to be `null`.
 2. Everything that the entity itself validates, should also be validated by the command object validator.
 3. A command object provides structure and ensures that all input data has been cast to the expected types.
 4. You can reuse the entity's validation logic and value objects when validating a command object.
-
-

9 Conclusion

This chapter covers:

- A deeper discussion on the distinction between core and infrastructure code
 - A summary of the strategy for pushing infrastructure to the sides
 - A recommendation for using a domain- and test-first approach to software development
 - A closer look at the concept of “pure” object-oriented programming
-

9.1 Core code and infrastructure code

In Chapter [1](#) we’ve looked at definitions for the terms *core code* and *infrastructure code*. What I personally find useful about these definitions is that you can look at a piece of code and find out if the definitions apply to it. You can then decide if it’s either core or infrastructure code. But there are other ways of applying these terms to software. One way is to consider the bigger picture of the application and its interactions with *actors*. You’ll find the term actor in books about user stories and use cases by authors like Ivar Jacobson and Alistair Cockburn, who make a distinction between:

1. Primary actors, which act upon our system
2. Secondary or supporting actors, upon which our system acts

As an example, a *primary* actor could be a person using their web browser to send an HTTP POST request to our application. A *supporting* actor could be the relational database that our application sends an SQL INSERT query to. Communicating with both actors requires many infrastructural elements to be in place. The web server should be up and running, and it should be accessible from the internet. The server needs to pass incoming requests to the application, which likely uses a web framework to process the HTTP messages and dispatch them to the right controllers. On the other end of the application some data may have to be stored in the database. PHP needs to

have a PDO driver installed before it can connect to and communicate with the database. Most likely you'll need a lot of supporting code as well to do the mapping from domain objects to database records. All of the code involved in this process, including a lot of third-party libraries and frameworks, as well as software that isn't maintained by yourself (like the web server), should be considered *infrastructure* code.

Most of the time between the primary actor sending an HTTP request to your server, and the database storing the modified data, will be spent by running infrastructure code and most of this code can be found in PHP extensions, frameworks, and libraries. But somewhere between ingoing and outgoing communication the server will call some of your own code, the so-called *user code*.

User code is what makes your application special: *what things can you do with your application?* You can order an e-book. You can pay for it. *What kind of things can you learn from your application?* You can see what e-books are available. And once you've bought one, you can download it. Frameworks, libraries, and PHP extensions could never help you with this kind of code, because it's domain-specific: it's your business logic.

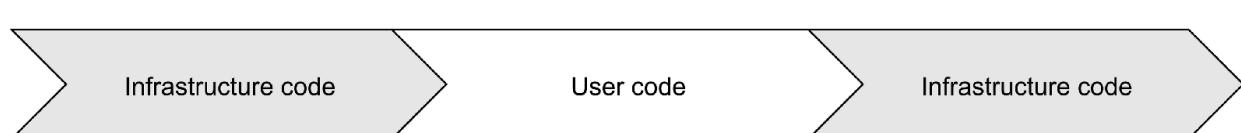


Figure 9.1: User code is in the middle, with infrastructure code to the left and right.

Figure 9.1 shows that user code is in the middle of a lot of infrastructure code. Even if we try to ignore most of the surrounding infrastructure while working on and testing user code, we'll often find that this code is hard to work with. That's because the code still contains many infrastructural details. A use case may be inseparable from the web controller that invokes it. The use of service locators and the likes prevents code from running in isolation, or in a different context. Calls to external services require the external service to be available when we want to locally test our code. And so on...

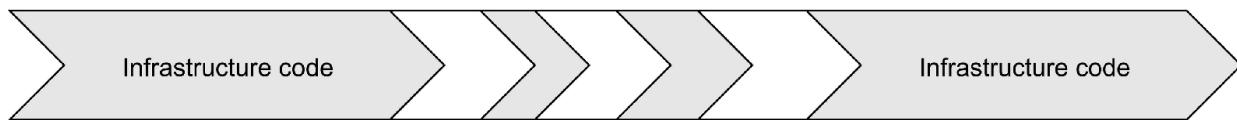


Figure 9.2: Core code is not strictly separated from infrastructure code.

If that's the case, user code consists of a mix of infrastructure code and core code. Figure 9.2 shows what this looks like. When I look at this diagram, I immediately feel the urge to push the bits of infrastructure code to the sides. Where they belong, I'd say, because infrastructure code is the code that connects core code to the outside world so it might as well live as close to the outside world as possible.

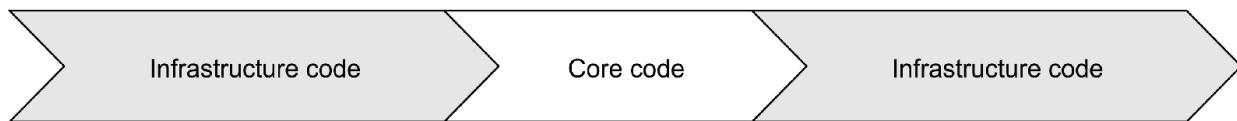


Figure 9.3: After defragmenting the user code, we can push even more infrastructure code to the sides, and keep only core code in the middle.

What remains in the middle, after “defragmenting” the user code is only *core code* (Figure 9.3). This is code that can be executed without relying on any actual infrastructure, and without making any connection to the world surrounding the application. No network, no database, no file system, etc. As we've seen in the previous chapters, this is great for testing (see also Chapter 14).

Completely isolated core code allows you to test the use cases of your application in a very early stage of development. You can prove that the application correctly implements its use case scenarios, without setting up routing for your web application, and without running schema migrations on any database. This is great because it allows you to work on your use cases from the start of the project, and collect design feedback, improve user stories, etc. You won't have to spend an entire *Sprint Zero* on choosing and setting up all the infrastructure. You won't have to find out that you've built

the wrong thing when most of the development budget has already been spent. You won't regret that you chose MongoDB instead of MySQL because you don't have to decide on day 1 of the project.

9.2 A summary of the strategy

What does it mean to push infrastructure code *to the sides*? In the previous chapters we've already seen many refactoring steps that helped us do it. All those "tactical" refactoring techniques can be summarized by a simple strategy. Separating core code from infrastructure code can be achieved by applying the following principles:

- Use dependency injection everywhere, let services depend on abstractions only
- Make use cases independent of the delivery mechanism of their input

The result will be that none of the core code depends on infrastructure code on either side. At the same time, any infrastructure code would be able to call core code. There are no special requirements for doing so. Both ingredients combined result in code that is completely *portable*. It provides a clear view on the application's use cases, without any distortions caused by infrastructural concerns. It can be easily tested, in complete isolation, without any special setup.

9.2.1 Use dependency injection and inversion everywhere

Dependency *injection* means that services will get everything they depend on (other services, as well as configuration values) injected as constructor arguments. Dependency *inversion* means that services depend on abstractions, instead of concrete classes^{[82](#)}.

The "inversion" part of dependency inversion indicates that the normal dependency direction gets *inverted*. Figure [9.4](#) illustrates the normal dependency direction, before applying dependency inversion. The `EbookOrderService` depends on `OrderRepositoryUsingSql`. This is a concrete class which knows how to store orders in a relational database. The

dependency arrow points from left to right, since the service depends on the repository.

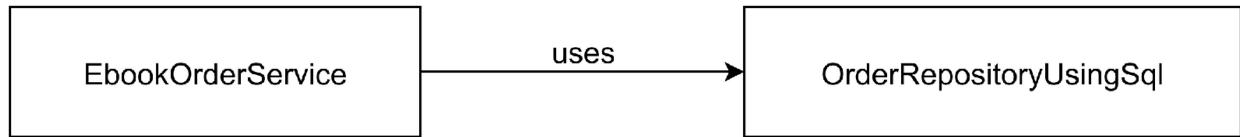


Figure 9.4: EbookOrderService uses OrderRepositoryUsingSql as a dependency, and gets it injected as a constructor argument.

Figure 9.5 shows what happens when we introduce a proper abstraction for the concrete OrderRepositoryUsingSql class. The abstraction is called OrderRepository and it's an interface. If the OrderRepositoryUsingSql is updated to implement the OrderRepository interface the EbookOrderService can start depending on the interface instead of the concrete class. In the diagram you can see that there's no longer an arrow pointing from EbookOrderService to OrderRepositoryUsingSql. We have successfully *inverted* this dependency.

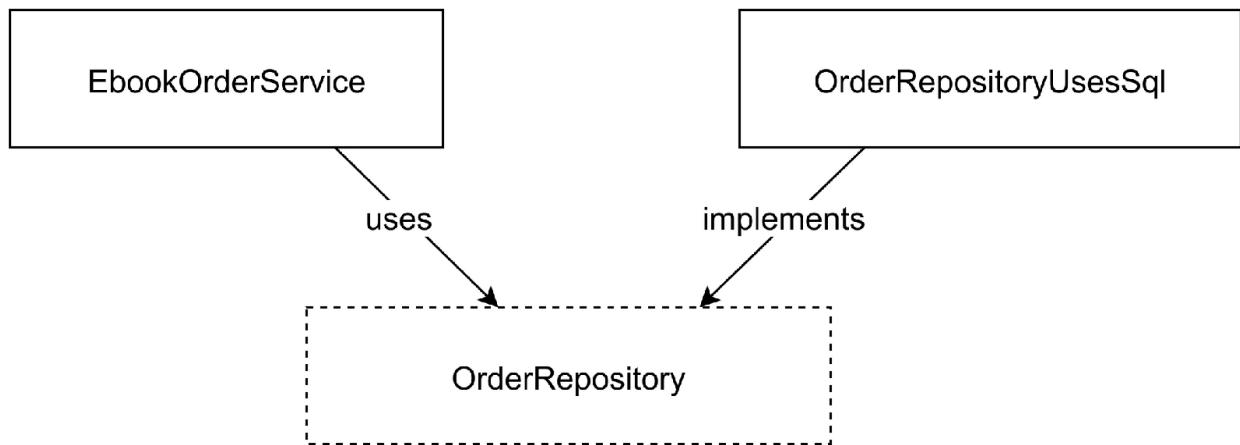


Figure 9.5: After applying *Dependency inversion*, EbookOrderService no longer depends on OrderRepositoryUsingSql.

This gives us great flexibility. We can change the implementation of the OrderRepository without getting in the way of the EbookOrderService. As

long as the modified implementation correctly implements the contract defined by the interface, everything will be fine. In fact, we can even completely replace the implementation of `OrderRepository`. We could experiment with a repository that uses MongoDB to store orders. And we can easily define a faster stand-in replacement when we write tests for the service.

Figure 9.6 shows what dependency inversion looks like in terms of core code versus infrastructure code. The repository interface and the service are core code. The repository implementation is infrastructure code. The dependency arrows stay inside the core area (from `EbookOrderService` to `OrderRepository`), or they go from the infrastructure area to the core area (from `OrderRepositoryUsingSql` to `OrderRepository`).

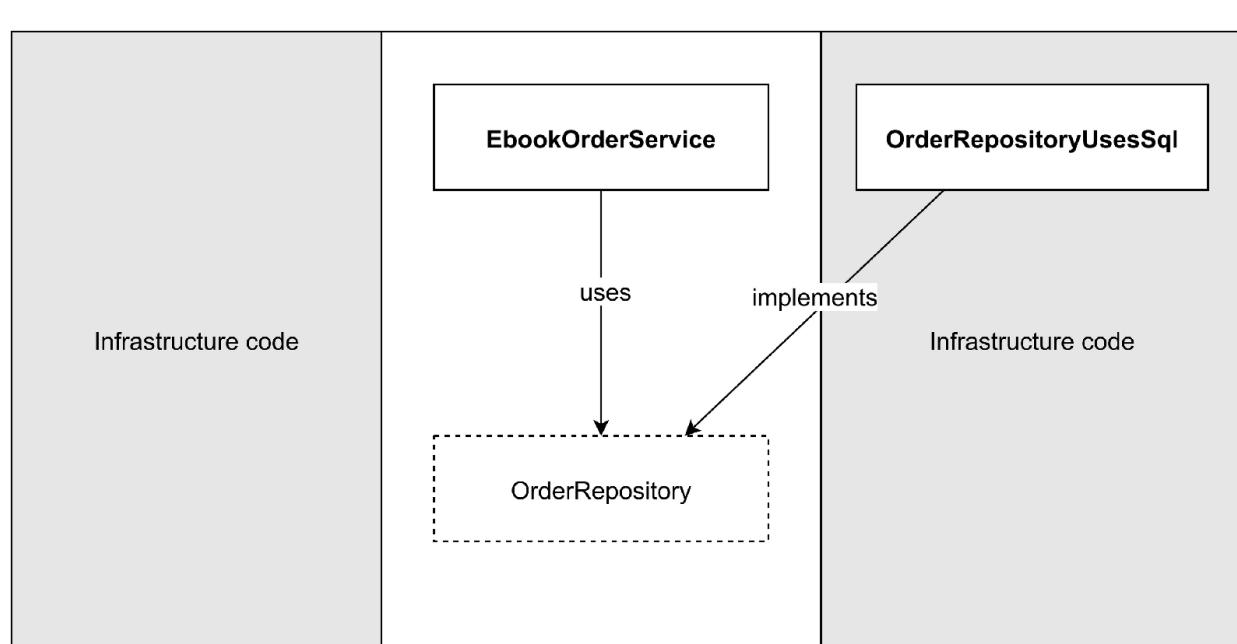


Figure 9.6: This diagram shows how applying dependency inversion helps separating core from infrastructure code.

Because core code never depends on infrastructure code, we can completely replace the infrastructure code and the core would not be influenced by it.

9.2.2 Make use cases universally invokable

Many applications hide important use cases in their controllers. As we saw in Chapter 4, with a few extra steps you can extract the code that represents the actual use case, independent of infrastructure code. We started in that chapter with the `OrderController::orderEbookAction()` method which contained all the steps of the “order an e-book” scenario. Figure 9.7 shows the initial situation. Although the controller depends on an abstraction provided by the core, the use case itself is sadly part of the infrastructure. This means we can’t easily invoke it, without also setting up and running other infrastructure code.

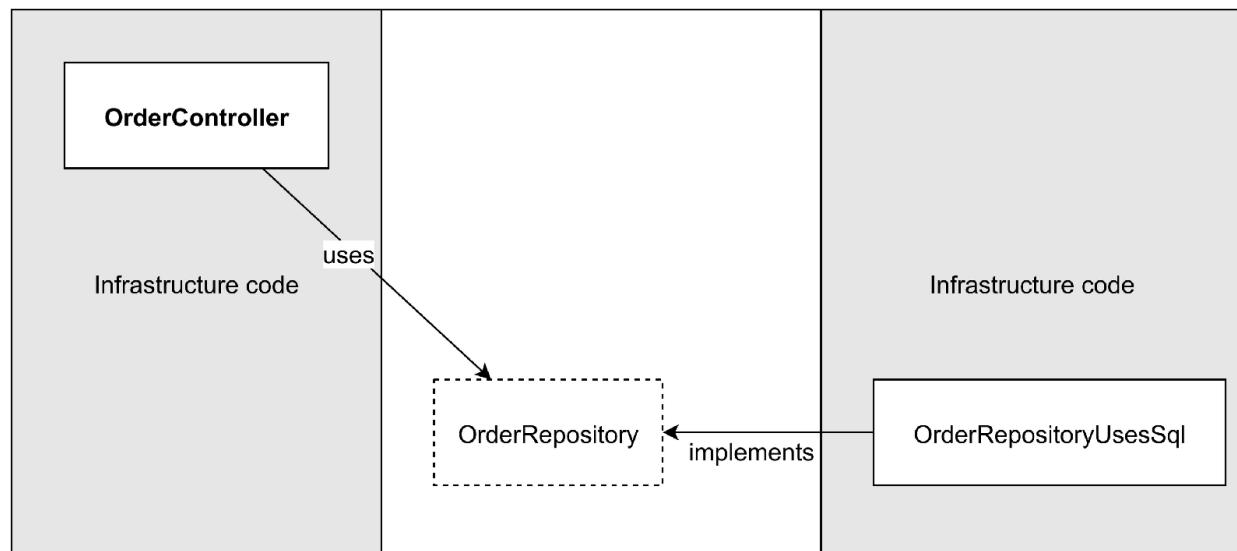


Figure 9.7: The use case is currently tied to a particular infrastructure, making it hard to invoke in other situations.

After some refactoring steps we were able to extract a use case service from the controller. This service would now count as core code, since it no longer depends on any infrastructural element. Its input is plain data, not an `HttpRequest` object or something. It manipulates only data in memory, and it only talks to abstractions, like `OrderRepository`. Figure 9.8 shows the result of the operation.

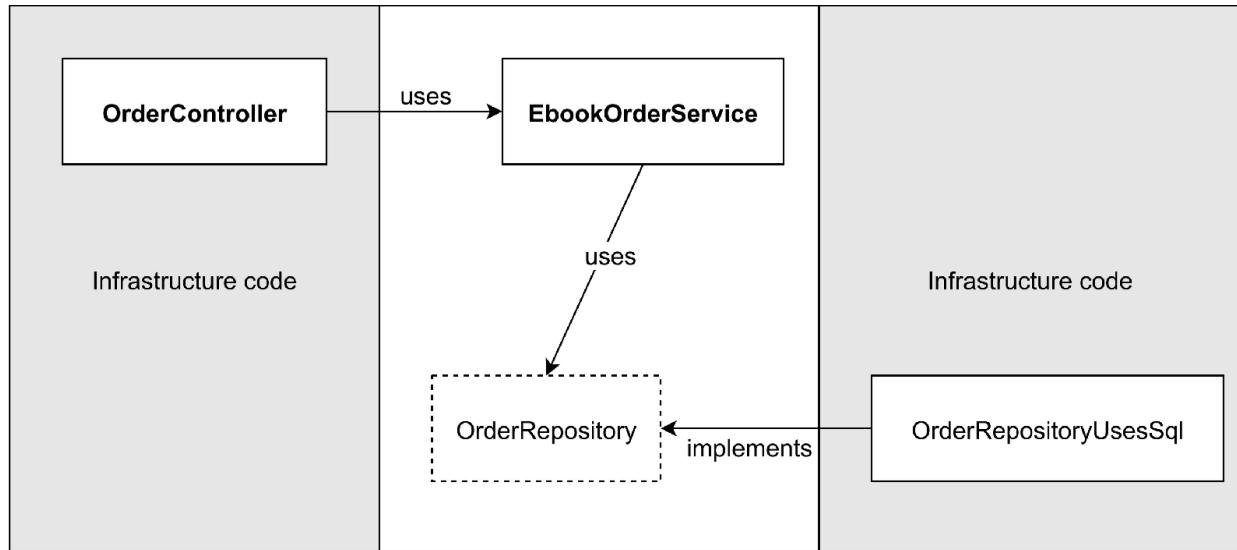


Figure 9.8: The use case is now fully represented as core code.

Again, the dependency arrows only point towards core code. This means that besides being able to completely replace the infrastructure code on the right side, we can now also replace the code on the left side. We can reuse the same use case with a different “front-end”, like an API endpoint, or a batch importer. Maybe we even build CLI commands that allow us to test some of the application’s use cases from the terminal. The application’s test suite will essentially be an alternative infrastructure for our core code as well. It will provide input for the use cases, and verify the outcomes of invoking them. Figure 9.9 shows how the only thing a test runner has to do is call the use case service, and use stand-in dependencies, to prevent the test from actually saving data to the database. Doing so, the test can verify that the use case scenarios have been implemented correctly, when it comes to domain objects, the domain invariants they protect, the interaction between them, and the way the use case is related to things outside the core.

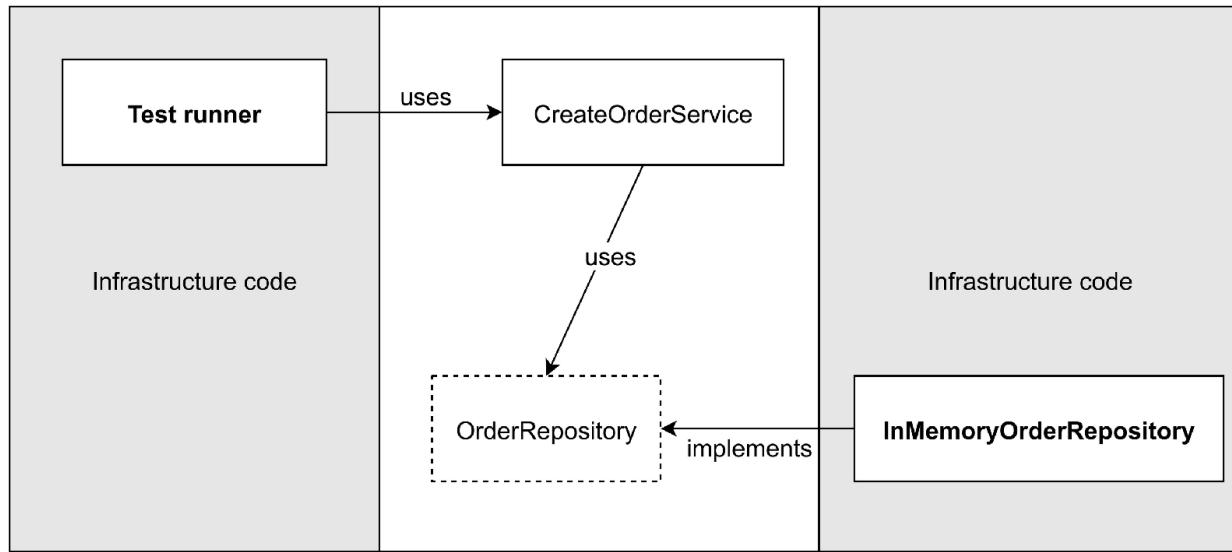


Figure 9.9: Because the use case is now represented by core code, it's easy to invoke it in other situations. It's also easy to run it in a testing environment.

9.3 Focus on the domain

An application’s code base is a codification of the team’s collective knowledge. It shows their understanding of the business domain. It also shows their understanding of the tools they work with. There’s a big difference between the two: the tools (frameworks, libraries, protocols, and so on) have a much shorter life expectancy than the business domain itself. A framework that is great today, will be deprecated in a year, maybe two. An ORM may be “rad” today, but considered a pain in just a couple of months from now. You’ll want to get rid of everything, but you don’t want to throw away the domain knowledge that is represented by your code. If you mix infrastructural details with your domain model, it’ll be really hard to let both evolve in different directions, or at a different speed.

Of course, your domain model will have to evolve itself. You want to incorporate new pieces of knowledge that you have gained along the way. But infrastructure shouldn’t get in the way of doing so. Testing the behavior of a domain model should happen in a really fast loop, giving you feedback on the design while you’re still working on that design. You shouldn’t be

forced to test core behavior with a *functional test*, which invokes the application including all layers of the stack.

9.4 Focus on testability

Production code needs tests which describe the developer's knowledge in terms of expectations about the code's runtime behavior. Code alone, without any tests, won't be able to preserve the relevant knowledge. When you encounter code without tests, you'll have to read the code to find out why it's there. And even then, it'll be very hard to find out why something is done the way it's done. Which is why Michael Feathers says:

To me, *legacy code* is simply code without tests.^{[83](#)}

You'll need special techniques to handle this kind of code. You'll feel the need to rewrite parts of it sooner than with tested code. However, changing anything about it will be a very scary thing to do, because it's very likely that you'll break things. As Martin Fowler puts it:

Whenever I do refactoring, the first step is always the same. I need to ensure I have a solid set of tests for that section of code. The tests are essential because even though I will follow refactorings structured to avoid most of the opportunities for introducing bugs, I'm still human and still make mistakes.^{[84](#)}

There's this interesting asymmetry about tests, as Robert Martin points out:

I know this sounds ridiculous; but consider. If somehow all your production code got deleted, but you had a backup of your tests, then you'd be able to recreate the production system with a little work.^{[85](#)}

If you only have production code, then it will be really hard to write tests for it. If you only have tests, then it will be much easier to write the production code that will make them pass. This shows the value of the tests: they specify what the production code is supposed to do, and, if you write good tests, they'll also explain why.

Of course, I know that testing can be hard sometimes. It will be annoying, it will sometimes feel like extra work. But all of this can in my experience be solved by simply investing more time in it. Make sure you get better at testing, every day. There's a rule that people use for this kind of problem: "if it hurts, do it more often". Fowler calls this "frequency reduces difficulty"⁸⁶. It's not just the repetition that makes it easier. The real trick is that repetition will automatically make you look for better ways of doing it. In other words, I think the rule should be: "if testing isn't easy, make it easy".

So focusing on testability is bound to increase the quality of your application code, as well as improving its life expectancy. Combined with a domain-first approach, the architectural techniques put forward in this book should help you build long-lasting applications.

9.5 Pure object-oriented code

In the previous chapters I've used the word "pure" several times as a qualifier for certain types of objects. I think it'll be useful to describe in more detail what I mean by pure objects. Knowing when code is pure or not will make a difference, because pure code can be unit-tested, and most of it can end up in the core of your application.

"Pure" as a qualifier for code originates from functional programming. A pure function is a function with a return value that completely depends on the arguments provided to it. Listing 9.1 shows an example of such a pure function. There's nothing that could influence the outcome of calling `sum()` other than the provided arguments, and the code of the function itself.

Listing 9.1: `sum()` is a pure function.

```
function sum(int $a, int $b): int
{
    return $a + $b;
}
```

Listing 9.2 on the other hand shows a function whose outcome is dependent on something other than the arguments and the code.

Listing 9.2: secondsPassed() is an impure function.

```
function secondsPassed(int $previousTimestamp): int
{
    return time() - $previousTimestamp;
}
```

Such a function would be called “impure” because its return value depends on the actual current time. There are many reasons for a function to be impure. For instance, if it tries to load a file, make a network connection, generates random data, etc. We’ve seen these situations in previous chapters and called the code that performs this kind of work infrastructure code.

Now, there are ways to make impure code pure again, or rather, split impure code into a pure part and an impure part. Listing [9.3](#) shows how to do it in the case of secondsPassed(): you only need to “push” out the part that made the function impure.

Listing 9.3: secondsPassed() is now made pure again.

```
function secondsPassed(
    int $currentTimestamp,
    int $previousTimestamp
): int {
    return $currentTimestamp - $previousTimestamp;
}
```

With objects, a similar approach can be used to make impure methods pure again. Take a look at Listing [9.4](#) which shows the rather silly Stopwatch class with an impure method, which is basically the same as the impure secondsPassed() function we just saw.

Listing 9.4: secondsPassed() is an impure object method.

```
final class Stopwatch
{
    public function secondsPassed(int $previousTimestamp): int
```

```

    {
        return time() - $previousTimestamp;
    }
}

```

If we want to make the method pure again, we have to get rid of the call to `time()`. In the functional example we modified the function to accept the current time as an argument instead of fetching it. This is certainly an option here as well (see Listing [9.5](#)).

Listing 9.5: One way of making `secondsPassed()` pure.

```

final class Stopwatch
{
    public function secondsPassed(
        int $currentTimestamp,
        int $previousTimestamp
    ): int {
        return $currentTime - $previousTimestamp;
    }
}

```

There is another option, which is unique to objects. We could inject a dependency that the `secondsPassed()` method can use to retrieve the current time. Listing [9.6](#) shows how you could inject some kind of `Timer` object. The `Timer` has a `time()` method that can be used to replace calls to `time()`.

Listing 9.6: Another way of making `secondsPassed()` pure.

```

final class Timer
{
    public function currentTimestamp(): int
    {
        return time();
    }
}

final class Stopwatch
{

```

```

private Timer $timer;

public function __construct(Timer $timer)
{
    $this->timer = $timer;
}

public function secondsPassed(int $previousTimestamp): int
{
    return $this->timer->currentTimestamp()
        - $previousTimestamp;
}
}

```

This leaves the `secondsPassed()` method unchanged: clients won't have to provide the current timestamp themselves. It does allow us to get rid of calls to `time()`, and replace them by calls to `Timer::currentTimestamp()`. But that doesn't make `Stopwatch` pure yet. Although the call to `time()` now only happens inside `Timer`, calling `Stopwatch::secondsPassed()` will inevitably call `Timer::currentTimestamp()`, which is impure. This indirectly makes `Stopwatch::secondsPassed()` impure as well.

We can fix this by applying a technique we've already seen several times now: dependency inversion. We should introduce a proper abstraction for "retrieving the current time". We already have a separate object for it (`Timer`), now we only need an interface for it. Let's turn `Timer` into an interface, and define a standard implementation for it that uses the system clock to get the current timestamp (see Listing 9.7).

Listing 9.7: The `Timer` interface and its standard implementation.

```

interface Timer
{
    public function currentTimestamp(): int;
}

final class TimerUsesSystemClock implements Timer
{
    public function currentTimestamp(): int
    {
        return time();
}

```

```

}

final class Stopwatch
{
    private Timer $timer;

    public function __construct(Timer $timer)
    {
        $this->timer = $timer;
    }

    public function secondsPassed(int $previousTimestamp): int
    {
        return $this->timer->currentTimestamp()
            - $previousTimestamp;
    }
}

```

By introducing the `Timer` interface, we have successfully built in the possibility to call `secondsPassed()` without indirectly calling `time()`. We can easily instantiate a `Stopwatch` object, with a stand-in `Timer` object, which simply returns a hard-coded timestamp (see Listing 9.8)

Listing 9.8: A `FakeTimer` that can be used when testing the `Stopwatch`.

```

final class FakeTimer implements Timer
{
    private int $timestamp;

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

    public function currentTimestamp(): int
    {
        return $this->timestamp;
    }
}

$stopwatch = new Stopwatch(new FakeTimer(1562845845));

```

Having seen the mechanism of making impure object methods pure, we could also rephrase “pure” as: deterministic. Because they only rely on method arguments and constructor-injected abstract dependencies the client has full control over the object. This results in deterministic objects, which is great for testability. There’s no special setup required. The only thing a test has to do is instantiate the object itself, providing any required dependency, and calling a method on it, providing any required argument.

Looking at the code of `Stopwatch` we can conclude that it’s now pure: it’s decoupled from any infrastructure class. But when the application is running in production `Stopwatch` will of course call `currentTimestamp()` on `TimerUsesSystemClock` instead of the `FakeTimer`. A dependency injection container will usually take care of the actual setup of your service object, including any of its dependencies. Figure 9.10 shows the differences between both perspectives.

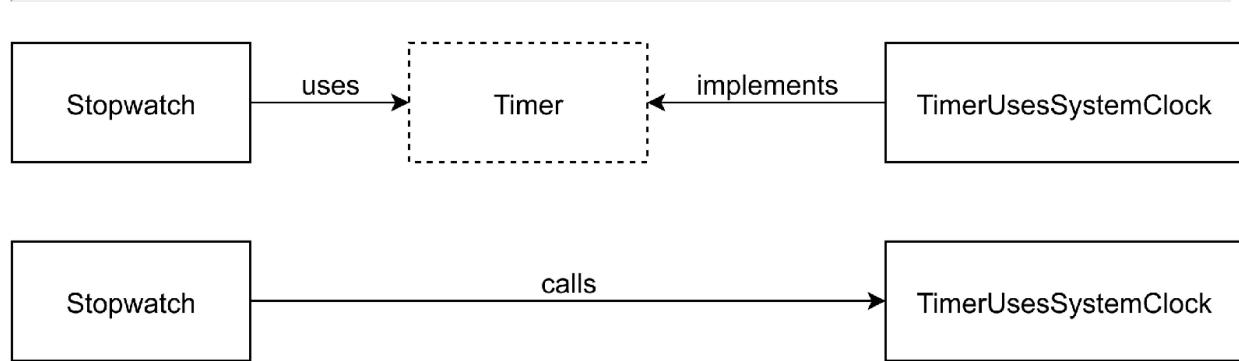


Figure 9.10: The code of `Stopwatch` depends only on the `Timer` interface, but at runtime the `Stopwatch` will use an object of type `TimerUsesSystemClock`, without being aware of that.

I find that when you start at a low level, the level of classes and methods, and you aim to write as much object-pure code as possible, while pushing all the infrastructure-related things to the sides, you will end up with a much better design at a higher level too. Which is why I’ve started this book focusing only on core versus infrastructure instead of architectural concepts like layering, and ports and adapters. Of course we’ll discuss these concepts in Part II, but the big win is separating core from infrastructure code. All the rest is nice-to-have, and you will get the rest more or less for free.

9.6 Summary

In this chapter we rephrased the definition of core code as code that represents the use cases of an application. Infrastructure code is code that connects these use cases to its external actors. Actors can be primary actors (users of the application) or secondary/supporting actors (other systems that our application uses). To decouple from primary actors we have to make our use cases *universally invokable*, regardless of the delivery mechanism that a specific type of actor supports. To decouple from secondary actors we have to apply *dependency inversion* and abstract any dependency that communicates directly with a secondary actor. A domain-first approach combined with a test-first approach will improve the quality of the application's core code, making it more likely to survive the infrastructural changes in the world around it. Core code has to be object-oriented code that is “object-pure” which is quite similar to the notion of functionally pure. All dependencies are made explicit and for every dependency that connects to something outside the application an abstraction has been introduced. Object-pure code is easy to test because, by definition, it behaves in a completely deterministic way.

Part II

Organizing principles

10 Introduction

This part covers:

- Framework-independent object types
 - Architectural layering
 - Hexagonal architecture, or: ports and adapters
-

In the previous part we covered many techniques for separating core from infrastructure code. Doing so is great for the future of your application. After some time though, maintaining these large buckets of code is still going to be a trouble. We need some more detailed organizing principles. In this part we'll discuss some of those principles, which have proven to be very effective. Combined, these principles result in an application that:

1. Is built around a standardized, recognizable catalog of object types (Chapter [11](#)),
2. Uses layers for separating concerns at a higher level, which assist the developer in deciding where to put things (Chapter [12](#)), and
3. Decouples application use cases from the way clients connect to the application using an architectural pattern called *Ports and Adapters* (Chapter [13](#)).

11 Key design patterns

This chapter covers:

- A catalog of design patterns
 - Implementation suggestions
 - A high-level design process based on these design patterns
-

11.1 Framework-inspired structural elements

Every framework comes with its own set of recognized element types. For instance, Symfony developers learn to create controllers, entities, form types, Twig templates, Yaml configuration files, and so on. Laravel developers also create controllers, but they need among other things: models, Blade templates, and PHP configuration files. When you take a look at the directory structure of most web application projects, you'll immediately notice the framework that's been used. Frameworks dictate your project structure. And frameworks also invade your code. This all sounds like frameworks are an enemy, instead of the helpful friend they presume to be, but this is a false contradiction. In infrastructure code, frameworks are your friend. In core code, they are not.

If frameworks determine the structure of your core code, you'll end up with:

1. Implicit use cases inside controllers,
2. A domain model that's coupled to its underlying infrastructure, and in general
3. Code that's coupled to the framework.

In Part I we've already seen many techniques to overcome these problems. We were able to extract a use case from a controller by modeling it as a framework-independent service. We extracted an entity from database interaction code. And we decoupled code from the framework by using dependency injection everywhere, and by passing contextual information as method arguments.

In this chapter we take a closer look at the types of objects that were the result of decoupling from infrastructure. Knowing more about the typical aspects of these objects will help you use them as building blocks instead of merely the result of refactoring activities. By using these objects as “primitives” you can implement all of the application’s use cases, without even choosing a framework. The framework will just be the finishing touch, the bridge between your application’s core and the outside world.

11.2 Entities

The first pattern to cover is the *Entity* pattern. In this book the concept of an entity is the same as the concept of an aggregate in Domain-Driven Design literature. An aggregate is an entity, including any of its child entities, and any of the value objects used inside of it. In my

experience the term “aggregate” leads to a lot of confusion so I decided to use the word “entity” in this book. We have talked about entity design in Chapter 2, and I’ve already mentioned several design rules for it there. Still, I want this chapter to be a reference guide to the standard design patterns you’ll need in decoupled application development, so I’ll briefly summarize the rules here. I’ll just declare the rules without defending them in detail⁸⁷

Entities are objects that preserve the state of your application. They are the only type of objects in your application that have persistent state. Most of the other objects should be designed to be immutable and stateless. Being mutable, entities should not be passed to clients that don’t intend to change their state. When a client needs to retrieve information from an entity, in most cases they should rely on a different type of object, that is, a *Read model* (see Section 11.6). The only type of client that is supposed to modify an entity is an *Application service* (see Section 11.4).

You shouldn’t be able to traverse from one entity to the other, e.g.

```
$this->getLine(1)->getProduct()->getProductGroup()->getProducts();
```

Changes should always be limited to a single entity. If you need to modify another entity, fetch it from its own repository. Don’t make changes to multiple entities in the same transaction. Make a clear distinction between the primary change and secondary effects. Handle these effects using *Event subscribers* (see Section 11.5).

11.2.1 Protect invariants

An entity should always protect its *domain invariants* (the things that are always true about it) and make sure that it’s in a consistent state. It should never contain invalid, incomplete, or meaningless data. Entities are meant to establish a basic level of consistency for your application, as well as protecting it against data corruption.

An entity’s constructor should be a named constructor and it should force clients to provide the minimum set of required data (see Listing 11.1). The constructor needs to verify that the data provided is valid, for instance that values are within the allowed range, have the minimum length, etc. You can use a standard set of assertions, or throw your own exceptions in case something is wrong.

An entity has a unique identity from the start. Provide it when calling the constructor (as discussed in Section 2.6).

Relations between entities should be established by their IDs, not by providing the entire object reference.

Listing 11.1: An `Order` always has an ID and a relation with a *Customer* entity.

```
final class Order
{
```

```

// ...

private function __construct(/* ... */)
{
    // ...
}

public static function create(
    OrderId $orderId,
    CustomerId $customerId
): Order {
    return new self(/* ... */);
}
}

```

11.2.2 Constrain updates

Only allow clients to update fields that can actually be modified. Force clients to update fields together when it makes sense (e.g. when you change the delivery address, clients should provide the street, number, postal code, and city in one go). Always validate the incoming data and throw exceptions when something is wrong. Verify that the requested change is possible given the current state of the entity, and that the entity won't end up in an invalid state (see Listing [11.2](#)).

Listing 11.2: `changeDeliveryAddress()` puts limits on when the delivery address can actually be changed.

```

final class Order
{
    private bool $wasCancelled = false;

    // ...
    public function changeDeliveryAddress(
        DeliveryAddress $deliveryAddress
    ): void {
        if ($this->wasCancelled) {
            throw new LogicException(
                sprintf(
                    'Order %s was already cancelled',
                    $this->id->asString()
                )
            );
        }
    }
}

```

11.2.3 Model state changes as actions with state transitions

When an update to a particular field actually represents an action performed on the object, define a method for it. The job of this method, again, is to validate the arguments provided to it. It also has to verify that the action is allowed, given the current state of the object. As an

example, an order may be cancelled, but only if it hasn't been delivered yet. Instead of a `setCancelled()` method, an `Order` entity would have a `cancel()` method, which performs the required checks (see Listing 11.3).

Listing 11.3: The `cancel()` action is only available when the object is in a certain state.

```
final class Order
{
    // ...
    private bool $wasCancelled;
    private bool $wasDelivered;

    public function cancel(): void
    {
        if ($this->wasDelivered) {
            throw new LogicException(
                sprintf(
                    'Order %s has already been delivered',
                    $this->id->asString()
                )
            );
        }

        $this->wasCancelled = true;
    }
}
```

Methods that change the state of the entity, or perform an action on it, should both be *Command* methods, that is, they should have a `void` return type.

While changing the delivery address of an order would be a simple update, cancelling an order has an impact on what you can do with the order. As we just saw, the same is true for delivering it. Once an order is in the "cancelled" state, it can no longer be delivered. Once an order is in the "delivered" state, it can no longer be cancelled. When you design an entity, it'll be helpful to create a state machine diagram for it, which documents the possible states of an entity and what actions (state transitions) are available for any given state (see Figure 11.1). Create unit tests for your entity to prove that it correctly implements the state machine you had in mind.

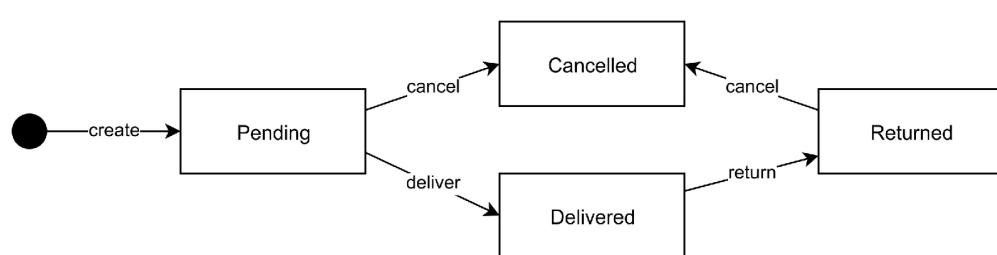


Figure 11.1: Entity states and state transitions

11.2.4 Don't think too much about tables

Design your object, its methods, and its properties without worrying about persistence. Mapping the data to any kind of database should be a separate task. Your entity should in the first place be a well-designed object regardless of the database that will eventually support storing it. You should always be able to define a mapping from your object to the database you want to use. In fact, you can let the design of your object determine what storage model would work best.

11.2.5 Record domain events

Keep an internal record of *Domain events* that have happened to the entity, like “Order was created”, “Order was cancelled”, etc. Domain events are simple immutable objects which are named after the event they represent, e.g. OrderWasCreated, OrderWasCancelled, etc. After saving the entity, it should be possible to retrieve a collection of these event objects so you can dispatch them and let other parts of the application respond to them. Listing 11.4 shows a simple setup for recording events and releasing them.

Listing 11.4: Keeping track of internally recorded domain events in a private array.

```
final class OrderWasCancelled
{
    private OrderId $orderId;

    public function __construct(OrderId $orderId)
    {
        $this->orderId = $orderId;
    }
}

final class Order
{
    private array $events = [];

    // ...

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

        $this->events[] = new OrderWasCancelled($this->id);
    }

    public function releaseEvents(): array
    {
        $events = $this->events;

        $this->events = [];

        return $events;
    }
}
```

11.3 Repositories

Every entity needs a repository. Because a repository crosses the application boundary to save and load entities it should be a service abstraction. This means the repository should be defined as an interface (e.g. `OrderRepository`). It also needs a standard implementation that implements the interface and the contract described by it. There are several alternative designs for repositories, but I prefer the simplest one. It doesn't make a distinction between adding and updating entities (see Listing 11.5).

Listing 11.5: A repository interface has a `save()` and a `getById()` method.

```
interface OrderRepository
{
    /**
     * @throws CouldNotSaveOrder
     */
    public function save(Order $order): void;

    /**
     * @throws CouldNotFindOrder
     */
    public function getById(OrderId $orderId): Order;
}
```

The contract that this interface represents is that when you have saved an `Order` with a certain `OrderId` you can at any time retrieve a copy of it by providing that same `OrderId` as an argument to `getById()`. After making a change to `Order` (in fact, after changing it in any of the ways that it allows), the object can be saved again using `save()`. And it can also be retrieved by calling `getById()`. The object you get back from `OrderRepository` could be the exact same instance, or an object that behaves in an identical way to the one you saved.

For `OrderRepository` implementations that save `Order` instances to an actual database, implementing this contract can take some effort. But creating a test double for this contract is actually really simple (see Listing 11.6).

Listing 11.6: A test double for `OrderRepository`

```
final class InMemoryOrderRepository
    implements OrderRepository
{
    /**
     * @var array<string, Order>
     */
    private array $orders = [];

    public function save(Order $order): void
```

```

{
    $this->orders[$order->orderId()->asString()] = $order;
}

public function getById(OrderId $orderId): Order
{
    if (!isset($this->orders[$orderId->asString()])) {
        throw CouldNotFindOrder::withId($orderId);
    }

    return $this->orders[$orderId->asString()];
}

```

“That’s too easy!”

Yes, the in-memory repository implementation is really simple. Things will be more complicated when you implement the repository that will save the entity to the real database. As we discussed in Chapter 2 there are different options. You can still use an ORM and delegate some of the logic at the cost of loosing some explicitness as well as control. If you decide to write your own mapping implementation keep the following requirements in mind.

When you save an entity you need to know if it’s a new entity or an existing one that’s being updated. Based on this knowledge you should do either an `INSERT` or an `UPDATE` query. You can create something known as an *identity map* where you keep track of entities that have been loaded from the database. The next time the application tries to `save()` an entity you look it up in the identity map. If it’s there, it needs to be updated, if it isn’t there it needs to be inserted. After the insert you add the entity to the identity map so saving it once more will trigger an update. An alternative (something I’ve used in TalisORM⁸⁸) is to let the entity itself keep track of whether or not it’s “new”.

If the entity has child entities, you need to do the same kind of change tracking for these entities as well. Child entities can often be deleted too so there should be a way to find out when a `DELETE` query is needed. And before you know it you’ll be implementing your own ORM so it’s good to keep an eye on the cost/benefit ratio. In the past few years I personally have experienced some great benefits from writing my own mapping code, but make sure to fully consider your own context before taking this route.

11.4 Application services

The `Order` entity can be created, you can change its delivery address, cancel it, etc. These behaviors are represented by command methods, with an intention-revealing name. Only *Application services* should have access to these methods. An application service coordinates

the requested change. For instance, it creates the `Order` and saves it to the `OrderRepository`. Or it loads an `Order` entity by its ID, calls one or more methods on it, and saves it again (see Listing [11.7](#) for some examples).

Listing 11.7: Some application services.

```
final class CreateOrderService
{
    private OrderRepository $orderRepository;

    public function __construct(OrderRepository $orderRepository)
    {
        $this->orderRepository = $orderRepository;
    }

    public function __invoke(/* ... */): OrderId
    {
        $orderId = $this->orderRepository->nextIdentity();

        $order = Order::create(
            $orderId,
            /* ... */
        );

        $this->orderRepository->save($order);

        return $order->id();
    }
}

final class ChangeDeliveryAddressService
{
    private OrderRepository $orderRepository;

    public function __construct(OrderRepository $orderRepository)
    {
        $this->orderRepository = $orderRepository;
    }

    public function __invoke(OrderId $orderId /* ... */): void
    {
        $order = $this->orderRepository->getById($orderId);

        $order->changeDeliveryAddress(/* ... */);

        $this->orderRepository->save($order);
    }
}
```

An application service could have a single method (like the `__invoke()` method in Listing [11.7](#)). In that case the service represents a single use case as well. But since several use cases may share the same set of dependencies you may also create a single class with multiple methods (see Listing [11.8](#)).

Listing 11.8: An application service with multiple methods (use cases).

```
final class OrderService
{
    private OrderRepository $orderRepository;

    public function __construct(
        OrderRepository $orderRepository
    ) {
        $this->orderRepository = $orderRepository;
    }

    public function createOrder(/* ... */): OrderId
    {
        $orderId = $this->orderRepository->nextIdentity();

        // ...

        return $orderId;
    }

    public function changeDeliveryAddress(/* ... */): void
    {
        // ...
    }

    public function markAsDelivered(/* ... */): void
    {
        // ...
    }

    public function cancel(/* ... */): void
    {
        // ...
    }
}
```

11.4.1 Return the identifier of a new entity

Application service methods are command methods: they change entity state and shouldn't return anything. However, when an application service creates a new entity (like the `createOrder()` method does in Listing 11.7), you may still return the ID of the new entity. One thing an application service definitely should not return is the complete entity. As explained before, an entity is a write model with built-in behaviors for changing its state. It should not be available to clients that don't want to change its state. Clients of application services are usually controllers and they certainly shouldn't change entity state.

What if you want to return some information about the entity in the response? In that case use the ID that was returned by the application service to fetch a view model from its view model repository (Listing 11.9).

Listing 11.9: Fetching a view model inside the controller.

```

final class OrderController
{
    // ...

    public function createOrderAction(Request $request): Response
    {
        $orderId = $this->orderService->createOrder(/* ... */);

        $order = $this->orderDetailsRepository->getById($orderId);

        return $this->templateRenderer->render(
            'order-details.html.twig',
            [
                'order' => $order
            ]
        );
    }
}

```

See Section [3.5](#) for more information about view models.

11.4.2 Input should be defined as primitive-type data

The client of an application service is usually some kind of a controller, like a web controller or a console command. Another client could be a test that invokes the service. We'll see an example of this in Chapter [14](#).

To make an application service fully portable, allowing any type of client to use the service, you should make the input arguments easy to create. The best way to do that is to use primitive-type parameters only (see Listing [11.10](#)).

Listing 11.10: This application service accepts simple, easy to create argument.

```

final class OrderService
{
    // ...

    public function changeDeliveryAddress(
        string $orderId,
        string $address,
        string $postalCode,
        string $city,
        string $country
    ): void {
        // ...
    }
}

```

11.4.3 Wrap input inside command objects

You can introduce *Parameter object* which combines all the parameters in a single object (see Listing 11.11).

Listing 11.11: Using a parameter object.

```
final class ChangeDeliveryAddress
{
    private string $orderId;
    private string $address;
    private string $postalCode;
    private string $city;
    private string $country;

    public function __construct(
        string $orderId,
        string $address,
        string $postalCode,
        string $city,
        string $country
    ) {
        $this->orderId = $orderId;
        $this->address = $address;
        $this->postalCode = $postalCode;
        $this->city = $city;
        $this->country = $country;
    }

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

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

    // ...
}

final class OrderService
{
    public function changeDeliveryAddress(
        ChangeDeliveryAddress $command
    ): void {
        // ...
    }
}
```

A parameter object for an application service is a data-transfer object (DTO). In the controller you should copy the data from the request into the DTO, which then carries it to the application service. Use the class name of the DTO to communicate intent. If the data is going to be used to change the delivery address of an order, the name should be `ChangeDeliveryAddress`. This makes it a *Command object*⁸⁹ and the application service becomes a *Command handler*.

When copying data from the request into the command object, don't throw exceptions when you encounter bad input (as discussed in Section [8.6](#)). The only thing you have to do is ensure that the request data is cast to the correct types, that all the fields have a value assigned to them or are `null` if that's an allowed value (see Listing [11.12](#)).

Listing 11.12: Using a DTO to define a shape for input data.

```
final class CreateOrder
{
    use Mapping;

    // ...

    public static function fromrequestData(array $data): self
    {
        return new self(
            self::getString($data, 'email'),
            self::getInt($data, 'ebook_id'),
            self::getInt($data, 'quantity'),
            self::getNonEmptyStringOrNull($data, 'buyer_name')
        );
    }
}
```

11.4.4 Translate primitive input to domain objects

An application service has to translate the primitive-type values from the DTO to the value objects that the entity can work with. Listing [11.13](#) shows how the application service takes the input and creates a value object which can be passed to the `Order::changeDeliveryAddress()` method.

Listing 11.13: Translating primitive-type values to rich domain objects.

```
final class OrderService
{
    private OrderRepository $orderRepository;

    // ...
    public function changeDeliveryAddress(
        ChangeDeliveryAddress $command
    ): void {
        $order = $this->orderRepository->getById(
            OrderId::fromString($command->orderId())
        );
        // ...

        $order->changeDeliveryAddress(
            DeliveryAddress::fromScalars(
                $command->address(),
                $command->postalCode(),
                $command->city(),
            )
        );
    }
}
```

```

        $command->country()
    );
// ...
}
}

```

This helps keep the knowledge about how to deal with domain objects inside the core of the application, and not in infrastructure code like the controller. It also ensures that entities and value objects will only throw exceptions once the application service has been invoked. This way, the controller still has a chance to validate the command object itself and show form errors to the user.

However, application services tend to become long lists of these primitive-value-to-value-object transformations, which obscures the view on the the actual use case that the service represents. As an alternative you can add accessor methods to the command object which instantiate and return the correct value objects themselves (see Listing [11.14](#)).

Listing 11.14: Translating primitive values to rich domain objects inside the DTO.

```

final class ChangeDeliveryAddress
{
    private string $orderId;
    private string $address;
    private string $postalCode;
    private string $city;
    private string $country;

    public function orderId(): OrderId
    {
        return OrderId::fromString($this->orderId);
    }

    public function deliveryAddress(): DeliveryAddress
    {
        return DeliveryAddress::fromScalars(
            $this->address,
            $this->postalCode,
            $this->city,
            $this->country
        );
    }
}

final class OrderService
{
    private OrderRepository $orderRepository;

    // ...

    public function changeDeliveryAddress(
        ChangeDeliveryAddress $command
    ): void {

```

```

    $order = $this->orderRepository->getById($command->orderId());
    // ...
    $order->changeDeliveryAddress(
        $command->deliveryAddress()
    );
    // ...
}
}

```

This approach has several advantages:

1. There's less noise inside the application service because it doesn't have to deal with all the type conversions itself.
2. The getters on the command DTO can be called multiple times inside the application service. There's no need to duplicate the instantiation logic.

A possible downside is that you could accidentally trigger a domain-level exception inside the controller by calling one of those getters. In practice I find that this doesn't get in the way and is just something to be aware of.

11.4.5 Add contextual information as extra arguments

Contextual data like the current user's ID, data from the current HTTP request, etc. should not be fetched when needed, nor should it be injected as constructor arguments of the application service. Instead, contextual information should always be provided as method arguments (as discussed in Section [5.5](#)). If you want to store the current user's ID on the Order entity, make sure to pass it as an argument to the CreateOrderService (see Listing [11.15](#)).

Listing 11.15: Passing contextual information as part of the command DTO.

```

final class OrderService
{
    // ...

    public function changeDeliveryAddress(
        ChangeDeliveryAddress $command
    ): void {
        // The current user ID is part of the command data:
        $userId = $command->userId();
        // ...
    }
}

// In the controller:
$this->orderService->changeDeliveryAddress(
    ChangeDeliveryAddress::fromRequestData(
        $request->request->all(),
        $user->userId()
    )
)

```

)
)

11.4.6 Save only one entity per application service call

An application service should only make changes to a single entity. This can help improve the domain model's performance, both in terms of processing changes in the database, and in preventing concurrent updates. It helps keep your use cases focused on a limited area of the domain. There is just one thing that the service has to do.

11.4.7 Move secondary tasks to a domain event subscriber

A change in one entity often requires other things to be done as well. Maybe another entity needs to be updated too. Maybe you have to send someone an email about the change, or push a message to a queue. For these secondary effects use domain events, an event dispatcher, and event subscribers. Figure 11.2 shows how these elements work together.

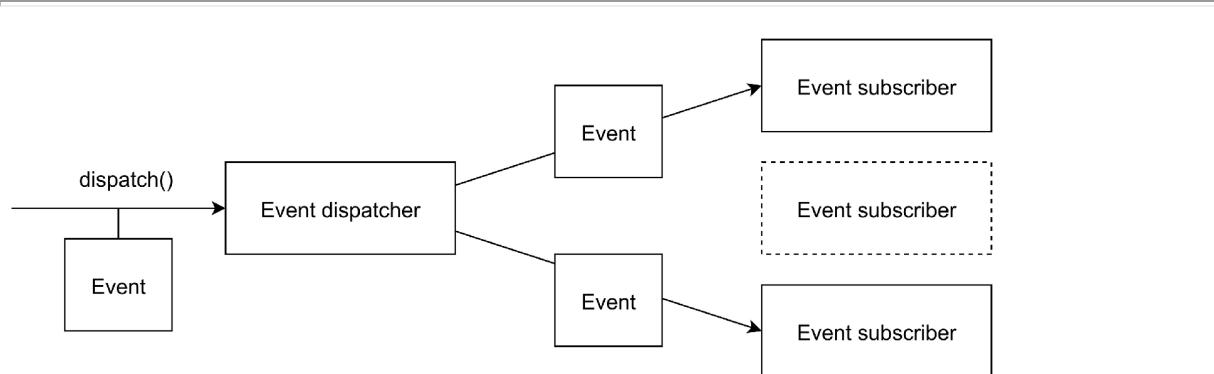


Figure 11.2: A client sends an event object to the event dispatcher. The dispatcher forwards the event to any subscriber that is known to be interested in events of that type.

Events are created inside the entity and after the entity itself has been saved they can be taken out by the application service and sent to the event dispatcher (as discussed in Section 4.5). So you shouldn't dispatch the events until you are certain that the entity's changes have been persisted. In order to dispatch events an application service should have the event dispatcher service as one of its dependencies (see Listing 11.16). It should be injected by its interface.

Listing 11.16: Dispatching domain events recorded inside the entity.

```
final class OrderService
{
    private OrderRepository $orderRepository;
    private EventDispatcher $eventDispatcher;

    public function __construct(
        OrderRepository $orderRepository,
```

```

        EventDispatcher $eventDispatcher
    ) {
    $this->orderRepository = $orderRepository;
    $this->eventDispatcher = $eventDispatcher;
}

public function changeDeliveryAddress(
    OrderId $orderId,
    ChangeDeliveryAddress $command
): void {
    // ...

    $order->changeDeliveryAddress(
        DeliveryAddress::fromScalars(
            $command->address,
            $command->postalCode,
            $command->city,
            $command->country
        )
    );
    $this->orderRepository->save($order);

    $this->eventDispatcher->dispatchAll(
        $order->releaseEvents()
    );
}
}

```

An application service may make multiple changes to an entity causing multiple events to be recorded and released. An entity may also record multiple events for a single action. This could happen when a change to an entity means different things to different observers. Or when an update brings the entity into some new state (see Listing 11.17).

Listing 11.17: Recording multiple events.

```

final class Order
{
    // ...

    public function markLineAsDelivered(int $lineNumber): void
    {
        $this->line($lineNumber)->markAsDelivered();
        $this->events[] = new LineDelivered($this->id, $lineNumber);

        if ($this->allLinesHaveBeenDelivered()) {
            $this->events[] = new OrderFullyDelivered($this->id);
        }
    }
}

```

11.5 Event subscribers

After saving an entity the event dispatcher should receive all the recorded domain events. The event dispatcher will then notify all the event subscribers that have been registered for that particular event. Listing 11.18 shows how a simple event dispatcher would do that. All subscribers in this example are provided as a constructor argument. Each subscriber is supposed to be a callable.

Listing 11.18: An EventDispatcher interface with a simple implementation.

```
interface EventDispatcher
{
    public function dispatchAll(array $events): void;
}

final class SimpleEventDispatcher implements EventDispatcher
{
    private array $subscribers;

    public function __construct(array $subscribersByEventType)
    {
        $this->subscribers = $subscribersByEventType;
    }

    public function dispatchAll(array $events): void
    {
        foreach ($events as $event) {
            foreach (
                $this->subscribersForEvent($event) as $subscriber
            ) {
                $subscriber($event);
            }
        }
    }

    private function subscribersForEvent(object $event): array
    {
        return $this->subscribers[get_class($event)] ?? [];
    }
}
```

Listing 11.19 shows how you'd register an event subscriber for the OrderFullyDelivered event.

Listing 11.19: CreateInvoice is an event subscriber which gets registered to the EventDispatcher service.

```
final class CreateInvoice
{
    private InvoicingService $invoicingService;

    public function __construct(
        InvoicingService $invoicingService
    ) {
        $this->invoicingService = $invoicingService;
```

```

    }

    public function whenOrderFullyDelivered(
        OrderFullyDelivered $event
    ): void {
        $this->invoicingService->createInvoiceFromOrder(
            $event->orderId(),
            /* ... */
        );
    }
}

$eventSubscriber = new CreateInvoice(/* ... */);

$eventDispatcher = new SimpleEventDispatcher(
    [
        OrderFullyDelivered::class => [
            [$eventSubscriber, 'whenOrderFullyDelivered']
        ]
    ]
);

```

Instantiating services and injecting them as dependencies is something a dependency injection container should do for you. The example just shows what the logic inside the container would look like.

11.5.1 Move subscribers to the module where they produce their effect

The class name of an event subscriber should describe *what* it's going to do, e.g. "create an invoice" (Listing 11.19). The methods of the event subscriber should describe *when* it's going to do this. Doing so allows you to move event subscribers to the area where they produce their effect. For instance, invoicing might be handled in a completely different part of the application. It would damage the ability to decouple modules from each other if the order module would reach out to the invoicing module and start calling methods there. Figure 11.3 shows how doing so establishes a dependency from the order module to the invoicing module.

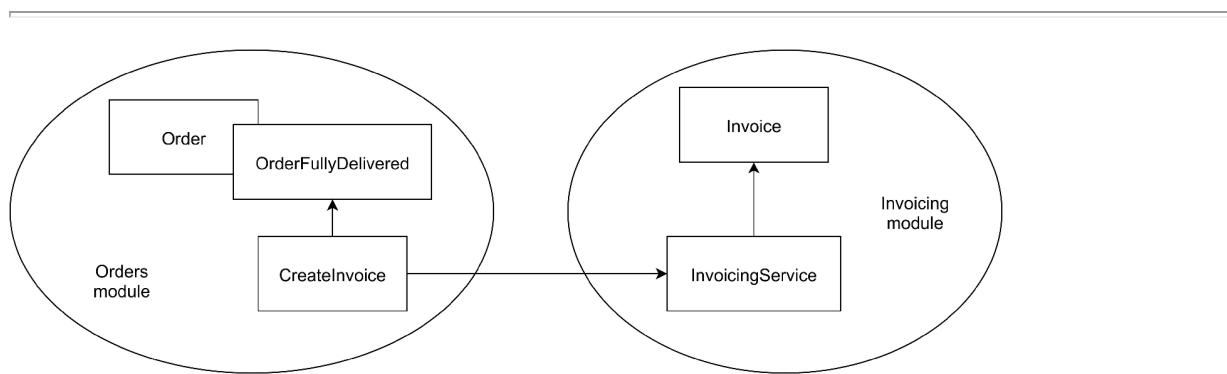


Figure 11.3: The CreateInvoice subscriber, which lives in the *Orders* module, uses the the InvoicingService from the *Invoicing* module, establishing a dependency from the *Orders* module to the *Invoicing* module.

The order comes first, and determines what needs to be invoiced. The invoice comes second, and it's based on data from the order. So the order module is *upstream*, the invoicing module is *downstream*. We should reflect that in the way we set up the event subscribers too. The `CreateInvoice` subscriber should live in the invoicing module and subscribe itself to events that are produced inside the order module. Whenever the `OrderFullyDelivered` event occurs it starts creating the invoice, which is an entity managed by the invoicing module. By arranging things that way the orders module doesn't have to know anything about the invoicing module. The invoicing module doesn't have to be explicitly told to create an invoice; it will respond to the fact that an order was fully delivered. Figure 11.4 shows how moving the `CreateInvoice` subscriber to the invoicing module establishes the desired dependencies between these modules.

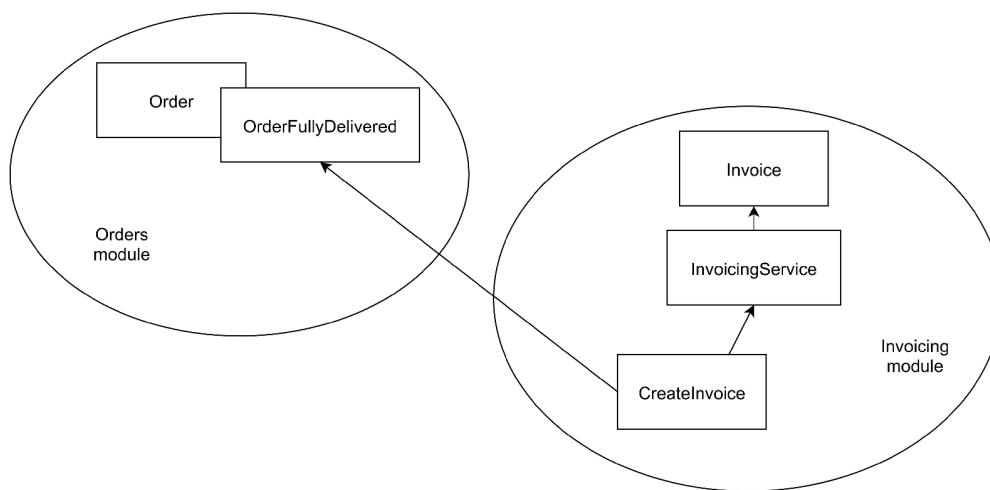


Figure 11.4: Moving the `CreateInvoice` subscriber to the *Invoicing* module, establishes the correct dependency direction: the *Invoicing* now depends on *Orders* instead of the other way around.

11.5.2 Delegate to an application service

The job of some event subscribers may be to respond to changes in one entity with an update of another entity. This would be a classic case of *Eventual consistency*: the system's state will be consistent only after all event subscribers are finished. Instead of making the change inside the event subscriber, delegate the call to an application service. This application service should then follow the standard pattern of retrieving an entity, making a change to it, and saving it again.

Most event subscribers should be independent from infrastructure and only contain core code because they are a crucial part of the use case. But there may be infrastructure-specific event subscribers in your application too. For instance, subscribers that log domain events, send them to a queue, store them in a database, or something like that. Infrastructure-specific event subscribers don't need to delegate their job to an application service. In fact, they can't, because an application service can't do any infrastructure work; it's supposed to be core code. So with

infrastructure-level event subscribers you just do whatever you need to do inside the event subscriber itself. Of course, you can always delegate some of the work to other (infrastructure) services that you inject as constructor arguments of the event subscriber.

11.6 Read models

While application services deal with entities, which are write models, a client that needs information from an entity shouldn't use the entity itself but a dedicated *Read model* instead. There are internal and external read models (which are often called view models, see Section [11.6.3](#)). Let's start with internal read models.

11.6.1 Use internal read models when you need information

Given a client and its need for information, start by defining a new type of object that would be able to provide this information. For example, the `InvoicingService` is going to create an invoice for an order so it needs to know a few things about the order. In `InvoicingService` act as if it already existed (see Listing [11.20](#)).

Listing 11.20: `InvoicingService` acts as if an `Order` read model already exists.

```
final class InvoicingService
{
    // ...

    public function createInvoiceForOrder(OrderId $orderId)
    {
        // ...

        $invoice = Invoice::create(
            $order->customerId(),
            $order->billingAddress()
        );

        foreach ($order->lines() as $line) {
            $invoice->addLine(
                $line->productDescription(),
                $line->quantity(),
                $line->tariff()
            );
        }
        // ...
    }
}
```

Your IDE will help you generate the outline for the new object by automatically creating the classes and methods that don't exist yet (see Listing [11.21](#) for the result).

Listing 11.21: The resulting `Order` read model.

```

final class Order
{
    // ...

    public function customerId(): CustomerId
    {
        // ...
    }

    public function billingAddress(): string
    {
        // ...
    }

    /**
     * @return array<Line>
     */
    public function lines(): array
    {
        // ...
    }
}

final class Line
{
    // ...

    public function productDescription(): string
    {
        // ...
    }

    public function quantity(): int
    {
        // ...
    }

    public function tariff(): Money
    {
        // ...
    }
}

```

Note that the `Order` read model is part of the *Invoicing* module. That way, *Invoicing* is the owner of the object's API so it can be easily modified to meet future needs of the `InvoicingService`.

The `InvoicingService` should be able to retrieve an instance of the `Order` read model from a *Repository*, which should also be owned by the *Invoicing* module. To separate the what from the how of retrieving a read model object, first create a repository interface on which `InvoicingService` can depend (see Listing 11.22).

Listing 11.22: The `InvoicingService` gets an `Order` object from the `OrderRepository` interface.

```

interface OrderRepository
{
    public function getById(OrderId $orderId): Order;
}

final class InvoicingService
{
    private OrderRepository $orderRepository;

    public function __construct(OrderRepository $orderRepository)
    {
        $this->orderRepository = $orderRepository;
    }

    public function createInvoiceForOrder(OrderId $orderId)
    {
        $order = $this->orderRepository->getById($orderId);

        $invoice = Invoice::create(
            $order->customerId(),
            $order->billingAddress()
        );
    }
}

```

Separating the interface and the implementation gives you a lot of flexibility. In a test scenario you can replace the `OrderRepository` dependency of the `InvoicingService` with a simpler, faster version. We can let it return any `Order` read model object we need.

Figure 11.5 shows the dependencies between the `Order` and `Invoicing` modules.

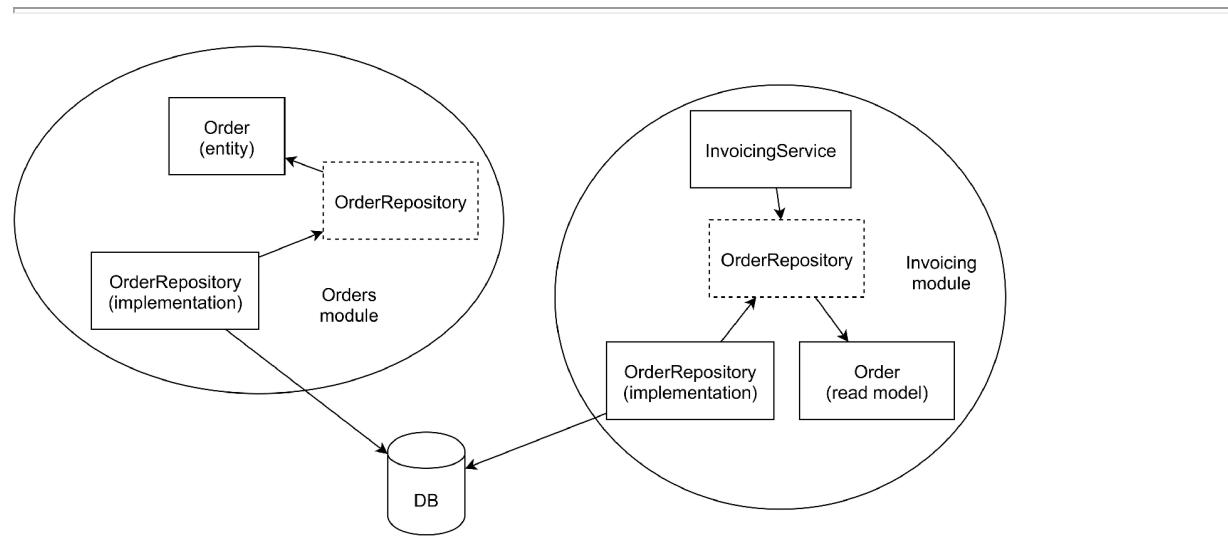


Figure 11.5: In this example, there is no code-level dependency between `Invoicing` and `Orders`. However, both modules use the same database to get their data.

11.6.2 Choose a standard implementation for the repository

The read model repository implementation will fetch the required data and create the desired `Order` read model objects. It can do this in several ways, for example:

- The repository can use the same database as the one used by the write model. In that case the repository will copy data from the relevant tables and populate the read model object with it.
- The application can build up a read model based on domain events from the write model. The repository would then return the current state of the read model.

We've already discussed these options in detail in Chapter [3](#).

Whatever changes are made outside of the module, as long as the repository implementation is able to provide the right read model objects, everything should be fine. It's like the *Dependency Inversion Principle* applied to models. Even if the `Orders` module gets replaced by a third-party platform for selling e-books, the `Invoicing` module doesn't need to suffer. The only thing that needs to be done is rewrite the `OrderRepository` implementation to use the third-party platform's API to retrieve information about an order (see Figure [11.6](#)). This makes the use of read models as local representations of remote entities a very powerful architectural technique.

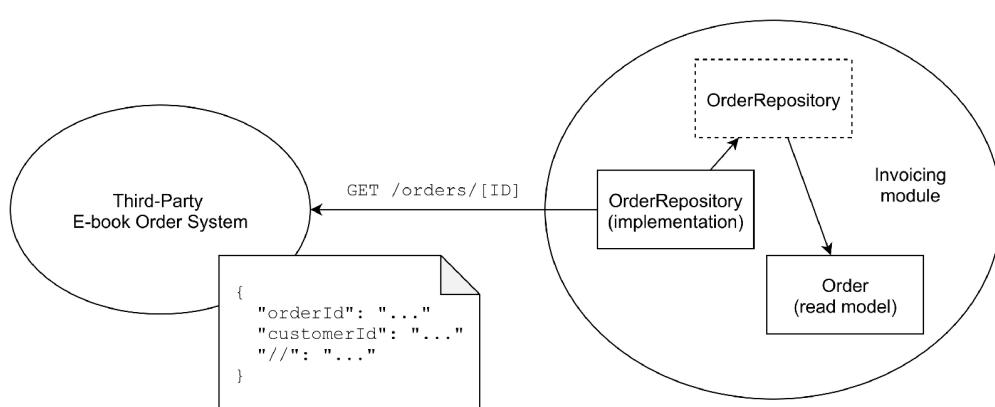


Figure 11.6: In this example, `Orders` has been completely replaced by a third-party system. `Invoicing`'s `OrderRepository` implementation now uses that system's API to fetch order information.

11.6.3 For view models, prepare the data for rendering

Besides internal read models there are also outward-facing models which expose data to primary actors. In our example, `Invoicing` may have an HTTP API too, allowing clients to fetch a JSON representation of an invoice. In that case, `Invoicing` manages the invoice data because it owns the `Invoice` entity but it also provides a view on that data. In fact, it provides multiple views because it also provides a list of unpaid invoices on the website, and allows users to

download an invoice as a PDF file. Figure 11.7 shows these different view models that *Invoicing* offers to its actors.

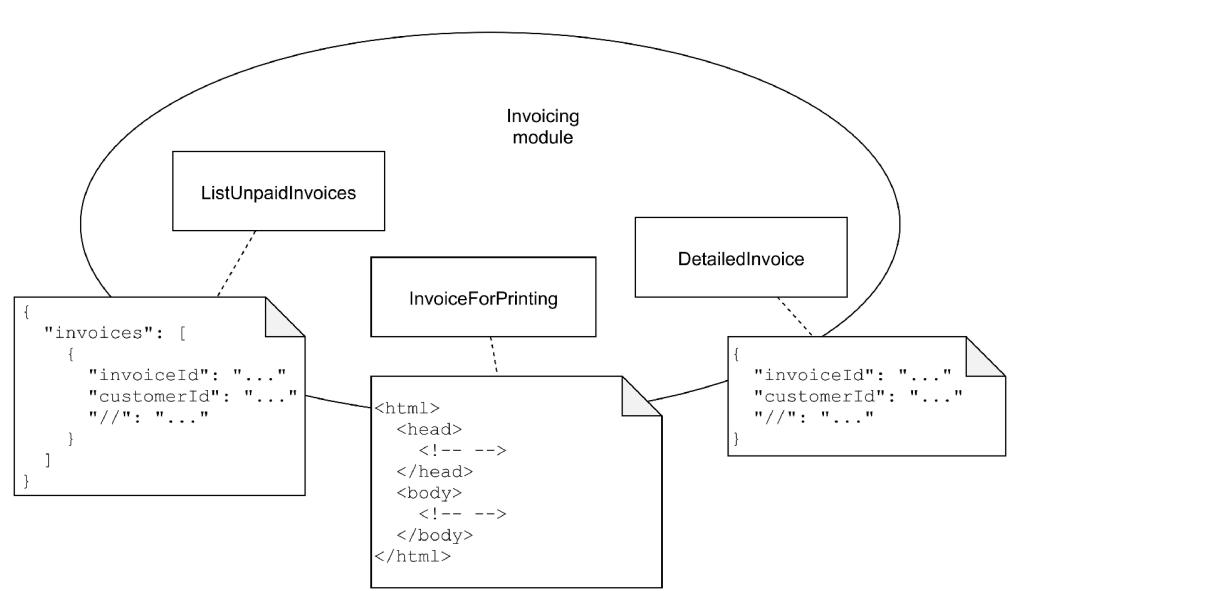


Figure 11.7: *Invoicing* offers several ways for clients to learn more about the data it manages.

The view models themselves need to be fully prepared for the view that renders them to the user. For instance, if the view model is going to be used to render an HTML template, make sure that the template renderer doesn't need to do much more than just echo a couple of properties, loop over some property, echo some more properties, etc. If the view model is going to be rendered as a JSON object, make sure that the object can be encoded to JSON in a single step.

11.7 Process modelling

We started this chapter with a discussion about framework-inspired architecture. Frameworks propose a number of elements that should be used to build your application: controllers, models, templates, etc. In the previous sections we discussed an alternative set of elements that can also be used to build your application: entities, repositories, application services, events, event subscribers, read models, and view models. The difference of course is that when using elements like application services and entities as building blocks, your design doesn't depend on a framework nor on any other piece of infrastructure.

This new set of elements clearly describes the use cases of your application: what you can do with it, what the consequences are, and which information the application exposes, without talking about implementation details. Being able to leave out implementation details is a sign that we can use these elements in high-level modelling sessions. You'll find this modelling technique explained by Alberto Brandolini in “Introducing Event Storming”⁹⁰. There he calls it “Process Modelling”. In my experience it's a very useful technique. It fits well between higher-level design sessions where the focus is on the problem domain, and lower-level design sessions

where the programmers want to take a step in the direction of the solution. In a process modelling session they can use domain concepts and knowledge about the desired process and create a useful model for the software based on design patterns that are familiar to them. Figure ?? can be used as a reference for such a session.

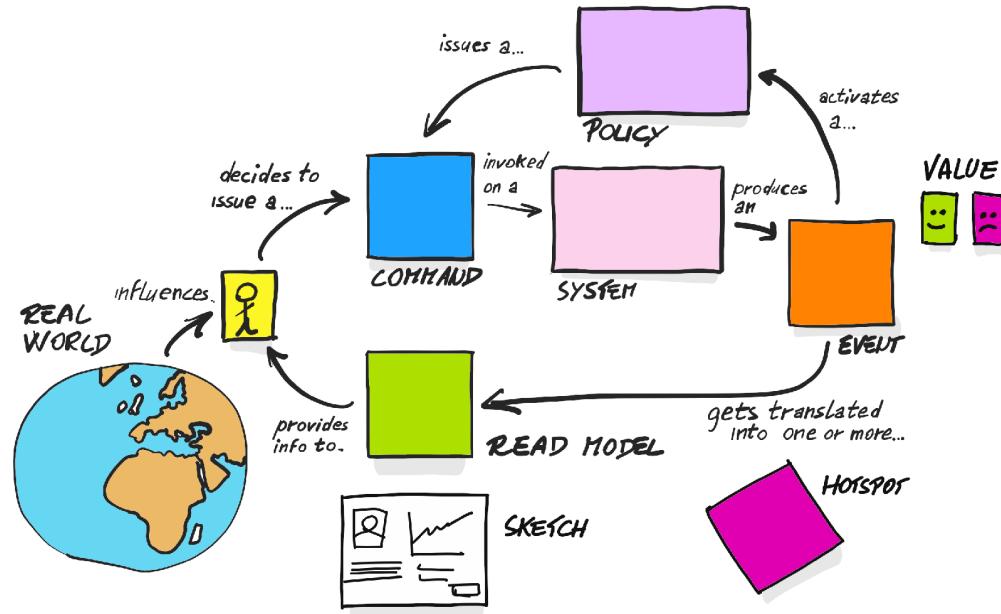


Figure 11.8: The picture that explains everything (copied with permission).

After years of developing software, for me it was a pretty revolutionary idea to consider the user as someone who is influenced by the real world, who lets the system inform them about something, and who then makes a decision based on this information. I realized that retrieving information from a system should be considered an important use case, just as important as a use case where the user decides to do something. I also realized that to successfully decouple from infrastructure both types of use cases should exist somewhere in the core code of the application.

As shown in Figure ??, during a process modelling session you will design commands, events, read models, effects, and decisions (policies). I personally like to take it one step further by zooming in on that “System” box. What happens inside the system should be an implementation detail and it doesn’t matter for the overall process. But since we have established some useful patterns, like application services and event subscribers, we can add those elements to the diagram as well. Figure 11.9 shows what’s inside the system. If you like, you can use an adapted process modelling session to describe these elements too.

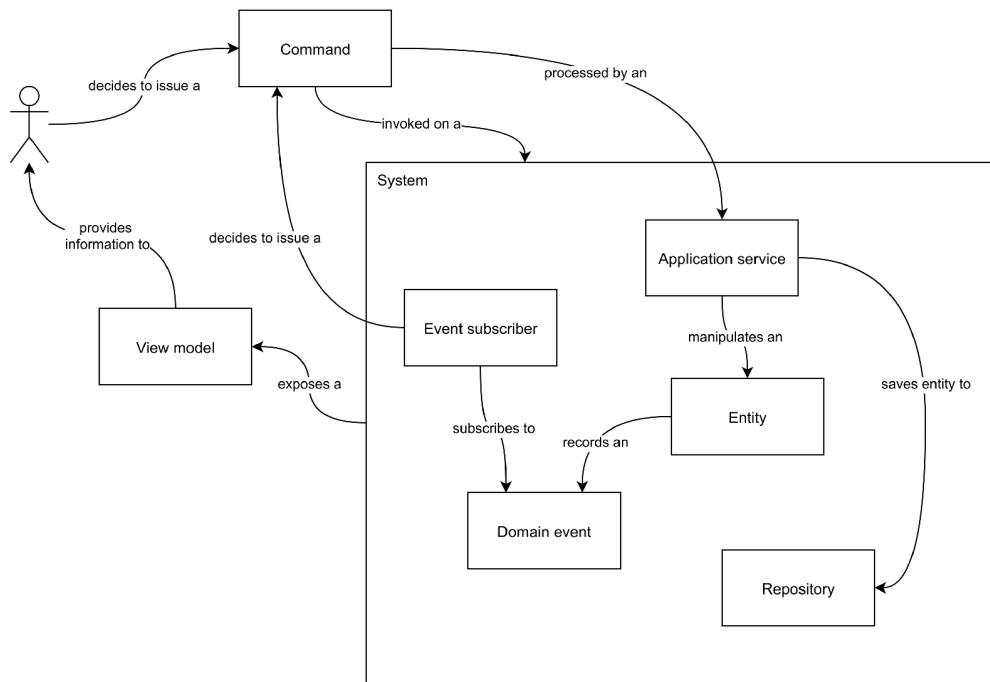


Figure 11.9: Modelling the elements that are inside the system.

Once you know the elements of the process, how they are named, and how they interact, you also have a rough idea about how to implement those elements because you know their underlying design patterns. So you can start working on the implementation with less fear and fewer doubts. Implementing use cases becomes more like following a recipe. Understanding use cases implemented by others also becomes easier, because you recognize the same patterns in their work.

Finally, because these design patterns are decoupled from infrastructure by design, they will come in very handy when you start specifying and testing your use cases using scenarios. We'll see how this works in Chapter [14](#).

11.8 Summary

In this chapter we took a closer look at some of the design patterns we discovered in Part I: entities, repositories, application services, read and view models, domain events, and event subscribers. Using these patterns in your application will automatically make it easier to keep core and infrastructural code separated. They allow you to clearly define all the use cases of your application, without mixing in any infrastructural concerns. These use cases are represented in code by:

1. *Application services*, which create or manipulate an entity, save it to the entity's repository, and dispatch domain events produced by the entity.
2. *View models*, which provide a useful representation of the application's data.

Event subscribers act as a bridge between a primary change, and any number of intended effects of that change.

You can use these design patterns in a process modelling session to find out which elements you need to build.

12 Architectural layers

This chapter covers:

- The Domain, Application, and Infrastructure layer
 - The Dependency rule
 - Using namespaces as a way to make layers visible
 - Verifying layer conventions with tools
-

12.1 MVC

Web frameworks often suggest something like an “MVC architecture” for your web applications. MVC stands for Model View Controller. Yes, you’ll need each of these things in a typical web application. You need a controller to process an incoming web request, and you need a view to present some information to the user. A good web framework should offer some nice tools for doing these things. It may offer convenient ways of extracting relevant data from the request, it may manage the session for you, and it may offer a cool templating engine that makes rendering HTML responses a breeze.

This is all great, and frameworks usually do a good job at this. But when it comes to making the jump from the controller to the model, things are likely to go wrong. As we saw in Chapter 4, controllers end up containing all the business logic for a given use case, and the model is likely going to be a simple data holder (see Figure 12.1).

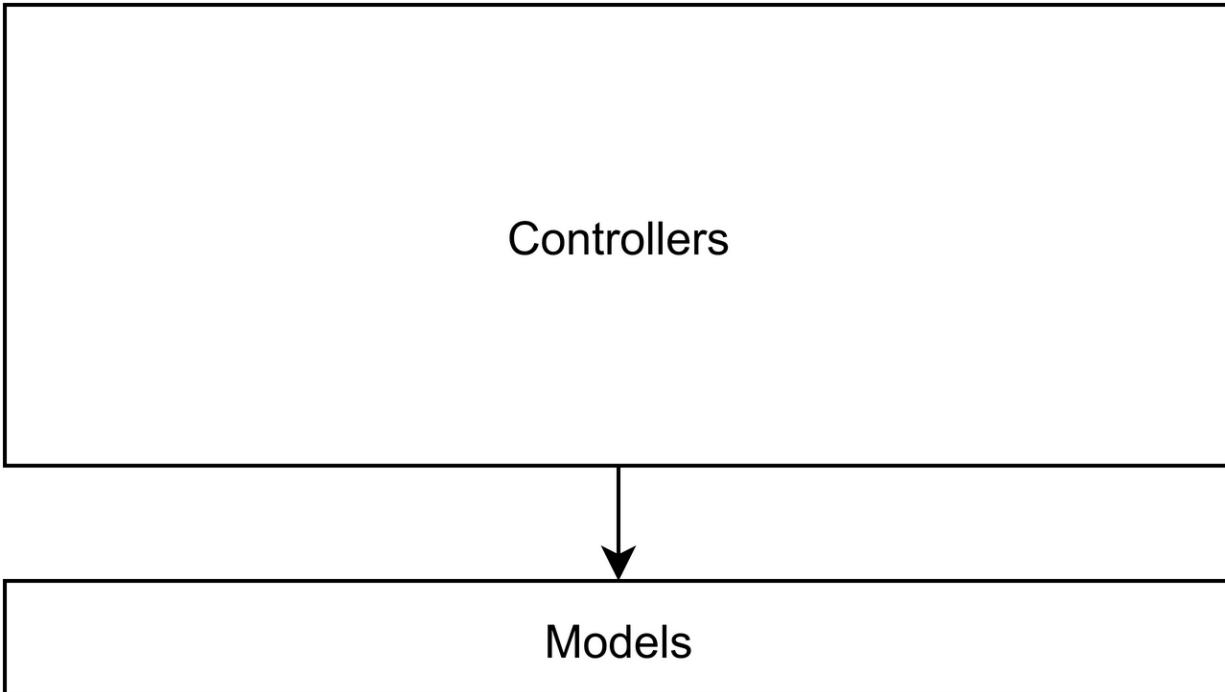


Figure 12.1: Controllers contain all business logic and become too big.
Models are only simple data holders.

Often developers have realized the shortcomings of this approach, and started moving business logic into services. In several projects that I've seen so far this move hasn't been very successful because the extracted services aren't truly decoupled from the framework. They still rely on the session, or the current web request. The methods in these services are often grouped around a certain domain concept and try to manage both the domain logic and the persistence logic for this particular concept. These services (sometimes even called "managers") then grow too large and become unmanageable. Another reason for failure in this area is that domain models still remain simple data holders and can't protect any of their invariants, nor implement any business rule all by themselves. This again results in services that have too much to do, and grow too large to deal with (see Figure 12.2). So the reason to start using services is a good one (we don't want our controllers full of business logic), but usually it's the implementation that's lacking. Without decoupling the service from the framework, without separating domain logic from persistence logic, and without taking the extra step of defining richer domain objects, the result will be disastrous.

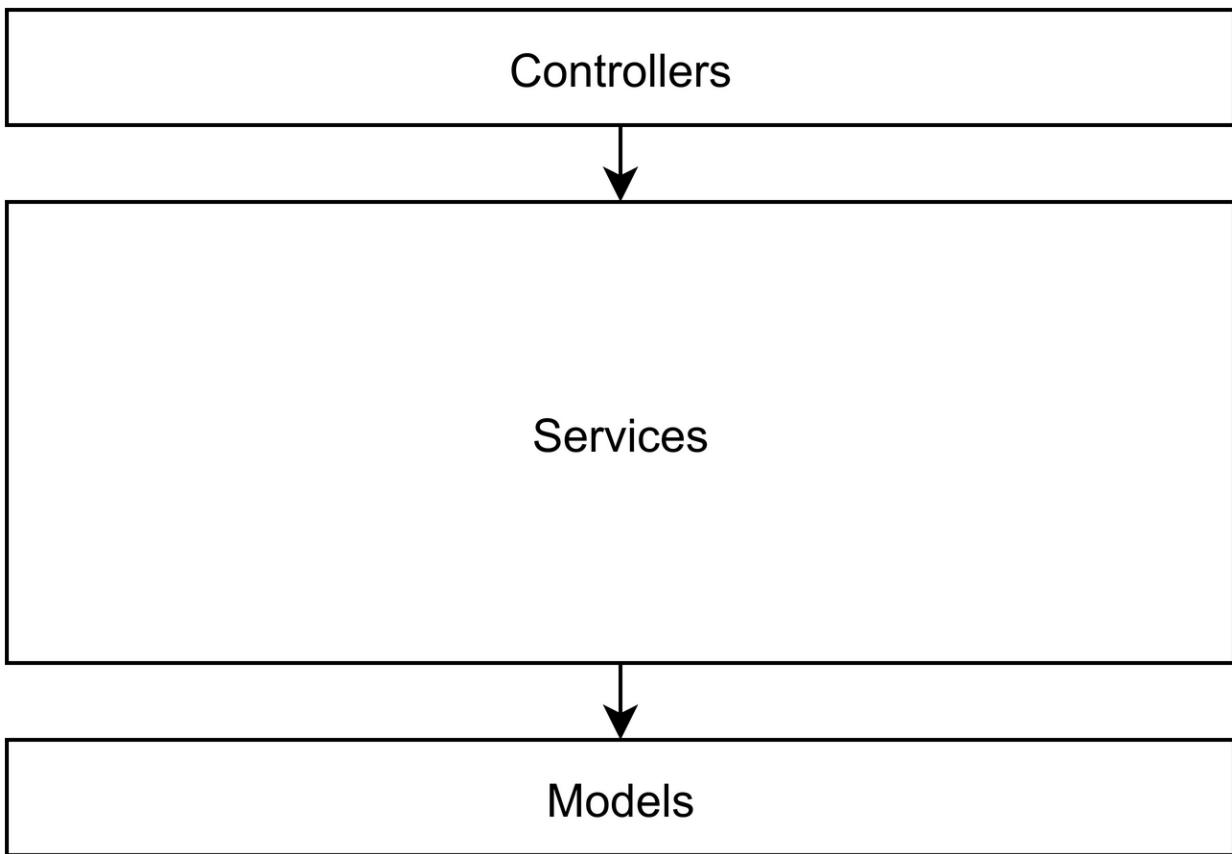


Figure 12.2: Controllers are small, as they should be, but services become too big. Models are still simple data holders.

So even after inventing an extra “service layer”, MVC is still lacking. For example, it doesn’t help us separate a use case in a primary action and its secondary effects. Services usually end up having large methods that start processing a request and perform all of the secondary tasks in the same method (or if you’re “lucky” in private methods of the same class). These services may even reach out to remote parts of the code base to do remotely related jobs. In the previous chapter we saw how to improve this situation by letting an application service take the first step, then dispatch events, to which event subscribers can respond by taking further steps. But MVC, with the added service layer, can’t help us with this.

At this point we should conclude that MVC isn’t a sufficient organizational principle for web applications. There is just too much that doesn’t have a

natural place within the categories of models, views, and controllers.

12.2 A standard set of layers

In this chapter I propose a set of layers, consisting of a *Domain*, *Application*, and *Infrastructure* layer. They provide broad categories for your code, helping you find the right place for each class in your application. Combined with the *Dependency rule*, they can even help you verify that you're correctly separating core code from infrastructure code.

In previous chapters we've made a distinction between core code and infrastructure code. Given the proposed set of layers, infrastructure code naturally belongs to the *Infrastructure* layer. The other two layers, *Domain* and *Application* should only contain core code (see Figure 12.3).

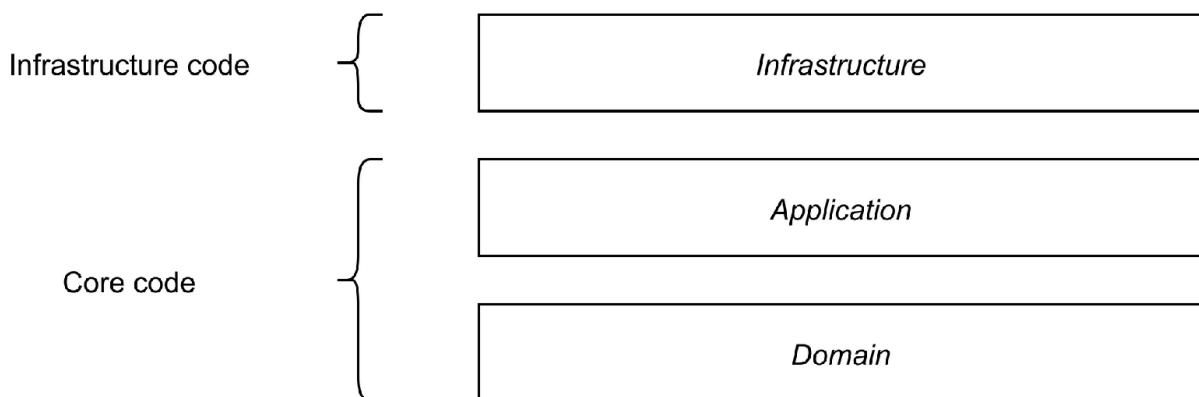


Figure 12.3: The infrastructure layer contains infrastructure code, the domain and application layer only core code.

12.2.1 The infrastructure layer

Every bit of infrastructure code we've encountered so far is going to end up in the infrastructure layer. This includes:

- Web controllers
- CLI commands
- Write and read model repository implementations

- Services that connect to external systems, like a remote API, or the file system
- Services that use the current time or generate random data

If you want to decide if code belongs in the infrastructure layer or in one of the other layers, you should compare the code to the definitions provided in Chapter [1](#):

1. Core code doesn't directly depend on external systems, nor does it depend on code written for interacting with a specific type of external system.
2. Core code doesn't need a specific environment to run in, nor does it have dependencies that are designed to run in a specific context only.

And as you know: if the code isn't core code, it's infrastructure code, and then it should go into the infrastructure layer. If you don't want it to be infrastructure code, you can refactor it using any of the techniques demonstrated in Part [I](#). After doing so you are “allowed” to move the code to one of the other layers.

12.2.2 The application layer

The application layer is the first layer that's free of infrastructure code. This layer includes:

- Application services/command handlers, and command DTOs
- View model repository *interfaces*, and view model DTOs
- Event subscribers that listen to domain events and perform secondary tasks
- Interfaces for infrastructure services

Looking at the classes in the application layer you should be able to recognize:

- What actors can do with your application and what data an actor has to provide for each task (represented by the application services and their method parameters, which could be command DTOs).

- What an actor can learn from your application (represented by the view model repository interfaces and the view model objects).
- How different use cases are connected to each other (represented by the event subscribers).
- On which things in the outside world your use cases depend (represented by the interfaces for infrastructure services).

12.2.3 The domain layer

The domain layer is also a layer without any infrastructure code. It contains:

- Entities
- Value objects
- Domain events
- Entity (write model) repository interfaces
- Domain services

These domain objects should be considered implementation details of the application layer. In fact, most of these details should stay behind the application layer. The infrastructure layer generally shouldn't be concerned with anything that's going on in the domain layer. Infrastructure code, like a web controller, should mainly have to deal with primitive-type data (or DTOs containing primitive-type data) when it communicates with the application layer. The application layer will use code from the domain layer to perform its task, and it will know about the rich domain objects it contains.

“Is that extra subdivision of core code into *Domain* and *Application* code really necessary?”

Great question. After all, the biggest win in terms of testability and life expectancy of your application comes from separating core from infrastructure code. The separation between application and domain code is not strictly necessary, and doesn't improve either of these quality aspects. However, I like to keep the distinction because it helps me clarify what the

use cases of my application are. What can an actor do with the application? What information can they retrieve from it? Without a separate application layer, this would not be immediately clear. The reader would see entities, but wouldn't know what an actor could do with them. They would see a repository, but they wouldn't know if the information exposed by it will end up being presented to a user, or if it's only for internal use.

12.2.4 Up and down the layer stack

In most cases, the flow of data through these layers in a runtime application is as follows (see Figure [12.4](#)):

1. The web framework (infrastructure layer) accepts an incoming HTTP request.
 2. It analyzes the request and finds the right controller (infrastructure layer) to call.
 3. The controller creates a DTO based on the data from the request and calls an application service (application layer).
 4. The application service creates or modifies an entity (domain layer).
 5. The application service then hands over the entity to its repository (infrastructure layer), in order to save it.
-

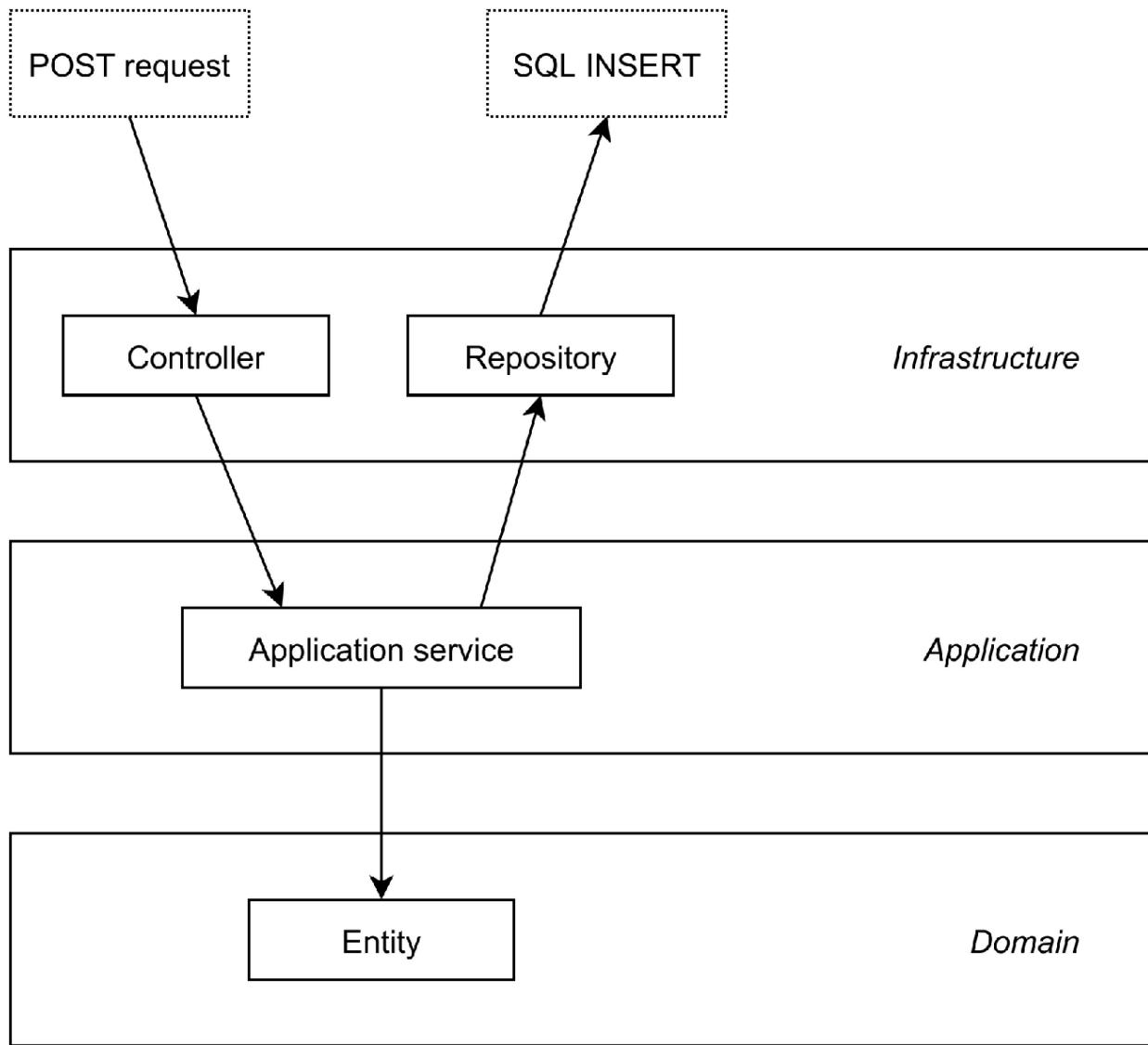


Figure 12.4: How data commonly flows through the layers.

There's a difference between the direction of the data in an application and the direction of dependencies in the code. For instance, the repository implementation is from the *Infrastructure* layer. So when an application service saves an entity, the data flows from the *Application* layer to the *Infrastructure* layer. But the application service class doesn't depend on an infrastructure class, it depends on an abstraction: the repository interface that lives in the *Domain* layer/ Figure 12.5 shows the resulting dependency diagram.

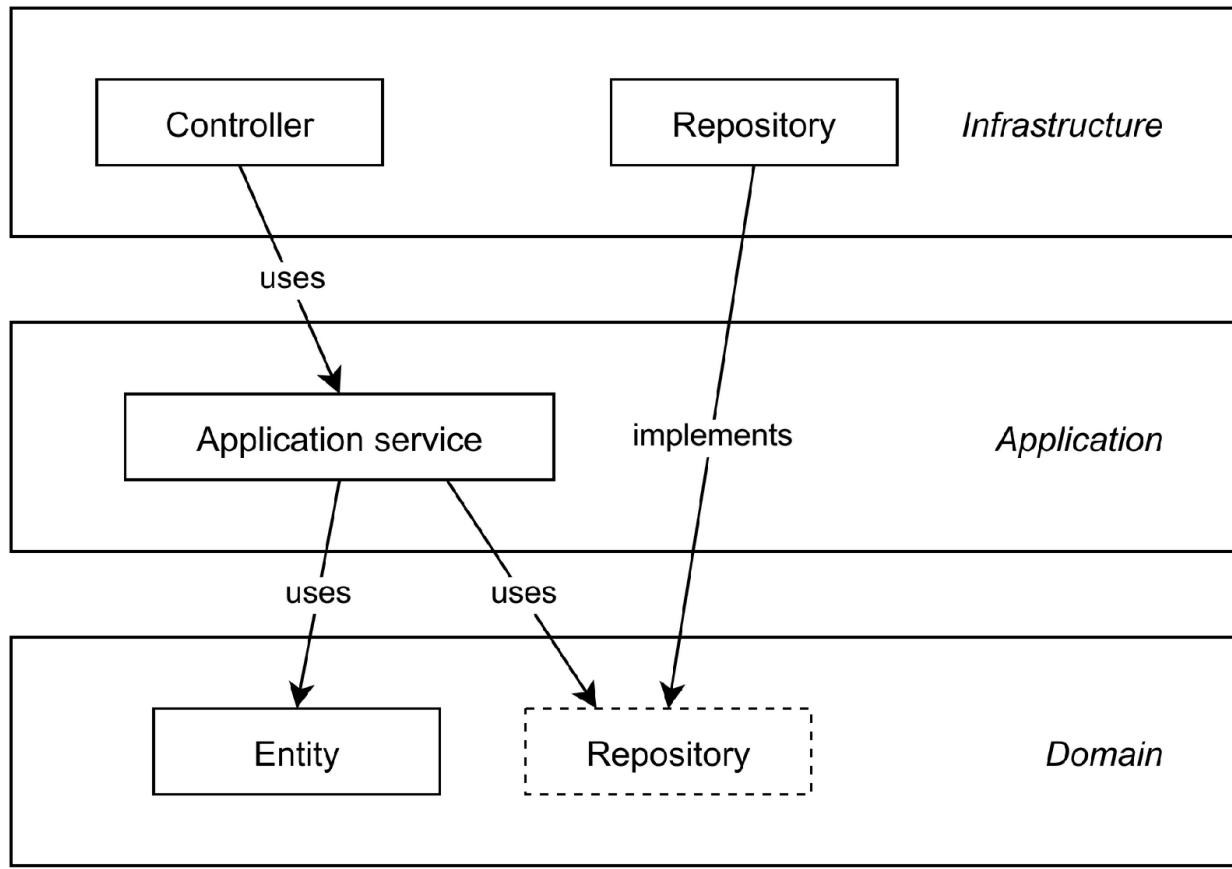


Figure 12.5: A dependency diagram showing dependencies between layers.

12.3 The Dependency rule

For class-level dependencies we have the *Dependency inversion principle*⁹¹, which tells us to depend on abstractions instead of concretions. In previous chapters we saw how applying this principle helps us separate core from infrastructure code. For layer-level dependencies, we have the *Dependency rule*⁹². This rule says that “source code dependencies [between layers] should only point inwards”. The word “inwards” hints at drawing layers as concentric circles instead of horizontal blocks. Once we redraw the previous dependency diagram (see Figure 12.5) using circles (see Figure 12.6), we can prove that none of the dependencies point inwards, so if we apply dependency inversion, the dependency rule will be followed as well.

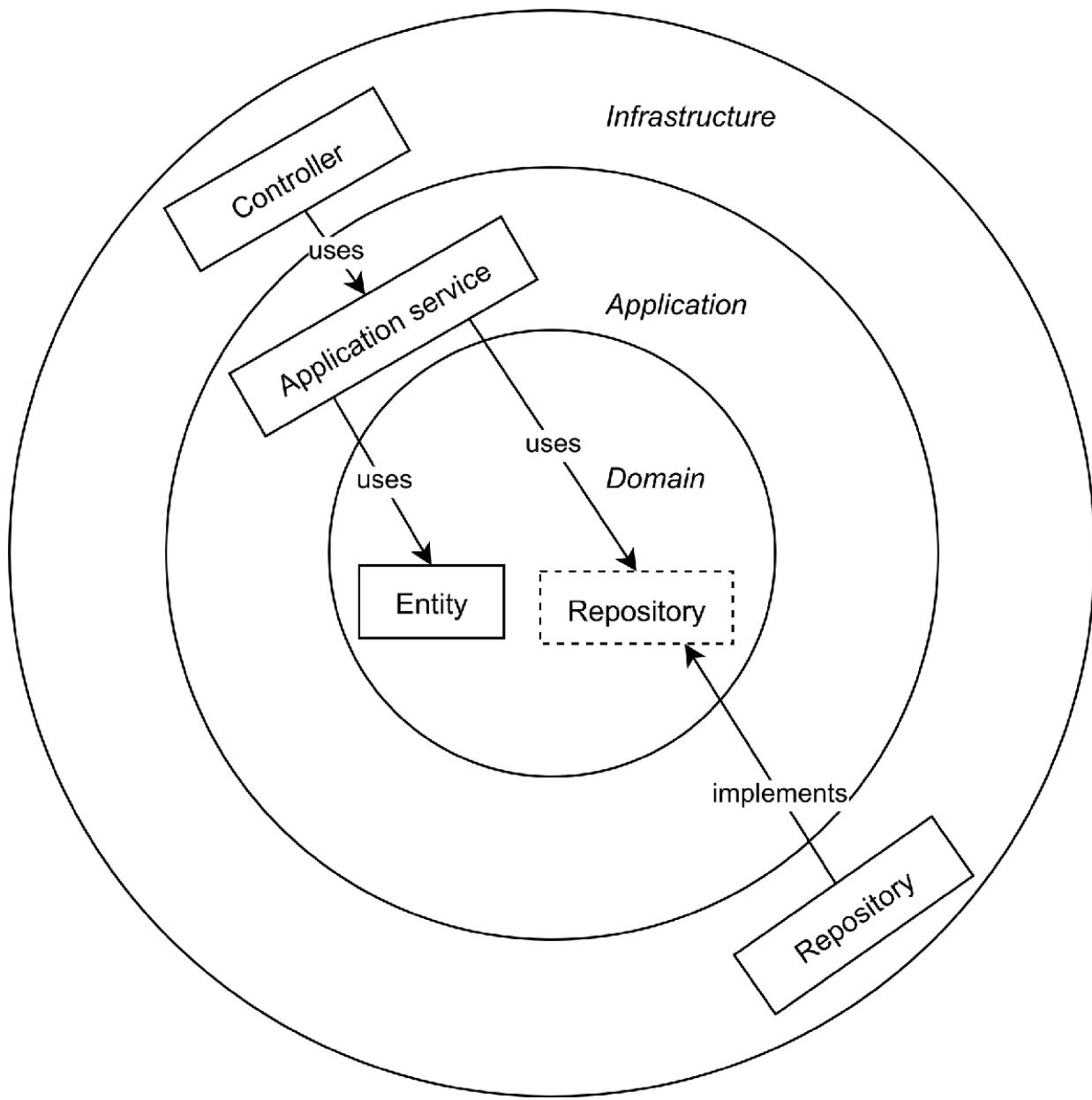


Figure 12.6: This modified dependency diagram using concentric circles shows that dependencies only point inwards.

The dependency arrows only point inwards: from *Infrastructure* to *Application*, from *Application* to *Domain*; never the other way around. This is a great design tool for a software architect like yourself, because you can reconnect your application to different types of actors without affecting the classes that are in the *Application* or *Domain* layer. If there are no dependencies upwards, this means we can rewrite or replace higher layers

without affecting the lower ones. This is also a great design tool for a software developer like yourself, because all the business logic represented by the *Application* and *Domain* layer can be tested in isolation from its surrounding infrastructure, by replacing only a few infrastructure implementations with fast and predictable test doubles.

12.4 Making layers tangible

We've been putting things into layers, we've defined a dependency rule for layers, but we never really talked about what a layer is. In fact, you could say a layer is nothing. You won't find a layer in your code, nor in your running application. It's just a way of grouping things. We say: these things belong to this layer, those things belong to the other layer. But once we've made this grouping, we can state the desired properties of these layers. For instance: "this layer will contain no infrastructure code", or "code dependencies between layers should only point inwards". That's why layering is an architectural activity. Layers are a high-level way of organizing our code, and it's one that influences the way we write our code.

12.4.1 Documenting the architecture

Layers are esoteric by nature: we can't see them in our code. We do work with them, base decisions on them (what to put where, which things can depend on other things), but we can't point them out in our code base. This makes it hard for people joining the project to find out what's going on. The high-level design choices we make about our application's architecture are choices we need to communicate. They need to be guarded and reinforced as well. If someone doesn't follow the dependency rule, or puts a class in the wrong layer, a code reviewer should be able to spot the issue without too much effort. Better yet, all team members should know about the layering system or they should be given the opportunity to find out about it. This means we have to make our layers visible, tangible. Of course, we can write a documentation page about it, and point team members to it. But this is not a feasible solution because we don't have satisfactory answers to these questions:

- Does everybody really read the documentation?

- Will they be able to understand it, and apply it to their own contributions?
- Who will make sure the documentation gets updated if it needs to?

It's better not to document the layering system separately. Instead, make sure the use of layers is apparent from the moment you take your first look at the code.

12.4.2 Using namespaces for layering

Layers are a way of grouping things. Those *things* are mostly classes, and classes can be grouped in namespaces. So using class namespaces as an indicator for the presence of a layering system would be a good idea. As an example, an `Order` class would be in the `Domain` namespace. The `EbookOrderService` class would be in the `Application` namespace. The `orderRepositoryUsingSql` class would be in the `Infrastructure` namespace.

Although a good solution, it's not a very scalable one. Given the amount of classes in each namespace we should make a subdivision within each layer namespace.

For starters, `Domain` could have two sub-namespaces, namely `Model` and `Service`. `Model` could have sub-namespaces for each entity (or actually, each aggregate) (see Figure [12.7](#) for an example).

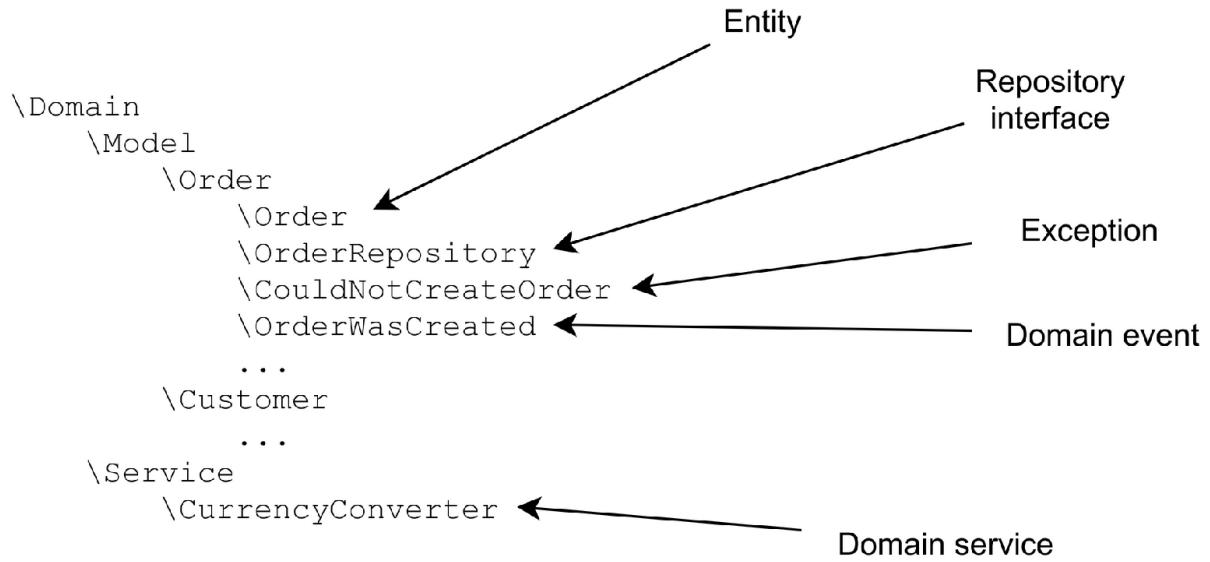


Figure 12.7: Subdividing the *Domain* namespace.

Note that this is really just an example. This layout is based on Vaughn Vernon’s “Implementing Domain-Driven Design”⁹³ and I have had some good results with it. You and your team can always settle on a different structure.

The `Application` namespace could have sub-namespaces for every use case it implements. You could provide actual phrases as the name for each use case. Make sure to include use cases for performing tasks (e.g. `createOrder`) as well as use cases that are about retrieving information (e.g. `ListAvailableEbooks`). In each sub-namespace you could keep the classes that are involved in handling the use case, like the application service itself, the command DTO, the view model repository interface, and the view model object.

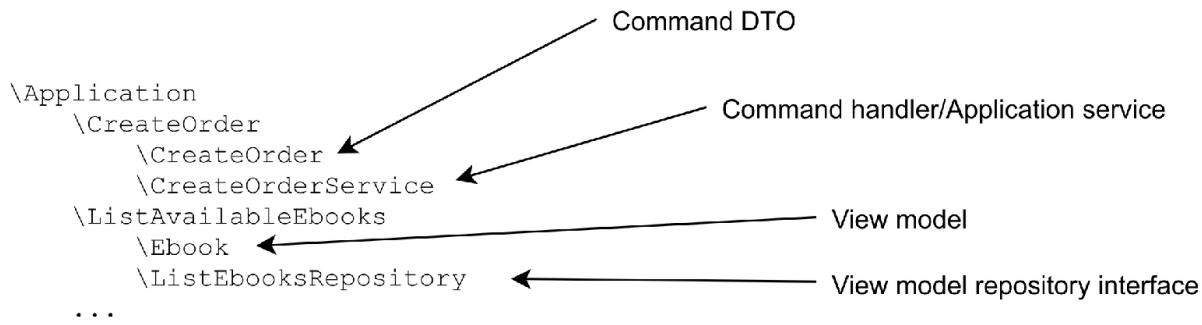


Figure 12.8: Subdividing the *Application* namespace.

The `Infrastructure` namespace needs some structure as well. There we could have sub-namespaces for every way in which the application is connected to the outside world. We'll get to that in Chapter [13](#), at the end of which we'll have a complete standard for the internal structuring of each layer.

12.4.3 Automated verification of design decisions

Once you've placed each class in a specific namespace, you can start verifying that the code actually follows the layering conventions we've established. For example, if you find an entity in the `Infrastructure` layer, it's in the wrong place. It belongs in the `Domain` layer. If the `Application` layer contains a class that makes an HTTP connection, you know that it should be moved to the `Infrastructure` layer instead. Also, looking at the source code dependencies between different classes, you can verify that all classes follow the *Dependency rule*. For instance, if you find a class in the `Domain` layer which uses a class from the `Infrastructure` layer, you know that the `Domain` layer depends on the `Infrastructure` layer. This is a violation of the *Dependency rule* (see Listing [12.1](#)).

Listing 12.1: Based on the `namespace` and `use` statements of a class we can check if the code follows the layer conventions.

```
namespace Infrastructure\Entity;
```

```
/*
```

```
* An entity doesn't belong in the Infrastructure layer
*/
final class Order
{
    // ...
}

namespace Application\RegisterUser;

final class WelcomeEmail
{
    public function __construct(Client $httpClient)
    {
        /*
         * An HTTP client doesn't belong inside the
         * Application layer
         */
        $this->httpClient = $httpClient;
    }

    // ...
}

namespace Domain\Service;

/*
 * A Domain class should not depend on a class from the
 * Infrastructure layer
 */
use Infrastructure\ExchangeRateProvider;

final class CurrencyConverter
{
    /**
     * @var ExchangeRateProvider
     */
    private $exchangeRateProvider;

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

To manually check all of this, and to keep checking it for every future change to our code base, would be a lot of work. However, this work could in theory be automated. We can tell a tool that a class in the `Domain`, `Application`, or `Infrastructure` namespace should be considered to be inside the layer corresponding to that name. The tool could then scan each class in the project and analyze its ‘namespace’ and ‘use’ statements. Based on this information the tool could trigger a warning if a dependency between layers goes in the wrong direction (e.g. from *Domain* to *Infrastructure*). If the tool would also look inside the code, it could warn us about the use of infrastructure code inside a layer that’s supposed to contain only core code (i.e. the *Domain* or *Application* layer).

As far as I know, there is no tool that can perform an “is this core or infrastructure code” check, meaning there is no tool to verify that code is in the correct layer.⁹⁴ However, there is a tool that can verify that the code follows the *Dependency rule*: `deptrac`⁹⁵ Listing 12.2 shows how you can configure it to recognize layering conventions based on namespaces. Under the `ruleset` key you can define the allowed dependency directions between layers.

Listing 12.2: Using this configuration `deptrac` can check if code follows the *Dependency rule*.

```
paths:
  - ./src
layers:
  - name: Infrastructure
    collectors:
      - type: className
        regex: .*\\Infrastructure\\.*
  - name: Domain
    collectors:
      - type: className
        regex: .*\\Domain\\.*
  - name: Application
    collectors:
      - type: className
        regex: .*\\Application\\.*
ruleset:
  Infrastructure:
    - Application
```

```
- Domain  
Application:  
- Domain  
Domain:  
# nothing
```

Running `deptrac` as part of your project's build can easily point out layering issues.

12.5 Summary

In this chapter we started out with a discussion about the MVC architecture for web applications and how it can't provide the necessary guidance in structuring our application. I then proposed a set of layers that provide a natural place for objects of all types:

- A *Domain* layer, which mainly consists of entities, value objects, and domain events.
- An *Application* layer, which contains classes that represent the application's use cases (what an actor can do with it and what information it can retrieve from it).
- An *Infrastructure* layer, which contains the code needed to connect the application to its primary and supporting actors.

We discussed the *Dependency rule*, which says that source code dependencies between layers can only go inward.

Using namespaces we can make our use of layers visible in the code base itself. Once every class has a namespace which corresponds to one of the proposed layers, we can let a tool automatically verify that every class follows the *Dependency rule*.

Exercises

1. What does the *Dependency rule* say?⁹⁶

1. Services should depend on abstractions
 2. Layers should only have inward dependencies
 3. Classes should be open for extension, closed for modification
2. Which of the following types of classes belong to the *Domain* layer?⁹⁷
1. Command DTO
 2. View model repository interface
 3. Entity/write model repository interface
 4. Entity
 5. Application service
3. Which of the following types of classes belong to the *Application* layer?⁹⁸
1. Application service
 2. Service container
 3. Symfony Console Application
 4. View model repository implementation
 5. View model repository interface
4. Which of the following types of classes belong to the *Infrastructure* layer?
⁹⁹
1. Entity
 2. Entity/write model repository implementation
 3. Service container
 4. Web controller
 5. Command DTO
-
-

13 Ports and adapters

This chapter covers:

- Hexagonal architecture
 - Ports and adapters
 - Structuring the Infrastructure layer
-

In the previous chapter we've explored *layers* as a way of grouping different types of objects in an application. In this chapter, we switch to a different perspective. We look at how an application is connected to *the outside world* and how we can group those connections in a logical way. We use ideas from *Hexagonal architecture* to accomplish this.

13.1 Hexagonal architecture

Hexagonal architecture is invented by Alistair Cockburn^{[100](#)}. Cockburn describes the intent of this architectural style as follows:

Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

I think this makes it pretty clear that hexagonal architecture is a suitable final destination for this book's quest of designing applications that are decoupled from surrounding infrastructure.

An alternative name for hexagonal architecture is *Ports & Adapters*. I think this name is actually better, because ports and adapters are the main concepts, and they are almost self-explanatory.

13.2 Ports

Let's start with the concept of a *Port*, which can only be explained using the concept of an *Actor*. In Chapter 9 we already talked about actors and the distinction between a primary and a secondary or supporting actor, but let's quickly repeat the definitions here. A primary actor is an actor that takes the initiative for communication. An example of a primary actor is a user who visits our web page, or an external system that talks to one of our API endpoints. When our application reaches out to an external system, for instance the database or a mail server, the external system should be considered a secondary or supporting actor.

With hexagonal architecture all the use cases are core code and they live inside the *Hexagon*. Whenever a primary actor needs to invoke one of the application's use cases the application should define a *Port* for that. Whenever a use case needs to communicate with a supporting actor, like the database, we also have to define a *Port* for that. A port is an "intention of communication". For example: our application may have a port "for creating an order" which can be used by primary actors to create an order. Our application will also have a port "for saving an order" which indicates that the application needs a supporting actor for saving orders.

A port that is used by a primary actor to communicate with our application could be called an *Incoming port*. A port that our application itself uses to communicate with a supporting actor could be called an *Outgoing port*. Figure 13.1 shows the hexagon with some of the use cases of our e-book webshop application. Each side of the hexagon represents a single port; the left side is usually used for incoming ports, the right side for outgoing ports.

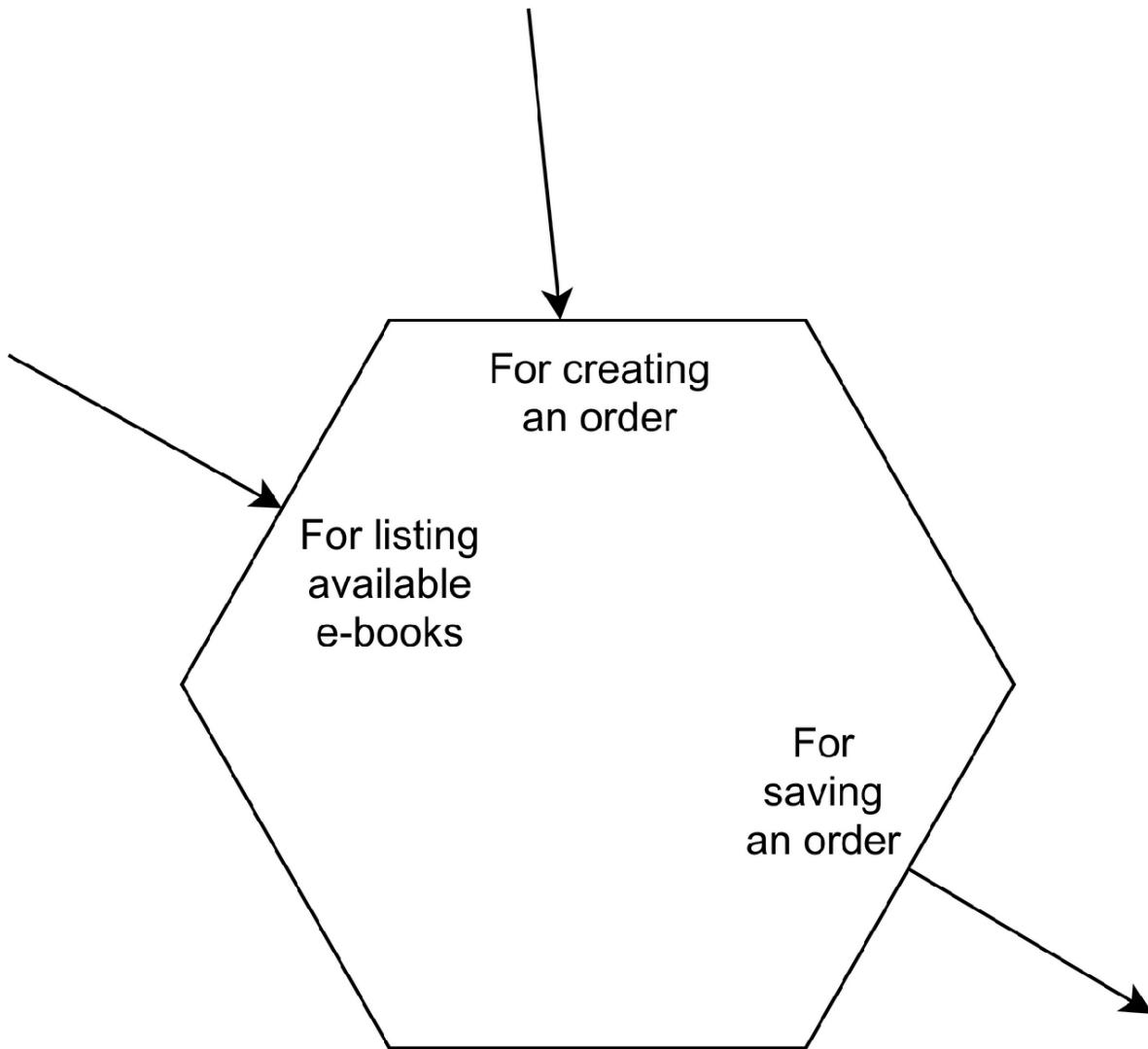


Figure 13.1: A hexagon with some of its incoming and outgoing ports.

Cockburn equates a port with an *interface*. It's a way in which actors can “interface” with our application. Every port will have a corresponding interface element inside the hexagon. We've already seen the port for saving an order (as one of the outgoing ports is called in Figure 13.1): the OrderRepository's save() method (see Listing 13.1).

Listing 13.1: The OrderRepository interface corresponds to the “for saving orders” port of the application.

```

interface OrderRepository
{
    // ...

    public function save(Order $order): void;
}

```

A port is only the *intention* of communication. A useful way to find good words for the intention of a port is to complete the sentence “for ...”. As an example, our e-book store would get ports “for listing the available e-books”, “for creating an order”, and “for saving an order”. Some developers who use hexagonal architecture also use these intentions as actual class or method names in their code, but personally I think it makes more sense to use imperative sentences like “list available e-books”, “create order” and “save order”.

A hexagon with ports alone is not enough to build a working application. A port is only an *intention*, now we need an *implementation*. In Cockburn’s terminology the implementation of a port is called an *Adapter*.

13.3 Adapters for outgoing ports

In the case of the outgoing port “for saving an order”, the adapter has to provide an implementation for the `OrderRepository` interface, which reaches out to some kind of database server to persist the `Order` entity. In Chapter 2 we already saw an example of such an implementation: the `OrderRepository-UsingSql` class (Listing 13.2). It connects to a relational database and stores the `Order` entity in an `orders` table, mapping the entity’s properties to table columns.

Listing 13.2: The `OrderRepositoryUsingSql` provides an implementation for the port “for saving an order”.

```

final class OrderRepositoryUsingSql implements OrderRepository
{
    private Connection $connection;

    public function __construct(Connection $connection)

```

```

{
    $this->connection = $connection;
}

public function save(Order $order): void
{
    // ...
}

// ...
}

```

An adapter often isn't a single class. All the collaborating objects should be considered part of the adapter too. For instance, if your `OrderRepository` implementation uses Doctrine ORM to store the `Order` entity, all of Doctrine ORM should be considered part of the adapter. Without it, the adapter wouldn't be functioning properly.

Since an outgoing port is defined in your code by an `interface` it's a natural option to provide alternative implementations for that interface, or in hexagonal terms: to provide an alternative adapter for the port. This is what hexagonal architecture is actually aiming for: the ability to replace adapters in order to make testing of the hexagon easier. An alternative implementation for the `OrderRepository` specifically designed for testing would be the `InMemoryOrderRepository` class in Listing [13.3](#). We'll use it in Section [14.5](#).

Listing 13.3: An in-memory implementation of `OrderRepository`.

```

final class InMemoryOrderRepository implements OrderRepository
{
    /**
     * @var array<string, Order>
     */
    private array $orders = [];

    public function save(Order $order): void
    {
        $this->orders[$order->orderId()->asString()] = $order;
    }
}

```

```

public function getById(OrderId $orderId): Order
{
    if (!isset($this->orders[$orderId->asString()])) {
        throw new RuntimeException(
            'Could not find order with ID ' . $orderId->asString()
        );
    }

    return $this->orders[$orderId->asString()];
}

```

Figure 13.2 shows the updated diagram with the two adapters of the port “for saving an order”.

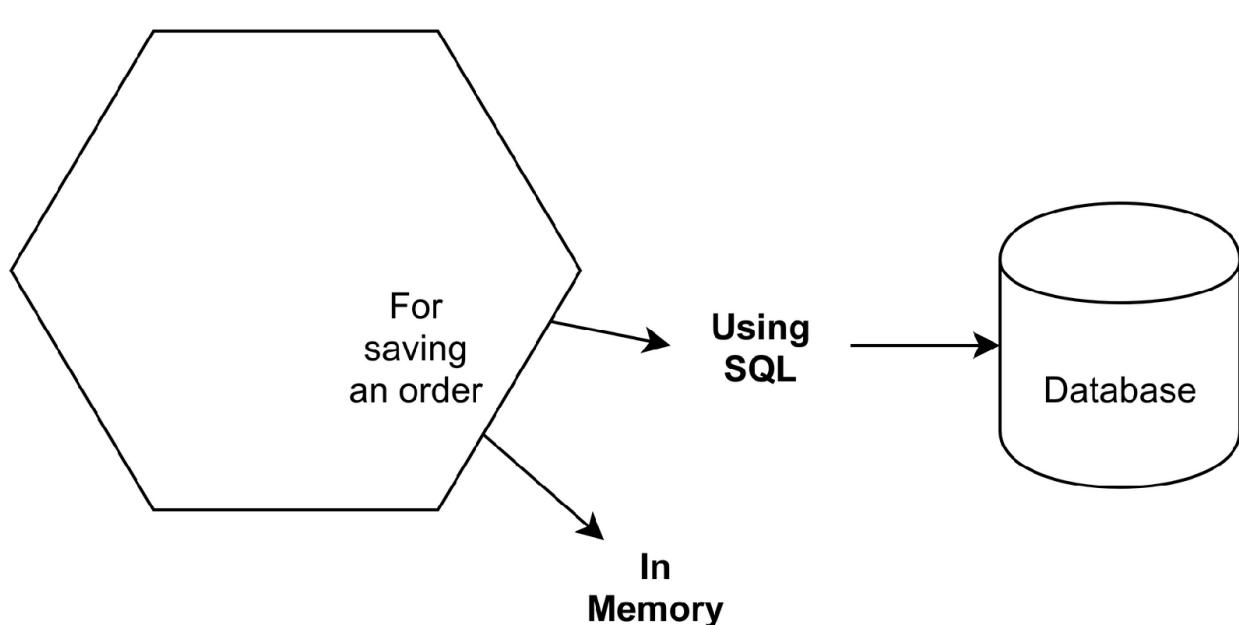


Figure 13.2: The hexagon with two adapters for the port “for saving an order”.

Besides providing an alternative adapter for testing purposes we can also experiment with alternative technologies. If you want to migrate from a relational database to a document database you can implement the same `OrderRepository` interface and see how that works without rewriting all the code inside the hexagon.

To show that two outgoing adapters (sometimes called “drivers”) are fully exchangeable, you’d need to write a *contract test* for the port. In such a test you specify how any implementation of the interface should behave. You can then run this test against each of the implementations you have. We’ll see an example of a contract test in Section [14.3](#).

13.4 Adapters for incoming ports

On the other side of the hexagon are the incoming ports. These are meant to accept incoming messages from users or external systems. They need code that facilitates this communication. In the case of the port “for creating an order” the web server needs to accept the incoming HTTP request, then forward it to PHP. PHP extracts relevant data and context from the request and then our web framework starts to further process it. Finally, it will call one of our controllers. All of the code involved in processing the incoming request should be considered part of the adapter. In fact, the controller itself is also part of the adapter, since the controller is specifically designed for HTTP communication. It uses web-specific objects and services like the current request or the user’s session. As soon as the controller calls an application service, we step out of the adapter and into the hexagon (see Listing [13.4](#)).

Listing 13.4: The controller is part of the adapter, the application service lives inside the hexagon.

```
final class OrderController
{
    private EbookOrderService $ebookOrderService;

    public function __construct(EbookOrderService $ebookOrderService)
    {
        $this->ebookOrderService = $ebookOrderService;
    }

    public function orderEbookAction(Request $request): Response
    {
        $orderId = $this->ebookOrderService->createOrder(
            CreateOrder::fromRequestData($request->request->all()
        );
    }
}
```

```

        ))
    );

    return new Response(/* ... */);
}
}

```

This is only true for as long as the application service itself is indeed fully decoupled from its delivery mechanism and the framework. It should be possible to call it not only from a web controller, but also from a CLI command handler as is shown in Listing [13.5](#).

Listing 13.5: A CLI adapter of the port for creating an order.

```

final class CreateOrderCommand extends Command
{
    private EbookOrderService $ebookOrderService;

    public function __construct(EbookOrderService $ebookOrderService)
    {
        $this->ebookOrderService = $ebookOrderService;
    }

    protected function execute(
        InputInterface $input,
        OutputInterface $output
    ): int {
        $orderId = $this->ebookOrderService->createOrder(
            new CreateOrder(
                (int)$input->getArgument('ebook_id'),
                (int)$input->getArgument('quantity'),
                $input->getArgument('email_address')
            )
        );

        $output->writeln(
            sprintf(
                '<success>Created a new order with ID %s',
                $orderId->asString()
            )
        );
    }

    return 0;
}

```

```
    }  
}
```

This level of decoupling guarantees that invoking an application service from a test scenario isn't substantially different from invoking it from a web controller. In fact, the test code itself should be considered an adapter since it communicates directly with the port.

Figure 13.3 shows the hexagon with several adapters of the port “for creating an order”.

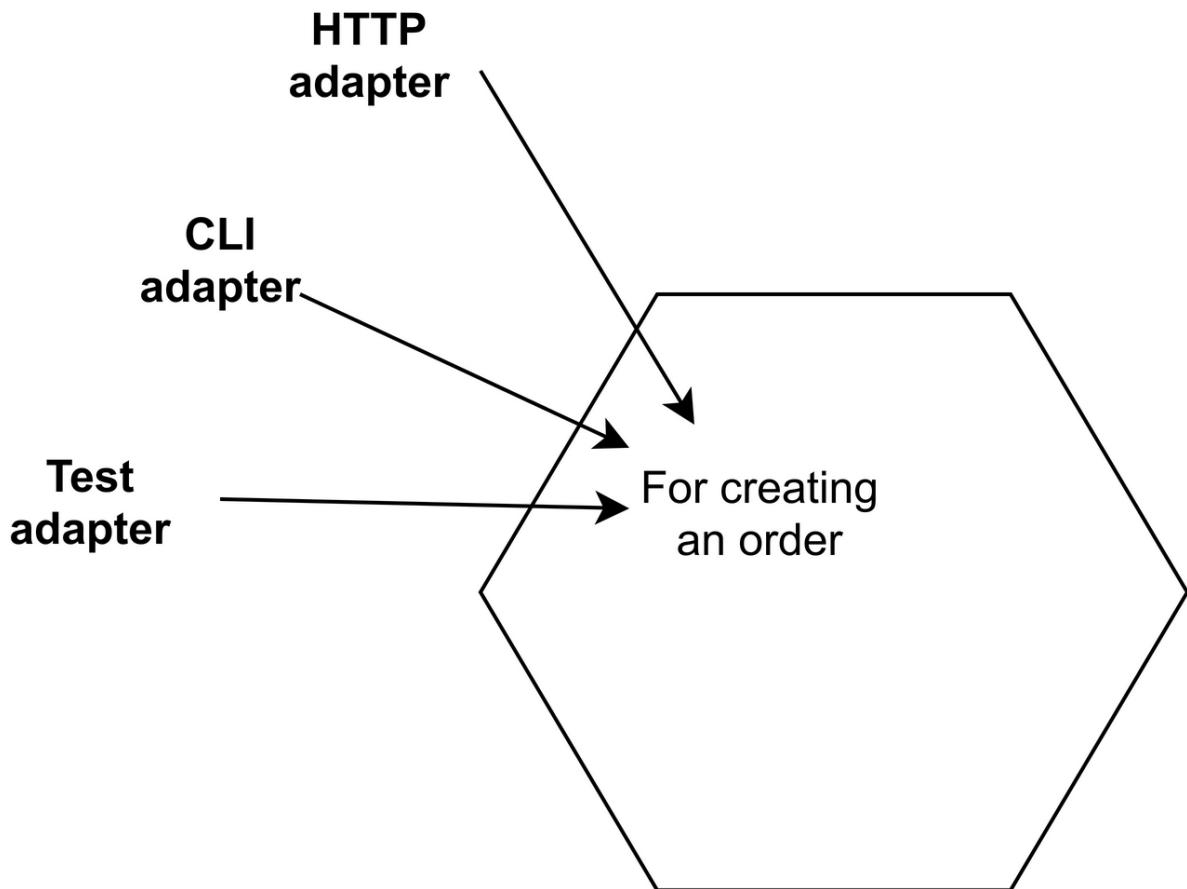


Figure 13.3: The hexagon with three adapters of the port “for creating an order”.

Incoming ports represent the intention of an actor to change something about the application’s state or to produce some other kind of effect with it.

Another intention could be to retrieve some information from the application. Actors may need a list of already created orders, or they want to know which e-books are available. Every possibility to retrieve information from an application should also be represented by a port. And again, the application has to provide at least one adapter for the port, to facilitate communication with the actor that wants to use the information. It needs to be delivered in a particular format (HTML, JSON, CSV, etc.), using a particular communication protocol (HTTP, AMQP, etc.).

As an example, the “listing available e-books” port of our application could be represented by a `ListAvailableEbooksRepository` interface and an `Ebook` DTO representing each available e-book (see Listing [13.6](#)).

Listing 13.6: The code representing the “listing available e-books” port.

```
interface ListAvailableEbooksRepository
{
    /**
     * @return array<Ebook>
     */
    public function listAll(): array;
}

final class Ebook
{
    private string $title;

    public function __construct(
        string $title
        // ...
    ) {
        $this->title = $title;
        // ...
    }

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

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

Provided that there is an implementation for this interface which is able to retrieve e-book records from the database and generate a list of `Ebook` objects, we also need a controller which can transform those objects into a format that an actor can deal with. Let's say the actor that needs this information is a JavaScript frontend application. In that case, it would be useful if the adapter for the port “for listing available e-books” would take the data from the `Ebook` objects and generate a JSON response based on it (see Listing [13.7](#))

Listing 13.7: Part of the adapter “for listing available e-books”.

```
final class EbookController
{
    private ListAvailableEbooksRepository $ebooksRepository;

    public function __construct(
        ListAvailableEbooksRepository $ebooksRepository
    ) {
        $this->ebooksRepository = $ebooksRepository;
    }

    public function listAvailableEbooksAction(): Response
    {
        $ebooks = $this->ebooksRepository->listAll();

        return new JsonResponse(
            array_map(
                fn (Ebook $ebook) => [
                    'title' => $ebook->title()
                ],
                $ebooks
            )
        );
    }
}
```

Besides offering a JSON response the application may also provide a regular HTML page rendering a nice list of e-books. It could just as well provide an

RSS feed of e-books, if that would be a desirable feature for some actors. Each example would require a separate adapter of the same port “for listing available e-books”.

13.5 The application as an interface

Earlier I said that ports are interfaces. For outgoing ports like the one “for saving an order” this immediately makes sense. When you want to save an Order you need to reach outside the application and connect to a database, so you can’t depend on a class there; you need an abstraction. The use case itself will rely on the port as an interface so that it isn’t tied to any implementation detail of the adapter. To test the port adapter we don’t have to invoke the entire use case. We only have to call the save() method on the OrderRepository implementation.

For an incoming port like the one “for creating an order” it may not be immediately clear that it needs to be an interface. And so far it wasn’t an interface either. In both the web controller and the console command we depended directly on the concrete class EbookOrderService. This means that when we invoke the port adapter (e.g. in the OrderController) in order to test it, we’ll also invoke the actual EbookOrderService, even though we’re only interested in testing the behavior of the port adapter itself. To prevent this we should try to replace the EbookOrderService with a test double. This is currently impossible because EbookOrderService is a final class which doesn’t allow subclassing; a necessity when creating test doubles. A quick solution would be to make the class non-final so you can create a test-double for it, but that doesn’t make sense to me; the class is not supposed to be extended so it should remain final¹⁰¹. A better solution is to define an interface for the incoming port as well. You can always create a test-double for an interface because it’s designed to be extended (in fact: implemented). The question is: what should the interface look like? Well, at least it should contain an abstract version of the EbookOrderService::create() method (see Listing 13.8).

Listing 13.8: An attempt at defining an interface for an application service

```
interface EbookOrderServiceInterface
```

```

{
    public function create(CreateOrder $command): OrderId;
}

final class EbookOrderService implements EbookOrderServiceInterface
{
    // ...

    public function create(CreateOrder $command): OrderId
    {
        // ...
    }
}

```

I think it'll be quite annoying when you have to always create an application service class and a separate interface that looks just like the class. So let's not go this way.

We can find a more generalized solution by taking the bigger picture into account. We are trying to define the incoming ports of our hexagon so we can invoke only the port adapters without also invoking the code inside the hexagon. This means we are trying to define an abstraction for the hexagon itself, which is in fact a collection of all the use cases of our application. This collection of use cases can be defined as a single API, which is both an *abstraction* (port adapters don't have to nor do they want to deal with what's going on behind the scenes) and a *contract* (port adapters rely on certain behaviors to be provided by the hexagon). We could define this API as a single interface and I think we may even be granted the `Interface` suffix here (See also Mathias Verraes, "Sensible Interfaces", <https://advwebapparch.com/sensible-interfaces>). Listing 13.9 shows what I mean.

Listing 13.9: The `ApplicationInterface` defining the API of the application.

```

interface ApplicationInterface
{
    public function createOrder(CreateOrder $command): OrderId;

    /**
     * @return array<Ebook>

```

```

    */
    public function listAvailableEbooks(): array;
    // ...
}

```

This `ApplicationInterface` defines all the use cases that incoming port adapters may invoke. A big advantage is that the `ApplicationInterface` can be used in any controller that wants to invoke a use case (see Listing [13.10](#)).

Listing 13.10: Using the *ApplicationInterface* in a controller.

```

final class EbookController
{
    private ApplicationInterface $application;

    public function __construct(
        ApplicationInterface $application
    ) {
        $this->application = $application;
    }

    public function listAvailableEbooksAction(): Response
    {
        $ebooks = $this->application->listAvailableEbooks();

        return new JsonResponse(/* ... */);
    }
}

final class OrderController
{
    private ApplicationInterface $application;

    public function __construct(
        ApplicationInterface $application
    ) {
        $this->application = $application;
    }

    public function orderEbookAction(Request $request): Response
    {

```

```

        $orderId = $this->application->createOrder(
            CreateOrder::fromRequestData($request->request->all(
        ))
    );

    return new Response(/* ... */);
}
}

```

Now that these port adapters depend on an interface we also need to provide a standard implementation which could be called `TheActualApplication` or just `Application`. This class will basically be a proxy for already existing services, as shown in Listing [13.11](#).

Listing 13.11: The standard implementation of `ApplicationInterface`.

```

final class Application implements ApplicationInterface
{
    private EbookOrderService $ebookOrderService;
    private ListAvailableEbooksRepository $listAvailableEbooksRe
pository;

    public function __construct(
        EbookOrderService $ebookOrderService,
        ListAvailableEbooksRepository $listAvailableEbooksReposi
tory
    ) {
        $this->ebookOrderService = $ebookOrderService;
        $this->listAvailableEbooksRepository = $listAvailableEbo
oksRepository;
    }

    public function createOrder(CreateOrder $command): OrderId
    {
        return $this->ebookOrderService->createOrder($command);
    }

    public function listAvailableEbooks(): array
    {
        return $this->listAvailableEbooksRepository->listAll();
    }
}

```

The size of this class may quickly get out of hand and we should take that as design feedback. Maybe the application is starting to do too many things and you need to subdivide it into modules. Another option is to try out the *Command bus* pattern. The idea is to have a generic interface, you could call it `CommandBus`, which has a single method: `handle()`. It accepts an untyped command object, examines the type of the command object, and uses it to find a service that can “handle” the command. There usually is some kind of mapping from command class to service class. Listing [13.12](#) shows how this could work.

Listing 13.12: A simple command bus for our application.

```
interface CommandBus
{
    /**
     * @return mixed
     */
    public function handle(object $command);
}

final class HardWiredCommandBus implements CommandBus
{
    private EbookOrderService $ebookOrderService;

    public function __construct(EbookOrderService $ebookOrderService)
    {
        $this->ebookOrderService = $ebookOrderService;
    }

    public function handle(object $command)
    {
        if ($command instanceof CreateOrder) {
            return $this->ebookOrderService->create($command);
        } elseif ($command instanceof /* ... */) {
            // and so on...
        }

        throw new RuntimeException(
            'Unknown command type: ' . get_class($command)
        );
}
```

```
}
```

See Listing [13.13](#) for the modified controller that now uses the `CommandBus` interface.

Listing 13.13: The controller uses the `CommandBus`.

```
final class OrderController
{
    private CommandBus $commandBus;

    public function __construct(
        CommandBus $commandBus
    ) {
        $this->commandBus = $commandBus;
    }

    public function orderEbookAction(Request $request): Response
    {
        $orderId = $this->commandBus->handle(
            CreateOrder::fromRequestData($request->request->all()
        ));
    }

    return new Response(/* ... */);
}
```

We don't have to limit ourselves to handling commands. The "command" bus could also handle queries like `listAvailableEbooks()`. Maybe it should just be called "bus" then, or "message bus".

The downside of using a *generic* interface like `CommandBus` is that we loose the parameter and return types that we have in the more *specific* `Application-Interface`. But when it comes to testing, the `CommandBus` has the same benefit as the `ApplicationInterface`. It's also a single thing that you can replace when you want to test incoming port adapters without also invoking code inside the hexagon.

Now that we have interfaces for both incoming ports and outgoing ports, and we've seen some examples of both incoming and outgoing port adapters, it's clear that the relation between port and adapter in both cases isn't symmetrical. For outgoing ports, the application (or hexagon) contains an interface (`OrderRepository`, `VatRateProvider`) for which the adapter has to provide an implementation by implementing the interface. For incoming ports the application also contains an interface (e.g. `ApplicationInterface` or `CommandBus`) but the adapters don't implement this interface, they *use* it.

13.6 Combining ports and adapters with layers

In Chapter [12](#) we introduced a layering system for applications, consisting of a *Domain*, *Application*, and *Infrastructure* layer. Hexagonal architecture is *orthogonal* to a layered architecture. This means you can apply hexagonal architecture, a layered architecture, or both. Neither one implies the other. However, they share the same origin: the desire to separate pure use cases from infrastructural concerns. This is why I think they fit well together. An application can have layers and at the same time explicitly define ports and adapters. Figure [13.4](#) visualizes how layers, ports and adapters could be combined. The inner hexagon contains the application's use cases, divided into an *Application* and *Domain* layer. Surrounding the hexagon is what you could call the *Outer hexagon*. This corresponds to the *Infrastructure* layer as we know it. It contains the adapters for the ports of the inner hexagon, effectively allowing communication between the inner hexagon and the outside world.

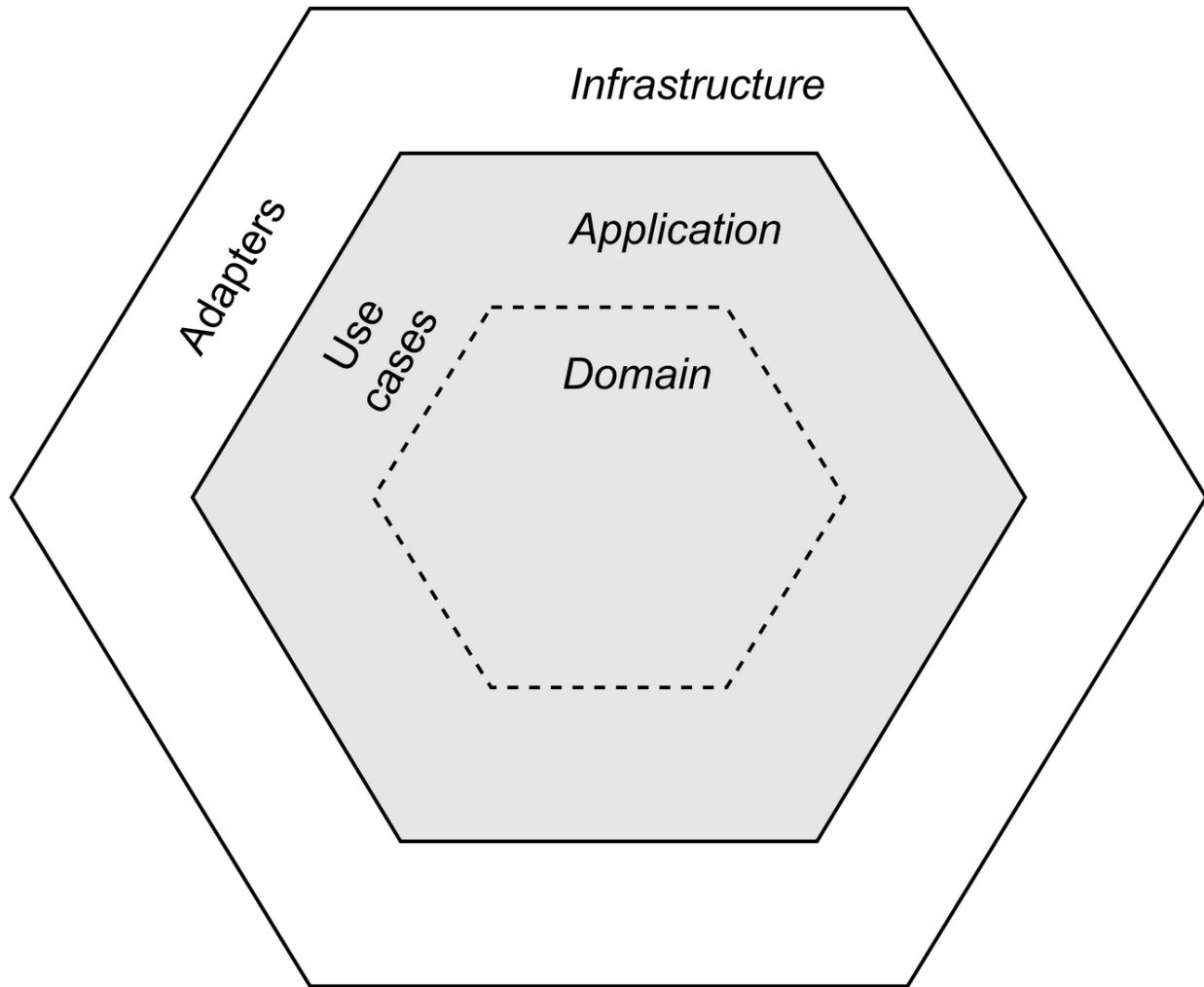


Figure 13.4: Layers, ports, and adapters combined.

13.7 Structuring the Infrastructure layer

In Section [12.4.2](#) we found a way to subdivide the *Domain* and *Application* layer namespaces and keep those namespaces from becoming large bags of classes. There is a similar risk when it comes to the *Infrastructure* namespace, although I find that in practice it isn't often a big problem. I don't often find myself browsing through the *Infrastructure* directory. If I need to change something about some adapter code, I'm usually working on the use case itself, meaning I'm looking at code in *Application* or *Domain*. When I'm looking at an application service or a repository interface for example, I can use the IDE's "Find usages" or "Find implementations"

functionality and quickly jump to the related infrastructure code. That said, it's often a good idea to at least make some kind of subdivision within the `Infrastructure` namespace.

If you have multiple classes related to a single adapter, you could group them in a sub-namespace of `Infrastructure`, just like you do when a use case relies on multiple classes. If you have multiple adapters which all use the same technology (framework, library, communication protocol, etc.), you could also group them in a sub-namespace of `Infrastructure`. This has the additional benefit of giving readers of the code a clear overview of the ways in which this application connects to external systems; something that an average code base isn't able to do. Figure ?? shows several examples of classes in `Infrastructure` and its sub-namespaces.

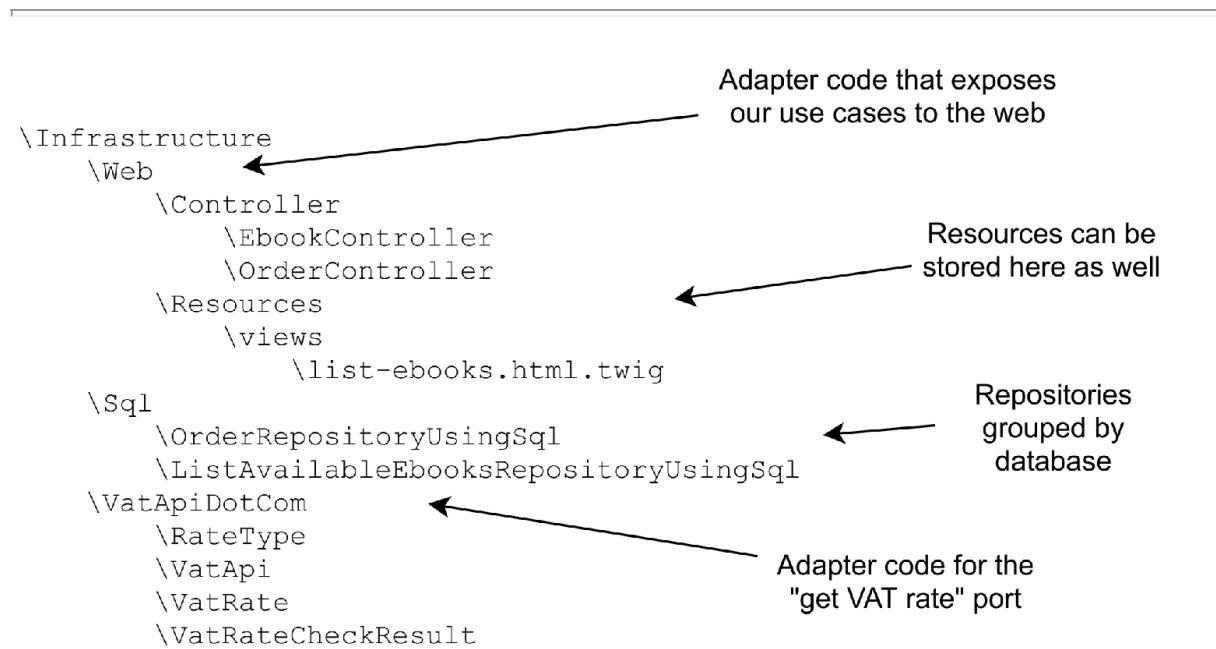


Figure 13.5: Sub-namespaces of the `Infrastructure` namespace.

Sometimes sub-namespaces of `Infrastructure` look like they could be extracted into a proper package. For example the `VatApiDotCom` classes in Figure ?? form a stand-alone HTTP client for `vatapi.com` and might as well

be extracted. If the code is not project-specific you may even make the library publicly accessible.

Something else that might happen in the `Infrastructure` namespace is the accumulation of shared code. If, for instance, you have some mapping utility that is used by all the SQL repositories, feel free to add it in the `Sql` sub-namespace. If it's more like a generic utility that could be used by any class in `Infrastructure`, you might as well make a generic sub-namespace for it too (e.g. `Shared` or `Common`). The advice about shared code is always to keep it to a minimum. If you don't keep an eye on it, it will grow quickly until everything ends up being "shared code".

13.8 Summary

In this chapter we discussed *Hexagonal architecture*. This approach to application architecture makes a clear distinction between pure application use cases, and how they are connected to the actors that invoke them or are invoked by them. We start with recognizing the ports of an application, which are intentions of communication between actors and our application. Ports are interfaces, and are represented as such in code. For each port, we need at least one adapter, which facilitates the actual communication, taking care of its implementation details.

Hexagonal architecture fits well with the layering system we explored in the previous chapter. In terms of layers, the inner hexagon consists of the *Domain* and *Application* layer combined. These layers contain the interfaces and application services which represent the ports of the application. The outer hexagon consists of the *Infrastructure* layer. This layer contains the port adapters.

Knowing about ports and adapters we can reorganize the classes in the *Infrastructure* layer. For instance, we can group classes by the type of the actor or the technology used.

Exercises

1. In hexagonal architecture a *Port* is:^{[102](#)}

1. An intention of communication
2. An implementation of communication

2. In hexagonal architecture an *Adapter* is:^{[103](#)}

1. An intention of communication
2. An implementation of communication

3. A database is a:^{[104](#)}

1. Primary actor
2. Secondary/supporting actor

4. A user is a:^{[105](#)}

1. Primary actor
2. Secondary/supporting actor

5. Should all ports be defined by an interface?^{[106](#)}

1. Yes, all ports should have an interface.
 2. No, only outgoing ports should have an interface.
-
-

14 A testing strategy for decoupled applications

This chapter covers:

- Different types of tests and how they cover all parts of the application
 - A development workflow for decoupled applications
-

Why did we want to decouple from infrastructure again? Because this allows us to focus on the high-level use cases of the application instead of the low-level implementation details. Because it enables the application core to survive all kinds of unrelated changes in the surrounding environment. And because it gives us a lot of code that is testable by default. In this chapter I'd like to take a closer look at this aspect: testability. And I'd like to give the outline for an answer to the question: what should be tested and how should we test it?

Test terminology has always been a problem for developers talking and writing about testing. The terms I use in this chapter for different types of tests may not be standard terms. But I think they are appropriate in the context of this book.

14.1 Unit tests

The simplest type of test is a unit test. It's quite a problematic name for such a simple test because it's really not clear what a "unit" is or how big it should be. A unit could be a function or a method, maybe a single class, or a class and all of the classes it directly relies on. I don't think a unit test should focus on any of these things. I want a unit test to test the behaviors of an *object*. I don't worry about which classes are involved in testing the behavior of this object. So I don't annotate my unit tests with `@covers` annotations either. For me it's about how the object behaves, which classes are involved is somewhat irrelevant.

Writing a unit test is a way to zoom in on the behavior of some of the system's smaller elements. These tests support development by allowing the developer to specify the behavior of the smaller elements, allowing them to ignore some of those behaviors when the larger elements are being tested (see Section [14.5](#)). Writing unit tests can therefore give you a steadily increasing sense of security. If the building blocks are safe to use, then you don't have to worry about all the details once you start using the blocks to build the bigger structures.

In most of the projects I've encountered so far there is a problem with the unit test suite. The tests in it are not all unit tests. Not because the unit is too large, but because the tests aren't isolated enough. What this means is nicely described by Michael Feathers in his definition of a unit test^{[107](#)}:

A test is not a unit test if:

1. It talks to the database
2. It communicates across the network

3. It touches the file system
4. It can't run at the same time as any of your other unit tests
5. You have to do special things to your environment (such as editing config files) to run it.

If you've read Part I you know that the first three points reflect only part of what could be considered "infrastructure" concerns. Additionally, we have also considered code that communicates with the system clock or the system's random device to be infrastructure code. Items 4 and 5 correspond to our discovery that code that relies on global state or statically accessible service and configuration locators shouldn't be considered core code; it needs special setup in order to run. So I think it makes sense to generalize Feathers' definition and say that:

A test is not a unit test if it invokes infrastructure code.

This certainly isn't true for many of the tests that developers call "unit tests". The confusion stems from the fact that unit tests and non-unit tests can both be executed by *PHPUnit*. The test framework has "unit" in its name, so all the tests that this framework runs are usually called "unit tests". But the test framework itself can't guarantee that the tests you write and run are true unit tests. So once again it will be your own responsibility to only invoke core code in a unit test. It may help to call actual unit tests "isolated tests" and move the tests that invoke infrastructure code to a different test suite called "integration tests" (see also Section [14.2](#)).

In my projects I usually write isolated tests only for some smaller building blocks like entities and value objects. These domain objects are great candidates for isolated tests because they have to protect all kinds of domain invariants, which all deserve to be separately tested. It's convenient to test these invariants close to the object that protects them, instead of in higher-level tests. I don't write unit tests for objects that coordinate changes in domain objects, like application services, event subscribers, and repositories. These will be tested with *Use case tests* (Section [14.5](#)) and *Adapter tests* (Section [14.2](#)).

[Listing 14.1](#) shows several examples of unit tests for the `Order` entity. I try to keep the test methods really small. Ideally each method consists of three statements. Wherever I can I use factory methods like `aPaidOrder()`. The method name can be used to describe what's special about the object it returns. This makes it easier to understand what the test tries to prove.

Listing 14.1: Examples of unit tests for an entity.

```
use PHPUnit\Framework\TestCase;

final class OrderTest extends TestCase
{
    // ...

    /**
     * @test
     */
    public function it_cant_be_cancelled_if_it_has_already_been_paid(): void
    {
        $order = $this->aPaidOrder();
```

```

        $this->expectException(CouldNotCancelOrder::class);

        $order->cancel();
    }

    // ...

    /**
     * @test
     */
    public function you_can_modify_the_external_reference(): void
    {
        $order = $this->aNewOrder();

        $order->setPaymentReference('ABC123');

        self::assertArrayContainsObjectOfType(
            ExternalReferenceWasModified::class,
            $order->releaseEvents()
        );
    }

    // ...
}

```

Value objects are also good candidates for unit-testing. By definition they are immutable objects without side-effects. Listing [14.2](#) shows some examples.

Listing 14.2: Unit-testing a value object

```

use PHPUnit\Framework\TestCase;

final class UrlTest extends TestCase
{
    /**
     * @test
     */
    public function it_can_be_created_from_and_converted_back_to_a_string(): void
    {
        $string = 'https://matthiasnoback.nl';

        self::assertEquals(
            $string,
            Url::fromString($string)->asString()
        );
    }

    /**
     * @test
     */
    public function it_can_extract_the_top_level_domain(): void
    {
        self::assertEquals(
            'nl',

```

```

        Url::fromString('https://matthiasnoback.nl')->tld()
    );
}

// ...
}

```

14.2 Adapter tests

We've already determined that you can't unit-test infrastructure code. But you still want to test it somehow, so what type of test can we use for infrastructure code? Infrastructure tests are often called integration tests, or integrated tests. These tests show that infrastructure code including third-party code correctly integrates with external actors. In Chapter [13](#) we used another name for code that integrates with external actors: "port adapters", or just "adapters". That's why it makes sense for me to call these tests *Adapter tests*. They prove that an adapter works correctly. Since there are two types of ports, incoming and outgoing, it's no surprise that there are two types of adapter tests too.

14.3 Contract tests for outgoing port adapters

Outgoing ports are the ports where the application needs to communicate with some external system, like a database, or a remote webservice. Outgoing ports are defined by separate interfaces, like the `OrderRepository` that can save an entity, and the `VatRateProvider` that we've used earlier to determine the `vatRate` of a product. These interfaces were introduced as an abstraction that allowed us to decouple core code from infrastructure code. The interface makes it easy to replace the communication with the external system with a fake implementation that we can use in our use case tests (see Section [14.5](#)). But the interface also defines a *contract* describing everything we want from an implementation. The contract for the `OrderRepository` is that it can save an `Order` entity and that you can get an equivalent object back from it if you call its `getById()` method. It is helpful to clearly specify and document the contract since the interface itself can't communicate these things. All it can describe about itself is the methods it has, their return types, and their parameter types. One option is to document the behavior using inline documentation, but it's actually more powerful to use a contract test to describe the contract. Listing [14.3](#) shows what this looks like.

Listing 14.3: Part of the contract test for the `OrderRepository`.

```

use PHPUnit\Framework\TestCase;

final class OrderRepositoryContractTest extends TestCase
{
    /**
     * @test
     * @dataProvider orders
     */
    public function it_can_save_and_load_order_entities(Order $order): void
    {
        foreach ($this->orderRepositories() as $orderRepository) {
            $orderRepository->save($order);
        }
    }
}

```

```

        $fromRepository = $orderRepository->getById($order->orderId());
        self::assertEquals($order, $fromRepository);
    }
}

/**
 * @return Generator<OrderRepository>
 */
private function orderRepositories(): Generator
{
    yield new InMemoryOrderRepository();

    yield new OrderRepositoryUsingDoctrineDbal(/* ... */);

    // yield any other implementation you have
}

/**
 * @return Generator<array<Order>>
 */
public function orders(): Generator
{
    yield [Order::create(/* ... */)];

    yield [Order::create(/* ... */) -> cancel()];

    yield [Order::create(/* ... */) -> markAsPaid()];

    // yield any other 'Order' entity that you might want to save
}
// ...
}

```

Several other parts of the contract should be tested as well, like the ability of all implementations to provide unique order IDs. Or the ability to save an entity after it has been modified.

Make sure that everything about the setup of the repository implementations is as real as possible. That is, use a real database, preferably the same database that you use in production. Don't create test doubles for anything. Make sure this test will expose any problem with the code that would otherwise only show up once the code is running in production. Figure 14.1 shows the result of this approach: a contract test for `OrderRepository` will invoke all the software elements involved in using the actual database. Such a test demonstrates that if an application service in the core of the application relies on the `OrderRepository` interface, and the service container has been configured to inject an instance of `OrderRepositoryUsingDoctrineDbal`, everything will work just fine.

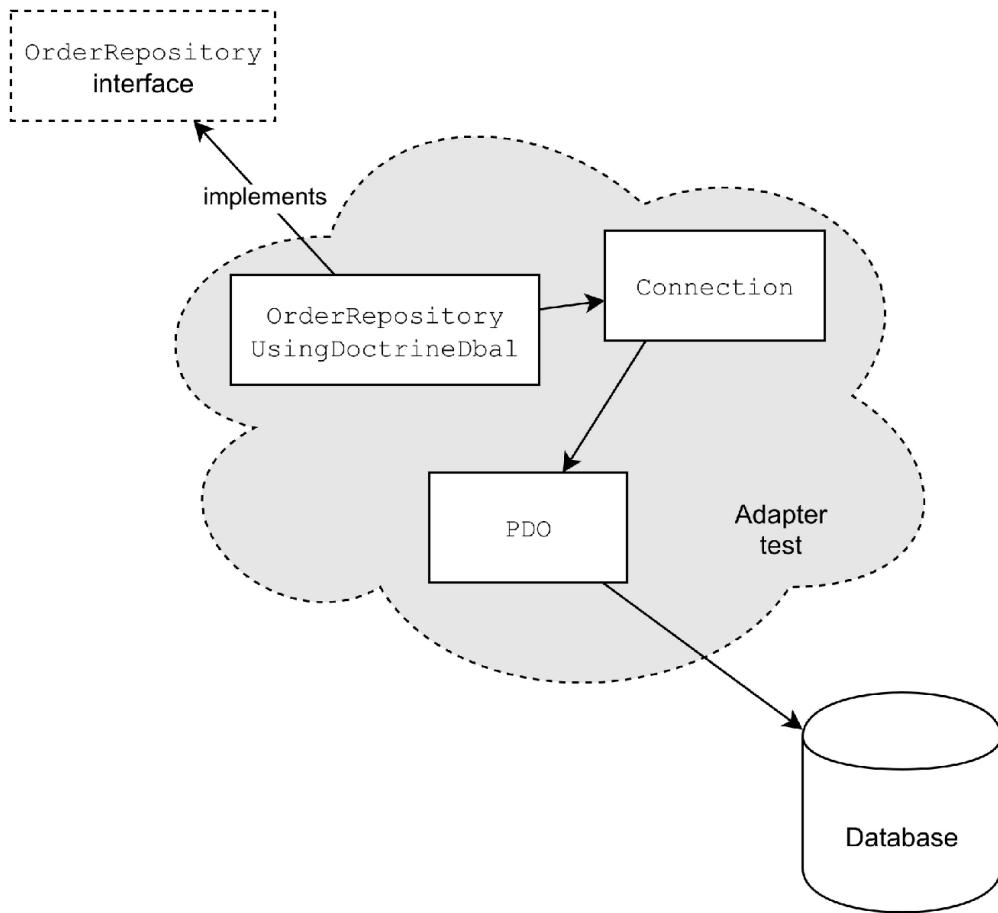


Figure 14.1: Test coverage for the `OrderRepository` contract test.

Something to keep in mind when writing a contract test is that you shouldn't rely on specific behavior of one of the implementations. The test methods should only call methods that are defined on the interface. This guarantees that the implementations follow the Liskov Substitution Principle, i.e. they can seemingly act as replacements for each other.

For repositories a contract test is the right kind of test to write. For other services, like the `VatRateProvider`, it may not work very well. The difference is that this service just needs to provide some kind of answer, but we can't really define in a contract what a good answer is, other than that it should be an answer of the right type, in this case `VatRate`. But this is already guaranteed at the language level so it doesn't make sense to write a test to verify that implementations of `VatRateProvider` return an instance of `VatRate`. If they don't, the “compiler” will complain. What would actually be useful is to test that a specific implementation like `VatRateProviderUsingVatApiDotCom` is able to communicate correctly and effectively with the remote API of `vatapi.com`. We already talked about testing this class in Section 6.4 but Listing 14.4 repeats the (slightly modified) example we discussed there.

Listing 14.4: Integration tests for `VatRateProviderUsingVatApiDotCom`.

```

final class VatRateProviderUsingVatApiDotComTest extends TestCase
{
    /**
     * @test
     */
    public function it_provides_the_dutch_vat_rate(): void
    {
        $provider = new VatRateProviderUsingVatApiDotCom(
            new VatApi(
                'TEST_API_KEY',
                new HttpClient(
                    'https://vatapi.com'
                )
            )
        );

        self::assertEquals(
            VatRate::fromInt(21),
            $provider->vatRateForSellingEbooksInCountry('NL')
        );
    }
}

```

In Chapter [6](#) we kept using `curl_*` functions inside the `VatApi` class. A more realistic approach is to inject an HTTP client into the `VatApi` class, so it doesn't have deal with those low-level implementation details itself. Injecting an HTTP client as a dependency makes it clear that this class is going to make a network connection. In Listing [14.4](#) it even shows the base URL for the requests. I've made this change to the example to make the point of this adapter test more clear: everything should be as real as possible.

When communicating with external services that are not managed by you or your team this can be really hard to do without sacrificing some test stability. Here's what I usually try:

1. Write the test against the real service
2. Write the test against a sandbox environment the third party offers
3. Write the test against a fake server that I run
4. Write the test against a fake or mock HTTP client offered by the client library that I use
5. Write the test against a fake or mock of the HTTP client interface that I use

Option 1 and 2 give you the most confidence, but the test may sometimes fail for reasons that you can't do anything about. Option 3 is great because it shows that you are using your HTTP client correctly: it can make actual HTTP requests. You can verify inside the fake server that the right requests are made. Option 4 is less optimal since it puts some trust in your library. You assume that it's able to talk to the particular server you're working with (but that assumption might be safe to make). Option 5 is even less optimal, since you might make some bad assumptions about how to use the library correctly. This is closely related to the testing rule “don't mock what you don't own”. I think this could be generalized as “don't mock an interface whose contract is bigger than the interface itself can describe”. But that's something for another day.

Figure 14.2 shows what software elements the adapter test for `VatRateProviderUsingVatApiDotCom` cover. This adapter tests demonstrates that when an application service relies on the `VatRateProvider` interface and the service container has been configured to inject an instance of `VatRateProviderUsingVatApiDotCom`, it will work and provide what's expected from it.

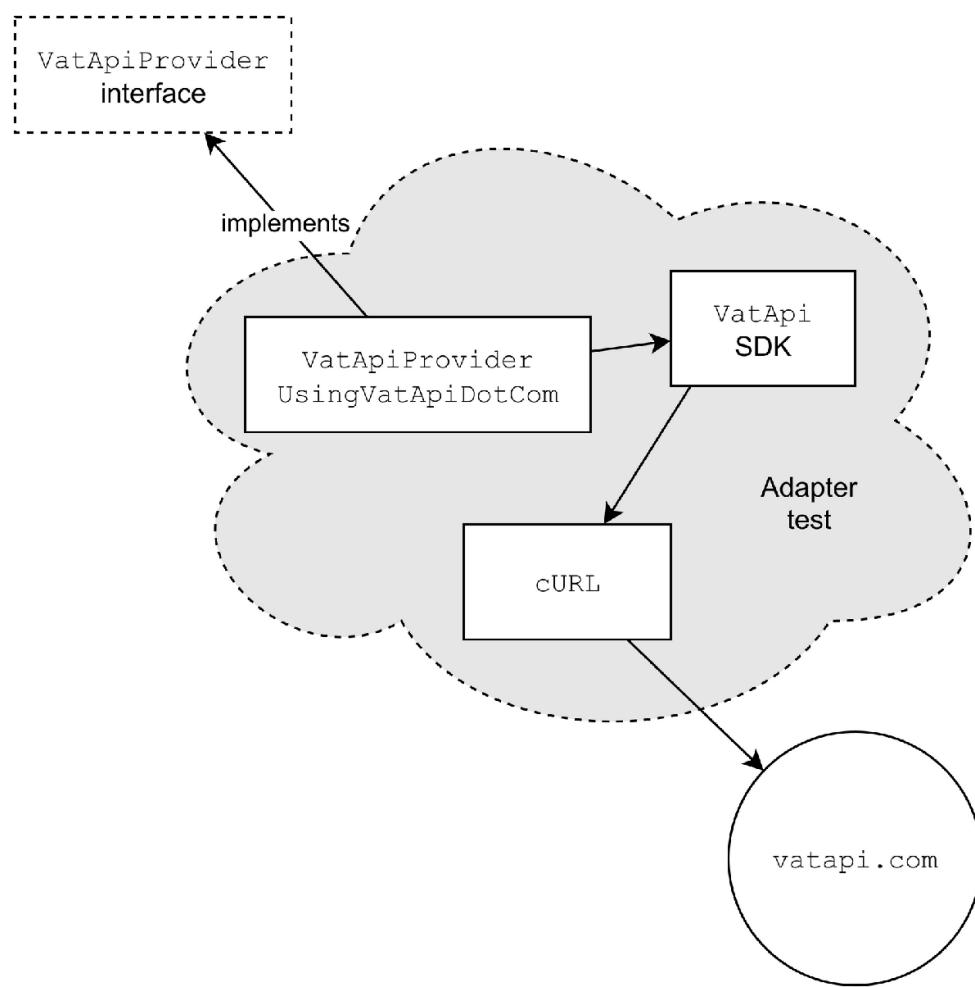


Figure 14.2: Test coverage for the `VatRateProvider` adapter test.

14.4 Driving tests for incoming port adapters

The implementation of an outgoing port adapter is correct if it fully implements the port interface and its contract. What defines the effectiveness of an incoming port adapter? Such an adapter processes incoming communication, like HTTP requests, or invocations from the command line. In the case of an HTTP request, there is a lot of code involved in processing it, from the web server itself, to the PHP SAPI, to the web framework you use in your project, to the controller that the framework invokes. What makes all of this code pass the test is whether the port adapter eventually makes the correct call to the application core. As an example, take a

look at the `createOrderAction` in Listing 14.5. This controller action processes the incoming HTTP request by copying the data from the request body into a DTO. It then passes the DTO to the `createOrder()` method of the `ApplicationInterface` (see Section 13.5).

Listing 14.5: `orderEbookAction()` invokes `ApplicationInterface::createOrder()`.

```
final class OrderController
{
    private ApplicationInterface $application;

    public function __construct(ApplicationInterface $application)
    {
        $this->application = $application
    }

    public function orderEbookAction(Request $request): Response
    {
        $orderId = $this->application->createOrder(
            CreateOrder::fromRequestData(
                $request->request->all()
            )
        );
        // ...
    }
}
```

An adapter test for this code wouldn't have to test the actual logic of creating the order. We only have to show that this code, provided an HTTP request, will make the right call to `createOrder()`, providing a `CreateOrder` object containing the expected data. This is a perfect case for a mock object. Inside a test we can create a mock for `ApplicationInterface` and define some expectations for it. We then prepare an HTTP request and invoke the controller. But should we directly invoke the controller ourselves, as shown in Listing 14.6?

Listing 14.6: Invoking the controller directly.

```
public function it_correctly_invokes_createOrder(): void
{
    $application = $this->createMock(ApplicationInterface::class);
    $application->expects($this->once())
        ->method('createOrder')
        ->with(new CreateOrder(/* ... */));

    $request = new Request('/create-order', /* ... */);
    $controller = new OrderController($application);
    $controller->createOrderAction($request);
}
```

I think this would leave too many things untested. We manually instantiate a `Request` object, but it's likely that the `Request` object created by the framework based on a real HTTP request will look very different. We manually instantiate the `OrderController`, but we don't know if the framework will be able to instantiate it too. We don't even know if the framework will correctly dispatch the request to the `OrderController` based on the '/create-order' request URI. All of these are currently unverified assumptions.

So instead of invoking the controller in isolation we should embed it in its natural environment and invoke it in a more realistic way. We're lucky that most frameworks offer tooling for creating controller tests that allow us to do so. I only have experience with Symfony's built-in tools^{[108](#)} and Panther^{[109](#)}. Listing 14.7 shows an example using Symfony's `WebTestCase` and related tools that show the equivalent of the test in Listing 14.6, while also testing the framework integration part.

Listing 14.7: Using the `WebTestCase` to test `createOrderAction()`.

```
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

final class OrderControllerTest extends WebTestCase
{
    public function it_correctly_invokes_createOrder(): void
    {
        $application = $this->createMock(ApplicationInterface::class);
        $application->expects($this->once())
            ->method('createOrder')
            ->with(new CreateOrder(2, 1, 'matthiasnoback@gmail.com'))
            ->will($this->returnValue(new OrderId(1001)));

        $client = self::createClient();
        $client->getContainer()->set(
            ApplicationInterface::class,
            $application
        );

        $client->request('POST', '/create-order', [
            'ebook_id' => '2',
            'quantity' => '1',
            'email_address' => 'matthiasnoback@gmail.com'
        ]);

        self::assertTrue(
            $this->client->getResponse()->isRedirect('/order-details/1001')
        );
    }
}
```

This test verifies that the framework will instantiate the `OrderController` and call the `createOrderAction()` method based on the '/create-order' URI. It verifies that `createOrderAction()` is able to extract the submitted data from the request body and copy it into the `CreateOrder` DTO, which is then passed as an argument to `ApplicationInterface::createOrder()`. And as a bonus, this test also verifies that we're being redirected to the correct order details page.

None of this code actually creates an `Order` entity, because we've replaced the `ApplicationInterface` with a mock. Because of this, none of this code needs any secondary actor to be available. This test will only invoke code on the left side of the application, where incoming communication is processed. Figure 14.3 shows what elements are involved in running this `OrderControllerTest`.

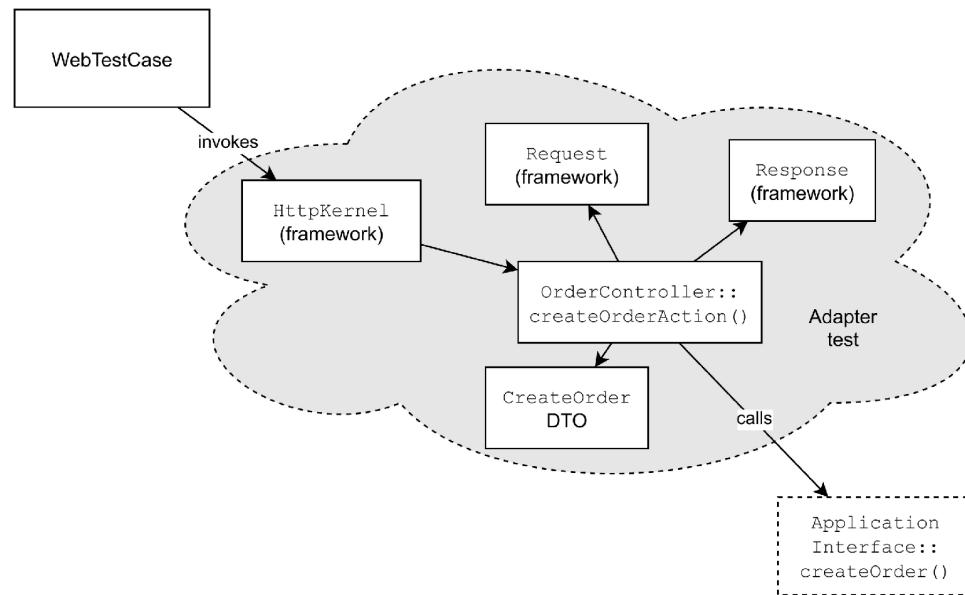


Figure 14.3: Test coverage of `OrderControllerTest`.

One thing to be aware of: even if we write a controller test like this, we can't be completely sure that the application works in production. The `WebTestCase` client doesn't communicate with a web server; it doesn't make an actual HTTP request. It instantiates a `Request` object just like we did before. This means that once the application runs behind a web server it may still not function correctly. There could be issues in the code between the moment the web server calls PHP and the moment the framework calls our controller. So should we test the application in a more production-like environment? Panther, the testing tool I mentioned before, is able to promote tests based on the `WebTestCase` to tests that start a web server and run the same tests while making actual HTTP requests. But I'm not sure if we should use this approach in an adapter test.

With any kind of adapter tests, the trade-off is between how many assumptions your test verifies and how fast and stable your tests will be. The first test we wrote, the one where we only tested the `OrderController` itself, could be considered a stable test. There aren't many elements involved, so there aren't many unrelated problems that could cause this test to fail. However, we decided it left too many assumptions unverified. Using the `WebTestCase` increases the confidence we have in our code, but there are many more steps involved. Such a test could fail for many unrelated reasons, and because it involves calling more code, it will be much slower too. Starting a web server and making actual HTTP requests will make the test even slower, and more likely to fail for hard-to-debug reasons.

In my experience, keeping the test suite fast and stable is very important. It makes my work much more fun, and I feel very confident about the quality of my work. So I think the majority of the tests for incoming adapters shouldn't make actual HTTP requests. They can invoke the framework programmatically and verify 80% of the assumptions. Still, it would be smart to have at least a few tests that mimic a production setting, just so we know that all the real infrastructure elements work well together. For this we write so-called *End-to-end tests* (see Section [14.6](#)).

“Why is it important to mock the application in an adapter test?

The reason we're mocking the `ApplicationInterface` is that we shouldn't use integration tests to test core code. Core code allows itself to be tested with unit tests or use case tests, which both run very quickly and are also very stable because they are completely deterministic and don't involve more elements than strictly needed. Adapter tests are integration tests and are by definition slower and less stable than isolated tests. So although we could test core logic using an adapter test, it's better to leave that to more isolated tests, and replace the entry point to the core (the `ApplicationInterface` in our example) with a test double.

`createService()` is a command method which produces a side-effect. This is why the request method is `POST`. Our web application also has controllers that don't produce a side-effect but only return some information. For example, the `/list-available-ebooks` page renders a list of e-books in HTML. The `EbookController::listAvailableEbooksAction()` calls the method `listAvailable-Ebooks()` on the `ApplicationInterface` and passes the return value to the template (see Listing [14.8](#)).

Listing 14.8: `listAvailableEbooksAction()` calls `listAvailableEbooks()`.

```
final class EbookController
{
    private ApplicationInterface $application;
    private TemplateRenderer $templateRenderer;

    public function __construct(
        ApplicationInterface $application,
        TemplateRenderer $templateRenderer
    ) {
        $this->application = $application;
        $this->templateRenderer = $templateRenderer;
    }

    public function listAvailableEbooksAction(): Response
    {
        $ebooks = $this->application->listAvailableEbooks();

        return $this->templateRenderer->render(/* ... */);
    }
}
```

```
    }
}
```

We could write a similar controller test for it like we've done before (see Listing [14.9](#)). But here we need to do a bit more work to prepare a set of fake `Ebook` objects that will be returned by the test double.

Listing 14.9: A test for the `EbookController`.

```
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

final class EbookControllerTest extends WebTestCase
{
    public function it_renders_a_list_of_ebooks(): void
    {
        $ebooks = [
            EbookBuilder::create()
                ->withTitle('Advanced Web Application Architecture')
                ->build()
        ];

        $application = $this->createMock(ApplicationInterface::class);
        $application->expects($this->any())
            ->method('listAvailableEbooks')
            ->will($this->returnValue($ebooks));

        $client = self::createClient();
        $client->getContainer()->set(
            ApplicationInterface::class,
            $application
        );

        $crawler = $client->request('GET', '/list-available-ebooks');

        self::assertStringContainsString(
            'Advanced Web Application Architecture',
            $crawler->filter('.ebook-title')->text()
        );
    }
}
```

The return value of `request()` is a so-called `DomCrawler` which you can use to select certain elements from the HTML response body using CSS selectors. You can then make assertions about their attributes or content.

To simplify the process of creating read model objects, I usually introduce a builder that allows me to focus only on the relevant values. The builder will add any other required data so you don't have to worry about that. Listing [14.10](#) shows what the builder looks like.

Listing 14.10: A builder for `Ebook` read model objects.

```
final class EbookBuilder
```

```

{
    private string $id = 'ad5075f1-be24-4ae1-8ba8-9efec6f4933b';
    private int $price = 2500;
    private string $title = 'The title';

    private function __construct()
    {
    }

    public static function create(): self
    {
        return new self();
    }

    public function withTitle(string $title): self
    {
        $this->title = $title;

        return $this;
    }

    public function build(): Ebook
    {
        return new Ebook(
            EbookId::fromString($this->id),
            new Money($this->price, new Currency('EUR')),
            $this->title
        );
    }
}

```

Figure 14.4 shows what elements are involved in the adapter test for the Ebook-Controller.

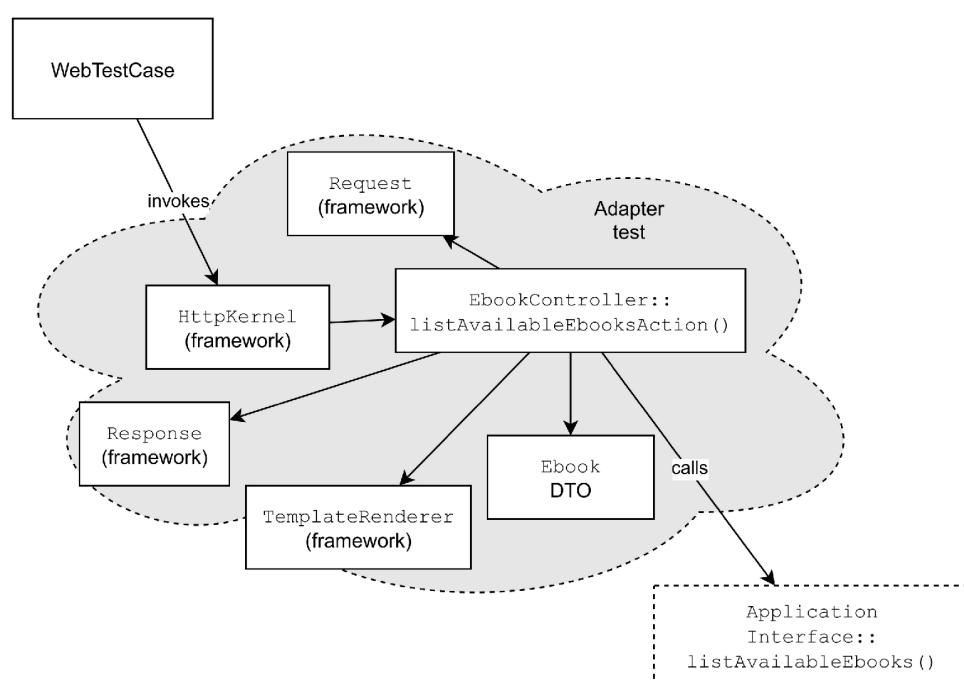


Figure 14.4: Test coverage of EbookControllerTest.

If all the ways in which your application communicates with the outside world have been divided into ports (interfaces) and adapters (supporting implementations), you can repeat the process described here and test all of your port adapters. You then know that incoming requests will be processed correctly and result in the right calls to the application core. You'll also know that return values from the application core will be transformed into the correct response. On the other side of the application, where outgoing communication happens, you know that the code correctly implements the interfaces from the core. Figure 14.5 depicts which areas of the application we have covered with tests so far.

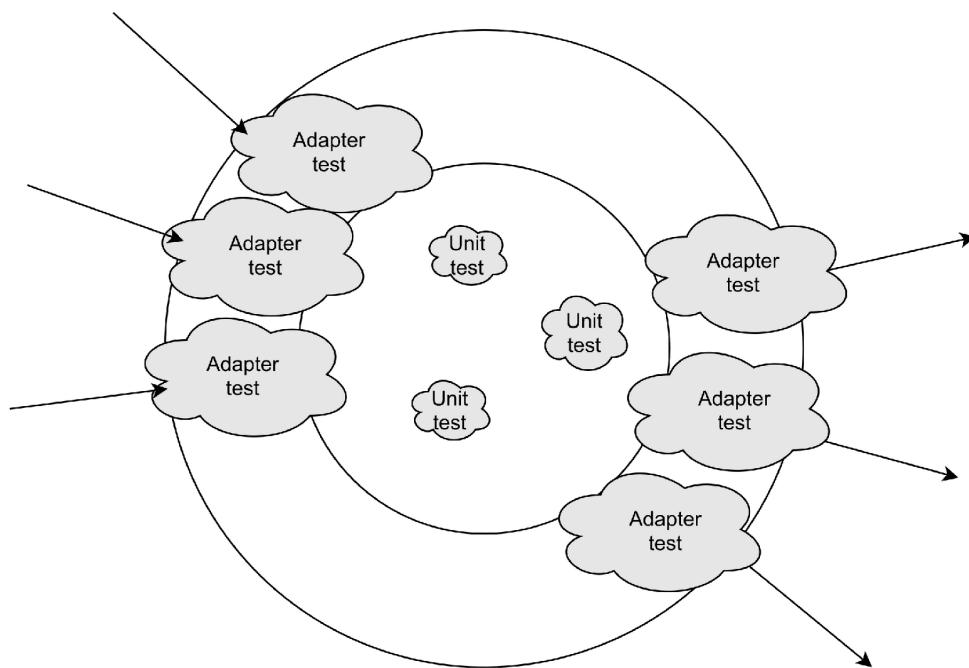


Figure 14.5: Test coverage with only adapter tests and some domain-level unit tests.

We're obviously not done yet. Even though we have a number of unit tests for domain model objects, we don't have a test for the use cases that our application has to offer. We don't know if inside the core of the application everything works well together, e.g. that after paying an order, the customer will receive a confirmation email. We need use case tests for that.

14.5 Use case tests

Use case tests test use cases, but most importantly they document the primary actions and their effects, for instance:

1. When the customer creates an order, they should receive an order confirmation email.
- 2.

When the administrator adds a new e-book to the catalog, it should appear in the list of available ebooks.

These scenarios can be implemented using the design patterns covered in Chapter [11](#). Mapping the first scenario to patterns, I'd say we need:

1. A `Customer` entity with a `CustomerId` value object we can use to link the order to the customer.
2. A `createOrder()` application service method that creates an `Order` entity, saves it using the `OrderRepository`, and dispatches the recorded domain events to the `EventDispatcher`.
3. An `Order` entity which produces an `OrderWasCreated` event.
4. An `OrderRepository` interface and an in-memory implementation that we can use in the use case test.
5. A `SendConfirmationMail` event subscriber which subscribes to `OrderWasCreated` events and sends the order confirmation email.

Another thing we need is something that can represent the actual sending of the order confirmation email. Again, this needs to be an abstraction, e.g. our own `Mailer` interface with a method `sendOrderConfirmationMail()`.

We'll also need an `EventDispatcher` service. We've seen an example of it in Section [11.5](#), including an implementation. Listing [14.11](#) shows the same interface with a similar implementation.

Listing 14.11: The `EventDispatcher` interface and an implementation.

```
interface EventDispatcher
{
    /**
     * @param array<object> $events
     */
    public function dispatchAll(array $events): void;
}

final class ConfigurableEventDispatcher implements EventDispatcher
{
    private array $subscribers = [];

    public function __construct()
    {
    }

    public function addSubscriber(
        string $eventType,
        callable $subscriber
    ): void {
        $this->subscribers[$eventType][] = $subscriber;
    }

    public function dispatchAll(array $events): void
    {
        foreach ($events as $event) {
```

```

        foreach (
            $this->subscribersForEvent($event) as $subscriber
        ) {
            $subscriber($event);
        }
    }

    private function subscribersForEvent(object $event): array
    {
        return $this->subscribers[get_class($event)] ?? [];
    }
}

```

“Can’t we use the framework’s event dispatcher?”

Technically you can. However, I find that third-party event dispatchers often don’t provide the kind of API that I’d like to use. For instance, they don’t have a `dispatchAll()` method, they require a base class for every domain event, or their event objects are mutable. Also, event dispatchers often allow listeners to break the chain and stop propagation of the event to other subscribers. All of this is undesirable, and since an event subscriber is a really simple piece of code, you could just write your own and it would not become a maintenance burden. Unless of course you find a good event dispatcher library that does the trick for you.

Since we’ll soon have several services (the application service for creating the order, an `OrderRepository`, a `Mailer`, etc.), we should create a *composition root* also known as a service container that manages service instances and creates them when they’re needed. This container isn’t the same as the container that the framework offers. While an event dispatcher could in theory be third-party code, our container will be a hand-written one^{[110](#)}. It only contains services needed by the core of our application, so you won’t find a router there, or a template renderer.

Listing [14.12](#) shows the container that we’ll use in the first scenario test.

Listing 14.12: A hand-written service container for testing

```

final class TestServiceContainer
{
    private ?EventDispatcher $eventDispatcher = null;
    private ?ApplicationInterface $application = null;
    private ?OrderRepository $orderRepository = null;

    public function __construct()
    {
    }
}

```

```

public function eventDispatcher(): EventDispatcher
{
    if ($this->eventDispatcher === null) {
        $this->eventDispatcher = new ConfigurableEventDispatcher();
    }

    return $this->eventDispatcher();
}

public function application(): ApplicationInterface
{
    if ($this->application === null) {
        $this->application = new Application(
            $this->orderRepository(),
            $this->eventDispatcher()
        );
    }

    return $this->application;
}

private function orderRepository(): OrderRepository
{
    if ($this->orderRepository === null) {
        $this->orderRepository = new InMemoryOrderRepository();
    }

    return $this->orderRepository;
}

// ...
}

```

Some development guidelines for hand-written containers are:

1. You can create a hierarchy of containers, e.g. `TestServiceContainer` extends from `DevelopmentServiceContainer` extends from an abstract `ServiceContainer`, which has a subclass `ProductionServiceContainer`.
2. Factory methods should be `private` when possible.
3. Within the hierarchy of containers you can override parts by defining (`abstract`) `protected` methods. This also allows you to widen the scopes for testing when needed (e.g. in production the `EventDispatcher` might be a private service, but when testing it could be a public one).
4. Define services as stateless objects so you can return a new instance from the factory method every time it's requested. If you have to make the service stateful or mutable, like the in-memory repositories or the event dispatcher, or if instantiation has to happen often, keep an instance of the service in a property of the container.
5. The constructor of the container can be used to force certain configuration values to be provided. It's recommended to define a configuration value object that provides sensible defaults.

Back to the example: we want to show that, when an order has been created, the customer receives an order confirmation email. Listing [14.13](#) shows what such a test looks like.

Listing 14.13: Using the `TestServiceContainer` in a test.

```
public function the_customer_receives_an_order_confirmation_mail(): void
{
    $container = new TestServiceContainer();

    $orderId = $container->application()->createOrder(
        new CreateOrder(2, 1, 'matthiasnoback@gmail.com')
    );

    // TODO verify that an email was sent
}
```

We haven't yet set up an event subscriber that can send the email, so let's do this now (see Listing [14.14](#)).

Listing 14.14: An event subscriber that will send the confirmation email.

```
final class TestServiceContainer
{
    // ...

    public function eventDispatcher(): EventDispatcher
    {
        if ($this->eventDispatcher === null) {
            $this->eventDispatcher = new ConfigurableEventDispatcher();

            // Register the event subscriber here:
            $this->eventDispatcher->addSubscriber(
                OrderWasCreated::class,
                [$this->sendOrderConfirmationEmail(), 'whenOrderWasCreated']
            );
        }

        return $this->eventDispatcher();
    }

    private function sendOrderConfirmationEmail(): SendOrderConfirmationEmail
    {
        return new SendOrderConfirmationEmail($this->mailer());
    }

    private function mailer(): Mailer
    {
        // TODO return a Mailer
    }
}
```

A use case test should only cover core code, so we should never actually send emails. Still, we want to verify that the `SendOrderConfirmationEmail` will ask the `Mailer` to send that confirmation email. The trick is once more to define a `Mailer` abstraction so we can replace the actual mailer with a test double when testing. Listing [14.5](#) shows the `Mailer` interface I had in mind for this, and how the `SendOrderConfirmationEmail` event subscriber calls it.

```

interface Mailer
{
    public function sendOrderConfirmationEmail(
        OrderId $orderId
    ): void;
}

final class SendOrderConfirmationEmail
{
    private Mailer $mailer;

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

    public function whenOrderWasCreated(
        OrderWasCreated $event
    ): void {
        $this->mailer->sendOrderConfirmationEmail(
            $event->orderId()
        );
    }
}

```

Eventually that `sendOrderConfirmationEmail()` method may need more input, but an `OrderId` should be fine for now.

In the test we'd like to make some kind of assertion to check that the `Mailer` was called. We could do that by making a mock object for the `Mailer` service and inject it into the service container (see Listing [14.15](#)).

Listing 14.15: Injecting a mock `Mailer`.

```

final class TestServiceContainer
{
    private ?Mailer $mailer = null;

    public function setMailer(Mailer $mailer): void
    {
        $this->mailer = $mailer;
    }

    private function mailer(): Mailer
    {
        Assert::that($this->mailer)->instanceOf(Mailer::class);

        return $this->mailer;
    }
}

// inside the test:

// ...

```

```

$mailer = $this->createMock(Mailer::class);
$mailer->expects($this->once())
    ->method('sendOrderConfirmationEmail');

$container->setMailer($mailer);

// ...

```

The downside is that we can't verify that the right argument (the `OrderId` from the created order) has been provided. Another downside is that this approach to creating mock objects is tied to PHPUnit as a test framework. We'll later look at a different test runner for use case tests, so it's smart to stay decoupled from PHPUnit.

The solution is to create a test double of a specific type, a so-called *Spy* object. It will keep a record of what has happened to it, so you can later make assertions about that. Listing [14.16](#) shows an example of a `Mailer` spy.

Listing 14.16: A spy implementation for `Mailer`.

```

final class MailerSpy implements Mailer
{
    /**
     * @var array<OrderId>
     */
    private array $emailsSentFor = [];

    public function sendOrderConfirmationEmail(OrderId $orderId): void
    {
        $this->emailsSentFor[] = $orderId;
    }

    /**
     * @return array<OrderId>
     */
    public function emailsSentFor(): array
    {
        return $this->emailsSentFor;
    }
}

```

We can now instantiate the mailer spy inside the `TestServiceContainer`. This makes it automatically available in other tests as well. We have to make its factory method `public` so we have access to `emailsSentFor()` inside the test (see Listing [14.17](#)).

Listing 14.17: Instantiating the `Mailer` spy in the container.

```

final class TestServiceContainer
{
    private ?Mailer $mailer = null;

    // ...

```

```

public function mailer(): MailerSpy
{
    if ($this->mailer === null) {
        $this->mailer = new MailerSpy();
    }

    return $this->mailer;
}
}

```

Note that I've removed the `setMailer()` method since we don't need it anymore. I've also narrowed the return type from `Mailer` to `MailerSpy`. Even if you have a parent class that returns a `Mailer` from the `mailer()` factory, this is still allowed (according to the Liskov Substitution Principle that is).

We can now finish our first test as shown in Listing 14.18.

Listing 14.18: Verifying that the order confirmation email gets sent.

```

$container = new TestServiceContainer();

$orderId = $container->application()->createOrder(
    new CreateOrder(2, 1, 'matthiasnoback@gmail.com')
);

self::assertEquals(
    $orderId,
    $container->mailer()->emailsSentFor()
);

```

It may be that this particular setup isn't useful in your situation, but I've added this example to give you an impression of the type of setup you need to create use case tests for your application.

“But how do we know if an actual email will be sent?”

Good point; the use case test we created doesn't prove that when the code runs in production an actual email will be sent. We can be certain that the `Mailer` will be called, because the test does prove that. What we also need to prove is that the production implementation of the `Mailer` interface is able to send emails. That would be the job for, you guessed it, an adapter test.

It's perfectly fine to write use case tests for the same test framework that you use for unit and adapter tests, e.g. PHPUnit. However, I've found that writing them in Gherkin, a language

designed for use case tests, makes them really stand out from code-level tests. For PHP the go-to tool is Behat^{[111](#)}, although Codeception^{[112](#)} also has support for scenario-based tests. See Listing [14.19](#) for the Gherkin-version of the test from Listing [14.18](#).

Listing 14.19: A Gherkin scenario

Feature: Ordering an e-book

```
Scenario: the customer receives an order confirmation email
  When a customer creates an order for an e-book
  Then they should receive an order confirmation email
```

Of course, the scenario itself doesn't test anything. It's mainly useful to establish a shared understanding between domain experts and software developers.

“Seems interesting, where can I learn more?”

The practice of describing application features using high-level scenarios that don't zoom in on technology, is known as Specification by Example, Acceptance Test-Driven Development, or Behavior-Driven Development. I'm sure there are good reasons to make a distinction between each of these approaches, but I just wanted to provide a few names that might help you find more information about it. I can recommend the following books on the topic:

1. The BDD Books series by Gáspár Nagy and Seb Rose^{[113](#)}
 2. Gojko Adzic, “Specification by Example”, Manning Publications (2011)
-

In order to use the scenario as an automated test, we have to write some code for every step in the scenario. This code is called a step definition, and the test runner will match each step with a step definition in a so-called *Context* class. Listing [14.20](#) shows an example that uses Behat as test runner.

Listing 14.20: Step definitions in the OrderContext class.

```
use Behat\Behat\Context\Context;
use PHPUnit\Framework\Assert;

final class OrderContext implements Context
{
    private TestServiceContainer $container;

    private ?OrderId $orderId = null;

    public function __construct()
```

```

{
    $this->container = new TestServiceContainer();
}

/**
 * @When a customer creates an order for an e-book
 */
public function aCustomerCreatesAnOrderForAnEbook(): void
{
    $this->orderId = $this->container->application()
        ->createOrder(
            new CreateOrder(2, 1, 'matthiasnoback@gmail.com')
        );
}

/**
 * @Then they should receive an order confirmation email
 */
public function theyShouldReceiveAnOrderConfirmationEmail(): void
{
    Assert::assertInstanceOf(OrderId::class, $this->orderId);

    Assert::assertContainsEqual(
        $this->orderId,
        $this->container->mailer()->emailsSentFor()
    );
}
}

```

Behat reads the first line of the scenario: “When a customer creates an order for an e-book” and looks for a method that has a `@When` annotation followed by the step itself. In this case it finds the method `aCustomerCreatesAnOrderForAnEbook()`. Behat then invokes this method. If this produces an exception, Behat considers the step “failed”. If all goes well, Behat considers the step to be successful and it proceeds with the next step “Then they should receive an order confirmation email”. It matches the step to the `theyShouldReceiveAnOrderConfirmationEmail()` method. Again, this method gets executed. It contains two assertions which might cause exceptions to be thrown. Whether or not this happens determines the success of the scenario.

Using Gherkin to describe use cases has many advantages in my opinion. To mention just a few:

1. It's easier to write in more abstract, high-level terms about the features of your application, leaving out the implementation details.
2. It's easier to specify what the behavior of the application should be in slightly different situations.
3. Scenarios can be written without the underlying code, so you can validate your understanding of the features with other stakeholders who may not know anything about programming.
- 4.

The scenarios are documentation of what your application can do, they specify what it should do, and because there's automation behind them, they can verify that what is documented and specified is actually true about your application. This makes it a form of *Living documentation*¹¹⁴.

Figure 14.6 shows what parts of the application have now been covered with tests.

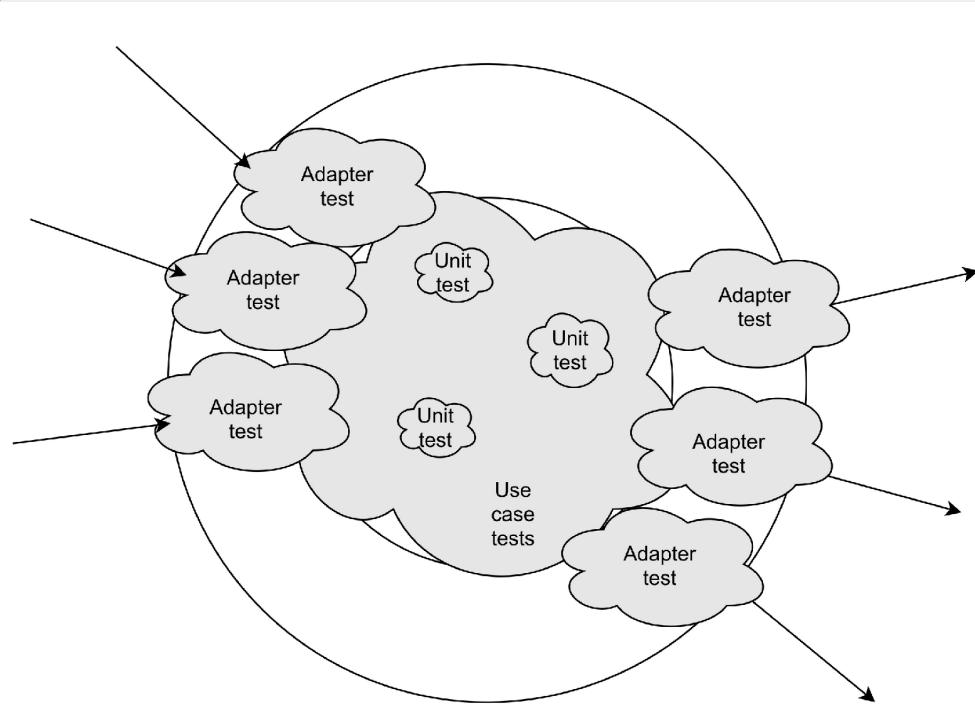


Figure 14.6: Test coverage after adding use case tests.

There's much more to say about Behat, writing scenarios, and automating them, but it's beyond the scope of this book. If you're interested, you can find some more examples of use case tests in the demo project¹¹⁵ that accompanies this book.

14.6 End-to-end tests

Looking at Figure 14.6 it's clear that we have tests for all the relevant parts of the application: the use cases themselves, some unit tests zooming in on a couple of domain objects, and integration tests for the port adapters. Something is missing here: a test that verifies that all these parts work well together once the application runs in a production environment. It turns out there's always the risk that something falls between the cracks and you deploy a broken application. The solution is to write a couple of end-to-end tests that show the application works, from one end of the deployed application to the other end. That is, if you'd draw another coverage diagram, you'd get a coverage cloud wrapping the entire application, including all of its secondary actors.

Ideally you'd run your end-to-end tests against the thing you're going to deploy. If (almost) everything is the same in your end-to-end test environment as it is in your production environment, then you can reduce the chance that once again a mistake slips past quality assurance and gets deployed to the production server.

An end-to-end test should treat the application as a black box. It should have no idea what's going on inside and only talk to it through public channels. In the case of a web application, it may only send HTTP requests and inspect responses. It should not take a peek inside the database or anything.

For web applications it makes sense to use something like Panther^{[116](#)}. The difference between Panther and the standard Symfony `WebTestCase` that we saw in Section [14.4](#) is that Panther talks to the web server itself instead of the framework's entry point.

Because we already have so many tests for different parts of our application, we shouldn't test every bit of functionality once more using an end-to-end test. We only want to increase our confidence a little bit by showing that the parts work well together. Most of the wiring issues can be discovered by running only a few important scenarios as end-to-end tests. It'll be smart to only write a few end-to-end tests because they tend to be slow and unstable. They break for many unrelated reasons, and those reasons are often not programming or configuration mistakes.

14.7 Development workflow

With all these different types of tests, the question is: where to start?

There are different ways to approach the development of a feature. If you don't create decoupled applications, a common starting point is to set up the routing, the controller, the templates. Then perhaps a model and a database migration for it. Maybe then some forms and form validation. All these things are framework-specific so if you'd like to create decoupled applications the workflow will be quite different.

We have an entire catalog of design patterns that we can use to build our decoupled applications with. They could be used as building blocks to build an application from scratch. For instance, we could start with designing our `Order` entity, including some value objects, then add a repository for it, then write the application service. But the risk of starting with the smaller elements and working your way up is that in the end it may turn out that you chose the wrong building blocks, or designed them in the wrong way. You may discover this yourself, or you may get some feedback from another stakeholder who isn't happy with the feature you've built.

This is why in my experience it works best to start with a high-level specification of the feature that you're going to work on. What are the different scenarios that need to be implemented? Challenge these scenarios, come up with possible edge cases. Then write them down in Gherkin.

I think it's a common misconception that scenarios need to be written by business people, not by programmers. I'm afraid this is the reason that many developers don't even think about

writing them. For me the ideal scenario is that the developer discusses the feature with the other stakeholders. The developer then write scenarios that reflect their understanding of the feature. When reading the scenarios, other stakeholders can verify that everybody has a shared understanding of what needs to be done. As a bonus, this increases their amount of trust in the development team.

When the developer starts working on implementing the feature they may do a process modelling session first (as discussed in Section [11.7](#)) and decide on the elements they have to build. Just like writing the high-level scenarios first, designing the high-level elements first may also save a lot of time spent on building the wrong solution. Instead of jumping right into the code, the process can be modelled in pseudo-code first (i.e. using sticky notes). Once the design is somewhat clear it makes sense to start programming.

The programming itself should be test-driven. You'll take a line from the scenario and start writing the code for it, using the elements you chose in the modelling session. Sometimes you may want to zoom in on a particular element, like an entity, or a value object. For these smaller objects you should write unit tests to demonstrate that they protect all their domain invariants.

You'll jump back and forth between unit tests and the scenarios until you have all the scenarios working. At this point you'll have all the *Application* and *Domain* layer code needed for the scenario. The feature is working, but the entire *Infrastructure* layer is still missing. So far the port adapters for outgoing ports are provided by test doubles, and the `Context` classes act as adapters for the incoming ports.

Of course, no stakeholder would consider the new feature “done” at this point. So it's time to wrap the infrastructure layer around it. On the left side you'll need to implement the port adapters for incoming communication. These consist of routing, controllers, templates, etc. On the right side you'll need to implement the port adapters for outgoing communication. There may be some repository implementations to build, some database migrations to generate, and maybe some API clients to develop. When it comes to testing these port adapters you can use the suggestions provided in Section [14.2](#).

Finally you can set up some end-to-end tests. An interesting option is to reuse the scenarios you wrote for the use case tests. Behat allows you to run the same scenario against a different context. During the first run you would test only core code by making calls to the `ApplicationInterface` directly. During the second run you would start the web server and any other required service and test the entire application by making actual HTTP requests.

For me this top-down approach to application development is great because:

- It starts with collaboration: developers and other stakeholders work together to establish a shared understanding of what needs to be created.
- When developers start out to build a feature, they focus on the structural elements first, without building them yet.
- The scenarios that were written to establish that shared understanding can now be used to verify step by step that the implemented solution is the one that was asked for; you won't end up building the wrong thing. You'll know automatically when the work is done so you won't build more than needed.

- Using different types of tests you can zoom in on specific parts, e.g. smaller objects like entities, or specific port adapters. You don't have to test domain logic through slow and unstable end-to-end tests.
- Confidence in the solution is very high. Running the test suite for a given feature proves that everything works, and keeps working.

14.8 Summary

Testing a decoupled application requires several different types of tests. In the first place there will be *Use case tests* that describe the behavior of the application core. Running these tests involves only core code. You can specify the behavior of smaller elements like entities and value objects inside the core with *Unit tests*. An application exposes a number of incoming ports and requires a number of outgoing ports. The adapters for these ports will be tested with *Adapter tests*. Adapter tests aren't unit tests because they involve infrastructure code; non-unit tests are often called integration or integrated tests. To verify that all the parts work well together you may add a number of *End-to-end tests*, which use the application as a real user would, in an environment that mimics as much as possible the production environment.

Exercises

1. For each of the following “things to test”, decide which type of test (unit test, adapter test, use case test, or end-to-end test) should be used for it.^{[117](#)}
 1. When the customer has paid for the order, they should receive an invoice.
 2. When you create an order it records an `OrderWasCreated` event.
 3. When you save an order using the repository’s `save()` method, you can get an equivalent object when you call `getById()` providing the same `OrderId`.
 4. When you send a `POST` request to the `/create-order` URL this causes a call to `ApplicationInterface::createOrder()`.
 5. When you send a `POST` request to the `/create-order` URL then the `/list-orders` page shows the newly created order.
2. Which of the following elements do you need for use case tests?^{[118](#)}
 1. An interface for the application
 2. A router
 3. A service container
 4. An event dispatcher
 5. A template renderer
3. Which of the following elements *shouldn't* be unit-tested?^{[119](#)}
 1. A controller
 2. An application service
 3. An entity

-
- 4. A value object
 - 5. A repository implementation
-

15 Conclusion

This chapter covers:

- Objections
 - Trade-offs
-
-

We've reached the last chapter of this book, and it's not going to introduce more concepts. I wanted to use this chapter to reflect on what has been discussed so far, and to figure out if or when you can use the techniques described in this book in your own projects.

Any advice given to you requires careful analysis in your own context. The advice may sound intuitively right at first, but when considering the specifics of your own project, your own work situation, etc. the advice may be problematic. You may be worried that it's going to take a lot of effort to follow the advice. Maybe you know in advance that some team members will have trouble following it. And maybe you're afraid of "doing it wrong" and this prevents you from finishing the project.

If you consider this book as my advice to design your application in a certain way, then you may have also found it questionable at times. Of course I'm not sure what your personal questions or objections are, but I have collected some questions from readers and workshop participants and I'll be responding to them in this chapter, hoping that they provide some answers to you as well. If this leaves some of your own questions unanswered, don't hesitate to get in touch and ask them directly.

15.1 Is a decoupled architecture the right choice for all projects?

Does something apply to all situations without any qualification? I think that question should always be answered with "No". In the famous words often

used by software developers: “There’s no silver bullet”. Still, there are some practices that are demonstrably better than others. And we’re lucky to have some research confirming that^{[120](#)}. When it comes to decoupled architecture, I didn’t do extensive research but I know that it definitely shouldn’t be applied everywhere. For instance I find that a single job application like the one that extracts URLs from the manuscript of this book and sets up a link registry for it, really doesn’t deserve to be decoupled. Maybe it’s the fact that its work is purely related to infrastructure anyway. Maybe it’s because it’s too small to deserve the effort.

Let’s take a look at some other types of applications that may not be better off with a decoupled architecture.

15.2 My application is not supposed to live longer than two years

One reason to ignore my advice is if your application isn’t supposed to live longer than, say, two years. I realize that learning a new style of architecture, teaching it to the entire team, and implementing it in a short-lived project, may take some more time than if you just keep doing everything the way you’re doing it now. In other words, the costs don’t outweigh the benefits for such a project. If you’re *absolutely certain* the project will be discarded within a couple of years, I agree. But you can’t always predict how long an application will live. It may start as a so-called “proof of concept”. Badly designed, programmed sloppily, just to help the project stakeholders prove some point. Then the business starts making money with it and the project has to survive a little longer. At some point, adding new functionality becomes really problematic. This is a known issue with software, and it has led me to believe that it’s better not to let go of certain practices because you think that the type of project may not live long enough to deserve them.

For instance, I think automated testing should never be considered “optional”. What *is* up to you is which type of tests you’ll create. In Chapter [14](#) we discussed a strategy for testing decoupled applications that may not apply to these POC projects. If you don’t care (for now) about having a decoupled *Domain* and *Application layer*, or clearly distinguishable ports and adapters, then you don’t even have the possibility to test these

elements in isolation. In that case you should still write end-to-end tests to have at least some kind of confidence in your application, now and in the future.

However, just be aware of the classic issues with end-to-end tests: they are slow and fragile, and within a couple of years they will become a real maintenance burden. So my advice is to start decoupling the application at some point, even if the business has been successfully running the software for a couple of years. You have to make sure every stakeholder is aware of the need to work hard, and keep working hard, to keep the software maintainable in the long term (see the totally unscientific Figure 15.1).

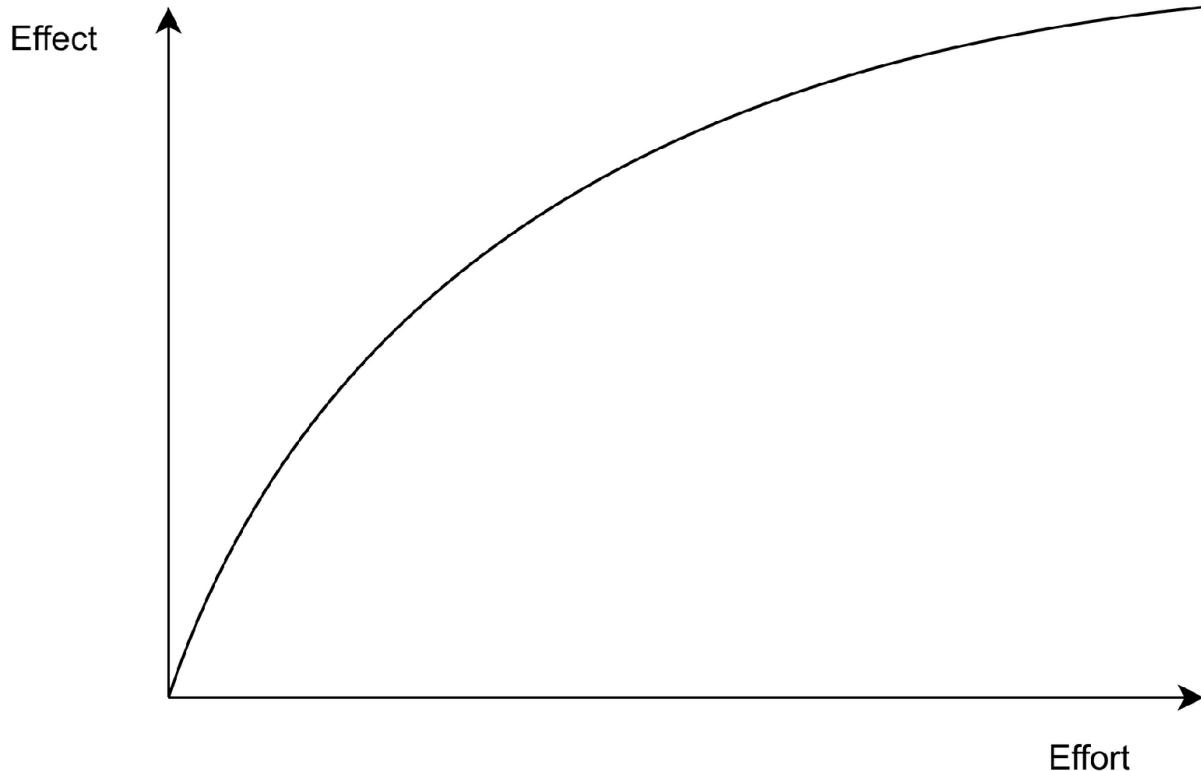


Figure 15.1: Effort needed from developers vs. the effect they produce.

15.3 My application offers only CRUD functionality

Another reason to ignore this book is if your application only offers CRUD functionality. That is, if your application can be generated based on a configuration file which describes the models, their fields and types, and the form field types and validation rules that apply to them. These applications exist. In fact, I've built many of them and I totally agree: they don't deserve a decoupled architecture.

However, I've also seen several applications that started as pure CRUD applications, but very soon CRUD turned out to be a very limited way of modelling the application's business domain. CRUD models don't protect their domain invariants very well. They often don't have any kind of actions defined on them. They don't protect themselves against invalid state transitions, and they don't produce domain events. But domain and process logic still has to be implemented somewhere, so a lot of this logic ends up inside controllers and framework or ORM-specific event listeners. The project becomes a mess, and soon enough you wish you had actually designed everything in a decoupled way. All I'm saying is: take a long and hard look before you decide that the project is CRUD-only and doesn't deserve a decoupled architecture.

15.4 My application is a legacy application

Another common objection to applying the architectural style described in this book is: we are living in a brownfield project, we can never achieve this. I have a lot of experience with legacy projects too, and I sometimes find it hard to stay positive. The force that's pulling you down is just so very strong. It's hard to resist the claim for bankruptcy. At the same time, I know there's always room for improvement. And my secret suggestion for you is to not ask permission for everything. If it helps make your work more bearable, and if it helps you deliver better things faster, be sure to spend a little time improving some aspects of the code base you're working with. Make it easier to write tests. Make it easier to move and rename classes, add parameter and return types to make your code more discoverable. Just don't spend hours or days in a row. Also, don't make too many changes at once, breaking everything, and making everybody angry. Improve your project just a little bit every day. Your legacy application will never be fully decoupled, but it doesn't have to be either. Many areas of an application remain

untouched most of the time, so improving those parts and risking everything isn't even worth your time.

15.5 I can never make my entire application decoupled

Not being able to make your entire application great is another objection to getting started with the decoupled architecture approach in the first place. I think this sentiment can be found in other areas of software development too, not just architecture. Setting a coding standard, installing static analysis tools, these things get postponed forever because you fear that you'll never be able to apply them everywhere. The urge for consistency is something that gets in the way a lot in software development teams. Why not start today? Every improvement you can make, even if it's just in one module, can be the starting point of a better life for everyone on the team, and for the application itself too. Trying to apply a good design idea everywhere is a waste of time, but not applying it anywhere is a wasted opportunity.

15.6 Isn't this over-engineering?

When you're the only one on the team trying to apply these design ideas anywhere at all, you might run into some resistance. All those extra classes, and the extra tests! Why not just do it the way the frameworks tells us to do it? Why not just write functional tests and be done with it? I hope this book has already provided most of the answers. But to address the “over-engineering objection”: I hope I've also been able to show you that the extra classes, interfaces, and tests deserve to be there. They are the result of making a distinction between core and infrastructure code. Keeping these separated requires extra elements in your software. If you feel like core and infrastructure deserve to be apart, doing that extra work can't be considered over-engineering. It's rather *just-right-engineering*. And since I believe that most web applications aren't short-lived projects, nor are they CRUD-only, I believe that we've been mostly under-engineering our web applications, and that it's time to start doing a better job.

While trying to do so, I hope you'll keep an eye on the cost and the benefits of what you're doing. I hope that you'll be pragmatic in your work and stay

away from dogmatism. I hope that you'll keep experimenting and won't be afraid to get it wrong. I hope that you'll succeed in your efforts.

Best of luck,

Matthias

P.S. Let me know how it went!

Notes

¹ <https://advwebapparch.com/blog>

² <https://advwebapparch.com/leanpub>

³ PDO is a PHP extension that provides an API for accessing relational databases. See <https://advwebapparch.com/pdo>.

⁴ Correct answer: Yes. To determine the current time, the application reaches out to surrounding infrastructure, in this case the system's clock.

⁵ Correct answer: No. Even though the code is part of the Symfony framework, it doesn't require any special setup to run. It doesn't require external systems to be available, and it is not designed to run in a specific context, like the terminal or a web server.

⁶ Correct answer: No. Even though the code depends on an interface, the abstraction of the dependency isn't complete. The `HttpClient` interface is designed for HTTP-based communication with external services and can't be replaced with a reasonable alternative in case HTTP is no longer the desired technology.

⁷ Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional (2003).

⁸ <https://advwebapparch.com/beberlei-assert>

⁹ <https://advwebapparch.com/native-assertions>

¹⁰ Eric Evans, "Domain-Driven Design", Addison-Wesley Professional (2003).

¹¹ Eric Evans, "Domain-Driven Design", Addison-Wesley Professional (2003).

¹² <https://advwebapparch.com/doctrine-orm>

¹³ <https://advwebapparch.com/doctrine-orm-and-ddd-aggregates>

¹⁴ <https://advwebapparch.com/effective-aggregate-design>

¹⁵ <https://advwebapparch.com/php-reflection>

¹⁶I've written more about this topic, the trade-offs that are involved, and different implementation options. See: "ORMless; a Memento-like pattern for object persistence",
<https://advwebapparch.com/ormless>.

¹⁷Although the term was coined by Bertrand Meyer, there's a useful summary by Martin Fowler: "CommandQuerySeparation", <https://advwebapparch.com/command-query-separation>. I discuss this topic in detail in "Style Guide for Object Design", Manning (2019).

¹⁸ <https://advwebapparch.com/ramsey-uuid>

¹⁹Eric Evans, "Domain-Driven Design", Addison-Wesley Professional (2003).

²⁰Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional (2003).

²¹Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional (2003).

²²Correct answer: it's a partial abstraction, because it is an interface which is more abstract than a class, but it still leaks implementation details about the underlying persistence mechanism. For instance, it mentions the word "table".

²³Correct answer: it's a value object, which in this case wraps the identity value of an entity.

²⁴Correct answer: it's an entity. Its identity is represented using a value object. Clients have to provide the identity as a constructor argument, making the entity consistent from the start.

²⁵Correct answers: 1 and 3. By using a smart implementation for identity generation, we won't be having duplicate identities.

²⁶Correct answer: No. According to the rules provided in Chapter 1, this is core code.

²⁷Correct answer: Infrastructure code.

²⁸ <https://advwebapparch.com/cqrs-documents>

²⁹ <https://advwebapparch.com/patterns-for-decoupling>

³⁰ <https://advwebapparch.com/udi-dahan-blog>

³¹ <https://advwebapparch.com/twig>

³² Correct answer: No, the recommended approach is to create a separate read model.

³³ Correct answer: 1. An interface method that represents the question, e.g. `listAvailableEbooks()`, 2. A class that represents the answer to the question, e.g. `Ebook`. 3. An implementation for the query method that is able to instantiate answer objects.

³⁴ Correct answer: 1. Using the data source of the write model for the read model as well. 2. Using events from the write model to reflect changes in the read model.

³⁵ Correct answer: 1. Entities and 2. Read models. View models are supposed to provide the data in a way that makes it immediately presentable. A value object would still need to be converted to a primitive type.

³⁶ Correct answer: 2. Options 1. and 3. are a possibility, not a necessity.

³⁷ <https://advwebapparch.com/symfony-console-component>

³⁸ Martin Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Professional (2018). From now on I’ll simply refer to the corresponding web page; in this case:

<https://advwebapparch.com/extract-variable>.

³⁹ <https://advwebapparch.com/extract-class>

⁴⁰ <https://advwebapparch.com/inline-variable>

⁴¹ <https://advwebapparch.com/introduce-parameter-object>

⁴² Correct answer: 1. This enables different types of clients to use the service. Combining input data in a *Parameter object* is optional.

⁴³ Correct answer: 3-1-4-2, although 4 and 2 can be swapped if you like.

⁴⁴ Some examples: a cron job that sends out emails on a daily basis, a message consumer that picks up a message from a queue and processes it as a command, a SOAP application that processes SOAP objects as commands, a long-running process that regularly polls an FTP server for uploaded files that need to be imported. Another very important client of a use case will be the test suite. We’ll get back to this topic in Chapter [14](#).

⁴⁵ <https://advwebapparch.com/laravel-helper-functions>

⁴⁶I've written in more detail about Laravel and its service container on my blog:
<https://advwebapparch.com/laravel-observations> .

⁴⁷ <https://advwebapparch.com/sfcontext>

⁴⁸ <https://advwebapparch.com/zend-registry>

⁴⁹ <https://advwebapparch.com/packagist>

⁵⁰Read more on contextual data and how to deal with it in my blog post "Context passing":
<https://advwebapparch.com/context-passing> .

⁵¹ <https://advwebapparch.com/hand-written-service-containers>

⁵² <https://advwebapparch.com/laravel-mocking>

⁵³ <https://advwebapparch.com/composition-root>

⁵⁴ <https://advwebapparch.com/composition-root-location>

⁵⁵Correct answer: declaring dependencies as constructor arguments makes it clear what those dependencies are. Clients can discover how to instantiate a service by inspecting the arguments and their types. There is no need to set up anything else than just the object itself. If you have provided all the dependencies, nothing else will be needed.

⁵⁶Although 1 could be the case, frameworks will offer a way to reconfigure those dependencies so that they don't need external systems. The correct answer is 2 because you need to do at least some work to prepare the context before you can run the code.

⁵⁷Correct answer: 1 and 5 are service dependencies that should be injected as constructor arguments. 2 is contextual information. 3 and 4 are job-specific data.

⁵⁸Correct answer: dependencies and configuration values should be injected as constructor arguments, the other things should be provided as method arguments.

⁵⁹Correct answer: 2. Core code should always use constructor injection. But there's a point where the first service has to be fetched from the container. This has to be close to the first user code that the framework executes.

⁶⁰ <https://advwebapparch.com/vat-api-docs>

⁶¹John Vlissides; Erich Gamma; Ralph Johnson; Richard HelmDesign, “Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional (1994)

⁶²Martin Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Professional (2018)

⁶³Correct answer: 3. An HTTP client abstraction is useful, but still forces the client to deal with low-level details, like where to make the request to, or what the response looks like. An abstract class is a step in the right direction, but only an interface is abstract enough to give you the flexibility you need.

⁶⁴Correct answer: 2. Being able to create a test double is a requirement for testing the client of the abstraction in isolation. Often the need for isolated testing is actually the reason we introduce an abstraction. But the true test for an abstraction is when you imagine or actually write an alternative implementation where the underlying technology is a completely different one. For instance when you switch from a remote service to a locally stored JSON file. If the abstraction survives such a switch and still works, it's a good one.

⁶⁵Correct answer: 2. A unit test is an isolated test. Given that the object connects to an external service, if you write a unit test for it, you somehow have to prevent it from actually communicating with the external service. But then you don't know if your code works well with the actual service. You need an integration test to prove that.

⁶⁶ <https://advwebapparch.com/ramsey-uuid>

⁶⁷ <https://advwebapparch.com/beberlei-assert>

⁶⁸Read more about object design techniques, like using named constructors, in my book “Object Design Style Guide”, Manning, 2019.

⁶⁹Ross Tuck provides some more suggestions regarding the use of value objects for dates and times in “Precision Through Imprecision: Improving Time Objects”,

<https://advwebapparch.com/precision-through-imprecision> .

⁷⁰Correct answer: 1 (because it fetches the current time from the system clock) and 2 (for the same reason). When you provide a specific value for the second argument of `date()`, it will use that value instead of the return value of `time()`. `checkdate()` also doesn't need the current time to determine if a given date is correct.

⁷¹ <https://advwebapparch.com/php-time-function>

⁷² <https://advwebapparch.com/php-date-function>

⁷³ <https://advwebapparch.com/php-checkdate-function>

⁷⁴ https://advwebapparch.com/php-mt_rand-function

⁷⁵ https://advwebapparch.com/php-mt_srand-function

⁷⁶ Admitted, this is a bit of a grey area. I would say it isn't infrastructure code, since it doesn't require a special context to run in, nor does it rely on external dependencies. One issue is that the code modifies global state by calling `mt_srand()`, which will influence other calls to `mt_rand()`. However, the object itself behaves in a completely predictable way, because the seeding is explicit here. Given the seed is the same, it will also produce the same object.

⁷⁷ <https://advwebapparch.com/symfony-validator-component>

⁷⁸ <https://advwebapparch.com/symfony-form-component>

⁷⁹ Correct answer: all, except validators. A validator should provide a list of errors instead of stopping execution as soon as it encounters an issue.

⁸⁰ Correct answer: in the controller. The application service is supposed to be agnostic about its surrounding infrastructure so it doesn't know anything about the web or HTML forms. A controller knows about the user and how it should present validation errors to them.

⁸¹ Correct answer: 3 and 4. A command object should have properties that are already cast to the correct primitive type, and not all of them have to be nullable. Also, you don't have to validate everything. If you have a good user interface, you don't need to validate most of the input. The entity will protect itself against bad data anyway.

⁸² Robert C. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice Hall (2003).

⁸³ Michael Feathers, "Working Effectively with Legacy Code", First Edition, Prentice Hall (2004)

⁸⁴ Martin Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional (2018)

⁸⁵ Robert C. Martin, "Test First", <https://advwebapparch.com/test-first>

⁸⁶ Martin Fowler, <https://advwebapparch.com/frequency-reduces-difficulty>

⁸⁷ You can always look up the reasoning in Eric Evans' "Domain-Driven Design – Tackling complexity in the heart of software", Addison-Wesley Professional (2003). A quick and accurate primer on the topic is Vaughn Vernon's article series "Effective Aggregate Design",
<https://advwebapparch.com/effective-aggregate-design>.

⁸⁸ <https://advwebapparch.com/talis-orm>

⁸⁹Not to be confused with the *Command* design pattern.

⁹⁰Available on Leanpub: <https://advwebapparch.com/event-storming> .

⁹¹Robert C. Martin, “Agile Software Development, Principles, Patterns, and Practices”, Prentice Hall (2003).

⁹²Robert C. Martin, “The Clean Architecture”, <https://advwebapparch.com/clean-architecture>

⁹³See Chapter 4: Architecture, in Vaughn Vernon, “Implementing Domain-Driven Design”, Addison-Wesley Professional (2013).

⁹⁴If you know about such a tool, or are building one, please let me know.

⁹⁵ <https://advwebapparch.com/deptra>

⁹⁶Correct answer: 2. 1 is paraphrasing the *Dependency inversion principle* which in fact helps implement 2, the *Dependency rule*. 3 is the *Open/Closed Principle*, which is nevertheless quite useful.

⁹⁷3 and 4. The other classes belong to the *Application* layer.

⁹⁸Correct answer: 1 and 5. The other classes belong to the *Infrastructure* layer.

⁹⁹Correct answer: 2, 3 and 4. An entity belongs to the *Domain* layer, a *Command DTO* is the parameter type of an application service, which belongs to the *Application* layer.

¹⁰⁰At the time of writing his original post is offline, but you can still find it in the web archive <https://advwebapparch.com/cockburn-hexagonal-architecture> . I also recommend watching the video of his talk at the DDD FR meetup in Paris, 2017: <https://advwebapparch.com/alistair-in-the-hexagone>

¹⁰¹Read more about marking classes as `final` in “Final classes by default, why?”: <https://advwebapparch.com/final-classes> .

¹⁰²Correct answer: 1.

¹⁰³Correct answer: 2.

¹⁰⁴Correct answer: 2.

¹⁰⁵Correct answer: 1.

¹⁰⁶Correct answer: 1. Only if all ports are defined as interfaces will it be possible to test the port adapters without invoking the code inside the hexagon.

¹⁰⁷Michael Feathers, “A Set of Unit Testing Rules” (2005): <https://advwebapparch.com/unit-test-definition>

¹⁰⁸ <https://advwebapparch.com/symfony-testing>

¹⁰⁹ <https://advwebapparch.com/panther>

¹¹⁰See also Matthias Noback (2019), “Hand-written service containers”: <https://advwebapparch.com/hand-written-service-containers>

¹¹¹ <https://advwebapparch.com/behat>

¹¹² <https://advwebapparch.com/codeception>

¹¹³ <https://advwebapparch.com/bdd-books>

¹¹⁴My favorite work on this topic is by Cyrille Martraire, “Living Documentation”, Addison-Wesley Professional (2019).

¹¹⁵ <https://advwebapparch.com/repository>

¹¹⁶ <https://advwebapparch.com/panther>

¹¹⁷Correct answer: 1 use case test, 2 unit test, 3 adapter test (more specifically: a contract test), 4 adapter test, 5 end-to-end test.

¹¹⁸Correct answer: 1, 3 and 4.

¹¹⁹Correct answer: 1 A controller should be tested with an adapter test or an end-to-end test because it’s not core code. 2 An application service is core code so it can be unit-tested, but it makes more sense to test it as part of a use case test. 5 A repository implementation is infrastructure code so it requires an adapter test, which is not a unit test but an integration test.

¹²⁰See for example Nicole Forsgren, Jez Humble, Gene Kim, “Accelerate”, IT Revolution Press (2018).