Advanced Web Application Architecture

Matthias Noback

Advanced Web Application Architecture

Matthias Noback

Contents

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 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.

1. 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. Tests will be easy to make and they 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.

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 takes more time to become a mess, and this is already a huge competitive advantage.

Overview of the contents

This book is divided into thee 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 **??**).
- 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 **??**)
- Separate *what* we need from external services from *how* we get it (Chapter **??**)
- Work with current time and random data independently from how the running application will retrieve this information (Chapter **??**)

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 **??**).

Part **??** ("Organizing principles") provides an overview of the organizational principles that can be applied to an application's design at the architectural scale. Chapter **??** 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 **??** 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 **??** 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 Part **??** we reach the book's conclusion by looking at a possible testing strategy for decoupled applications (Chapter **??**). We finish the book with a discussion of the particular approach to architecture put forward in this book, and how you can decide if it's the right approach for your projects.

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 Gitstore[1] to get access right away.

[1] `https://enjoy.gitstore.app/repositories/read-with-the-author/read-with-the-author`

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 blogging about all sorts of programming-related topics on matthiasnoback.nl. 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

Changelog

1. April 15th, 2020 (initial version)

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

2. April 23rd, 2020

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

1. Released: Chapter 2: 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)

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 PDO[1] 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;
```

[1]PDO is a PHP extension that provides an API for accessing relational databases. See https://www.php.net/manual/en/intro.pdo.php.

```
    }
}
```

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.

```php
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 {
        // ...
    }
}
```

It's not a coincidence that the classes in this example are domain-oriented. In Chapter **??** we will discuss architectural layering and we'll 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.

"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.

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.

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?[a]

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

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

2. Should the code below be considered *infrastructure* code?[b]

```
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?[c]

```
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'
        );

        // ...
    }
}
```

[a]Correct answer: Yes. To determine the current time, the application reaches out to surrounding infrastructure, in this case the system's clock.

[b]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.

[c]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.

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);
$values = array_map(
    function ($value) use ($connection) {
        return $connection->escape($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 **??**

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 pattern[1]. 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'),
```

[1] Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional (2003).

```
            '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`[2]), or using native assertions[3] 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);
```

[2] https://github.com/beberlei/assert
[3] https://www.php.net/manual/en/function.assert.php

```
        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 **??**.

A stateful object that guarantees its own consistency, and is going to be persisted somehow, is often called an *Entity*[4]. 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(
```

[4] Eric Evans, "Domain-Driven Design", Addison-Wesley Professional (2003).

```
            [
                // 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 pattern[5] 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
     */
```

[5] Eric Evans, "Domain-Driven Design", Addison-Wesley Professional (2003).

```
    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);
        $values = array_map(
            function ($value) {
                return $this->connection->escape($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[6]. 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
{
```

[6] https://github.com/doctrine/orm

```
    /**
     * @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.[7]

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 Vernon[8]. We'll get back to this topic in Section **??**.

How is Doctrine able to get the data out of the entity? It uses *reflection*[9] 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.

```php
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`...
```

[7] https://matthiasnoback.nl/2018/06/doctrine-orm-and-ddd-aggregates/
[8] https://dddcommunity.org/library/vernon_2011/
[9] https://www.php.net/manual/en/book.reflection.php

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

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.[10]

[10] I'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://matthiasnoback.nl/2018/03/ormless-a-memento-like-pattern-for-object-persistence/.

"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:

```php
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()`.

```php
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.

```php
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 creat 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.*[11]

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`[12] library for generating a random UUID.

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

[11] Although the term was coined by Bertrand Meyer, there's a useful summary by Martin Fowler: "CommandQuerySeparation" (`https://martinfowler.com/bliki/CommandQuerySeparation.html`). I discuss this topic in detail in "Style Guide for Object Design", Manning (2019).

[12] `https://github.com/ramsey/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*[13]. 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

[13]Eric Evans, “Domain-Driven Design”, Addison-Wesley Professional (2003).

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[14] 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[15]. 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:

[14] Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional (2003).

[15] Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional (2003).

- By inheriting a lot of infrastructure code, we loose 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 **??** 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 Part **??** 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 **??**, 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?[a]

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

2. Is the following class an entity or a value object?[b]

```
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?[c]

```
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?[d]

   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.

[a] Correct 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".

[b] Correct answer: it's a value object, which in this case wraps the identity value of an entity.

[c] Correct 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.

[d] Correct answers: 1 and 3. By using a smart implementation for identity generation, we won't be having duplicate identities.

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 **??**? 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...
}
```

```php
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`.

```php
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 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, $bookingPeriod 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($vatReturn->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 to synchronize the read model

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 **??**, 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.
 */
```

```php
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.

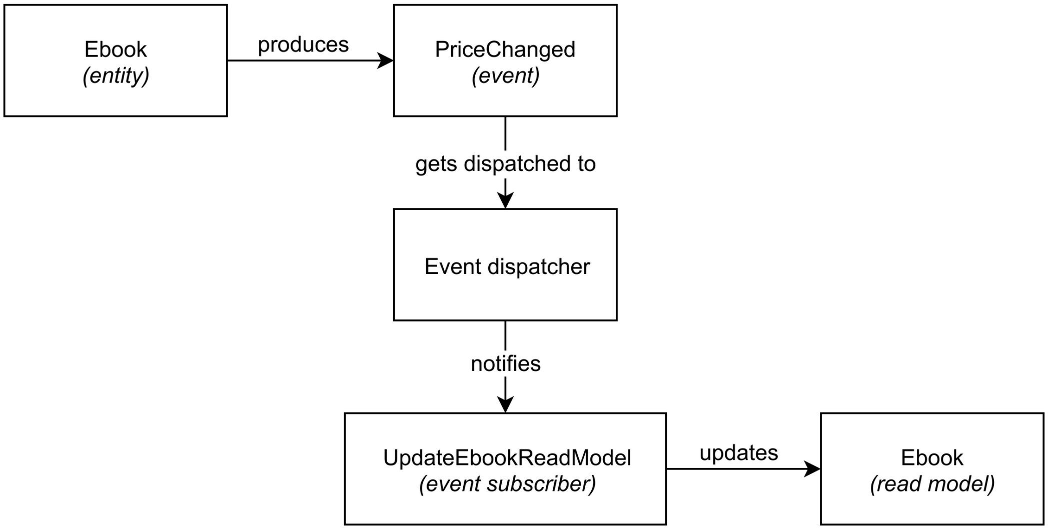

Figure 3.1. Using write model events to update a read model.

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(/* ... */);
}
```

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 **??** 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...
$ebookPrice = $ebook->price();

// ... so we have to convert it to an integer before multiplying it:
$orderAmountInCents = (int)$request->get('quantity')
    * $ebookPrice->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 = $ebookPrice->price()->multipliedBy(
    (int)$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 world outside, 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 Twig[1] 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>

<table>
    <thead>
        <tr>
            <th>Title</th>
            <th>Number of readers</th>
            <th>Price</th>
            <th>Actions</th>
        </tr>
    </thead>
    {% for ebook in ebooks %}
        <tr>
            <td>
```

[1] https://twig.symfony.com/

```
                {{ ebook.title }}
            </td>
            <td>
                {{ ebook.numberOfTimesSold }}
            </td>
            <td>
                {{ ebook.price }}
            </td>
            <td>
                <a href="{{ path('order_ebook', { ebookId: ebook.ebookId }) }}"
                    Order now
                </a>
            </td>
        </tr>
    {% endfor %}
</table>

{% endblock body %}
```

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(
            function (array $record): Ebook {
                return 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, and 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.

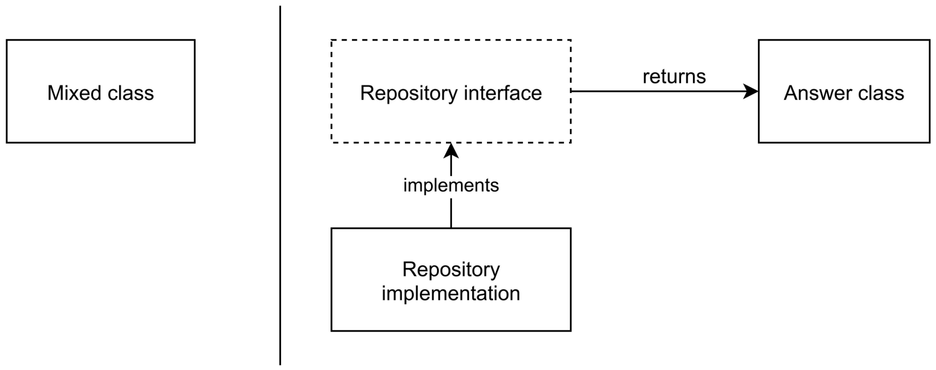

Figure 3.2. After decoupling a query from underlying infrastructure, you will have three elements.

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 `echo`-ing 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(
            function (Ebook $ebook): array {
                return $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?[a]
2. A read model consists of three class/interface elements. What are they?[b]
3. What are two common ways of keeping a read and a write model synchronized?[c]
4. Some models return value objects from their methods. Which ones?[d]
    1. Entities
    2. Read models
    3. View models
5. What is a true requirements for read models?[e]
    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.

[a]Correct answer: No, the recommended approach is to create a separate read model.

[b]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.

[c]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.

[d]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.

[e]Correct answer: 2. Options 1. and 3. are a possibility, not a necessity.

4. The end (for now)

I will keep publishing chapters on a regular basis, until the book is done. Thank you very much for buying this book and being my early supporter!

Please tell me about things in this book that are unclear, incomplete, or plain wrong.

Context me:

- By email: info@matthiasnoback.nl
- On Twitter: @matthiasnoback