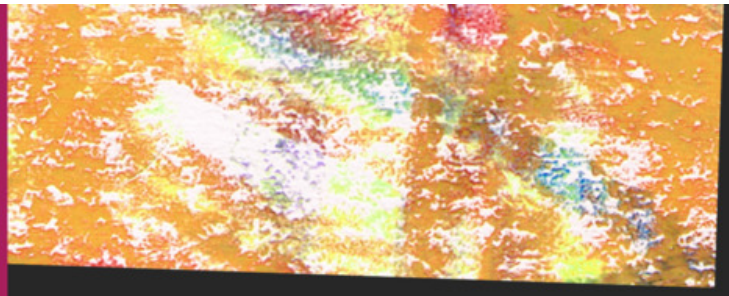

Advanced Web Application Architecture

Matthias Noback

Advanced Web Application Architecture

Matthias Noback

Advanced Web Application Architecture

Matthias Noback

ISBN 978-90-821201-6-5

Cover drawings by Julia Noback

Other books by Matthias Noback:

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

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

Contents

Introduction

1. Preface

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

2. Why is decoupling from infrastructure so important?

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

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

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

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

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

3. Who is this book for?

This book is for you:

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

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

4. Overview of the contents

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

- Extract a domain model from code that mixes SQL queries with business decisions (Chapter **??**)
- 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 **??**)
- 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 Chapter **??** we look at a possible testing strategy for decoupled applications. With Chapter **??** we reach the book's conclusion.

5. The accompanying demo project

Of course all the design techniques and principles discussed in this book are illustrated with many code samples. However, these samples are always abbreviated, idealized, and they show only the most essential aspects. To get a full understanding of how all the different parts of an application work together, we need a demo project. Again, not an idealized or simplified one, but a real-world project that is running in production. People have often asked for such a project and I've always answered: I'd love to work on that, now I need to find some time. This time I decided to make it happen. The demo project is the source code for the new *Read with the Author* platform. This software runs in production. In fact, you may have used the software

already if you bought a ticket for it on Leanpub. But the most important quality of the project is that it shows the design techniques and principles from this book in practice. You can explore the source code on GitHub `https://advwebapparch.com/repository` to get access right away.

6. About the author

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

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

Since 2011 he's been writing about all sorts of programming-related topics on his blog[1]. Other books by Matthias are *Principles of Package Design* (Apress, 2018), and *Object Design Style Guide* (Manning, 2019).

You can reach Matthias:

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

7. Acknowledgements

This has been the sixth book I published using the Leanpub[2] platform. It has always been a great experience, so thanks again Peter Armstrong and Lenn Ep.

[1] `https://advwebapparch.com/blog`
[2] `https://advwebapparch.com/leanpub`

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

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

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 infrastruc-

ture 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;
    }
}
```

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

[1] PDO is a PHP extension that provides an API for accessing relational databases. See https://advwebapparch.com/pdo.

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 **??**.

1.3. Rule no 2: No special context needed

The second rule for core code is:

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

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

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

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

        $host = $_SERVER['HTTP_HOST'];
```

```
        // ...
    }

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

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

        // ...
    }

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

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

        // ...
    }
}
```

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

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

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

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

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

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

Listing 1.6. Some examples of core code.

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

final class MemberService
{
    private MemberRepository $memberRepository;

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

```
            PurchaseId::fromString($purchaseId)
        );

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

final class EmailAddress
{
    private string $emailAddress;

    private function __construct(string $emailAddress)
    {
        if (!filter_var($emailAddress, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException('...');
        }

        $this->emailAddress = $emailAddress;
    }

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

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

Not having to create a special context for code to run in is, again, great for testability. The only thing you have to do in a test scenario is instantiate the class and call a method on it. But following the rules for core code isn't just great for testing. It also helps keeping your core code protected against all

kinds of external changes, like a major framework upgrade, a switch to a different database vendor, etc.

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

This also explains why I'm using the words "core" and "infrastructure". Infrastructure is a common term used to describe the technical aspects of an interaction. In a web application, infrastructure supports the communication between your application and the outside world. The core is the center of your application, the infrastructure is around it, both protecting the core and connecting it to external systems and users (Figure 1.1).

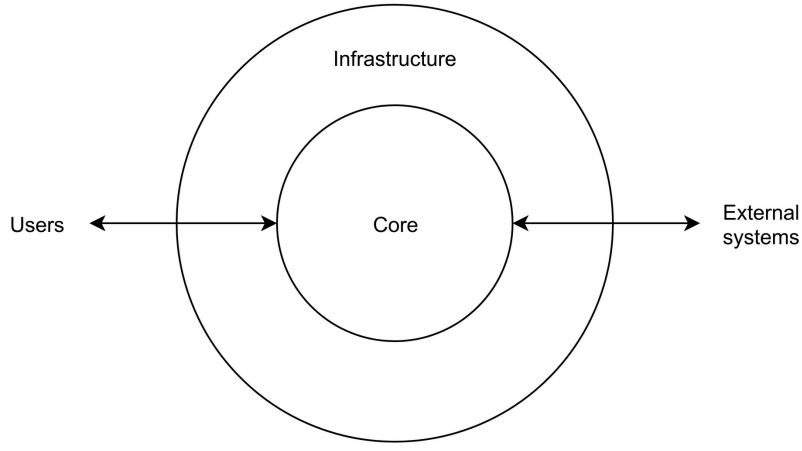

Figure 1.1. Connecting the core to external systems and users through infrastructure

---

"Is all code in my vendor directory infrastructure code?"

Great question. In `/vendor` you'll find your web framework, which facilitates communication with browsers and external systems using HTTP. You'll

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

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

1.4. Summary

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

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

Exercises

1. Should the code below be considered *infrastructure* code?[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 End of the Sample

I hope you liked this sample file and are ready to buy the whole book now so you can enjoy hundreds of additional pages explaining how to design your application in such a way that:

1. Domain-driven design becomes a natural approach
2. Tests are easy to write, are stable, and run fast
3. Your project will be maintainable for years to come

When you buy the book, make sure to use the following link to get a 10% discount:

```
http://leanpub.com/web-application-architecture/c/SAMPLE
```

Best regards,

Matthias Noback

2. The End of the Sample