DOMAIN-DRIVEN DESIGN

IN PHP

Carlos Buenosvinos
Christian Soronellas
Keyvan Akbary

Domain-Driven Design in PHP

Real examples written in PHP showcasing DDD Architectural Styles, Tactical Design, and Bounded Context Integration

Tweet This Book!

Please help Carlos Buenosvinos, Christian Soronellas and Keyvan Akbary by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought ”Domain-Driven Design in PHP” (@dddbook) by @theUniC, @keyvanakbary and @buenosvinos https://leanpub.com/ddd-in-php #ddd #php

The suggested hashtag for this book is #DDDinPHP.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#DDDinPHP

This book is dedicated to my dearest Vanessa, Valentina and Gabriela. Thanks for your love, your support, and your patience. Carlos

To my dear Elena. Without your encouragement, your love and your patience this book would not have been possible. Christian.

Contents

Foreword

TBC

Vaughn Vernon Shift Method, Inc.

Preface

Since 2012 and after two years reading and working with Domain-Driven Design approaches, in 2014, Christian and Carlos went to Berlin to get trained by Vaughn Vernon, author of "Implementing Domain-Driven Design" book. The training was fantastic, all the concepts that were going around on their minds up to that moment, got stuck into the ground in the 3 days IDDD workshop. However, they were the only two PHP developers there in a room full of Java and .NET. That was quite funny. Four years after, in 2016, Vaughn was giving his workshop in Barcelona with their help.

In 2014, Carlos sent a paper about Hexagonal Architecture to the php[tek]. His talk was not approved but Eli White got in touch with him. Are you interested in writing an article about Hexagonal Architecture for the php|arch magazine? Great! So in June 2014, "Hexagonal Architecture with PHP" was published. You'll find it included in the Appendix.

That article was the beginning of this book. Christian was excited about the idea behind the book and decided to get involved since the first chapter. Then, we met Keyvan, the third rider pushing for such a challenging aim.

Carlos has been leading agile teams from 20 to 100 people. He's Certified Scrum Master since 2010 and has seen quite a lot of different teams facing the challenge about writing code cheap to maintain. DDD has played a significant role when dealing with big teams.

So, we've written the book we wanted to have when started with Domain-Driven Design. Full of examples, production-ready code, shortcuts and our recommendations based in our experience about what worked and didn't work for our teams. We arrived to DDD via the tactical patterns and fallen in love then with the strategical parts.

DDD and PHP Community

Domain-Driven Design has arrived to the PHP community with lots of talking but no real code or scenario detailing how to implement Tactical DDD patterns or how to integrate Bounded Contexts with REST and/or messaging.

In 2016, Christian and Carlos went to the first DDD conference. It was the "DDD Europe". They were really happy to see so many PHP tech leads attending the conference for top PHP open-source projects (Doctrine, PHPUnit, etc.).

Inspired by "Implementing Domain-Driven Design", aka, the Red Book of DDD, and "Domain-Driven Design: Tackling complexity in the Heart of Software" by Eric Evans, Carlos, Christian and Keyvan show, with tons of details and examples, how to properly design Entities, Value Objects, Services, Domain Events, Aggregates, Factories, Repositories, Services and Application Services with PHP. What is the role of the main PHP libraries and frameworks used today (Doctrine, Symfony, Silex, etc.). They show how to apply Hexagonal Architecture within your application whether you use an open source framework or your own. They show how to integrate Bounded Contexts using REST frameworks and messaging mechanisms.

It would be nice to know the basic concepts of DDD in order to see their implementations in PHP, however, a totally newbie to DDD can use our book as a guide to get into. The book is currently being written, so please feel free to send them suggestions or comments.

Summary of Chapters

The book is arranged in different chapters exploring each of the tactical building blocks of Domain-Driven Design. It also includes a introduction to DDD, how to integrate different Bounded Context or Applications and some interesting appendixes.

Chapter 1: Getting Started with DDD (Finished)

What is Domain-Driven Design about? What role does it play in complex systems? Is it worthy? What are the main concepts a developer needs to know when jumping into it?

Chapter 2: Architectural Styles (Finished)

Bounded Contexts can be implemented in different ways and using different approaches. However, two styles are getting more popular, Hexagonal Architecture and CQRS + ES. In this chapter, we'll see both solutions and understand what are the key considerations in using them.

Chapter 3: Value Objects (Finished)

Value Objects are probably the basic pieces for rich modeling. We'll learn what are their properties and what make them so important. We'll check how to persist them using Doctrine and custom ORMs, how to validate them and properly unit test them considering immutability.

Chapter 4: Entities (Finished)

Entities are the identified by identity building blocks of DDD. We'll see how to create and validate them, how to properly map them using a custom ORM and Doctrine. We'll also review the annotations yes-or-no flame and the different strategies for generating identity.

Chapter 5: Domain Services (Finished)

In this chapter, you'll learn about what a Domain Service is and when to use it. We'll review what are Anemic Domain Models and Rich Domain Models. Last, we'll deal with infrastructure issues when writing Domain Services.

Chapter 6: Domain Events (Finished)

Domain Events are a great Inversion of Control (IoC) mechanism. In DDD, they are used for asynchronous communication between Bounded Contexts, decoupling infrastructure and eventual consistency.

Chapter 7: Modules (Finished)

With so many tactical building block it's a bit difficult to know where to place them in code. Specially if you are dealing with a framework like Symfony. We'll check what's our suggestion that is working quite well for the teams practicing it.

Chapter 8: Aggregates (Work in progress)

Aggregates are probably the most difficult part of tactical DDD. We'll see what are the key concepts when dealing with and how to design them. We'll see a practical scenario where two aggregates become one when adding a business rule and how the rest of the objects must be refactor.

Chapter 9: Factories (Finished)

Factory methods and Factory objects help us to keep business invariants, that's why they are so important in DDD. Last, we'll check the relation between Factories and Aggregates.

Chapter 10: Repositories (Finished)

Repositories are key for retrieving and adding Entities and Aggregates to collections. We'll review the different types of repositories. We'll learn how to implement them using Doctrine, custom ORMs, and Redis.

Chapter 11: Application Services (Finished)

Application is the thin layer that connects clients from outside to your Domain. In this chapter, we'll show you how to write your Application Services so they are easy to test and keep thin. We'll review how to prepare request objects, dependencies, and returning results.

Chapter 12: Integrating Bounded Contexts (Finished)

We'll explore the different tactical approaches to communicate Bounded Contexts and see real implementations. REST is our suggestion for synchronous communication and messaging with RabbitMQ for asynchronous.

Appendix A: Hexagonal Architecture with PHP (Finished)

Here, you'll find the original article written by Carlos Buenosvinos and published in June 2014 by the php|architect magazine that was the seed for this book.

Code and examples

The authors have created an organization at GitHub.com (https://github.com/dddinphp) where all the code examples from this book, additional snippets and some whole sample projects are available. If you are interested, watching those projects and providing feedback is totally recommended.

Acknowledgements

We are three Spaniards working on an English book. You guess right if you think that our English is far from perfect. Edd Mann has been supporting us with the language since the beginning. He is not just a great collaborator but also a big friend. We owe him a huge thanks. We would also say thanks to Albert Casademont and Ricard Clau for helping with the revision process.

We would like to thank to all the early adopters that bought the book at the beginning and gave us the needed love and support for keep pushing.

Github contributors

We would like to thank also to all the people that have reported issues, make suggestions, etc. in our GitHub repository[1]. To all of them, thank you very much, you have helped us to make this book better and what is more important, help the community and other developers to be better developers. As Dave Thomas[2] wrote in his book "The Pragmatic Programmer": "If you're reading this book is because you want to be a better developer. Great, we need better developers.".

So thanks to Jordi Abad, Jonathan Wondrusch, César Rodríguez, Yannick Voyer, Oriol González, Henry Snoek, Tom Jowitt, Nico Oelgart, Sascha Schimke, Sven Herrmann, Daniel Abad, Luis Rovirosa, and Marc Aube.

[1] https://github.com/dddinphp/ddd-in-php-book-issues

[2] https://twitter.com/pragdave

About the Authors

Carlos Buenosvinos

PHP Extreme Programmer working as CTO at Atrapalo.com, board member of the PHP Barcelona User Group, Agile Coach, Valu and Gabriela's father, and Vanessa's husband. He has worked in ecommerce (Atrapalo.com and Ebay International), payment processing (Vendo), and classifieds (Emagister.com). He likes developing for mobile, Raspberry Pi, and games. Zend PHP, Scrum Master, and MySQL 5.6 certified.

- Twitter: @buenosvinos[3]
- GitHub: https://github.com/carlosbuenosvinos

Christian Soronellas

Passionate software developer. Software Journeyman and craftsman apprentice. Extreme Programmer soul. Over 10 years of experience in web development. Zend PHP 5.3 Certified Engineer. Zend Framework Certified Engineer. SensioLabs Certified Symfony Developer. He has worked as a freelance, at Privalia, Emagister, and currently is working as a Software Architect at Atrapalo.

- Twitter: @theUniC[4]
- GitHub: https://github.com/theUniC

Keyvan Akbary

Polyglot Software Developer. Loves Software fundamentals, Craftsmanship movement, Extreme Programming, SOLID principles, Clean Code, Design Patterns and Testing. Sporadic Functional Programming adventurer. He understands technology as a medium for providing value to the business. He has worked as freelance for small and big clients, at Youzee, MyBuilder, founded Funddy, and currently working as a Lead Developer at TransferWise London.

- Twitter: @keyvanakbary[5]
- GitHub: https://github.com/keyvanakbary

[3] https://twitter.com/buenosvinos
[4] https://twitter.com/theUniC
[5] https://twitter.com/keyvanakbary

1. Getting Started with DDD

So what is all the fuss about? Domain-Driven Design, or DDD, is an approach to help us succeed in understanding and building software model designs. It provides us with *strategic* and *tactical* modeling tools to aid designing high-quality software that meets our business goals.

More importantly, **Domain-Driven Design is not about technology**. DDD is about developing knowledge around the business and using the technology to provide value. Only once you are capable of understanding the business your company works within, you will be able to participate in the software model discovery process to produce a **Ubiquitous Language**.

1.1 Why Domain-Driven Design?

- Software should not make sense only for coders but also for the business. DDD empathises making sure business and software talk the same language.
- Software priorities are aligned with business priorities.
- With DDD everybody learns and contributes discovering the business domain.
- Knowledge no longer belongs just to developers, with DDD everyone knows what is going on with the business.
- There are no translations between domain experts, meaning no information loss or tedious syncing. Everyone talks the same language.
- The design is the code and the code is the design, the only implemented truth for the common language. Focused on delivering software continuously through agile discovery processes.
- DDD provides a framework for strategic and tactical design. Strategic to pin-point the most important areas to develop based on business value and tactical about battle-tested building blocks and patterns.

1.2 How Domain-Driven Design helps?

Domain-Driven Design is an approach for delivering software, focused on three pillars:

1. *Ubiquitous Language*: Domain experts and software developers work together to build a common language for the business areas that are being developed. There is no “us versus them”, it is always *us*. Developing software is a business investment not just a cost. The effort in building the ubiquitous language helps spread deep domain insight among all team members.

2. *Strategic Design*: Domain-Driven Design addresses the strategy behind the direction of the business, not just the technical aspects. It helps define the internal relationships and early-warning feedback systems. On the technical side, strategic design protects each *business service* by providing the motivation for how service-oriented architecture should be achieved.
3. *Tactical Design*: Domain-Driven Design provides the tools and the building blocks for iterative software deliverables. Tactical design tools produces software that is correct as maps domain experts mental model, is testable and less error prone.

1.2.1 Ubiquitous Language

Along with Bounded Contexts the Ubiquitous Language is one of the main strengths of DDD.

In terms of context

For now, consider a *Bounded Context* is a conceptual boundary around a system.

The Ubiquitous Language inside a boundary has a specific contextual meaning. Concepts out of this context can have a different meaning.

So, how to capture this special language?

- Label with names for actions physical and conceptual domain concepts.
- Create a glossary of terms and definitions.
- Capture important software concepts with some kind of documentation.
- Share and evolve the knowledge collected with the rest of the team.

1.3 Should I start considering Domain-Driven Design as an option?

Domain-Driven Design is not a silver bullet, as everything in software, it depends on the context. As a rule of thumb, use it to simplify your domain, never to add more complexity.

If your application is data-centric and use-cases evolve around data manipulation and CRUD operations - this is, Create, Read, Update and Delete - you do not need DDD. The only thing your company needs is a fancy face in front of your database.

If your application has less than 30 use-cases, it might be simpler to use a framework like Symfony or Laravel to handle your business logic.

If you have more than 30 use-cases, your system maybe moving towards the dredded 'big ball of mud'. If you know for sure your system will grow in complexity, you should start considering using DDD to fight complexity.

If you know your application is gonna grow and is likely to change often, DDD will definitively help in managing the complexity and refactoring your model over time.

If you do not understand the Domain you are working on because it is new and nobody invested on a solution before, this might mean it is complex enough to start applying DDD. You will need to work closely with *Domain Experts* to get the models right.

1.4 Main challenges of applying Domain-Driven Design

On your journey for applying Domain-Driven Design, you will encounter several challenges.

Applying Domain-Driven Design completely will require thinking about the business domain, terminology, research and collaboration with domain experts rather than coding jargon. It will require time and effort.

You need to have the commitment of Domain experts for getting involved in the process of building software. You will need domain experts to uncover deep knowledge of the domain. It will require an open, healthy, respectful and continuous conversation with the experts to model their spoken language into software.

We developers are technical thinkers. Technical solutions are our speciality. Thinking in technical problems is not bad, the only problem is that sometimes thinking less technically is better. In order to think in the behaviours of objects we need to think in the Ubiquitous Language first.

1.5 The business value of using Domain-Driven Design

The best way of justify a technology or technique is to provide value to the business. Summarising, the main benefits of applying DDD are:

- Useful and meaningful model of its domain
- Domain experts contribute to software design
- Better user experience
- Clear boundaries
- Better architecture organization
- Iterative and continuous modeling on agile fashion
- Better tools, strategic and tactical

1.6 Wrap-up

Implementing Domain-Driven Design requires effort. If it were easy everybody would be writing high-quality code. Get ready because through this journey you will learn how to make your design look exactly how your software works. During this chapter you have learned:

- Domain-Driven Design is not about technology, is actually about providing value in the field you are working on, by focusing on model. Everyone takes part in the process of discovering the domain, developers and domain experts team up to build the knowledge base by sharing the same language, the *Ubiquitous Language*.
- Domain-Drive Design provides *tactical* and *strategic* modeling tools to design high-quality software. Strategic design targets the business direction, helps defining the internal relationships and technically protects each *business service* by defining strong boundaries. Tactical design provides useful building blocks for iterative design.
- We have studied the context in which DDD makes sense. DDD is not a silver bullet for every problem in Software, and highly depends on the amount of complexity you are dealing with.
- We have also seen that applying DDD is a long-term investment, it requires active effort. Domain experts will be required to collaborate closely with developers, and developers will have to think in terms of the business. In the end, it is the business customer that is the one that has to be pleased.

2. Architectural Styles

In order to be able to build complex applications, one of the key requirements is having an architectural design that fits the applications needs. A good advantage of Domain-Driven Design is that it is not tied to any particular architecture style. Instead, we are free to choose the architecture that best fits the needs of every *Bounded Context* inside the *Core Domain*, offering a diverse set of architectural choices for every specific domain problem.

For example, an *Order Processing System* can use **Event Sourcing** to track all the different order operations, a *Product Catalog* can use **CQRS** to expose the product details to the different clients and a *Content Management System* can use plain **Hexagonal Architecture** to expose requirements such as blogs, static pages, and so on.

This chapter presents an introduction to every relevant architecture style in the land of PHP, through an evolution from traditional 'old-school' PHP code to a more sophisticated architecture. Please note that although there are many other existing architecture styles, like *Data Fabric* or *SOA*, we found them a bit complex to introduce from the PHP perspective.

2.1 The Good Old Times

Before the release of PHP version 5, the language did not embrace the Object-Oriented paradigm. Back in these days, the usual way to write applications was by using procedures and global state. Concepts like *Separation of Concerns*, *MVC* and such were very alien among the PHP community. The example below, is an application written in this *'traditional way'*, where applications were composed of many front controllers mixed with HTML code. During this time *Infrastructure*, *Presentation* or *UI* and *Domain* layer code was tangled all together.

```
include __DIR__ . '/bootstrap.php';

$db = new PDO('mysql:host=localhost;dbname=my_database', 'a_username', '4_p4ssw0\
rd', [
    PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8',
]);

$errormsg = null;

if (isset($_POST['submit'] && isValid($_POST['post'])) {
    $post = getFrom($_POST['post']);
    $db->beginTransaction();
```

```
    try {
        $stm = $db->prepare('INSERT INTO posts (title, content) VALUES (?, ?)');
        $stm->exec([
            $post['title'],
            $post['content']
        ]);
        $db->commit();
    } catch (Exception $e) {
        $db->rollback();
        $errormsg = 'Post could not be created! :(';
    }
}

$stm = $db->prepare('SELECT id, title, content FROM posts');
$posts = $stm->fetchAll(PDO::FETCH_ASSOC);
?>
<html>
    <head></head>
    <body>
        <?php if (null !== $errormsg): ?>
        <div class="alert error"><?php echo $errormsg; ?></div>
        <?php else: ?>
        <div class="alert success">Bravo! Post was created successfully!</div>
        <?php endif; ?>
        <table>
            <thead><tr><th>ID</th><th>TITLE</th><th>ACTIONS</th></tr></thead>
            <tbody>
            <?php foreach ($posts as $post): ?>
                <tr>
                    <td><?php echo $post['ID']; ?></td>
                    <td><?php echo $post['TITLE']; ?></td>
                    <td><?php editPostUrl($post['ID']); ?></td>
                </tr>
            <?php endforeach; ?>
            </tbody>
        </table>
    </body>
</html>
```

This style of coding is often referred to as the *Big Ball of Mud*[1]. An improvement seen in this style

[1]Extracted from the c2.com wiki: *A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, DuctTape and bailing wire, SpaghettiCode jungle.*

however, was to encapsulate the header and the footer of the web page in their own separate files, which were included in the others. This avoided duplication and favoured reuse.

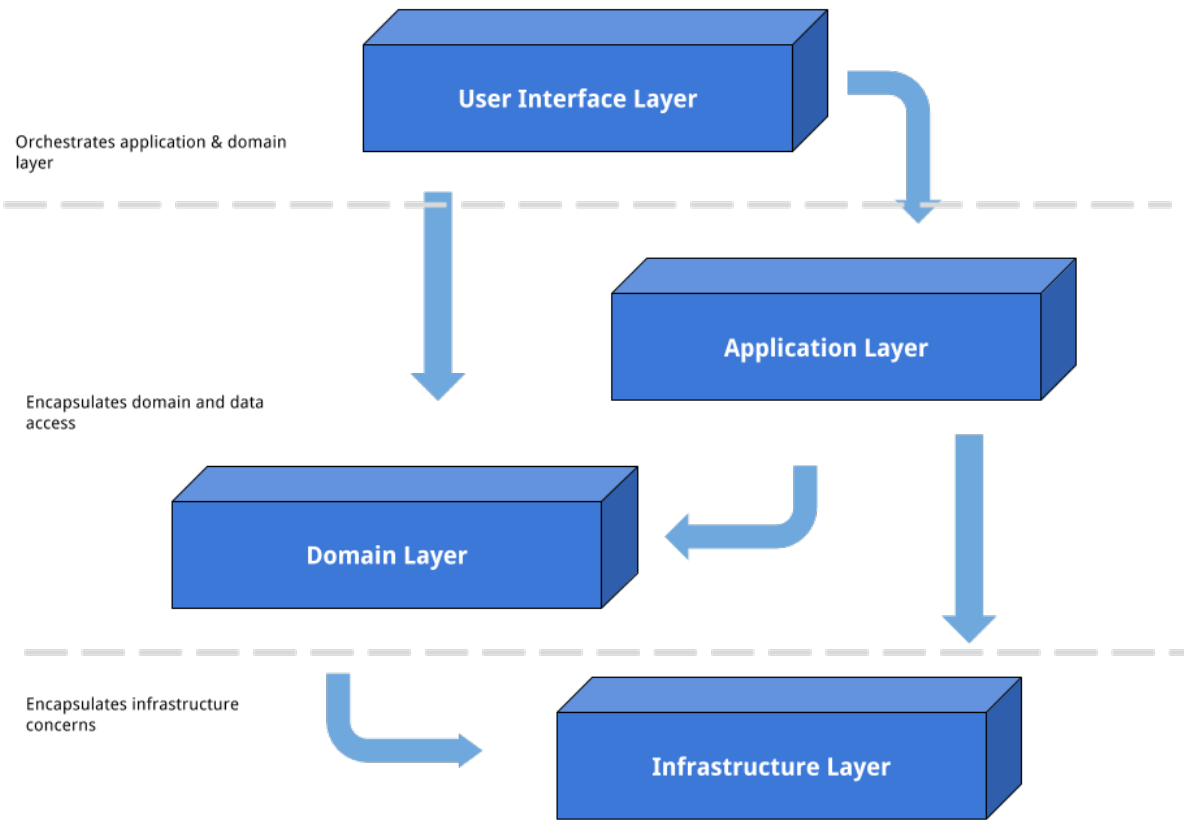
```
include __DIR__ . '/bootstrap.php';

$db = new PDO('mysql:host=localhost;dbname=my_database', 'a_username', '4_p4ssw0\
rd', [
    PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8',
]);

$errormsg = null;

if (isset($_POST['submit'] && isValid($_POST['post'])) {
    $post = getFrom($_POST['post']);
    $db->beginTransaction();
    try {
        $stm = $db->prepare('INSERT INTO posts (title, content) VALUES (?, ?)');
        $stm->exec([
            $post['title'],
            $post['content']
        ]);
        $db->commit();
    } catch (Exception $e) {
        $db->rollback();
        $errormsg = 'Post could not be created! :(';
    }
}

$stm = $db->prepare('SELECT id, title, content FROM posts');
$posts = $stm->fetchAll(PDO::FETCH_ASSOC);
?>
<?php include __DIR__ . '/header.php'; ?>
<?php if (null !== $errormsg): ?>
<div class="alert error"><?php echo $errormsg; ?></div>
<?php else: ?>
<div class="alert success">Bravo! Post was created successfully!</div>
<?php endif; ?>
<table>
    <thead><tr><th>ID</th><th>TITLE</th><th>ACTIONS</th></tr></thead>
    <tbody>
    <?php foreach ($posts as $post): ?>
        <tr>
            <td><?php echo $post['ID']; ?></td>
```

```
            <td><?php echo $post['TITLE']; ?></td>
            <td><?php editPostUrl($post['ID']); ?></td>
        </tr>
    <?php endforeach; ?>
    </tbody>
</table>
<?php include __DIR__ . '/footer.php'; ?>
```

Nowadays, and although it is highly discouraged, there are still applications that use this procedural way of coding. The main disadvantage of this style of architecture is that there is no real separation of concerns - maintenance and cost of change increases drastically in relation to other well-known and proven architectures.

2.2 Layered Architecture

From the code maintainability and reuse perspectives, the best way to make this code a bit easier to maintain would be splitting up concepts - creating layers for each different concern. In our previous example, it is easy to shape some different layers like the one encapsulating the data access and manipulation, another one to handle infrastructure concerns and a final one for encapsulating the orchestration of the previous two. An essential rule of the *Layered architecture* is that *each layer may be tightly coupled with the layers beneath it*, as shown in the following picture



Layered architecture

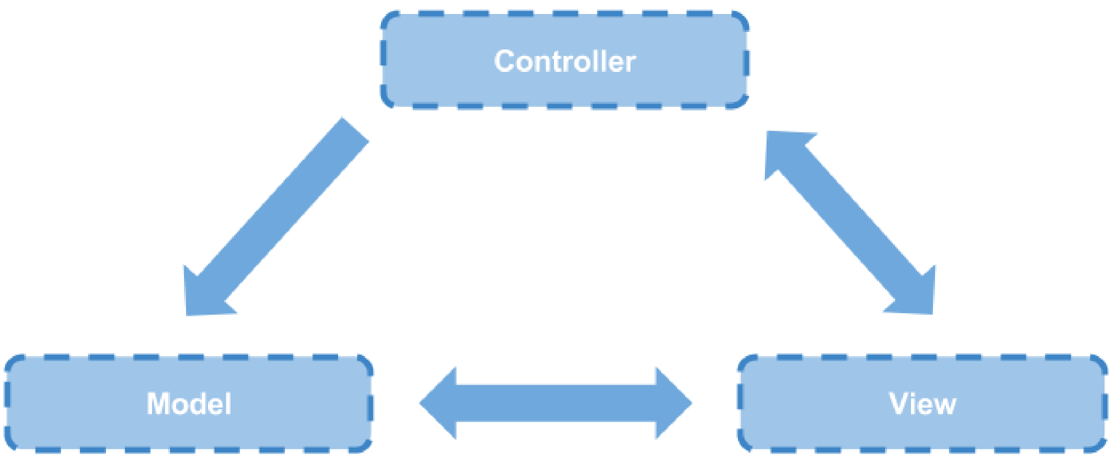
What the layered architecture really seeks is the separation of the different components of an application. For instance, in terms of the previous example, a blog post representation must be completely independent of a blog post as a conceptual entity. A blog post as a conceptual entity can instead be associated with one or more representations, as opposed to being tightly coupled to a specific representation. This is commonly named *Separation of Concerns*.

Another architecture paradigm and pattern that seeks the same purpose is the *Model-View-Controller* pattern. It was initially thought and widely-used for building desktop GUI applications, and now it is mainly used in web applications thanks to popular web frameworks like *Symfony*, *Zend Framework* or *Codeigniter*.

2.2.1 Model-View-Controller

Model-View-Controller is an architectural pattern and paradigm that divides the application into three main layers:

- *The Model*: Captures and centralizes all the domain model behaviour. This layer manages all the data, logic and business rules independently of the data representation. It can be said that the **Model layer is the heart and soul of every MVC application**.
- *The Controller*: Orchestrates interactions between the other layers. Triggers actions on the model in order to update its state and refreshes the representations associated to the model. Additionally, the Controller can also send messages to the View layer in order to change the specific Model representation.
- *The View*: A layer whose main purpose is to expose the differing representations of the Model layer and to give a way to trigger changes on the Model's state.



The MVC pattern

2.2.2 Example of Layered Architecture

2.2.2.1 The Model

Following the previous example, we mentioned that different concerns should be split up. In order to do so, all layers should be identified in our original tangled code. Through this process we need to pay special attention to the code conforming to the *Model* layer, which will be the beating heart of the application.

```
class Post
{
    private $title;
    private $content;

    public static function writeNewFrom($title, $content)
    {
        return new static($title, $content);
    }

    private function __construct($title, $content)
    {
        $this->setTitle($title);
        $this->setContent($content);
    }

    private function setTitle($title)
    {
        $this->assertNotEmpty($title);

        $this->title = $title;
    }

    private function setContent($content)
    {
        $this->assertNotEmpty($title);

        $this->content = $content;
    }
}

class PostRepository
{
    private $db;

    public function __construct()
    {
        $this->db = new PDO(
            'mysql:host=localhost;dbname=my_database',
            'a_username',
            '4_p4ssw0rd',
            [
```

```
                PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8',
            ]
        );
    }

    public static function add(Post $post)
    {
        $this->db->beginTransaction();

        try {
            $stm = $db->prepare(
                'INSERT INTO posts (title, content) VALUES (?, ?)'
            );

            $stm->exec([
                $post->getTitle(),
                $post->getContent(),
            ]);

            $db->commit();
        } catch (Exception $e) {
            $db->rollback();
            throw new UnableToCreatePostException($e);
        }
    }
}
```

The *Model* layer is now defined by a `Post` class and a `PostRepository` class. The `Post` class represents a blog post and the `PostRepository` class represents the whole collection of blog posts available. Additionally, another layer inside the *Model* is needed, a layer that coordinates and orchestrates the domain model behaviour: the **Application Layer**.

```
class PostService
{
    public function createPost($title, $content)
    {
        $post = Post::writeNewFrom($title, $content);

        (new PostRepository())->add($post);

        return $post;
    }
}
```

The `PostService` is what is known as an **Application Service** and its purpose is to orchestrate and organize the domain behaviour. In other words, the **Application services are the ones that make things happen** and they are the direct clients of a Domain Model. No other type of object should be able to directly talk to the internal layers of the *Model* layer.

2.2.2.2 The View

The *View* is a layer that can both receive and send messages from the *Model* layer and/or from the *Controller* layer. Its main purpose is to represent the Model to the user at the UI level, and refresh the representation in the UI each time the Model is updated. Generally speaking, the View layer receives an object, often a **Data Transfer Object (DTO)** instead of instances of the *Model* layer, gathering all the needed information to be successfully represented. For PHP there are several template engines that can help a great deal in separating the Model representation from the Model itself and from the Controller. The most popular one by far is called **Twig**[2]. Lets see how the View layer would look like with Twig

> **DTOs instead of Model instances?**
>
> This is an old and active topic. Why create a DTO instead of giving an instance of the Model to the View layer? The main reason and the short answer is, again, *Separation of Concerns*. Letting the view inspect and use a *Model* instance leads to tight coupling between the View layer and the Model layer. In fact, a change in the Model layer can potentially break all the views that make use of the changed Model instances.

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

{% block content %}
    {% if errormsg is defined %}
    <div class="alert error">{{ errormsg }} </div>
    {% else %}
    <div class="alert success">Bravo! Post was created successfully!</div>
    {% endif %}
    <table>
        <thead><tr><th>ID</th><th>TITLE</th><th>ACTIONS</th></tr></thead>
        <tbody>
        {% for post in posts %}
            <tr>
                <td>{{ post.id }}</td>
                <td>{{ post.title }}<?php echo $post['TITLE']; ?></td>
                <td><a href="{{ editPostUrl(post.id) }}">Edit Post</a></td>
```

[2] http://twig.sensiolabs.org/

```
            </tr>
        {% endforeach %}
        </tbody>
    </table>
{% endblock %}
```

Most of the time, when the *Model* triggers a state change, it also notifies the related *Views* so that the UI can get refreshed. In a typical web scenario the synchronization between the Model and its representations can be a bit tricky because of the client-server nature. In this kind of environments some `JavaScript` defined interactions are usually needed to maintain that synchronization. For this reason, JavaScript MVC frameworks like the ones below have become widely popular in recent years:

- AngularJS[3]
- EmberJS[4]
- Marionette[5]
- ReactJS[6]

2.2.2.3 The Controller

The Controller layer is responsible for organizing and orchestrating the View and the Model. It receives messages from the View layer and triggers Model behaviour in order to perform the desired action. Furthermore, it sends messages to the View in order to display Model representations. Both operations are performed thanks to the **Application Layer**, responsible for orchestrating, organizing and encapsulating domain behaviour.

In terms of a web application in PHP, the Controller usually comprehends a set of classes, which in order to fulfill their purpose *"speak HTTP"*. That is, they receive an HTTP request and return an HTTP response.

```
class PostsController
{
    public function updateAction(Request $request)
    {
        if ($request->request->has('submit')
            && Validator::validate($request->request->post)
        ) {
            $postService = new PostService();
```

[3] https://angularjs.org/
[4] http://emberjs.com/
[5] http://marionettejs.com/
[6] https://facebook.github.io/react/

```
            try {
                $postService->createPost(
                    $request->request->get('title'),
                    $request->request->get('content')
                );

                $this->addFlash(
                    'notice',
                    'Post has been created successfully!'
                );
            } catch (Exception $e) {
                $this->addFlash(
                    'error',
                    'Unable to create the post!'
                );
            }
        }

        return $this->render('posts/update-result.html.twig');
    }
}
```

2.3 Inverting Dependencies. Hexagonal Architecture

Following the essential rule of Layered Architecture, there is a risk to end up implementing domain interfaces that need to make use of infrastructural concerns within the domain model layer.

As an example, the `PDORepository` from the previous example should be placed in the Domain Model if we were following MVC. However, placing infrastructural details right in the middle of our domain violates separation of concerns. This can result in issues, it is hard to avoid violating the essential rules of Layered Architecture, leading to a style of code which can become hard to test if the Domain Layer is aware of technical implementations.

2.3.1 The Dependency Inversion Principle (DIP)

How can we fix this? As the Domain Model layer depends on concrete infrastructure implementations, the *Dependency Inversion Principle*[7] could be applied by relocating the Infrastructure layer on top of the other three layers.

[7] http://www.objectmentor.com/resources/articles/dip.pdf

The Dependency Inversion Principle

High level modules should not depend upon low level modules. Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

Robert C. Martin

By using the *Dependency Inversion Principle*, the architecture schema changes and the Infrastructure layer which can be referred to as *low level modules* now depend on the *UI*, the *Application Layer* and the *Domain Layer*, which are the *high level modules*. The dependency has been inverted.

But then, what is Hexagonal Architecture?, and how does it fit within of all this? Hexagonal Architecture (also known as *Ports and Adapters*) was defined by Alistair Cockburn and represents the application as an hexagon where each side represents a **Port** with one or more **Adapters**. A Port is a connector with a pluggable Adapter which transforms an **outside** input to something the **inside** application can understand. In terms of the *DIP*, the Port would be a *high level module* and an Adapter would be a *low level module*. Furthermore, if the application needs to emit some message to the *outside* it will also use a Port with an Adapter to send it and transform it to something that the outside can understand. For this reason, Hexagonal Architecture brings up the concept of symmetry in the application and it is also the main reason why the schema of the architecture changes. It is often represented as a hexagon, because it does no longer make sense to talk about a "top" layer nor a "bottom" layer. Instead, Hexagonal Architecture talks mainly in terms of the *'outside'* and the *'inside'*.

2.3.2 Applying Hexagonal Architecture

Following on with the blog example application, the first concept we need is a Port where the *outside world* could talk to the application. For this case, we will use an HTTP Port and its corresponding Adapter. The outside will use the port to send messages to the application. The example was using a database to store the whole collection of blog posts so, in order to allow the application retrieve blog posts from the database, a Port is needed

```
interface PostRepository
{
    public function byId(PostId $id);
    public function add(Post $post);
}
```

This interface states the Port by which the application will retrieve information about blog posts, *and it will be located in the Domain Layer*. Now, an Adapter for this Port is needed. The Adapter is in charge of defining the way in which the blog posts will be retrieved using a specific technology.

```
class PDOPostRepository implements PostRepository
{
    private $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function byId(PostId $id)
    {
        $stm = $this->db->prepare(
            'SELECT * FROM posts WHERE id = ?'
        );

        $stm->execute([$id->id()]);

        return recreateFrom($stm->fetch());
    }

    public function add(Post $post)
    {
        $stm = $db->prepare(
            'INSERT INTO posts (title, content) VALUES (?, ?)'
        );

        $stm->exec([
            $post->getTitle(),
            $post->getContent(),
        ]);
    }
}
```

Once we have the Port and its Adapter defined, the last step to do is to refactor the `PostService` class so that it uses them. And this can be easily achieved by using **Dependency Injection**[8]

[8] http://www.martinfowler.com/articles/injection.html

```
class PostService
{
    private $postRepository;

    public function __construct(PostRepository $postRepository)
    {
        $this->postRepository = $postRepository;
    }

    public function createPost($title, $content)
    {
        $post = Post::writeNewFrom($title, $content);

        $this->postRepository->add($post);

        return $post;
    }
}
```

This is just a simple example of Hexagonal Architecture. It is a flexible architecture that promotes *separation of concerns* like *layered architecture* and *symmetry* in that there is an inside application that communicates using ports with the outside. From now on, this will be the foundational architecture used to build and explain *CQRS* and *Event Sourcing*.

For a deeper insight about this architecture, **you can checkout the appendix** with a detailed example, explaining advanced topics like *transactionability* and other *cross cutting concerns*.

2.4 Command Query Responsibility Segregation

Hexagonal Architecture is a good foundational architecture but it has some limitations. For example, complex UIs can require **aggregate** information displayed in diverse forms or they can require data obtained from multiple aggregates. And in this scenario, we can end up with a lot of *finder* methods inside the Repositories (maybe as many as the UI views which exist within the application). Or maybe we can decide to move this complexity to the Application Services - using complex structures to accumulate data from multiple aggregates. Here is an example:

```
interface PostRepository
{
    public function save(Post $post);
    public function byId(PostId $id);
    public function all();
    public function byCategory(CategoryId $categoryId);
    public function byTag(TagId $tagId);
    public function withComments(PostId $id);
    public function groupedByMonth();
    // ...
}
```

When these techniques are abused, the construction of the UI views can become really painful and we should evaluate the trade-offs between making Application Services return domain model instances and using some kind of **Data Transfer Object (DTO)** in order to avoid tight coupling between the Domain Model and infrastructure code like web controllers, CLI controllers, and so on.

Luckily, there is another approach. If the problem is having multiple and disparate views, we can exclude them from the Domain Model and start treating them as a purely infrastructural concern. This option is based on a design principle, named **Command Query Separation CQS**, defined by *Bertrand Meyer* which gave birth to a new architectural pattern named **Command Query Responsibility Segregation** defined by *Greg Young*.

Command Query Separation (CQS)

"Asking a question should not change the answer" – Bertrand Meyer

This design principle states that every method should be either a *Command*, that performs an action, or a *Query*, that returns data to the caller, but not both.

CQRS seeks an even more aggressive *separation of concerns* splitting the *Model* in two:

- The **Write Model**: Also known as the **Command Model**, it performs the writes and takes responsibility for the true domain behaviour.
- The **Read Model**: It takes responsibility of the reads within the application and treats them as something that should be out of the Domain Model.

Every time someone triggers a command to the *write model*, this performs the write to the desired datastore and additionally triggers the *read model* update in order to display the latest changes on the *read model*.

This strict separation triggers another problem, **Eventual Consistency**. The consistency of the *read model* now is subject to the commands performed by the *write model*. In other words, it is said that *the read model is eventually consistent.* This is, every time the *write model* performs a command it will pull up a process that will be responsible to update the *read model* according to the last updates on the *write model.* There is such a window of time were the UI may present stale information to the user. In the web scenario this happens often as we are somewhat limited by the current technologies. Think about a caching system in front of a web application. Every time the database is updated with new information, the data on the cache layer may potentially be stale, so every time it gets updated there should be a process that updates the cache system. **Cache systems are eventually consistent**.

This kind of processes, speaking in *CQRS* terminology, are called **Write Model Projections** or just **Projections**. We *project* the *write model* onto the *read model.* This process can be synchronous or asynchronous, depending on your needs, and it can be done thanks to another useful tactical design pattern that will be explained in detail later on in the book, **Domain Events**. The basis of the *write model* projections is to gather all the published *domain events* and update the *read model* with all the information coming from the events.

2.4.1 The Write Model

This is the true holder of domain behaviour.

Following on with the example, the *Repository* interface would be simplified to

```
interface PostRepository
{
    public function save(Post $post);
    public function byId(PostId $id);
}
```

Now the `PostRepository` has been freed from all the *read* concerns except one, the `byId` which is responsible for loading the *aggregate* by its' ID so that we can operate on it.

And once this is done, all the `query` methods are also stripped down from the `Post` model, leaving it only with *command* methods. This means we will effectively get rid of all the getter methods and any other methods exposing information about it. Instead, *domain events* will be published in order to be able to trigger *write model projections* by subscribing to them.

```
class AggregateRoot
{
    private $recordedEvents = [];

    protected function recordApplyAndPublishThat(DomainEvent $domainEvent)
    {
        $this->recordThat($domainEvent);
        $this->applyThat($domainEvent);
        $this->publishThat($domainEvent);
    }

    protected function recordThat(DomainEvent $domainEvent)
    {
        $this->recordedEvents[] = $domainEvent;
    }

    protected function applyThat(DomainEvent $domainEvent)
    {
        $modifier = 'apply' . get_class($domainEvent);

        $this->$modifier($domainEvent);
    }

    protected function publishThat(DomainEvent $domainEvent)
    {
        DomainEventPublisher::getInstance()->publish($domainEvent);
    }

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

    public function clearEvents()
    {
        $this->recordedEvents = [];
    }
}

class Post extends AggregateRoot
{
    private $id;
```

```
    private $title;
    private $content;
    private $published = false;
    private $categories;

    private function __construct(PostId $id)
    {
        $this->id = $id;
        $this->categories = new Collection();
    }

    public static function writeNewFrom($title, $content)
    {
        $post = new Post(PostId::create());

        $post->recordApplyAndPublishThat(
            new PostWasCreated(PostId::generate(), $title, $content)
        );
    }

    public function publish()
    {
        $this->recordApplyAndPublishThat(
            new PostWasPublished($this->id)
        );
    }

    public function categorizeIn(CategoryId $categoryId)
    {
        $this->recordApplyAndPublishThat(
            new PostWasCategorized($this->id, $categoryId)
        );
    }

    public function changeContentFor($newContent)
    {
        $this->recordApplyAndPublishThat(
            new PostContentWasChanged($this->id, $newContent)
        );
    }

    public function changeTitleFor($newTitle)
```

```
    {
        $this->recordApplyAndPublishThat(
            new PostTitleWasChanged($this->id, $newTitle)
        );
    }
}
```

All actions that trigger a state change are implemented via *domain events*. For each *domain event* published there is an *apply* method responsible to reflect the state change.

```
class Post extends AggregateRoot
{
    // ...

    protected function applyPostWasCreated(PostWasCreated $event)
    {
        $this->id = $event->id();
        $this->title = $event->title();
        $this->content = $event->content();
    }

    protected function applyPostWasPublished(PostWasPublished $event)
    {
        $this->published = true;
    }

    protected function applyPostWasCategorized(PostWasCategorized $event)
    {
        $this->categories->add($event->categoryId());
    }

    protected function applyPostContentWasChanged(PostContentWasChanged $event)
    {
        $this->content = $event->content();
    }

    protected function applyPostTitleWasChanged(PostTitleWasChanged $event)
    {
        $this->title = $event->title();
    }
}
```

2.4.2 The Read Model

The *read model*, also known as the *Query Model*, is a pure denormalized data model lifted from domain concerns. In fact, with CQRS all the *read concerns* are treated as reporting processes, an infrastructure concern. In general, when using CQRS, the *read model* is subject to the needs of the UI and how complex the views compounding the UI are.

In a situation where the the *read model* is defined in terms of relational databases, the simplest approach would be to set one-to-one relationships between database tables and UI views. These database tables / UI views will be updated using *write model projections* triggered from the *domain events* published by the *write* side.

```
-- Definition of a UI view of a single post with its comments
CREATE TABLE single_post_with_comments (
  id INTEGER NOT NULL,
  post_id INTEGER NOT NULL,
  post_title VARCHAR(100) NOT NULL,
  post_content TEXT NOT NULL,
  post_created_at DATETIME NOT NULL,
  comment_content TEXT NOT NULL
);

-- Set up some data
INSERT INTO single_post_with_comments(1, 1, "Layered architecture", "Lorem ipsum\
 dolor sit amet, ...", NOW(), "Lorem ipsum dolor sit amet, ...");
INSERT INTO single_post_with_comments(2, 1, "Layered architecture", "Lorem ipsum\
 dolor sit amet, ...", NOW(), "Lorem ipsum dolor sit amet, ...");
INSERT INTO single_post_with_comments(3, 2, "Hexagonal architecture", "Lorem ips\
um dolor sit amet, ...", NOW(), "Lorem ipsum dolor sit amet, ...");
INSERT INTO single_post_with_comments(4, 2, "Hexagonal architecture", "Lorem ips\
um dolor sit amet, ...", NOW(), "Lorem ipsum dolor sit amet, ...");
INSERT INTO single_post_with_comments(5, 3, "Command - Query Responsability Segg\
regation", "Lorem ipsum dolor sit amet, ...", NOW(), "Lorem ipsum dolor sit amet\
, ...");
INSERT INTO single_post_with_comments(6, 3, "Command - Query Responsability Segg\
regation", "Lorem ipsum dolor sit amet, ...", NOW(), "Lorem ipsum dolor sit amet\
, ...");

-- Query it
SELECT * FROM single_post_with_comments WHERE post_id = 1;
```

An important feature of this architectural style is that the *read model* should be completely disposable since the *true state* of the application is handled by the *write model*. This means the *read model* can be removed and recreated when needed using *write model projections*.

Here we can see some examples of possible views within a blog application

```
SELECT * FROM posts_grouped_by_month_and_year ORDER BY month DESC, year ASC;
SELECT * FROM posts_by_tags WHERE tag = "ddd";
SELECT * FROM posts_by_author WHERE author_id = 1;
```

It is important to point out that *CQRS* does not constrain the definition and implementation of the *read model* to a relational database. It depends exclusively on the needs of the application being built. It could be a relational database, a document-oriented database, a key-value store or whatever best suits the needs of your application.

Following the blog post application, we will use Elasticsearch[9] (a document-oriented database) to implement a *read model*.

```
class PostsController
{
    public function listAction()
    {
        $client = new \Elasticsearch\ClientBuilder::create()->build();

        $response = $client->search([
            'index' => 'blog-engine',
            'type'  => 'posts',
            'body' => [
                'sort' => [
                    'created_at' => ['order' => 'desc']
                ]
            ]
        ]);

        return [
            'posts' => $response
        ];
    }
}
```

The *read model* code has been drastically simplified to a query to an Elasticsearch index. This reveals that the *read model* does not really need an *object-relational mapper* as doing so might be an overkill. However, the *write model* **might benefit from the use of an object-relational mapper** as they allow you to organize and structure the *read model* according to the needs of the application.

[9] https://en.wikipedia.org/wiki/Elasticsearch

2.4.3 Synchronizing the Write Model with the Read Model

Here comes the tricky part. How do we synchronize the *read model* with the *write model*? We already said we would do it by using *domain events* captured in a *write model* transaction. For each type of domain event captured, a specific projection will be executed. So a one-to-one relationship between *domain events* and projections will be set.

Let's have a look at an example of configuring projections, so that we can get a better idea.

```
$client = new \Elasticsearch\ClientBuilder::create()->build();

$projector = new Projector();
$projector->register([
    new Infrastructure\Projection\Elasticsearch\PostWasCreated($client),
    new Infrastructure\Projection\Elasticsearch\PostWasPublished($client),
    new Infrastructure\Projection\Elasticsearch\PostWasCategorized($client),
    new Infrastructure\Projection\Elasticsearch\PostContentWasChanged($client),
    new Infrastructure\Projection\Elasticsearch\PostTitleWasChanged($client),
]);

$events = [
    new PostWasCreated(/* ... */),
    new PostWasPublished(/* ... */),
    new PostWasCategorized(/* ... */),
    new PostContentWasChanged(/* ... */),
    new PostTitleWasChanged(/* ... */),
];

$projector->project($event);
```

This code is kind of synchronous, but the process can be asynchronous if needed. You could make your customers aware of this out-of-sync data by placing some alerts.

For the next example, we will use the ampq-lib PHP extension in combination with ReactPHP[10].

[10] http://https://github.com/GeniusesOfSymfony/ReactAMQP

```
// Connect to an AMQP broker
$cnn = new AMQPConnection();
$cnn->connect();

// Create a channel
$ch = new AMQPChannel($cnn);

// Declare a new exchange
$ex = new AMQPExchange($ch);
$ex->setName('events');
$ex->declare();

// Create an event loop
$loop = React\EventLoop\Factory::create();

// Create a producer that will send any waiting messages every half a second
$producer = new Gos\Component\ReactAMQP\Producer($ex, $loop, 0.5);

$serializer = JMS\Serializer\SerializerBuilder::create()->build();

$projector = new AsyncProjector($producer, $serializer);

$events = [
    new PostWasCreated(/* ... */),
    new PostWasPublished(/* ... */),
    new PostWasCategorized(/* ... */),
    new PostContentWasChanged(/* ... */),
    new PostTitleWasChanged(/* ... */),
];

$projector->project($event);
```

And the event consumer on the RabbitMQ exchange would be something like

```php
// Connect to an AMQP broker
$cnn = new AMQPConnection();
$cnn->connect();

// Create a channel
$ch = new AMQPChannel($cnn);

// Create a new queue
$queue = new AMQPQueue($ch);
$queue->setName('events');
$queue->declare();

// Create an event loop
$loop = React\EventLoop\Factory::create();

$serializer = JMS\Serializer\SerializerBuilder::create()->build();

$client = new \Elasticsearch\ClientBuilder::create()->build();

$projector = new Projector();
$projector->register([
    new Infrastructure\Projection\Elasticsearch\PostWasCreated($client),
    new Infrastructure\Projection\Elasticsearch\PostWasPublished($client),
    new Infrastructure\Projection\Elasticsearch\PostWasCategorized($client),
    new Infrastructure\Projection\Elasticsearch\PostContentWasChanged($client),
    new Infrastructure\Projection\Elasticsearch\PostTitleWasChanged($client),
]);

// Create a consumer
$consumer = new Gos\Component\ReactAMQP\Consumer($queue, $loop, 0.5, 10);

// Check for messages every half a second and consume up to 10 at a time.
$consumer->on(
    'consume',
    function($envelope, $queue) use ($projector, $serializer) {
        $event = $serializer->unserialize($envelope->getBody(), 'json');
        $projector->project($event);
    }
);

$loop->run();
```

From now on, it could be as simple as making all the needed repositories consume an instance of the projector and make them invoke the projection process.

```
class DoctrinePostRepository implements PostRepository
{
    private $em;
    private $projector;

    public function __construct(EntityManager $em, Projector $projector)
    {
        $this->em = $em;
        $this->projector = $projector;
    }

    public function save(Post $post)
    {
        $this->em->transactional(function(EntityManager $em) use ($post) {
            $em->persist($post);

            foreach ($post->recordedEvents() as $event) {
                $em->persist($event);
            }
        });

        $this->projector->project($post->recordedEvents());
    }

    public function byId(PostId $id)
    {
        return $this->em->find($id);
    }
}
```

The `Post` instance and the recorded events are triggered and persisted in the same transaction. This ensures that no events are lost, as we will project them to the *read model* if the transaction is successful. So, no inconsistencies will exist between the *write model* and the *read model*.

2.5 Event Sourcing

CQRS is a powerful and flexible architecture. There is an added benefit in regard to gathering and saving the *domain events* (which occurred during an aggregate operation), giving you a high-level degree of detail of what is going on within your domain. *Domain Events* are one of the key tactical patterns because of their significance within the domain, as they describe past occurrences.

An ever growing number of events is a smell of the business overlooking insight in the Domain. By using *CQRS* we gained a highly sophisticated history of all the relevant occurrences at a level that the whole state of the domain model can be expressed just by reproducing *domain events*. We just need a tool for storing all those events in a consistent way. This store is called an **eventstore**.

> **The fundamental idea behind Event Sourcing is to express the state of Aggregates as a linear sequence of events**

With *CQRS* we partially achieved the following - the `Post` entity alters its state by using *domain events* but it is persisted as always, mapping the object to a database table. *Event Sourcing* takes this a step further. If we were using a database table to store the state of all the blog posts and another to store the state of all the blog post comments and so on, by using *Event Sourcing* we could use just a single database table. A single append-only database table that would store all the *domain events* published by all the aggregates within the domain model. Yes, you have read that correctly, a **single** database table.

With this model in mind, tools like *object-relational mappers* are not needed any more. They would be an overkill for a single database table. The only tool needed would be a simple database abstraction layer by which *events* can be appended.

```
interface EventSourcedAggregateRoot
{
    public static function reconstitute(EventStream $events);
}

class Post extends AggregateRoot implements EventSourcedAggregateRoot
{
    public static function reconstitute(EventStream $history)
    {
        $post = new static($history->getAggregateId());

        foreach ($events as $event) {
            $post->applyThat($event);
        }
```

```
        return $post;
    }
}
```

Now the `Post` aggregate has a method which given a set of *events* (or in other words an **event stream**) is able to replay the state step by step until it reaches the current state before saving. The next step would be building an adapter of the `PostRepository` port that will fetch all the published events from the `Post` aggregate and append them to the data store where all the events are appended. This is what we call an **eventstore**.

```
class EventStorePostRepository implements PostRepository
{
    private $eventS tore;
    private $projector;

    public function __construct($eventStore, $projector)
    {
        $this->eventStore = $eventStore;
        $this->projector = $projector;
    }

    public function save(Post $post)
    {
        $events = $post->recordedEvents();

        $this->eventStore->append(new EventStream($post->id(), $events));
        $post->clearEvents();

        $this->projector->project($events);
    }
}
```

This is how the implementation of the `PostRepository` looks like when we use an **eventstore** to save all the events published by the `Post` aggregate. Now we need a way to restore an aggregate from its events history. A `reconstitute` method implemented by the `Post` aggregate to be used to rebuild a blog post state from triggered events comes in very handy.

```
class EventStorePostRepository implements PostRepository
{
    public function byId(PostId $id)
    {
        return Post::reconstitute(
            $this->eventStore->getEventsFor($id)
        );
    }
}
```

The **eventstore** is the work-horse that carries out all the responsibility in regard to saving and restoring **eventstreams**. Its public API is composed of two simple methods: `append` and `getEventsFrom`. The former appends an *eventstream* to the *eventstore* and the later loads *eventstreams* to allow aggregate rebuilding.

We could use a key-value implementation to store all events

```
class EventStore
{
    private $redis;
    private $serializer;

    public function __construct($redis, $serializer)
    {
        $this->redis = $redis;
        $this->serializer = $serializer;
    }

    public function append(EventStream $eventstream)
    {
        foreach ($eventstream as $event) {
            $data = $this->serializer->serialize(
                $event,
                'json'
            );

            $date = (new DateTimeImmutable())->format('YmdHis');

            $this->redis->rpush(
                'events:' . $event->getAggregateId(),
                $this->serializer->serialize([
                    'type' => get_class($event),
                    'created_on' => $date,
```

```
                'data' => $data
            ], 'json')
        );
    }
}

public function getEventsFor($id)
{
    $serializedEvents = $this->redis->lrange(
        'events:' . $id,
        0,
        -1
    );

    $eventStream = [];

    foreach ($serializedEvents as $serializedEvent) {
        $eventData = $this->serializer->deserialize(
            $serializedEvent,
            'array',
            'json'
        );

        $eventStream[] = $this->serializer->deserialize(
            $eventData['data'],
            $eventData['type'],
            'json'
        );
    }

    return new EventStream($id, $eventStream);
}
}
```

This *eventstore* implementation is built upon **Redis**[11], a widely used key-value store. The events are appended in a list using the prefix *"events:"*. In addition, before persisting the events we extract some metadata like the event class or the creation date, as it will come handy later.

Obviously, in terms of performance, it is expensive for an aggregate to go over its full event history to reach its final state all of the time. This is especially the case when an eventstream has hundreds or even thousands of events. The best way to overcome this situation is to take a snapshot from the *aggregate* and replay only the events in the eventstream since the snapshot was taken. A snapshot

[11] http://redis.io

is just a simple serialized version of the aggregate state at a given moment. It can be based on the number of events of the aggregate's eventstream or time-based. With the first approach, a snapshot will be taken every *n triggered events* (every 50, 100 or 200 for example). With the second approach a snapshot will be taken every *n seconds*.

To follow the example, we will use the first way of snapshotting. In the event's metadata we store an additional field, the *version*, from which we will start replaying the aggregate history.

```
class SnapshotRepository
{
    public function byId($id)
    {
        $key = 'snapshots:' . $id;
        $metadata = $this->serializer->unserialize(
            $this->redis->get($key)
        );

        if (null === $metadata) {
            return;
        }

        return new Snapshot(
            $metadata['version'],
            $this->serializer->unserialize(
                $metadata['snapshot']['data'],
                $metadata['snapshot']['type'],
                'json'
            )
        );
    }

    public function save($id, Snapshot $snapshot)
    {
        $key = 'snapshots:' . $id;
        $aggregate = $snapshot->aggregate();

        $snapshot = [
            'version' => $snapshot->version(),
            'snapshot' => [
                'type' => get_class($aggregate),
                'data' => $this->serializer->serialize(
                    $aggregate,
                    'json'
```

```
            )
        ]
    ];

    $this->redis->set($key, $snapshot);
    }
}
```

And now we need to refactor the `EventStore` class so that it starts using the `SnapshotRepository` to load the aggregate with acceptable performance times.

```
class EventStorePostRepository implements PostRepository
{
    public function byId(PostId $id)
    {
        $snapshot = $this->snapshotRepository->byId($id);

        if (null === $snapshot) {
            return Post::reconstitute(
                $this->eventStore->getEventsFrom($id)
            );
        }

        $post = $snapshot->aggregate();

        $post->replay(
            $this->eventStore->fromVersion($id, $snapshot->version())
        );

        return $post;
    }
}
```

We just need to take aggregate snapshots periodically. We could do this synchronously or asynchronously by a process responsible for monitoring the **eventstore**.

The following code is a simple example demonstrating the implementation of aggregate snapshotting.

```php
class EventStorePostRepository implements PostRepository
{
    public function save(Post $post)
    {
        $id = $post->id();

        $events = $post->recordedEvents();
        $post->clearEvents();

        $this->eventStore->append(
            new EventStream($id, $events)
        );

        $countOfEvents = $this->eventStore->countEventsFor(
            $id
        );

        $version = $countOfEvents / 100;

        if (!$this->snapshotRepository->has($post->id(), $version)) {
            $this->snapshotRepository->save(
                $id,
                new Snapshot(
                    $post,
                    $version
                )
            );
        }

        $this->projector->project($events);
    }
}
```

2.6 Wrapping Up

As there are plenty of options for architecture styles you could get a bit confused in this chapter. You will have to consider the trade-offs for each one of them in order to choose wisely. One thing is clear, the **Big Ball of Mud** approach is not an option as the code will rot pretty fast. **Layered architecture** is a better option but it presents some disadvantages like tight coupling between layers. Arguably, the most balanced option would be the **Hexagonal Architecture**, as it can be used as a foundational base architecture. It promotes a high-level degree of decoupling and symmetry between the *inside* and *outside* of the application.

We have also seen *CQRS* and *Event Sourcing* as pretty flexible architectures that will help you in fighting serious complexity. *CQRS* and *Event Sourcing* have their place but do not let *'the coolness factor'* distract you from the value they provide. As they come with some overheads, you should have a technical reason for justifying its use. These architectural styles are indeed really useful and the heuristics to start using them can be discovered in the number of *finders* on the repositories for *CQRS* and the volume of triggered events for *Event Sourcing*. If the number of *finder* methods starts growing and repositories become hard to maintain then it is time to consider the use of *CQRS*, to split read and write concerns. And after that, if the volume of events on each aggregate operation tends to grow and the business is interested in more granular information then an option to consider is whether a move towards *Event Sourcing* would pay off.

3. Value Objects

Value Objects are a fundamental building block in Domain-Driven Design, used to model concepts of your Ubiquitous Language in code. A Value Object is not just a *thing* in your domain, it measures, quantifies, or describes something. They can be seen as small, simple objects such as money or a date range - whose equality is not based on identity, but instead on the content held.

For example, a product price could be modelled using a Value Object. In this case it is not representing a *thing*, but instead a value that allows us to measure how much money a product is worth. The memory footprint for these objects is trivial to determine (calculated by their constituent parts) and very little overhead. As a result, new instance creation is favoured over reference reuse, even when being used to represent the same value. Equality is then checked based on the comparability of both instances fields.

3.1 Definition

Ward Cunningham defines[1] a Value Object as

> a measure or description of something. Examples of value objects are things like numbers, dates, monies and strings. Usually, they are small objects which are used quite widely. Their identity is based on their state rather than on their object identity. This way, you can have multiple copies of the same conceptual value object. Every $5 note has its own identity (thanks to its serial number), but the cash economy relies on every $5 note having the same value as every other $5 note.

Martin Fowler defines[2] a Value Object as

> a small object such as a Money or date range object. Their key property is that they follow value semantics rather than reference semantics. You can usually tell them because their notion of equality isn't based on identity, instead two value objects are equal if all their fields are equal. Although all fields are equal, you don't need to compare all fields if a subset is unique - for example currency codes for currency objects are enough to test equality. A general heuristic is that value objects should be entirely immutable. If you want to change a value object you should replace the object with a new one and not be allowed to update the values of the value object itself - updatable value objects lead to aliasing problems.

[1] http://c2.com/cgi/wiki?ValueObject
[2] http://martinfowler.com/bliki/ValueObject.html

Examples of Value Objects are numbers, text strings, dates, times, a person's full name (composed of first, middle, last name, and title), currencies, colours, phone numbers, and postal addresses.

> **Exercise**
>
> Try to locate more examples of potential Value Objects in your current Domain.

3.2 Value Object vs Entity

Consider the following examples from Wikipedia[3], to better understand the difference between Value Objects and Entities.

When people exchange dollar bills, they generally do not distinguish between each unique bill. They are instead only concerned with the face value of the dollar bill. In this context, dollar bills are Value Objects. However, the Federal Reserve might be interested in tracking bills as unique identities and therefore treat them as Entities.

Another example could be that many airlines differentiate among seats, treating them as unique locations. In this instance, a seat can be considered an Entity. On the other hand, there are airlines such as Southwest Airlines (or EasyJet/Ryanair in Europe) that do not differentiate among seats. In this context, a seat could be treated as a Value Object.

> **Exercise**
>
> In regard to the concept of an address (street, number, zip code, etc.). When would be a possible context where an address could be modelled as an Entity and not as a Value Object? Discuss your findings with a peer.

3.3 Currency and Money Example

`Currency` and `Money` Value Objects are probably the most used examples for explaining Value Objects thanks to the Money pattern[4]. This design pattern provides a solution to model the problem in order to avoid floating-point rounding issue, allowing for deterministic calculations to be performed.

In the real world a currency describes monetary units in the same way as meters and yards describe distance units. Each currency is represented with a three upper-case letter ISO code.

[3] http://en.wikipedia.org/wiki/Domain-driven_design#Building_blocks_of_DDD

[4] http://martinfowler.com/eaaCatalog/money.html

```php
class Currency
{
    private $isoCode;

    public function __construct($anIsoCode)
    {
        $this->setIsoCode($anIsoCode);
    }

    private function setIsoCode($anIsoCode)
    {
        if (!preg_match('/^[A-Z]{3}$/', $anIsoCode)) {
            throw new \InvalidArgumentException();
        }

        $this->isoCode = $anIsoCode;
    }

    public function isoCode()
    {
        return $this->isoCode;
    }
}
```

One of the main goals of Value Objects is also the holy grail of Object Oriented design: encapsulation. By following this abstraction, you will end up with a dedicated location to put all the validation, comparison logic and behaviour for a given concept.

Money is used to measure a specific amount of currency. It is modelled using an amount and a Currency. Amount, in the case of the Money pattern, is implemented using an integer representation of the currency's least-valuable fraction - i.e. in the case of USD or EUR, cents.

As a bonus point, you will also notice in the example that we are using self-encapsulation[5] to set the ISO code, centralising changes in the Value Object itself.

[5] http://martinfowler.com/bliki/SelfEncapsulation.html

```
class Money
{
    private $amount;
    private $currency;

    public function __construct($anAmount, Currency $aCurrency)
    {
        $this->setAmount($anAmount);
        $this->setCurrency($aCurrency);
    }

    private function setAmount($anAmount)
    {
        $this->amount = (int) $anAmount;
    }

    private function setCurrency(Currency $aCurrency)
    {
        $this->currency = $aCurrency;
    }

    public function amount()
    {
        return $this->amount;
    }

    public function currency()
    {
        return $this->currency;
    }
}
```

Now that you know the formal definition of a Value Object, let's dive deeper into some of the powerful features that they offer.

3.4 Characteristics

Whilst modelling an Ubiquitous Languages concept in code, you should always favour Value Objects over Entities. Value Objects are easier to create, test, use and maintain.

With this in mind, you can decide on whether the concept in question could be modelled as a Value Object if…

- It measures, quantifies, or describes a *thing* in the domain
- It can be kept immutable
- It models a conceptual whole, by composing related attributes as an integral unit
- It is completely replaceable when the measurement or description changes
- It can be compared with others through value equality
- It supplies its collaborators with Side-Effect-Free behaviour

3.4.1 Measures, Quantifies, or Describes

As discussed before, a Value Object should not be considered just a *thing* in your Domain. As a value, it measures, quantifies, or describes a concept in the Domain.

In our example, the `Currency` object describes what type of a money is. The `Money` object measures or quantifies units of a given `Currency`.

3.4.2 Immutability

This is one of the most important aspects of a Value Object to grasp. Object values should not be able to be altered over their lifetime. Because of this immutability, Value Objects are easy to reason, test and are free of undesired/unexpected side-effects.

As such, Value Objects should be created through their constructor. In order to build one, you usually pass the required primitive types or other value objects through this constructor. Value Objects are always in a valid state, that is why we create them in a single atomic step. Empty constructors with multiple setters and getters move the creation responsibility to the client, resulting in the Anemic Domain Model[6], which is considered an anti-pattern.

It is also good to point out that it is not recommended to hold references to entities in your Value Objects. Entities are mutable, and as such this could lead to undesirable side-effects occurring in the Value Object.

In languages with method overloading[7] such as Java, you can create multiple constructors with the same name. Each of these constructors are provided with different options to build the same type of resulting object. In PHP, we are able to provide a similar capability by way of factory methods[8].

In our `Money` object we could add some useful factory methods, such as:

[6] http://www.martinfowler.com/bliki/AnemicDomainModel.html

[7] http://en.wikipedia.org/wiki/Function_overloading

[8] http://en.wikipedia.org/wiki/Factory_method_pattern

```
class Money
{
    // ...

    public static function fromMoney(Money $aMoney)
    {
        return new self(
            $aMoney->amount(),
            $aMoney->currency()
        );
    }

    public static function ofCurrency(Currency $aCurrency)
    {
        return new self(0, $aCurrency);
    }
}
```

By using the `self` keyword we do not couple the code with the class name. As such, a change to the class name or namespace will not effect these factory methods. This small implementation detail aids when refactoring the code at a later date.

> **static vs. self**
>
> Using `static` over `self` can result in undesirable issues when a Value Object inherits from another Value Object.

Due to this immutability we must consider how to handle mutable actions which are commonplace in a stateful context. If we require a state change, we must now instead return a brand new Value Object representation with this change.

If we want to increase the amount of a `Money` value object for example, we are required to now instead return a new Money instance with the desired modifications. Fortunately, it is relativity simple to abide by this rule, as shown in the example below.

```
class Money
{
    // ...

    public function increaseAmountBy($anAmount)
    {
        return new self(
            $this->amount() + $anAmount,
            $this->currency()
        );
    }
}
```

The object returned by `increaseAmountBy` is different from the one used to invoke the method. This can be observed in the example comparability checks below.

```
$aMoney = new Money(100, new Currency('USD'));
$otherMoney = $aMoney->increaseAmountBy(100);

var_dump($aMoney === $otherMoney); // bool(false)

$aMoney = $aMoney->increaseAmountBy(100);
var_dump($aMoney === $otherMoney); // bool(false)
```

3.4.3 Conceptual Whole

So you may be thinking, why not just implement something similar to the following example, avoiding the need for a new Value Object class altogether?

```
class Product
{
    private $id;
    private $name;

    /**
     * @var int
     */
    private $amount;

    /**
     * @var string
     */
```

```
    private $currency;

    // ...
}
```

This approach has some noticeable flaws, if say for example you want to validate the ISO. It does not really make sense for the `Product` to be responsible for the currency's ISO validation (breaking the Single Responsibility Principle). This is highlighted even more so if you want to reuse the accompanying logic in other parts of your domain (to abide by the DRY principle). With these factors in mind, this use-case is a perfect candidate to be abstracted out into a Value Object. Using this abstraction not only gives you the opportunity to group related properties together, but also to create higher-order concepts and a more concrete Ubiquitous Language.

> **Exercise**
>
> Discuss with a peer if an email could be considered a Value Object or not. Does the context it is used in matter?

3.4.4 Value Equality

As discussed at the beginning of the chapter, two Value Objects are equal if the content they measure, quantify, or describe is the same.

For example, conceptualise two `Money` objects representing 1 USD. Can we consider them equal? In the 'real' world are two coins of 1 USD valued the same? Of course they are. Directing our attention back to the code, the Value Objects in question refers to separate instances of `Money`. However, we can consider them to both represent the same value, so in-turn they are *equal*.

In regard to PHP, it is common place to compare two Value Objects using the == operator. Examining the PHP documentations[9] definition of the operator highlights an interesting behaviour.

> When using the comparison operator (==), object variables are compared in a simple manner, namely: Two object instances are equal if they have the same attributes and values, and are instances of the same class.

This behaviour works in agreement to our formal definition of a Value Object. However, as an exact class match predicate is present, you should be wary when handling sub-typed Value Objects.

With this in mind, the even stricter === operator unfortunately does not help us.

> When using the identity operator (===), object variables are identical if and only if they refer to the same instance of the same class.

The following example should help confirm these subtle differences.

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

```
$a = new Currency('USD');
$b = new Currency('USD');

var_dump($a == $b);  // bool(true)
var_dump($a === $b); // bool(false)

$c = new Currency('EUR');

var_dump($a == $c);  // bool(false)
var_dump($a === $c); // bool(false)
```

With this in mind a solution is to implement a conventional `equals` method in each Value Object. This method is tasked with checking the type and equality of its composite attributes. Abstract data type comparability is easy to implement using PHP's built-in type hinting. On the other hand you can also use the `get_class()` function to aid in the comparability check if necessary. The language however, is unable to decipher what equality truly means in your domain concept, meaning it is your responsibility to provide the answer.

In order to compare `Currency` objects, we just need to compare both their associated ISO codes are the same. The === operator does the job pretty well in this case.

```
class Currency
{
    // ...

    public function equals(Currency $currency)
    {
        return $currency->isoCode() === $this->isoCode();
    }
}
```

Because `Money` objects use `Currency` objects, the `equals` method needs to perform both this comparability check, along with comparing the amounts.

```
class Money
{
    // ...

    public function equals(Money $money)
    {
        return
            $money->currency()->equals($this->currency()) &&
            $money->amount() === $this->amount();
    }
}
```

3.4.5 Replaceability

Consider a `Product` Entity that contains a `Money` Value Object used to quantify its price. Consider also two `Product` Entities whose price is identical, for example 100 USD. This scenario could be modelled using two individual `Money` objects or two references pointing to a single Value Object.

Sharing the same Value Object can be risky, if one is altered, both will reflect the change. This behaviour can be considered an unexpected side-effect. For example, if Carlos was hired on February, 20th, and we know that Christian was also hired on the same day, we may set Christian's hire date to be the same instance as Carlos's. If Carlos then changes the month in his hire date to May, Christian's hire date changes too. Whether it is correct or not, it is not what people expect.

Due to the problems highlighted in this example when holding a reference to a Value Object, rather than modifying its value, it is recommended instead to replace the object as a whole.

```
$this->price = new Money(100, new Currency('USD'));
// ...
$this->price = $this->price->increaseAmountBy(200);
```

This kind of behaviour is similar to how basic types such as strings work in PHP. Consider the function `strtolower`, it returns a new string rather than modifying the original one. No reference is used, but instead a new value is returned.

3.4.6 Side-Effect-Free Behaviour

If we want to include some additional behaviour into our `Money` class, like an `add` method, it feels natural to check that the input fits any preconditions and maintains any invariance. In our case, we only wish to add monies with the same currency.

```
class Money
{
    // ...

    public function add(Money $money)
    {
        if ($money->currency() !== $this->currency()) {
            throw new \InvalidArgumentException();
        }

        $this->amount += $money->amount();
    }
}
```

If the two currencies do not match, an exception is raised. Otherwise, the amounts are added. However, this code has some undesirable pitfalls. Now imagine we have another method `otherMethod`.

```
class Banking
{
    public function aMethod()
    {
        $aMoney = new Money(100, new Currency('USD'));
        $this->otherMethod($aMoney);
        // ...
    }
}
```

Everything is fine until for some reason we start seeing unexpected results in `$aMoney`. What happens if `otherMethod` uses our previously defined `add` method? Maybe you are unaware that `add` mutates the state of the `Money` instance. This is what we call a *side-effect*. You should never mutate arguments, as the client never expects this behaviour.

So, how can we fix this? Simple, by making sure that the Value Object remains immutable we avoid this kind of unexpected problem. A simple solution could be returning a new instance for every potentially mutable operation, like the `add` method

```
class Money
{
    //...

    public function add(Money $money)
    {
        if (!$money->currency()->equals($this->currency())) {
            throw new \InvalidArgumentException();
        }

        return new self(
            $money->amount() + $this->amount(),
            $this->currency()
        );
    }
}
```

With this simple change, immutability is guaranteed. Each time two `Money` are added together, a new resulting instance is returned. Other classes can perform any number of changes, without affecting the original copy. Code free of side-effects is easy to understand, easy to test and less error-prone.

3.5 Basic Types

Consider the following code snippet.

```
$a = 10;
$b = 10;
var_dump($a == $b);
// bool(true)
var_dump($a === $b);
// bool(true)
$a = 20;
var_dump($a);
// integer(20)
$a = $a + 30;
var_dump($a);
// integer(50)
```

Although `$a` and `$b` are different variables, stored low-level in different memory locations, when compared they are the same. They hold the same value. We consider them equal. You can change the value of `$a` from `10` to `20` at anytime you want, the new value is `20` and `10` has disappeared. You can replace integer values as much as you want without consideration of the previous value because you are not modifying it, you are just replacing it. If you apply any operation on them such as addition, $a + $b, you get another new value that can be assigned to another variable or a previously defined one. When you pass $a to another function, except if explicitly passed by reference, you are passing a value. It does not matter if $a gets modified within that function because in your current code, you will still have the original copy. Value Objects behave as basic types.

3.6 Testing

Value Objects are tested in the same way normal objects are. However, the immutability and side-effect-free behaviour must be tested too. A solution is to create a copy of the Value Object you are testing before performing any modifications. Assert both are equal using the implemented equality check. Perform the actions you want to test and assert the results. Finally, assert that the original object and copy are still equal. Let's put this into practice and test the side-effect-free implementation of our `add` method in the `Money` class.

```
class MoneyTest extends \PHPUnit_Test_TestCase
{
    /**
     * @test
     */
    public function copiedMoneyShouldRepresentSameValue()
    {
        $aMoney = new Money(100, new Currency('USD'));

        $copiedMoney = Money::fromMoney($aMoney);

        $this->assertTrue($aMoney->equals($copiedMoney));
    }

    /**
     * @test
     */
    public function originalMoneyShouldNotBeModifiedOnAddition()
    {
        $aMoney = new Money(100, new Currency('USD'));

        $aMoney->add(new Money(20, new Currency('USD')));

        $this->assertEquals(100, $aMoney->amount());
    }

    /**
     * @test
     */
    public function moneysShouldBeAdded()
    {
        $aMoney = new Money(100, new Currency('USD'));
```

```
        $newMoney = $aMoney->add(new Money(20, new Currency('USD')));

        $this->assertEquals(120, $newMoney->amount());
    }

    // ...
}
```

3.7 Persisting Value Objects

Value Objects are not persisted on their own, they are typically persisted within an Aggregate. Value Objects should not be persisted as complete records, though it is an option in some cases. Instead it is best to use Embedded Value or Serialize LOB patterns. Both patterns can be used when persisting your objects with an open-source ORM such as Doctrine or with a bespoke ORM. As Value Objects are small, Embedded Value is usually the best choice because it allows an easy way to query Entities by any of the attributes the Value Object has. However, if querying by those fields is not important to you, Serialize strategies can be very easy to implement.

Consider the following `Product` Entity with a string id, name, and price (`Money` Value Object) attributes. We have intentionally decided to simplify this example with the id being a string and not a Value Object.

```
class Product
{
    private $productId;
    private $name;
    private $price;

    public function __construct(
        $aProductId,
        $aName,
        Money $aPrice
    ) {
        $this->setProductId($aProductId);
        $this->setName($aName);
        $this->setPrice($aPrice);
    }

    // ...
}
```

Assuming you have a Repository for persisting `Product` Entities, an implementation to create and persist a new `Product` could look like the following.

```
$product = new Product(
    $productRepository->nextIdentity(),
    'Domain-Driven Design in PHP',
    new Money(999, new Currency('USD'))
);

$productRepository->persist($product);
```

Let's now look at both the ad-hoc ORM and the Doctrine implementations which could be used to persist a `Product` Entity which contains Value Objects. We will highlight the application of the Embedded Value and Serialized LOB patterns, and the differences between persisting a single Value Object and a collection of them.

> **Why Doctrine?**
>
> Doctrine[10] is a great ORM. It solves 80% of the requirements a PHP application faces. It has a great community. With a correctly-tuned set-up, it can perform the same or even better than a bespoke ORM (without losing maintainability). We recommend using Doctrine in most cases when dealing with Entities and business logic. It will save you a lot of time and headaches.

3.7.1 Persisting Single Value Objects

Many different options are available to persist a single Value Object. These range from using Serialize LOB or Embedded Value as mapping strategies, to use an ad-hoc ORM or an open-source alternative, such as Doctrine. We consider an ad-hoc ORM to be a custom built ORM that your company may have developed in order to persist Entities in a database. In our scenario, the ad-hoc ORM code is going to be implemented using the DBAL[11] library. The Doctrine database abstraction and access layer (DBAL) offers a lightweight runtime around a PDO-like API, along with additional features such as, database schema introspection and manipulation through an OO API.

3.7.1.1 Embedded Value with an ad-hoc ORM

If we are dealing with an ad-hoc ORM using the Embedded Value pattern, we need to create a field in the entity table for each attribute in the Value Object. In this case, two extra columns are needed when persisting a `Product` Entity, one for the amount of the Value Object, and the second for its currency ISO code.

[10] http://www.doctrine-project.org/projects/orm.html
[11] http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/

```
CREATE TABLE `products` (
    id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
    price_amount INT NOT NULL,
    price_currency VARCHAR(3) NOT NULL
)
```

For persisting the Entity in the database, our Repository has to map one-to-one each of the fields of the Entity and the ones from the `Money` Value Object. If using an ad-hoc ORM repository based on DBAL, the `DbalProductRepository` should create the `INSERT` statement, bind the parameters and execute it.

```
class DbalProductRepository extends DbalRepository implements ProductRepository
{
    public function add(Product $aProduct)
    {
        $sql = 'INSERT INTO products VALUES (?, ?, ?, ?)';
        $stmt = $this->connection()->prepare($sql);
        $stmt->bindValue(1, $aProduct->id());
        $stmt->bindValue(2, $aProduct->name());
        $stmt->bindValue(3, $aProduct->price()->amount());
        $stmt->bindValue(4, $aProduct->price()->currency()->isoCode());
        $stmt->execute();

        // ...
    }
}
```

After executing this snippet of code to create a `Product` Entity and persist it into the database, each column has been filled with the desired information.

```
mysql> select * from products \G
*************************** 1. row ***************************
            id: 1
          name: Domain-Driven Design in PHP
  price_amount: 999
price_currency: USD
1 row in set (0.00 sec)
```

As you can see, you can map your Value Objects and query parameters in an ad-hoc manner to persist your Value Objects. However, everything is not as easy as it seems. Let's try to fetch the persisted `Product` with its associated `Money` Value Object. A common approach would be to execute a `SELECT` statement and return a new Entity.

```
class DbalProductRepository extends DbalRepository implements ProductRepository
{
    public function productOfId($anId)
    {
        $sql = 'SELECT * FROM products WHERE id = ?';
        $stmt = $this->connection()->prepare($sql);
        $stmt->bindValue(1, $anId);
        $res = $stmt->execute();
        // ...

        return new Product(
            $row['id'],
            $row['name'],
            new Money(
                $row['price_amount'],
                new Currency(
                    $row['price_currency']
                )
            )
        );
    }
}
```

There are some benefits to this approach. First is that you can easily read step-by-step how the persistence and subsequent creation is occurring. Second, you can perform queries based on any of the attributes of the Value Object. Finally, the space required to persist the Entity is just what is required, no more, no less.

However, using the ad-hoc ORM approach has its drawbacks. As explained in the Domain Events chapter, Entities (in Aggregate form) should fire an Event in the constructor if your Domain is interested in the Aggregates creation. If you use the `new` operator, you would be firing the event as many times as the Aggregate is fetched from the database.

That is one of the reasons why Doctrine uses internally Proxies, `serialize`, and `unserialize` methods to reconstitute an object with its attributes in a specific state without using its constructor. An Entity should be created with the `new` operator just once in its lifetime.

Constructors

Constructors do not need to include a parameter for each attribute in the object. Think about a blog Post. A constructor may need an `id` and a `title`, however, internally it can also be setting its status attribute to draft. When publishing the post, a `publish` method should be called in order to alter its status accordingly and set a published date.

If your intention is still on rolling out your own ORM, be ready to solve some fundamental problems, such as events, different constructors, Value Objects, lazy load relations, etc. That is why we recommend giving Doctrine a try for DDD applications.

Besides, in this instance, you need to create a DbalProduct Entity that extends from the `Product` Entity and is able to reconstitute the Entity from the database without using the `new` operator, using a static factory method.

3.7.1.2 Embedded Value (Embeddables) with Doctrine >= 2.5.*

As of this writing, Doctrine stable release is currently 2.4. Doctrine 2.5 is under development and it comes with support for mapping Value Objects, removing the need to do this yourself. For now however, it looks like Doctrine has no support for nested embeddables. For now, check the Doctrine Embeddables reference[12] for more information. This option, if implemented correctly, is definitely the one that we most recommend. This would be the simplest, most elegant solution, also providing search support in your DQL queries. However, at this time, it is not an option in a production environment. Rest assured we will update the book as soon as this scenario changes.

3.7.1.3 Embedded Value with Doctrine <= 2.4.*

So what is an acceptable solution for using embedded values with Doctrine < 2.5? We need to now surrogate all the Value Objects attributes in the `Product` Entity, meaning to create new *artificial* attributes that will hold the information of the Value Object. With this in place, we can map all those new attributes using Doctrine. Let's see what impact this has on the `Product` Entity.

```
class Product
{
    protected $productId;
    protected $name;
    protected $price;

    protected $surrogateCurrencyIsoCode;
    protected $surrogateAmount;

    public function __construct($aProductId, $aName, Money $aPrice)
    {
        $this->setProductId($aProductId);
        $this->setName($aName);
        $this->setPrice($aPrice);
    }
```

[12] http://doctrine-orm.readthedocs.org/en/latest/tutorials/embeddables.html

```
    private function setPrice(Money $aMoney)
    {
        $this->price = $aMoney;
        $this->surrogateAmount = $aMoney->amount();
        $this->surrogateCurrencyIsoCode = $aMoney->currency()->isoCode();
    }

    private function price()
    {
        if (null === $this->price) {
            $this->price = new Money(
                $this->surrogateAmount,
                new Currency($this->surrogateCurrency)
            );
        }

        return $this->price;
    }

    // ...
}
```

As you can see, there are two new attributes. One for the amount and another for the ISO code of the currency. We have also updated the `setPrice` method in order to keep attribute consistency when setting it. On top of this we have updated the `price` getter in order to return the `Money` Value Object built from the new fields. Let's see how the corresponding XML Doctrine mapping should be changed.

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping
    xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
    https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
    <entity
        name="Product"
        table="product">
        <id
            name="id"
            column="id"
            type="string"
            length="255">
```

```
            <generator
                strategy="NONE">
            </generator>
        </id>
        <field
            name="name"
            type="string"
            length="255"
        />
        <field
            name="surrogateAmount"
            type="integer"
            column="price_amount"
        />
        <field
            name="surrogateCurrencyIsoCode"
            type="string"
            column="price_currency"
        />
    </entity>
</doctrine-mapping>
```

Surrogate attributes

These two new fields do not strictly belong to the Domain, as they do not refer to infrastructure details, but are an necessity due to the lack of embeddable support in Doctrine. There are alternatives that can push these two attributes outside the pure Domain, however, this approach is simpler, easier, and as a trade-off, acceptable. There is another use in this book of surrogate attributes, you can find it when surrogating Entity identities.

If we wish to push these two attributes outside of the Domain, this can be achieved through the use of an Abstract Factory[13]. First, we need to create a new Entity in our Infrastructure folder, `DoctrineProduct` that would extend from `Product` Entity. All surrogate fields will be placed in the new class, and methods such as `price` or `setPrice` should be reimplemented. We will map Doctrine to use the new `DoctrineProduct` as opposed to the `Product` Entity. Now, we are able to fetch Entities from the database, but, what about creating a new `Product`? At some point, we are required to call `new Product`, but because we need to deal with `DoctrineProduct` and we do not want our Application Services to know about infrastructure details, we will need to use Factories to create `Product` Entities. So, in every instance where Entity creation occurs with `new`, you will instead now call `createProduct` on `ProductFactory`.

[13] http://en.wikipedia.org/wiki/Abstract_factory_pattern

There could be many additional classes required to avoid placing the surrogate attributes in the original Entity. As such, it is our recommendation to surrogate all the Value Objects to the same Entity, though this leads to a less pure solution.

3.7.1.4 Serialized LOB and ad-hoc ORM

If the addition of searching capabilities to the Value Objects attributes is not important, there is another pattern that can be considered, the Serialized LOB. This pattern works by serializing the whole Value Object into a string format that can be persisted and fetched easily. The most significant difference between this solution and the Embedded alternative is that in the latter option the persistence footprint requirements get reduced to a single column.

```
CREATE TABLE `products` (
    id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
    price TEXT NOT NULL
)
```

In order to persist `Product` Entities using this approach, a change in the `DbalProductRepository` is required. The `Money` Value Object must be serialized into a string before persisting the final Entity.

```
class DbalProductRepository extends DbalRepository implements ProductRepository
{
    public function add(Product $aProduct)
    {
        $sql = 'INSERT INTO products VALUES (?, ?, ?)';
        $stmt = $this->connection()->prepare($sql);
        $stmt->bindValue(1, $aProduct->id());
        $stmt->bindValue(2, $aProduct->name());
        $stmt->bindValue(
            3,
            $this->serialize(
                $aProduct->price()
            )
        );

        // ...
    }

    private function serialize($object)
    {
        return serialize($object);
```

```
    }
}
```

Let's see how our `Product` is now represented in the database. The table column `price` is a `TEXT` type column that contains a serialization of a `Money` object representing 9,99 USD.

```
mysql> select * from products \G
*************************** 1. row ***************************
   id: 1
 name: Domain-Driven Design in PHP
price: O:22:"Ddd\Domain\Model\Money":2:{s:30:" Ddd\Domain\Model\Money amount";i:\
999;s:32:" Ddd\Domain\Model\Money currency";O:25:"Ddd\Domain\Model\Currency":1:{\
s:34:" Ddd\Domain\Model\Currency isoCode";s:3:"USD";}}
1 row in set (0.00 sec)
```

This approach does the job, however, it is not recommended due to problems occurring when refactoring classes in your code. Could you imagine the changes that would be required in our database representation, when moving the `Money` class from one namespace to another? Another trade-off, as explained before, is the lack of querying capabilities. It does not matter whether you use Doctrine or not, writing a query to get the products cheaper than say 200 USD is almost impossible whilst using a serialization strategy.

The querying issue can only be solved by using Embedded Values, however, the serialization refactoring problems can be fixed using a specialised library for handling serialization processes.

3.7.1.4.1 Improved Serialization with JMS Serializer

`serialize/unserialize` native PHP strategies have a problem when dealing with class and namespace refactoring. One alternative is use your own serialization mechanism, for example, concatenating the amount, a one character separator such as "|" and the currency ISO code. However, there is another better favored approach, using an open-source serializer library such as JMS Serializer[14]. Let's see an example of applying it for serializing a `Money` object.

[14] http://jmsyst.com/libs/serializer

```
$myMoney = new Money(
    999,
    new Currency('USD')
);

$serializer = JMS\Serializer\SerializerBuilder::create()->build();
$jsonData = $serializer->serialize($myMoney, 'json');
```

In order to unserialize the object, the process is straight forward.

```
$serializer = JMS\Serializer\SerializerBuilder::create()->build();
// ...
$myMoney = $serializer->deserialize($jsonData, 'Ddd\Domain\Model\Money', 'json');
```

With this example, you can refactor your `Money` class without having to update your database. JMS Serializer can be used in many different scenarios, for example, when working with REST APIs. An important feature is the ability to specify what attributes of an object should be omitted in the serialization process, a password, for example.

Check the Mapping Reference[15] and the Cookbook[16] for more information. JMS Serializer is a must in any DDD project.

3.7.1.5 Serialized LOB with Doctrine

In Doctrine, there are different ways of serializing objects in order to eventually persist them.

3.7.1.5.1 Doctrine Object Mapping Type

Doctrine has support for the Serialize LOB pattern. There are plenty of predefined mapping types you can use in order to match Entity attributes with database columns or even tables. One of those mappings is the `object` type. It maps a SQL CLOB to a PHP object using `serialize()` and `unserialize()`.

As the Documentation says: “Object Type maps and converts object data based on PHP serialization. If you need to store an exact representation of your object data, you should consider using this type as it uses serialization to represent an exact copy of your object as string in the database. Values retrieved from the database are always converted to PHP’s object type using unserialization or null if no data is present.

This type will always be mapped to the database vendor’s `text` type internally as there is no way of storing a PHP object representation natively in the database. Furthermore this type requires a

[15] http://jmsyst.com/libs/serializer/master/reference/xml_reference
[16] http://jmsyst.com/libs/serializer/master/cookbook

SQL column comment hint so that it can be reverse engineered from the database. Doctrine cannot correctly map back this type correctly using vendors that do not support column comments, and will instead fall back to the text type instead. Because the built-in text type of PostgreSQL does not support NULL bytes, the object type will result in unserialization errors. A workaround to this problem is to serialize()/unserialize() and base64_encode()/base64_decode() PHP objects and store them into a text field manually."

Let's see a possible XML mapping for the Product Entity using the `object` type.

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping
    xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
    https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
    <entity
        name="Product"
        table="products">
        <id
            name="id"
            column="id"
            type="string"
            length="255">
            <generator
                strategy="NONE">
            </generator>
        </id>
        <field
            name="name"
            type="string"
            length="255"
        />
        <field
            name="price"
            type="object"
        />
    </entity>
</doctrine-mapping>
```

The key addition is the `type="object"` that tells Doctrine that we are now going to be using an `object` mapping. Let's now see how we could create and persist an Product entity using Doctrine.

```
// ...
$em->persist($product);
$em->flush($product);
```

Let's check that if we now fetch our Product Entity from the database it is returned in an expected state.

```
// ...
$repository = $em->getRepository('Ddd\\Domain\\Model\\Product');
$item = $repository->find(1);
var_dump($item);

/*
class Ddd\Domain\Model\Product#177 (3) {
  protected $productId =>
  int(1)
  protected $name =>
  string(41) "Domain-Driven Design in PHP"
  protected $money =>
  class Ddd\Domain\Model\Money#174 (2) {
    private $amount =>
    string(3) "100"
    private $currency =>
    class Ddd\Domain\Model\Currency#175 (1) {
      private $isoCode =>
      string(3) "USD"
    }
  }
}
*/
```

Last, but not least, as the Doctrine documentation states: "Object types are compared by reference, not by value. Doctrine updates this value if the reference changes and therefore behaves as if these objects are immutable value objects." Check the Doctrine Basic Mapping Types reference[17] for more information.

This approach suffers from the same refactoring issues as did the ad-hoc ORM. The `object` mapping type is internally using `serialize/unserialize`. What about instead using our own serialization?

[17] http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#doctrine-mapping-types

3.7.1.5.2 Doctrine Custom Types

Another option is to handle the Value Object persistence using a Doctrine Custom Type. A Custom Type adds to Doctrine a new mapping type that describes a custom transformation between an Entity field and the database representation to persist it.

As the Doctrine documentation explains "just redefining how database types are mapped to all the existing Doctrine types is not at all that useful. You can define your own Doctrine Mapping Types by extending `Doctrine\DBAL\Types\Type`. You are required to implement 4 different methods to get this working."

With the `object` type, the serialization step includes information, such as the class, which makes it quite difficult to safely refactor our code. Let's try to improve on this solution. Think about a custom serialization process that could solve the problem. One such way could be to persist the `Money` Value Object as a string in the database encoded in "amount|isoCode" format?

```
use Ddd\Domain\Model\Currency;
use Ddd\Domain\Model\Money;
use Doctrine\DBAL\Types\TextType;
use Doctrine\DBAL\Platforms\AbstractPlatform;

class MoneyType extends TextType
{
    const MONEY = 'money';

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        $value = parent::convertToPHPValue($value, $platform);

        $value = explode('|', $value);
        return new Money(
            $value[0],
            new Currency($value[1])
        );
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        return implode(
            '|',
            [
                $value->amount(),
                $value->currency()->isoCode()
            ]
```

```
        );
    }

    public function getName()
    {
        return self::MONEY;
    }
}
```

Using Doctrine you are required to register all Custom Types. It is common to use an EntityManagerFactory that centralizes this EntityManager creation. You could alternatively do this step in bootstrapping your application.

```
use Doctrine\DBAL\Types\Type;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Tools\Setup;

class EntityManagerFactory
{
    public function build()
    {
        Type::addType(
            'money',
            'Ddd\\Infrastructure\\Persistence\\Doctrine\\Type\\MoneyType'
        );

        return EntityManager::create(
            array(
                'driver'   => 'pdo_mysql',
                'user'     => 'root',
                'password' => '',
                'dbname'   => 'ddd',
            ),
            Setup::createXMLMetadataConfiguration(
                [__DIR__.'/config'],
                true
            )
        );
    }
}
```

Now, we need to specify in the mapping that we want to use our Custom Type.

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping>
    <entity
        name="Product"
        table="product">
        <!-- ... -->
        <field
            name="price"
            type="money"
        />
    </entity>
</doctrine-mapping>
```

Why use XML mapping?

Thanks to the XSD schema validation in the headers of the XML mapping file, most IDE's provide auto-complete functionality for all the elements and attributes present in the mapping definition.

Let's check the database how the price was persisted using this approach.

```
mysql> select * from products \G
*************************** 1. row ***************************
   id: 1
 name: Domain-Driven Design in PHP
price: 999|USD
1 row in set (0.00 sec)
```

This approach is an improvement on the one before in terms of future refactoring, however, searching capabilities remain limited due to the format of the column. With the Doctrine Custom types you can improve the situation a little, but still not the best option for building your DQL queries. Check the Doctrine Custom Mapping Types reference[18] for more information.

Time to discuss

Think and discuss with a peer how would you create a Doctrine Custom Type using JMS to serialize and unserialize a Value Object.

[18] http://doctrine-orm.readthedocs.org/en/latest/cookbook/custom-mapping-types.html

3.7.2 Persisting a Collection of Value Objects

Imagine that now we would like to add to our `Product` Entity a collection of prices to be persisted. These prices could represent the different prices the product has bore throughout its lifetime, or the product price in different currencies. This could be named `HistoricalPrice` as shown below.

```
class HistoricalProduct extends Product
{
    /**
     * @var Money[]
     */
    protected $prices;

    public function __construct($aProductId, $aName, Money $aPrice, array $someP\
rices)
    {
        parent::__construct($aProductId, $aName, $aPrice);
        $this->setPrices($somePrices);
    }

    private function setPrices(array $somePrices)
    {
        $this->prices = $somePrices;
    }

    public function prices()
    {
        return $this->prices;
    }
}
```

`HistoricalProduct` extends from `Product` so it inherits the same behaviour plus the price collection functionality.

As in the previous sections, Serialization is a plausible approach if you do not care about querying capabilities, however, Embedded Values should be a possibility if we know exactly how many prices we want to persist. But, what happens if we want to persist a undetermined collection of historical prices?

3.7.2.1 Collection Serialized into a Single Column

Serializing a collection of Value Objects into a single column is most likely the easiest solution. Everything that has previously been discussed through persisting a single Value Object applies in this situation. With Doctrine you can use an Object or Custom Type, with some additional considerations to bear in mind: Value Objects should be small in size, however, if you wish to persist a large collection, be sure to consider the maximum column length and the max row width that your database engine can handle.

Exercise

Think up both Doctrine Object Type and Doctrine Custom Type implementation strategies for persisting a `Product` with different prices.

3.7.2.2 Collection backed by a Join Table

In the case of needing to persist an Entity with a collection of Value Objects and need querying capabilities, you have the choice to persist the Value Objects as Entities. In terms of the Domain, those objects would still be Value Objects but we will need to give them an id and relate them in a "one-to-many"/"one-to-one" relation with the owner, a real Entity. To summarise, your ORM handles the collection of Value Objects as Entities, but in your Domain they are still treated as Value Objects.

The main idea behind the "Join Table" strategy is to create a table that connects the owner Entity and its Value Objects. Let's see a database representation.

```
CREATE TABLE `historical_products` (
  `id` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `name` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `price_amount` int(11) NOT NULL,
  `price_currency` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

`historical_products` table will look the same as `products`. Remember that `HistoricalProduct` extends `Product` Entity in order to easily show how to deal with persisting an collection. A new `prices` table is now required in order to persist all the different `Money` Value Objects that a `Product` Entity can handle.

```
CREATE TABLE `prices` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `amount` int(11) NOT NULL,
  `currency` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Finally, a table that relates `products` and `prices` is needed.

```
CREATE TABLE `products_prices` (
  `product_id` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `price_id` int(11) NOT NULL,
  PRIMARY KEY (`product_id`,`price_id`),
  UNIQUE KEY `UNIQ_62F8E673D614C7E7` (`price_id`),
  KEY `IDX_62F8E6734584665A` (`product_id`),
  CONSTRAINT `FK_62F8E6734584665A` FOREIGN KEY (`product_id`) REFERENCES `histor\
ical_products` (`id`),
  CONSTRAINT `FK_62F8E673D614C7E7` FOREIGN KEY (`price_id`) REFERENCES `prices` \
(`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

3.7.2.2.1 Collection backed by a Join Table with Doctrine

Doctrine requires that all database entities to have a unique identity. Because we want to persist `Money` Value Objects we need to then add an *artificial* identity so Doctrine can handle its persistence. There are two options: including the surrogate identity in the `Money` Value Object or placing it in an extended class.

The issue with the first option is that the new identity is only required due to the Database persistence layer. This identity is not part of the Domain.

An issue with the second option is the amount of alteration that are required in order to avoid this said *boundary leak*. With a class extension, creating `new` instances of the `Money` Value Object class from any Domain Object is not recommended, as it would break the Inversion Principle. The solution is to again create a Money Factory that would need to be passed into Application Services and any other Domain objects.

In this scenario, we recommend to use the first option. Let's review the changes required to the `Money` Value Object.

```
class Money
{
    private $amount;
    private $currency;

    private $surrogateId;
    private $surrogateCurrencyIsoCode;

    public function __construct($amount, Currency $currency)
    {
        $this->setAmount($amount);
        $this->setCurrency($currency);
    }

    private function setAmount($amount)
    {
        $this->amount = $amount;
    }

    private function setCurrency(Currency $currency)
    {
        $this->currency = $currency;
        $this->surrogateCurrencyIsoCode = $currency->isoCode();
    }

    public function currency()
    {
        if (null === $this->currency) {
            $this->currency = new Currency($this->surrogateCurrencyIsoCode);
        }

        return $this->currency;
    }

    public function amount()
    {
        return $this->amount;
    }

    public function equals(Money $aMoney)
    {
        return
```

```
            $this->amount() === $aMoney->amount()
            && $this->currency()->equals($this->currency());
    }
}
```

As seen, two new attributes have been added. The first one, `surrogateId` is not used by our Domain, but is required for the persistence infrastructure to persist this Value Object as an Entity in our Database. The second one, `surrogateCurrencyIsoCode` holds the ISO code for the currency. Using these new attributes it is really easy to map our Value Object with Doctrine.

The `Money` mapping is quite straight forward.

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mappin\
g" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="htt\
p://doctrine-project.org/schemas/orm/doctrine-mapping https://raw.github.com/doc\
trine/doctrine2/master/doctrine-mapping.xsd">
  <entity name="Ddd\Domain\Model\Money" table="prices">
    <id name="surrogateId" type="integer" column="id">
      <generator strategy="AUTO"></generator>
    </id>
    <field name="amount" type="integer" column="amount"/>
    <field name="surrogateCurrencyIsoCode" type="string" column="currency"/>
  </entity>
</doctrine-mapping>
```

Using Doctrine, the `HistoricalProduct` Entity would have following mapping.

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mappin\
g" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="htt\
p://doctrine-project.org/schemas/orm/doctrine-mapping https://raw.github.com/doc\
trine/doctrine2/master/doctrine-mapping.xsd">
  <entity name="Ddd\Domain\Model\HistoricalProduct" table="historical_products" \
repository-class="Ddd\Infrastructure\Domain\Model\DoctrineHistoricalProductRepos\
itory">
    <many-to-many field="prices" target-entity="Ddd\Domain\Model\Money">
      <cascade>
        <cascade-all/>
      </cascade>
      <join-table name="products_prices">
        <join-columns>
```

```
          <join-column name="product_id" referenced-column-name="id" />
        </join-columns>
        <inverse-join-columns>
          <join-column name="price_id" referenced-column-name="id" unique="true"\
 />
        </inverse-join-columns>
      </join-table>
    </many-to-many>
  </entity>
</doctrine-mapping>
```

3.7.2.2.2 Collection backed by a Join Table with ad-hoc ORM

It is possible to do the same with an ad-hoc ORM, where Cascade INSERTS and JOIN queries are required. The only consideration to be careful about is how removal of Value Objects are handled, in order not to leave orphan `Money` Value Objects.

> **Exercise**
>
> Think up a solution for `DbalHistoricalRepository` that would handle the `persist` method.

3.7.2.3 Collection backed by a Database Entity

"Database Entity" is the same strategy as "Join Table" with the addition that the Value Object is only managed by the owner Entity. In the current scenario, consider that the `Money` Value Object is only used by the `HistoricalProduct` Entity. A "Join Table" would be over-complex. So, the same result could be achieved using a "one-to-many" database relation.

> **Exercise**
>
> Think of the mapping required between `HistoricalProduct` and `Money` if a "Database Entity" approach was used.

3.7.3 NoSQL

What about NoSQL mechanisms such as Redis, MongoDB, or CouchDB? You unfortunately do not escape from these problems. In order to persist an Aggregate using Redis, you need to serialize it into a string before *setting* the value. If you use PHP `serialize/unserialize` methods you will face namespace or class name refactoring issues again. If you choose to serialize in a custom way (json, custom string, etc.) you are required to again rebuild the Value Object during Redis retrieval.

3.7.3.1 PostgreSQL and JSONB

If our database engine would allow us to not only use Serialized LOB strategy but also search based on its value, we would have the best of both approaches. Well, now you can. As of PostgreSQL version 9.4 support for JSONB[19] has been added. Value Objects can now be persisted as JSON serializations and subsequently queried within this JSON serialization.

3.8 Security

Another interesting detail about modeling your Domain concepts using Value Objects is about its security benefits. Consider an application within a selling flight tickets context. If you deal with International Air Transport Association airport codes, also known as the IATA codes[20], you can decide to use a string or model yhe concept using a Value Object. If you choose to go with the string, think about all the places where you will be checking that the string is a valid IATA code. What's the chance to forget anywhere important? On the other side, think about trying to instantiate a `new IATA("BCN'; DROP TABLE users;--")`. If you centralize the guards[21] in the constructor and the pass into your model a IATA Value Object avoiding SQL Injections or similar attacks get easier.

If you want to know more about the security side of Domain-Driven Design, you can follow Dan Bergh Johnsson[22] or read his blog[23].

3.9 Wrap-up

Using Value Objects for modeling concepts in your Domain that measure, quantify or describe is highly recommended. As shown, Value Objects are easy to create, maintain and test. In order to handle persistence within a DDD application, using an ORM is a must. However, in order to persist Value Objects using Doctrine without current support for embedded values (scheduled for version 2.5), there are two options: adding the Value Object fields directly into your Entity and mapping them (less elegant, but easier) or extending your entities (far more elegant, but more complex).

[19] http://www.postgresql.org/docs/9.4/static/functions-json.html
[20] https://en.wikipedia.org/wiki/International_Air_Transport_Association_airport_code
[21] https://en.wikipedia.org/wiki/Guard_(computer_science)
[22] https://twitter.com/danbjson
[23] http://dearjunior.blogspot.com.es/search/label/domain%20driven%20security

4. Entities

4.1 Introduction

We have talked about the benefits of trying to model out everything in the domain as a value object first. But when modeling the domain, there will be probably situations where you will find that some concept in the ubiquitous language will be demanding a thread of identity.

Clear examples of this would be

- A **person**. A person has always an identity and it's always the same regarding their name, or document identifier.
- An **order** in an e-commerce system. In that context every new order created has its own identity and it's the same over time.

This kind of concept, have an identity that endures over the time. In PHP that would be plain old classes. For example, in the case of a person

```
namespace Ddd\Identity\Domain\Model;

class Person
{
    private $identificationNumber;
    private $firstName;
    private $lastName;

    public function __construct($anIdentificationNumber, $aFirstName, $aLastName)
    {
        $this->identificationNumber = $anIdentificationNumber;
        $this->firstName = $aFirstName;
        $this->lastName = $aLastName;
    }

    public function identificationNumber()
    {
        return $this->identificationNumber;
    }
```

```
    public function firstName()
    {
        return $this->firstName;
    }

    public function lastName()
    {
        return $this->lastName;
    }
}
```

Or in the case of an order, would be

```
namespace Ddd\Billing\Domain\Model\Order;

class Order
{
    private $id;
    private $amount;
    private $firstName;
    private $lastName;

    public function __construct($anId, Amount $amount, $aFirstName, $aLastName)
    {
        $this->id = $anId;
        $this->amount = $amount;
        $this->firstName = $aFirstName;
        $this->lastName = $aLastName;
    }

    public function id()
    {
        return $this->id;
    }

    public function firstName()
    {
        return $this->firstName;
    }

    public function lastName()
    {
```

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

4.2 Objects vs Primitive types

Most of the time the identity of an entity is represented as a primitive type: usually a string or an integer. But using a *value object* to represent it has more advantages:

- Value Objects are immutable, so they cannot be modified.
- Value Objects are complex types that can have custom behaviours that otherwise with primitive types cannot have. Put for example **the equality operation**. With value objects, equality operations can be modelled and encapsulated in their own classes, making concepts go from implicit to explicit.

```
namespace Ddd\Billing\Domain\Model;

class OrderId
{
    private $id;

    public function __construct($anId)
    {
        $this->id = $anId;
    }

    public function id()
    {
        return $this->id;
    }

    public function equalsTo(OrderId $anOrderId)
    {
        return $anOrderId->id === $this->id;
    }
}

class Order
{
    private $id;
```

```
    private $amount;
    private $firstName;
    private $lastName;

    public function __construct(OrderId $anOrderId, Amount $amount, $aFirstName,\
 $aLastName)
    {
        $this->id = $anOrderId;
        $this->amount = $amount;
        $this->firstName = $aFirstName;
        $this->lastName = $aLastName;
    }

    public function id()
    {
        return $this->id;
    }

    public function firstName()
    {
        return $this->firstName;
    }

    public function lastName()
    {
        return $this->lastName;
    }

    public function amount()
    {
        return $this->amount;
    }
}
```

4.3 Identity Operation

As stated before the identity of an entity is what it defines it. So then, handling it is an important aspect of the entity. There are usually 4 ways to define the identity of an entity: A client provides the identity, the application itself provides an identity, the persistence mechanism provides the identity or another bounded context provides an identity.

4.3.1 Persistence Mechanism Generates Identity

Usually, the simplest way is to let the persistence mechanism to generate the identity because the vast major of persistence mechanisms supports some kind of identity generation, like MySQL's AUTO_INCREMENT attribute or Oracle's/Postgres sequences. This, although simple, have a major drawback: **We won't have the identity of the entity until we persist it**. So to some degree, if we are going with persistence mechanism generated identities we will couple the identity operation with the underlying persistence store.

```
CREATE TABLE `orders` (
  `id` int(11) NOT NULL auto_increment,
  `amount` decimal(10, 5) NOT NULL,
  `first_name` varchar(100) NOT NULL,
  `last_name` varchar(100) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

And then we might consider this code

```
namespace Ddd\Identity\Domain\Model;

class Person
{
    private $identificationNumber;
    private $firstName;
    private $lastName;

    public function __construct($anIdentificationNumber, $aFirstName, $aLastName)
    {
        $this->identificationNumber = $anIdentificationNumber;
        $this->firstName = $aFirstName;
        $this->lastName = $aLastName;
    }

    public function identificationNumber()
    {
        return $this->identificationNumber;
    }

    public function firstName()
    {
        return $this->firstName;
```

```
    }

    public function lastName()
    {
        return $this->lastName;
    }
}
```

4.3.1.1 Surrogate Identity

Sometimes using an ORM to map entities to a persistence store, some constraints are imposed, for example Doctrine demands of an integer field if an *IDENTITY* generator strategy is used. This can conflict with the domain model if it requires another kind of identity.

The simplest way to handle that situation is by using a *Layer SuperType*[1] where we put the identity field created for the persistence store.

```
namespace Ddd\Common\Domain\Model;

abstract class IdentifiableDomainObject
{
    private $id;

    protected function id()
    {
        return $this->id;
    }

    protected function setId($anId)
    {
        $this->id = $anId;
    }
}
```

[1] http://martinfowler.com/eaaCatalog/layerSupertype.html

```
namespace Acme\Billing\Domain;

use Acme\Common\Domain\IdentifiableDomainObject;

class Order extends IdentifiableDomainObject
{
    private $orderId;

    public function orderId()
    {
        if (null === $this->orderId) {
            $this->orderId = new OrderId($this->id());
        }

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

4.3.1.2 Active Record vs Data Mapper for Rich Domain Models

Every project always face the decision of which ORM should use. There are a lot of good ORMs for PHP out there: Doctrine, Propel, Eloquent, Paris and many many more.

Most of them are *Active Record*[2] implementations. An Active Record implementation is fine mostly for CRUD applications, but it's not the ideal solution for rich domain models, for the following reasons

- The *Active Record* pattern assumes a one-to-one relation between an entity and a database table. So **it couples the design of the database to the design of the object system**. And in a rich domain model sometimes entities are constructed with information that may come from different data sources.
- Advanced things like collections or inheritance are tricky to implement
- Most of the implementations force the use, through inheritance, of some sort of constructions to impose several conventions. This can lead to *persistence leakage* into the domain model by coupling the domain model with the ORM. The only Active Record implementation the author has seen that does not impose inheriting from a base classes is Castle ActiveRecord[3] from Castel Project[4], a .NET framework. While this leads to some degree of separation between persistence and domain concerns in the produced entities, it doesn't prevent from coupling the data design with the objects design.

[2] http://www.martinfowler.com/eaaCatalog/activeRecord.html
[3] http://docs.castleproject.org/Active%20Record.MainPage.ashx
[4] http://www.castleproject.org/

Currently the best ORM for PHP out there is *Doctrine*[5]. It's an implementation of the *Data Mapper pattern*[6]. Data Mapper decouples the persistence concerns from the domain concerns, leading to persistence-free entities. This makes that tool the best to use, if someone cares to build a rich domain model.

4.3.2 Client Provides Identity

Sometimes, dealing with certain domains, the identities come naturally with the client consuming the domain model. Probably this is the ideal case, because the identity can be modelled quite easy

```
namespace Ddd\Catalog\Domain\Model\Book;

class ISBN
{
    private $isbn;

    private function __construct($anIsbn)
    {
        $this->setIsbn($anIsbn);
    }

    private function setIsbn($anIsbn)
    {
        $this->assertIsbnIsValid($anIsbn, 'The ISBN is invalid.');

        $this->isbn = $anIsbn;
    }

    public static function create($anIsbn)
    {
        return new static($anIsbn);
    }

    private function assertIsbnIsValid($anIsbn, $string)
    {
        // ... Validates an ISBN code
    }
}
```

[5] http://doctrine-project.org

[6] http://www.martinfowler.com/eaaCatalog/dataMapper.html

```
class Book
{
    private $isbn;
    private $title;

    public function __construct(ISBN $anIsbn, $aTitle)
    {
        $this->isbn = $anIsbn;
        $this->title = $aTitle;
    }
}


$book = new Book(
    ISBN::create('...'),
    'Domain-Driven Design with PHP by Examples'
);
```

4.3.3 Application Generates Identity

If the client cannot provide the identity generally the preferred way to handle the identity operation is to let the application generate the identities, usually through a UUID. There are several libraries in PHP that generate UUIDs. An they can be found at packagist.

```
https://packagist.org/search/?q=uuid
```

The best recommended would be the one developed by *Ben Ramsey* at **https://github.com/ramsey/uuid** because it has about 500 watchers on Github and about 240.000 installations on packagist, at the time of writing.

The preferred place to put the creation of the identity would be inside a *Repository*

```
namespace Ddd\Billing\Domain\Model\Order;

interface OrderRepository
{
    public function nextIdentity();
    public function add(Order $anOrder);
    public function remove(Order $anOrder);
}

class OrderId
```

```
{
    private $id;

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

    public static function create($anId)
    {
        return new static($anId);
    }
}
```

```
namespace Ddd\Billing\Infrastructure\Doctrine\Order;

use Ddd\Billing\Domain\Model\Order\Order;
use Ddd\Billing\Domain\Model\Order\OrderId;
use Ddd\Billing\Domain\Model\Order\OrderRepository;

use Doctrine\ORM\EntityRepository;
use Rhumsaa\Uuid\Uuid;

class DoctrineOrderRepository
    extends EntityRepository
    implements OrderRepository
{
    public function nextIdentity()
    {
        return OrderId::create(
            strtoupper(Uuid::uuid4())
        );
    }

    public function add(Order $anOrder)
    {
        $this->getEntityManager()->persist($anOrder);
    }

    public function remove(Order $anOrder)
    {
        $this->getEntityManager()->remove($anOrder);
```

```
    }
}
```

4.3.4 Other Bounded Context Generates Identity

Probably this would be the most complex identity generation strategy, because it enforces to have a local entity to be dependent not only on local bounded context events, but in external bounded contexts events. So in terms of maintenance, the cost would be high.

Other Bounded Context provides of some UI widget to select the identity of the local entity. This can even grab some properties of the remote entity to its own.

When synchronization is needed between the entities of the Bounded Contexts, usually can be achieved with an *Event Driven* architecture on each of the Bounded Context that need to be notified when the original entity is changed.

4.4 Persisting Entities

Currently, as discussed earlier in the chapter, the best tool to use to save entity state to a persistent store is Doctrine ORM.

Doctrine has several ways to specify entity metadata: by annotations in entities code, by XML, by YAML or by plain PHP. In this chapter we are going to discuss in deep why annotations are not the best idea to use when mapping entities.

4.4.1 Setting Up Doctrine

First of all we need to require it through composer. In the root of the project the command below has to be executed

```
> php composer.phar require "doctrine/orm=~2.4"
```

And then these lines will allow to setup doctrine

```php
require_once "/path/to/vendor/autoload.php";

use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$paths = ["/path/to/entity-files"];
$isDevMode = false;

// the connection configuration
$dbParams = array(
    'driver'   => 'pdo_mysql',
    'user'     => 'the_database_username',
    'password' => 'the_database_password',
    'dbname'   => 'the_database_name',
);

$config = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create($dbParams, $config);
```

4.4.2 Mapping Entities

By default Doctrine documentation presents the code examples using annotations. So we do start the code example using annotations and discussing why they should be avoided, whenever possible.

To do so, we will bring back the Order class discussed earlier in this chapter.

4.4.2.1 Mapping Entities Using Annotated Code

One of the features used to present Doctrine when it was released was that mapping information can be specified using annotated code

What's an annotation?

An annotation is a special form of metadata. In PHP is put under source code comments. For example, PHPDocumentor makes use of annotations to build API information or PHPUnit uses some annotations to specify dataProviders or to provide expectations about exceptions thrown by a piece of code

```
class SumTest extends PHPUnit_Framework_TestCase {
       /**
        * @dataProvider aMethodName
        */
       public function testAddition() {
           // ...
       }
}
```

So in order to map the Order entity to the persistence store first of all the source code for the Order should be modified to add the Doctrine annotations

```
use Doctrine\ORM\Mapping\Entity;
use Doctrine\ORM\Mapping\Id;
use Doctrine\ORM\Mapping\GeneratedValue;
use Doctrine\ORM\Mapping\Column;

/** @Entity */
class Order
{
    /** @Id @GeneratedValue(strategy="AUTO") */
    private $id;

    /** @Column(type="decimal", precision="10", scale="5") */
    private $amount;

    /** @Column(type="string") */
    private $firstName;

    /** @Column(type="string") */
    private $lastName;

    public function __construct(Amount $anAmount, $aFirstName, $aLastName)
    {
        $this->amount = $anAmount;
```

```
        $this->firstName = $aFirstName;
        $this->lastName = $aLastName;
    }

    public function id()
    {
        return $this->id;
    }

    public function firstName()
    {
        return $this->firstName;
    }

    public function lastName()
    {
        return $this->lastName;
    }

    public function amount()
    {
        return $this->amount;
    }
}
```

And then to persist the entity to the persistent store it's just as easy as

```
$order = new Order(
    new Amount(15, Currency::EUR()),
    'AFirstName',
    'ALastName'
);

$entityManager->persist($order);
$entityManager->flush();
```

At a first glance, this code can look simple and this can be an easy way to specify mapping information. But this way comes at a cost. What's odd about the final code?

First of all, **domain concerns are mixed with infrastructure concerns**. *Order* is a domain concept whereas *Table*, *Column* and so on are infrastructure concerns.

And so it is, that this entity is tightly coupled to the mapping information specified by the annotations in the source code. If the entity were required to be persisted using another entity manager and with a different mapping metadata, it would not be possible.

Annotations tend to lead to side-effects and tight coupling. So it would be better to not use them.

So what's the best way to specify mapping information? The best way is the one that allows to separate the mapping information from the entity itself. And this can be achieved by using XML mapping, YAML mapping or PHP mapping. In this book we are going to cover XML mapping.

4.4.2.2 Mapping Entities Using XML

To map the Order entity using the XML mapping, first the setup code of Doctrine should be changed slightly.

```
require_once "/path/to/vendor/autoload.php";

use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$paths = ["/path/to/mapping-files"];
$isDevMode = false;

// the connection configuration
$dbParams = [
    'driver'   => 'pdo_mysql',
    'user'     => 'the_database_username',
    'password' => 'the_database_password',
    'dbname'   => 'the_database_name',
];

$config = Setup::createXMLMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create($dbParams, $config);
```

The mapping file should be created on the path where Doctrine will search for the mapping files. And the mapping files should be named after the fully qualified class name and replacing the backslash (\) for dots. So following the example

```
Acme\Billing\DomainModel\Order
```

Would have the mapping file named as

```
Acme.Billing.DomainModel.Order.dcm.xml
```

In addition, it's convenient that all the mapping files use a special XML Schema created specially to specify mapping information

```
https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd
```

And the final mapping file would be

```
<?xml version="1.0" encoding="UTF-8"?>
<doctrine-mapping
    xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
    https://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

    <entity name="Ddd\Billing\Domain\Model\Order"
            table="orders">

        <id name="id" column="id" type="guid">
            <generator strategy="NONE" />
        </id>

        <field name="amount"
               type="decimal"
               nullable="false"
               scale="10"
               precision="5" />

        <field name="firstName" type="string" nullable="false" />
        <field name="lastName" type="string" nullable="false" />
    </entity>

</doctrine-mapping>
```

4.5 Testing entities

It's relatively easy to test entities. Just because they are plain old PHP classes with actions derived of the domain concept they represent. The focus of the test should be the invariants that the entity protects, because probably the behaviour on the entities will be modelled around those invariants.

For the example and for the sake of simplicity, suppose a domain model for a blog is needed. A possible one could be

```
class Post
{
    private $title;
    private $content;
    private $status;
    private $createdAt;
    private $publishedAt;

    public function __construct($aContent, $title)
    {
        $this->setContent($aContent);
        $this->setTitle($title);

        $this->unpublish();
        $this->createdAt(new DateTimeImmutable());
    }

    private function setContent($aContent)
    {
        $this->assertNotEmpty($aContent);

        $this->content = $aContent;
    }

    private function setTitle($aPostTitle)
    {
        $this->assertNotEmpty($aPostTitle);

        $this->title = $aPostTitle;
    }

    private function setStatus(Status $aPostStatus)
    {
        $this->assertIsAValidPostStatus($aPostStatus);

        $this->status = $aPostStatus;
    }

    private function createdAt(DateTimeImmutable $aDate)
    {
        $this->assertIsAValidDate($aDate);
```

```
        $this->createdAt = $aDate;
    }

    private function publishedAt(DateTimeImmutable $aDate)
    {
        $this->assertIsAValidDate($aDate);

        $this->publishedAt = $aDate;
    }

    public function publish()
    {
        $this->setStatus(
            Status::published()
        );

        $this->publishedAt(new DateTimeImmutable());
    }

    public function unpublish()
    {
        $this->setStatus(
            Status::draft()
        );

        $this->publishedAt = null;
    }

    public function isPublished()
    {
        return $this->status->equalsTo(Status::published());
    }

    public function publicationDate()
    {
        return $this->publishedAt;
    }
}

class Status
{
    const PUBLISHED = 10;
```

```
    const DRAFT     = 20;

    private $status;

    public static function published()
    {
        return new self(self::PUBLISHED);
    }

    public static function draft()
    {
        return new self(self::DRAFT);
    }

    private function __construct($aStatus)
    {
        $this->status = $aStatus;
    }

    public function equalsTo(self $aStatus)
    {
        return $this->status === $aStatus->status;
    }
}
```

In order to test this domain model we must ensure the test covers all the Post invariants

```
class PostTest extends PHPUnit_Framework_TestCase
{
    /** @test */
    public function aNewPostIsNotPublishedByDefault()
    {
        $aPost = new Post(
            'A Post Content',
            'A Post Title'
        );

        $this->assertFalse(
            $aPost->isPublished()
        );

        $this->assertNull(
```

```
            $aPost->publicationDate()
        );
    }

    /** @test */
    public function aPostCanBePublishedWithAPublicationDate()
    {
        $aPost = new Post(
            'A Post Content',
            'A Post Title'
        );

        $aPost->publish();

        $this->assertTrue(
            $aPost->isPublished()
        );

        $this->assertInstanceOf(
            'DateTimeImmutable',
            $aPost->publicationDate()
        );
    }
}
```

4.6 Validation

Validation is a highly important process in our domain model. It not only checks for the correctness of attributes, but entire objects and even the composition of those objects. Different levels of validation are required in order to keep this model in a valid state. Only because an object consists of valid attributes (on a per-basis), does not necessary mean the object (as a whole) is valid. Vice versa, valid objects do not necessarily mean we have valid compositions.

4.6.1 Attribute Validation

Some people understand **validation** as the process whereby a service validates the state of a given object. In this case, the validation conforms to a design-by-contract[7] approach - consisting of pre-conditions, post-conditions and invariants. One such way to protect a single attribute is by using Value Objects. In order to make our design more flexible for change, we focus only on asserting domain pre-conditions that must be met. Here we will be using guards as an easy way of validating the pre-conditions:

[7] http://en.wikipedia.org/wiki/Design_by_contract

```
class Username
{
    const MIN_LENGTH = 5;
    const MAX_LENGTH = 10;
    const FORMAT = '/^[a-zA-Z0-9_]+$/';

    private $username;

    public function __construct($username)
    {
        $this->setUsername($username);
    }

    private setUsername($username)
    {
        $this->assertNotEmpty($username);
        $this->assertNotTooShort($username);
        $this->assertNotTooLong($username);
        $this->assertValidFormat($username);
        $this->username = $username;
    }

    private function assertNotEmpty($username)
    {
        if (empty($username)) {
            throw new InvalidArgumentException('Empty username');
        }
    }

    private function assertNotTooShort($username)
    {
        if (strlen($username) < self::MIN_LENGTH) {
            throw new InvalidArgumentException(sprintf('Username must be %d char\
acters or more', self::MIN_LENGTH));
        }
    }

    private function assertNotTooLong($username)
    {
        if (strlen($username) > self::MAX_LENGTH) {
            throw new InvalidArgumentException(sprintf('Username must be %d char\
acters or less', self::MAX_LENGTH));
```

```
        }
    }

    private function assertValidFormat($username)
    {
        if (preg_match(self::FORMAT, $username) !== 1) {
            throw new InvalidArgumentException('Invalid username format');
        }
    }
}
```

As you can see in the example above, there are four pre-conditions that must be satisfied in order to build a `Username` value object:

- Must not be empty
- Must be at least 5 characters
- Must be less than 10 characters
- Must follow a format of alphanumeric characters or underscore

If all the pre-conditions are met, the attribute will be set and the object will be successfully built. Otherwise, a `InvalidArgumentException` will be raised, execution halted and the client will be shown an error.

Some developers may see this kind of validation as *defensive programming*. However, here we are not checking that the input is a string, or that nulls are not permitted. We cannot avoid people using our code incorrectly, but we can control the correctness of our domain state.

4.6.2 Entire Object Validation

There are times when an object composed of valid properties, as a whole, can still be deemed invalid. It can be tempting to add this kind of validation to the object itself, but generally this is an anti-pattern. Higher-level validation is likely to change at different times to the object itself. Also it is good practice to separate these responsibilities.

The validation informs the client about any errors that have been found, or collect the results to be reviewed later. Sometimes we do not want to stop the execution at the first sign of trouble.

An abstract and reusable `Validator` could be something like:

```
abstract class Validator
{
    private $validationHandler;

    public function __construct(ValidationHandler $validationHandler)
    {
        $this->validationHandler = $validationHandler;
    }

    protected function handleError($error)
    {
        $this->validationHandler->handleError($error);
    }

    abstract public function validate();
}
```

As a concrete example, we want to validate an entire `Location` object, composed of valid `Country`, `City` and `Postcode` value objects. These individual values however, might be in an invalid state at the time of validation. Maybe the city does not form part of the country or maybe the postcode does not follow the city format.

```
class Location
{
    private $country;
    private $city;
    private $postcode;

    public function __construct(Country $country, City $city, Postcode $postcode)
    {
        $this->country = $country;
        $this->city = $city;
        $this->postcode = $postcode;
    }

    public function getCountry()
    {
        return $this->country;
    }

    public function getCity()
    {
```

```
        return $this->city;
    }

    public function getPostcode()
    {
        return $this->postcode;
    }
}
```

The validator checks the state of the `Location` object in its entirety, analysing the meaning of the relationships between properties:

```
class LocationValidator extends Validator
{
    private $location;

    public function __construct(Location $location, ValidationHandler $validatio\
nHandler)
    {
        parent::__construct($validationHandler);
        $this->location = $location;
    }

    public function validate()
    {
        if (!$this->location->getCountry()->hasCity($this->location->getCity()))\
 {
            $this->handleError('City not found');
        }

        if (!$this->location->getCity()->isPostcodeValid($this->location->getPos\
tcode())) {
            $this->handleError('Invalid postcode');
        }
    }
}
```

Once all the properties have been set we are able to validate the entity, most likely after some described process. On the surface it looks as if the `Location` validates itself, this however is not the case. `Location` delegates this validation to a concrete validator instance, splitting these two clear responsibilities.

```
class Location
{
    //...

    public function validate(ValidationHandler $validationHandler)
    {
        $validator = new LocationValidator($this, $validationHandler);
        $validator->validate();
    }
}
```

4.6.2.1 Decoupling Validation Messages

With some minor changes to our existing implementation, we are able to decouple the validation messages from the validator:

```
class LocationValidationHandler implements ValidationHandler
{
    public function handleCityNotFoundInCountry();

    public function handleInvalidPostcodeForCity();
}

class LocationValidator
{
    private $location;
    private $validationHandler;

    public function __construct(Location $location, LocationValidationHandler $v\
alidationHandler)
    {
        $this->location = $location;
        $this->validationHandler = $validationHandler;
    }

    public function validate()
    {
        if (!$this->location->getCountry()->hasCity($this->location->getCity()))\
 {
            $this->validationHandler->handleCityNotFoundInCountry();
        }
```

```
        if (!$this->location->getCity()->isPostcodeValid($this->location->getPos\
tcode())) {
            $this->validationHandler->handleInvalidPostcodeForCity();
        }
    }
}
```

We also need to change the signature of the validation method to:

```
class Location
{
    //...

    public function validate(LocationValidationHandler $validationHandler)
    {
        $validator = new LocationValidator($this, $validationHandler);
        $validator->validate();
    }
}
```

4.6.3 Validating Object Compositions

Validating object compositions can be complicated, because of this, the preferred way of achieving this is through a Domain Service. The service then communicates with repositories in order to retrieve the valid Aggregate. Due to the likely complex object graphs that can be created, an Aggregate could be in an intermediate state, requiring other aggregates to be validated before-hand. We can use Domain Events to notify other parts of the system that a particular element has been validated.

4.7 Wrap-up

Some concepts in the domain demand identity, this is, mutations in their state don't change them as a concept. We've seen how modeling identity as a Value Object brings some benefits like immutability and logic for operating the identity itself. We've shown several ways of providing identity:

- Persistence mechanism: Easy to implement but you'll not have the identity before persisting the object, delaying and complicating event propagation.
- Surrogate id: Some ORMs require an extra field on your Entity to map the identity with the persisting mechanism Provided by the client: Sometimes the identity fits a domain concept and you could model it inside your domain.

- Generated by the application: You could use a library to generate IDs.
- Generated by Bounded Context: Probably the most complex strategy. Other bounded context provides an interface for generating Identities.

We've seen and discussed Doctrine as a persistence mechanism, the drawbacks of using the Active Record pattern and finally we've checked different levels of Entity validation:

- Attribute validation: Check for specifics inside the object state through pre-conditions, post-conditions and invariants.
- Entire object validation: Looks for consistency of an object as a whole. Extracting the validation to an external service is a good practice.
- Object compositions: Complex object compositions could be validated through Domain Services. A good way of communicating this to the rest of the application is through Domain Events.

5. Services

5.1 Introduction

When there are operations that need to be represented, we can consider them to be *services*.

There are typically three different types of service which you will encounter, these are:

Type	Characteristics
Application	Operate on scalar types, transforming them into domain types. Scalar types can be considered any type that is unknown to the domain model. This includes primitive and types that do not belong to the domain. Services of this kind do not contain any business rules nor domain logic. They simply exist to coordinate, orchestrate and execute operations that belong to the domain model.
Domain	Operate only on types belonging to the domain. They contain meaningful concepts that can be found within the domains ubiquitous language.
Infrastructure	Operations that fulfil infrastructure concerns, such as sending emails, logging meaningful data.

5.2 Application Services

Application services are the middleware between the outside world and the domain logic. The purpose of such a mechanism is to transform commands from the outside world into meaningful domain instructions.

Let's consider the *User signs in into our platform* use case. Starting with an outside-in approach, from the delivery mechanism we need to compose the input request for our domain operation. Using a framework like Symfony 2 as the delivery mechanism the code would be something like

```
class SignInController extends Controller
{
    public function signInAction(Request $request)
    {
        $signInService = new SignInUserService($this->get('user_repository'));

        try {
            $response = $signInService->execute(new SignInUserRequest(
                $request->request->get('email'),
                $request->request->get('password')
            ));
        } catch(UserAlreadyExistsException $e) {
            $this->render('error.html.twig', $response);
        }

        return $this->render('success.html.twig', $response);
    }
}
```

On the domain side, the Application Service that coordinates the logic that fulfils the *User signs in* use case

```
class SignInUserService implements Service
{
    private $userRepository;

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

    public function execute(SignInUserRequest $request)
    {
        $user = $this->userRepository->userOfEmail($request->email);
        if ($user) {
            throw new UserAlreadyExistsException();
        }

        $user = new User(
            $this->userRepository->nextIdentity(),
            $request->email,
            $request->password
```

```
        );

        $this->userRepository->persist($user);

        return new SignInUserResponse($user);
    }
}
```

As following the same contract for every service is convenient (will see that later), the communication between the delivery mechanism and the domain is carried by data structures called DTOs (Data Transfer Objects).

```
class SignInUserRequest
{
    public $email;
    public $password;

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

class SignInUserResponse
{
    public $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }
}
```

5.2.1 Transactions

In Domain-Driven Design, transactions are handled at the Application Service level. You are not going to find any `beginTransaction` or similar anywhere in your Domain code. All the operations performed during the execution of the Application Service are going to be run atomically against your database.

In PHP, we have an elegant solution in order to make your Application Services be executed in a transactional scope without worrying about transactions in your Domain code.

You can possibly think that persisting a new user is just going to run a single insert on a table, but what if we have more than one table storing different information about the users?

```
interface Service
{
    public function execute($request);
}

class TransactionalService
{
    private $session;
    private $service;

    public function __construct(Service $service, TransactionalSession $session)
    {
        $this->session = $session;
        $this->service = $service;
    }

    public function execute($request)
    {
        if (empty($this->service)) {
            throw new \LogicException('A use case must be specified');
        }

        $operation = function () use ($request) {
            return $this->service->execute($request);
        };

        return $this->session->executeAtomically($operation->bindTo($this));
    }
}

interface TransactionalSession
{
    /**
     * @return mixed
     */
    public function executeAtomically(callable $operation);
}
```

5.2.2 Testing Application Services

How could we test this Application Service? As Kent Beck suggests in "TDD by Examples", test everything that could possibly break. That means the happy and the sad paths. Happy path is the one that does the job right when all the input is valid and all the resources are available. In the registration use case, the email and password are valid, the user does not exist already, the database is available, etc. The sad path is every other case.

```
class SignInUserServiceTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @test
     * @expectedException \Ddd\Domain\Model\User\UserAlreadyExistsException
     */
    public function alreadyExistingEmailShouldThrowAnException()
    {
        $service = new SignInUserService(new UserRepository());
        $service->execute('user@example.com', 'password');
        $service->execute('user@example.com', 'password');
    }

    /**
     * @test
     */
    public function afterUserSignUpItShouldBeInTheRepository()
    {
        $userRepository = new UserRepository();
        $service = new SignInUserService($userRepository);
        $user = $service->execute('user@example.com', 'password');

        $this->assertSame(
            $user,
            $userRepository->userOfId($user->id())
        );
    }
}
```

5.3 Domain Services

Throughout conversations with domain experts, you will come across concepts in the *Ubiquitous Language* that cannot be neatly represented as either an *Entity* or *Value*.

- A user being able to sign-in to a system by themselves?
- A cart being able to be promoted to an order by itself?

The examples above are two concrete concepts which can not naturally be bound to either an Entity or a Value Object. Further highlighting this oddity, we can attempt to model the behavior as follows

```
class User
{
    public function signIn($aUsername, $aPassword)
    {
        // ...
    }
}


class Cart
{
    public function createOrder()
    {
        // ...
    }
}
```

In the case of the first implementation, we are not able to know that the given username and password relate to the invoked-upon user instance. Clearly this operation does not suit this Entity, instead it should be extracted out into a separate class, making its intention explicit.

With this thought in mind we could create a domain service with the sole responsibility to authenticate users.

```
class SignIn
{
    public function execute($aUsername, $aPassword)
    {
        // ...
    }
}
```

Or similarly, in the case of the second example, a domain service specialised in creating orders from a supplied cart.

```
class CreateOrderFromCart
{
    public function execute(Cart $aCart)
    {
        // ...
    }
}
```

A domain service can be defined as an operation that fulfills a domain task and naturally does not fit into either an Entity nor a Value Object. As a concept that represents an operation in the domain, they should be used by clients regardless of its run history. Domain services don't hold any kind of state by themselves, so **domain services are stateless operations**.

5.4 Domain Services With Multiple Implementations

It is common to encounter infrastructural dependencies when modeling a domain service. For example, in the case were an authentication mechanism which handles password encryption is required. In this instance you could use a *Separated Interface*[1], allowing for multiple encryption mechanisms to be defined. Using this pattern still provides you with a clear separation of concerns between the domain and the infrastructure.

```
interface SignIn
{
    public function execute($aUsername, $aPassword);
}
```

Using the above interface found in the domain, we could create an implementation in the infrastructure like follows

```
class DefaultEncryptionSignIn implements SignIn
{
    private $userRepository;

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

    public function execute($aUsername, $aPassword)
```

[1]http://martinfowler.com/eaaCatalog/separatedInterface.html

```
    {
        if (!$this->userRepository->has($aUsername)) {
            throw \InvalidArgumentException(
                sprintf('The user "%s" does not exist.', $aUsername)
            );
        }

        $aUser = $this->userRepository->byUsername($aUsername);

        if (!$this->isPasswordValidForUser($aUser, $aPassword)) {
            throw new BadCredentialsException($aUser, $aPassword);
        }

        return $aUser;
    }

    private function isPasswordValidForUser(
        User $aUser,
        $anUnencryptedPassword
    ) {
        return password_verify($anUnencryptedPassword, $aUser->hash());
    }
}
```

Another implementation based instead on the MD5 strategy

```
class Md5EncryptionSignIn implements SignIn
{
    const SALT = 'S0m3S41T';

    private $userRepository;

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

    public function execute($aUsername, $aPassword)
    {
        if (!$this->userRepository->has($aUsername)) {
            throw new InvalidArgumentException(
                sprintf('The user "%s" does not exist.', $aUsername)
```

```
        );
    }

    $aUser = $this->userRepository->byUsername($aUsername);

    if ($this->isPasswordInvalidFor($aUser, $aPassword)) {
        throw new BadCredentialsException($aUser, $aPassword);
    }

    return $aUser;
}

private function salt()
{
    return md5(self::SALT);
}

private function isPasswordInvalidFor(
    User $aUser,
    $anUnencryptedPassword
) {
    $encryptedPassword = md5($anUnencryptedPassword . '_' . $this->salt());
    return $aUser->hash() !== $encryptedPassword;
}
}
```

Opting for this choice allows us to have multiple implementations of the domain service interface, each responsible for handling different encryption mechanisms. Deciding on the implementation to use can be easily managed through an inversion of control container, such as Symfony's Dependency Injection component, for example

```
<?xml version="1.0"?>
<container
    xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services
        http://symfony.com/schema/dic/services/services-1.0.xsd">

    <services>

        <service id="sign_in"
                 alias="sign_in.default" />
```

```
        <service id="sign_in.default"
                 class="Ddd\Auth\Infrastructure\Authentication\DefaultEncryption\
SignIn">
            <argument type="service" id="user_repository" />
        </service>

        <service id="sign_in.md5"
                 class="Ddd\Auth\Infrastructure\Authentication\Md5EncryptionSign\
In">
            <argument type="service" id="user_repository" />
        </service>

    </services>
</container>
```

If in the future we wish to handle a new type of encryption, we can simply start by implementing the domain service interface. Then it is a matter of declaring the service in the inversion of control container and replacing the service alias dependency with the newly created one.

5.4.1 An Issue on Code Reuse

Although the implementation described above clearly defines the separation of concerns, we are required to repeat the password verification algorithm **every time we wish to implement a new encryption mechanism**.

An alternative means of solving this problem, aiding code reuse, is by separating out these two responsibilities. We could instead extract the password encryption logic out into a specialised class, using the Strategy Pattern[2] for all defined encryption algorithms. This leaves the design open for extension and closed for modification.

```
class SignIn
{
    private $userRepository;
    private $passwordEncryption;

    public function __construct(
        UserRepository $userRepository,
        PasswordEncryption $passwordEncryption
    ) {
        $this->userRepository = $userRepository;
```

[2] http://en.wikipedia.org/wiki/Strategy_pattern

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

    public function execute($aUsername, $aPassword)
    {
        if (!$this->userRepository->has($aUsername)) {
            throw new InvalidArgumentException(
                sprintf('The user "%s" does not exist.', $aUsername)
            );
        }

        $aUser = $this->userRepository->byUsername($aUsername);

        if ($this->isPasswordInvalidFor($aUser, $aPassword)) {
            throw new BadCredentialsException($aUser, $aPassword);
        }

        return $aUser;
    }

    private function isPasswordInvalidFor(User $aUser, $plainPassword)
    {
        return !$this->passwordEncryption->verify(
            $plainPassword,
            $aUser->hash()
        );
    }
}

interface PasswordEncryption
{
    /**
     * @param string $password
     * @param string $hash
     * @return boolean
     */
    public function verify($plainPassword, $hash);
}
```

Defining different encryption strategies is as easy as implementing the `PasswordEncryption` interface.

```
class BasicPasswordEncryption implements PasswordEncryption
{
    public function verify($plainPassword, $hash)
    {
        return password_verify($plainPassword, $hash);
    }
}

class Md5PasswordEncryption implements PasswordEncryption
{
    const SALT = 'S0m3S41T';

    public function verify($plainPassword, $hash)
    {
        return $hash === $this->calculateHash($plainPassword);
    }

    private function calculateHash($plainPassword)
    {
        return md5($plainPassword . '_' . $this->salt());
    }

    private function salt()
    {
        return md5(self::SALT);
    }
}
```

5.5 Testing Domain Services

Given the user authentication example from *multiple domain service implementations*, it is extremely beneficial to be able to easily test the service. Typically however, testing Template Method implementations can be tricky, as a result we will be using a plain password encryption implementation for testing purposes.

```
class PlainPasswordEncryption implements PasswordEncryption
{
    public function verify($plainPassword, $hash)
    {
        return $plainPassword === $hash;
    }
}
```

Now we can test all cases in the domain service

```
class SignInTest extends PHPUnit_Framework_TestCase
{
    private $signIn;
    private $userRepository;

    protected function setUp()
    {
        $this->userRepository = new InMemoryUserRepository();
        $this->signIn = new SignIn(
            $this->userRepository,
            new PlainPasswordEncryption()
        );
    }

    /**
     * @test
     * @expectedException InvalidArgumentException
     */
    public function itShouldComplainIfTheUserDoesNotExist()
    {
        $this->signIn->execute('test-username', 'test-password');
    }

    /**
     * @test
     * @expectedException BadCredentialsException
     */
    public function itShouldTellIfTheUserIsFoundButThePasswordDoesNotMatch()
    {
        $this->userRepository->add(
            new User(
                'test-username',
```

```
                'test-password'
            )
        );

        $this->signIn->execute('test-username', 'no-matching-password')
    }

    /**
     * @test
     */
    public function itShouldTellIfTheUserIsFoundAndMatchesTheProvidedPassword()
    {
        $this->userRepository->add(
            new User(
                'test-username',
                'test-password'
            )
        );

        $this->assertInstanceOf(
            'Ddd\Domain\Model\User\User',
            $this->signIn->execute('test-username', 'test-password')
        );
    }
}
```

5.6 Anemic Domain Models vs Rich Domain Models

Caution must be had to not overuse domain service abstractions within your system. Following this path can lead to entities and value objects stripped of all behaviour, becoming mere data containers. This is contrary to the goal of OOP, which can be thought of as the gathering of both data and behaviour into semantic units called objects. Their intent being to express real-world concepts and problems. This can be considered an anti-pattern and is referenced to as the *Anemic domain model*.

Typically when starting a new project or feature, it is easy to fall into the trap of modeling the data first. This commonly includes thinking that each database table has a direct one-to-one object form representation. This thinking may or may not however be the exact case all the time.

Suppose we are task with modeling an order processing system. If we do start by modeling the data first, we could end up with an SQL script like so

```
CREATE TABLE `orders` (
`ID` INTEGER NOT NULL AUTO_INCREMENT,
`CUSTOMER_ID` INTEGER NOT NULL,
`AMOUNT` DECIMAL(17, 2) NOT NULL DEFAULT '0.00',
`STATUS` TINYINT NOT NULL DEFAULT 0,
`CREATED_AT` DATETIME NOT NULL,
`UPDATED_AT` DATETIME NOT NULL,
PRIMARY KEY (`ID`)
) ENGINE=INNODB DEFAULT CHARSET=utf8 COLLATION;
```

From this, it is relatively easy to create an `Order` class representation. This representation includes the required accessor methods, used to set/get data from and to the underlying *orders* database table.

```
class Order
{
    const STATUS_CREATED    = 10;
    const STATUS_ACCEPTED   = 20;
    const STATUS_PAID       = 30;
    const STATUS_PROCESSED  = 40;

    private $id;
    private $customerId;
    private $amount;
    private $status;
    private $createdAt;
    private $updatedAt;

    public function __construct(
        $customerId,
        $amount,
        $status,
        DateTimeInterface $createdAt,
        DateTimeInterface $updatedAt
    ) {
        $this->customerId = $customerId;
        $this->amount = $amount;
        $this->status = $status;
        $this->createdAt = $createdAt;
        $this->updatedAt = $updatedAt;
    }

    public function setId($id)
```

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

    public function getId()
    {
        return $this->id;
    }

    public function setCustomerId($customerId)
    {
        $this->customerId = $customerId;
    }

    public function getCustomerId()
    {
        return $this->customerId;
    }

    public function setAmount($amount)
    {
        $this->amount = $amount;
    }

    public function getAmount()
    {
        return $this->amount;
    }

    public function setStatus($status)
    {
        $this->status = $status;
    }

    public function getStatus()
    {
        return $this->status;
    }

    public function setCreatedAt(DateTimeInterface $createdAt)
    {
        $this->createdAt = $createdAt;
```

```
    }

    public function getCreatedAt()
    {
        return $this->createdAt;
    }

    public function setUpdatedAt(DateTimeInterface $updatedAt)
    {
        $this->updatedAt = $updatedAt;
    }

    public function getUpdatedAt()
    {
        return $this->updatedAt;
    }
}
```

An example use-case for this implementation could be to update the order status, as follows

```
// Fetch an order from the database
$anOrder = $orderRepository->find(1);

// Update order status
$anOrder->setStatus(Order::STATUS_ACCEPTED);

// Update updatedAt field
$anOrder->setUpdatedAt(new DateTimeImmutable());

// Save the order to the database
$orderRepository->save($anOrder);
```

This code has a similar problem to the initial user authentication solution, in regard to code reuse. To resolve this issue, defenders of such practice suggest the use of a Service Layer[3], making the operations explicit and reusable. This above implementation could now instead be encapsulated into a separate class.

[3]http://martinfowler.com/eaaCatalog/serviceLayer.html

```
class ChangeOrderStatusService
{
    private $orderRepository;

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

    public function execute($anOrderId, $anOrderStatus)
    {
        // Fetch an order from the database
        $anOrder = $this->orderRepository->find($anOrderId);

        // Update order status
        $anOrder->setStatus($anOrderStatus);

        // Update updatedAt field
        $anOrder->setUpdatedAt(new DateTimeImmutable());

        // Save the order to the database
        $this->orderRepository->save($anOrder);
    }
}
```

Or in the case of updating an order amount

```
class UpdateOrderAmountService
{
    private $orderRepository;

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

    public function execute($orderId, $amount)
    {
        $anOrder = $this->orderRepository->find(1);

        $anOrder->setAmount($amount);
        $anOrder->setUpdatedAt(new DateTimeImmutable());
```

```
        $this->orderRepository->save($anOrder);
    }
}
```

The client code would be drastically decreased into following clearly intentioned operation.

```
$updateOrderAmountService = new UpdateOrderAmountService(
    $orderRepository
);

$updateOrderAmountService->execute(1, 20.5);
```

Implementing this approach can result in a large degree of code re-usability. Someone who wishes to update the order amount simply only has to retrieve an instance of the *UpdateOrderAmountService* and invoke the *execute* method with the appropriate parameters.

However, choosing this path breaks the discussed object-oriented design principles, and incurs the costs of building a domain model without taking advantage of any of the benefits.

5.6.1 Anemic Domain Model Breaks Encapsulation

If we re-look at the code used to define the services within our *Service Layer*, we can see that as a client making use of the *Order* entity, we are **required to know every detail of its internal representation**. This finding goes against the fundamental rule of object-oriented programming, combining data with subsequent behaviour.

5.6.2 Anemic Domain Model Brings a False Sense of Code Reuse

Say there is an instance were a client bypasses the `UpdateOrderAmountService` and instead fetches, updates and persists directly to the OrderRepository. If the `UpdateOrderAmountService` included any other relevant business logic regarding the order amount, it would not have be executed. This could lead to the order being stored in an inconsistent state. As such, invariants should be correctly guarded, and the best way to do this is to let the true domain model handle it. In the case of this example the `Order` entity would be the best place to ensure this.

```
class Order
{
    // ...

    public function changeAmount($amount)
    {
        $this->amount = $amount;
        $this->setUpdatedAt(new DateTimeImmutable());
    }
}
```

Note that by pushing this action down into the entity and naming it in terms of the *Ubiquitous Language*, the system achieves great code reuse. Anyone who now wishes to change the amount of the order has to invoke the `Order::changeAmount` method directly.

This leads to far richer classes, were behaviour is the ideal direction to aim for resulting code reuse. This is commonly referred to as a rich domain model.

5.6.3 How to Avoid Anemic Domain Model?

The way to avoid falling into an anemic domain model is to instead when starting a new project or feature, to think of the behaviour first. Databases, ORMs, and so on are just implementation details, and we should strive to push the decision to use these tools as late in the development process as we can. In doing this we can focus on the one true attribute that matters, the behaviour.

5.7 Wrap-up

As we've seen, Services represent operations inside our system. We can differentiate between:

- **Application Services**: Help coordinate requests from the outside world into the domain. These Services should not contain domain logic. Transactions are handled in the application level, wrapping your services inside Transactional decorators will make your code transaction-agnostic.
- **Domain Services**: Operate with domain concepts only, those expressed by the Ubiquitous Language. Remember to postpone implementation details and think in behaviour first, Domain Services abuse will lead to anaemic domain models and bad Object-Oriented Design.
- **Infrastructure Services**: Operate over infrastructure like sending emails or logging information.

6. Domain Events

Software events are something happened that none, one or more components care about. PHP developers are not generally used to work with events. It's not a feature in the language. However, it's more common to see how new frameworks and libraries embrace them to provide new ways of decoupling, reusing and speeding up code.

Domain Events are events related to Domain changes. Domain Events are things that happen in our Domain that domain experts care about.

6.1 Introduction

Think about a JavaScript 2D platform game. There are tons of different components interacting with each other on the screen at the same time. There is a component that indicates the number of lives remaining, another one that shows all the points scored, or another one counting down the time remaining to finish the current level. Each time the player jumps on an enemy your points scored get increased. When your scoring goes higher than a certain number of points, you get an extra life. When a player collides against a key, it gets captured and probably a door opens. How all these components interact with each other? What's the optimal architecture for this scenario?

There are probably two main options: the first one is to couple each component with the ones it is connected to. In the example, the player components would be coupled with too many other components probably. When a new component is added to the game, the developer needs to modify the code of the first one. Do you remember the Open-Closed principle[1]? Adding a new component shouldn't make the first component to be updated. What would happen with too many components? Is it easy to maintain? Not at all.

The second approach is connecting all the components to a single object that handles all the important events in the game. It receives events from each component and it forwards them to specific components. For example, the scoring component would be interested in an `EnemyKilled` event, or the `LifeCaptured` event is quite useful for the player entity and the remaining lives component. In this way, all components are coupled to a single component that manages all the notifications. With this approach, adding new components or removing existing ones do not affect the remaining ones.

While developing a single application, events come handy to decoupling components. When developing a whole domain in a distributed way, events are very useful to decouple each service or application that plays a role in the domain. The key points are the same but at a different scale.

[1] http://en.wikipedia.org/wiki/Open/closed_principle

6.2 Definition

Domain Events are one specific type of event used for notifying Domain changes to local or remote Bounded Contexts.

Vaughn Vernon defines[2] a Domain Event as

> as an occurrence capture of something what happened in the domain.

Eric Evans defines[3] a Domain Event as

> a full-fledged part of the domain model, a representation of something that happened in the domain. Ignore irrelevant domain activity while making explicit the events that the domain experts want to track or be notified of, or which are associated with state change in the other model objects.

Martin Fowler defines[4] a Domain Event as

> captures the memory of something interesting which affects the domain.

Examples of Domain Events in a Web application are `UserRegistered`, `OrderPlaced`, `UserRelocated` or `ProductAdded`.

6.2.1 Short story

In a Ticket Sales Agency, a content manager decides to increase the price of a U2 show. Using her back-office, edits the show. A `ShowPriceChanged` Domain Event is published and persisted in the same transaction with the new show price into the database.

A batch process takes the Domain Event and queues it into RabbitMQ. The Domain Event gets distributed in two queues, one for the same local Bounded Context and another remote one for Business Intelligence purposes.

In the first queue, a worker fetches the corresponding `Show` using the id in the event and push it into an Elasticsearch server, so the user can see the new price when searching. It could also update the new price in a different database table.

In the second queue, a worker inserts the info into a Logs Server or a Data Lake where reporting or Data Mining processes can be run.

[2] http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8
[3] https://domainlanguage.com/ddd/patterns/DDD_Reference_2011-01-31.pdf
[4] http://martinfowler.com/eaaDev/DomainEvent.html

An external application that cannot be integrated using Domain Events, could access to all the `ShowPriceChanged` events using a REST API that the local Bounded Context provides.

As you can see, Domain Events are useful for dealing with eventual consistency and integrating different Bounded Contexts. Aggregates create Events and publish them. Subscribers may store Events and then forward them to remote subscribers.

6.2.2 Metaphor

I go to Babur's for a meal on Tuesday, and pay by credit card. This might be modelled as an event, whose event type is `PurchasePlaced`, whose subject is my credit card, and whose occurred date is Tuesday. If Babur's uses an old manual system and doesn't transmit the transaction until Friday, the noticed date would be Friday.

Things happen. Not all of them are interesting, some may be worth recording but don't provoke a reaction. The most interesting ones cause a reaction. Many systems need to react to interesting events. Often you need to know why a system reacts in the way it did.

By funneling inputs to a system into streams of Domain Events you can keep a record of all the inputs to a system. This helps you organize your processing logic, and also allows you to keep an audit log of the inputs to the system.

Exercise

Try to locate examples of potential Domain Events in your current Domain.

6.2.3 RealLife(tm) Example

Before going into the detail about Domain Events, let's see a real example about using Domain Events and how they can help us in our application and our whole Domain.

Let's consider a simple Application Service that will register a new user. For example, in an e-commerce context. Application Services will be explained in its own chapter, so don't worry too much about its interface, just focus on the `execute` method.

```
class SignInUserService implements ApplicationService
{
    private $userRepository;
    private $userFactory;
    private $userTransformer;

    public function __construct(
        UserRepository $userRepository,
        UserFactory $userFactory,
        UserTransformer $userTransformer
    )
    {
        $this->userRepository = $userRepository;
        $this->userFactory = $userFactory;
        $this->userTransformer = $userTransformer;
    }

    /**
     * @param SignInUserRequest $request
     * @return User
     * @throws UserAlreadyExistsException
     */
    public function execute($request = null)
    {
        $email = $request->email();
        $password = $request->password();

        $user = $this->userRepository->userOfEmail($email);
        if (null !== $user) {
            throw new UserAlreadyExistsException();
        }

        $user = $this->userFactory->build(
            $this->userRepository->nextIdentity(),
            $email,
            $password
        );

        $this->userRepository->add($user);
        $this->userTransformer->write($user);
    }
}
```

As shown, the Application Service checks if the user already exists. If not, it creates a new `User` and adds it to the `UserRepository`.

Consider now a new requirement. A new user must be notified by email when registered. Without thinking too much, the first approach coming to mind is updating our Application Service to include a piece of code that would do the job. Probably some sort of `EmailSender` that would be run after the `add` method. However, let's consider another approach.

What about firing a `UserRegistered` event so another component listening to such sort of event can react and send that email? There are some cool benefits about this new approach. First of all, we don't need to update the code of our Application Service every time a new action must be performed when a new user is registered.

Second, it's easier to test. The Application Service remains simpler and each time a new action is developed, we just write the tests for the action.

Later in the same e-commerce project, we are told to integrate an open-source gamification platform not written in PHP. Each time a user places a purchase or reviews a product in our e-commerce Bounded Context, she can get badges that can be shown in the e-commerce user profile page or be notified by email. How could we model the problem?

Following the first approach, we would update the Application Service to integrate with the new platform having a similar situation as the confirmation email feature. With the `DomainEvent` approach, we would create another listener for the `UserRegistered` event that will connect directly, by REST or SOA, to the gamification platform or even better, would spread it through some messaging system, such as RabbitMQ, so that event can be received by the gamification platform and react accordingly so our e-commerce BC doesn't know anything about our new gamification BC.

6.3 Characteristics

Domain events are ordinarily **immutable**, as they are a record of something in the past. In addition to a description of the event, a domain event typically contains a timestamp for the time the event occurred and the identity of entities involved in the event. Also, a domain event often has a separate timestamp indicating when the event was entered into the system and the identity of the person who entered it. When useful, an identity for the domain event can be based on some set of these properties. So, for example, if two instances of the same event arrive at a node they can be recognized as the same.

The essence of a Domain Event is that you use it to capture things that can trigger a change to the state of the application you are developing or to another applications in your Domain interested in those changes. These event objects are then processed to cause changes to the system, and stored to provide an audit log.

6.3.1 Naming Conventions

All events should be represented as verbs in the past tense such as `CustomerRelocated`, `CargoShipped`, or `InventoryLossageRecorded`. They are things that have completed in the past. There are interesting examples in the English language where one may be tempted to use nouns as opposed to verbs in the past tense, an example of this would be "Earthquake" or "Capsize", as a congressman recently worried about Guam. We suggest to avoid the temptation of using names like those for Domain Events and stick with the usage of verbs in the past tense. Nouns tend to match up with "Transaction Objects" discussed later from Streamlined Object Modeling. It's imperative to model events as past tense verbs as they are part of the Ubiquitous Language.

6.3.2 Domain Events and Ubiquitous Language

Consider the differences in the Ubiquitous Language when we discuss the side effects from relocating a customer, the event makes the concept explicit where as previously the changes that would occur within an aggregate or between multiple aggregates were left as an implicit concept that needed to be explored and defined. As an example, in most systems the fact that a side effect occurred is simply found by a tool such as Hibernate or Entity Framework, if there is a change to the side effects of a use case, it is an implicit concept. The introduction of the event makes the concept explicit and part of the Ubiquitous Language; relocating a customer does not just change some stuff, relocating a customer produces a CustomerRelocatedEvent which is explicitly defined within the language.

6.3.3 Immutability

Domain Events describe changes in your Domain that have already happened. They talk about the past. By definition, it's impossible to change the past, except if you are *Marty McFly* and have a *Delorean*, but that might be not the case. Domain Events are immutable, that's it.

> **Symfony Event Dispatcher**
>
> Some PHP frameworks support events. However, don't confuse those events with Domain Events. They are different in characteristics and goals. For example, Symfony has the Event Dispatcher component. If you need to implement an event system for a state machine, for example, you can rely on it. The whole request to response Symfony trip is based in events too. However, Symfony Events are mutable, each of the listeners are capable of modifying the event to add or update the information in it.

6.4 Modeling Events

When modeling Events, name them and their properties according to the Ubiquitous Language in the Bounded Context where they originate. If an Event is the result of executing a command operation on an Aggregate, the name is usually derived from the command that was executed. It is important that the Event name reflects the past nature of the occurrence. It is not occurring now. It occurred previously. The best name to choose is the one that reflects that fact.

Let's consider our user registration feature and the `DomainEvent` needs to represent that fact. The following code shows a minimal interface for a base `DomainEvent`.

```
interface DomainEvent
{
    /**
     * @return \DateTime
     */
    public function occurredOn();
}
```

As seen, the minimum information required is a `DateTime` in order to know when the event happened.

Let's model now the new user registration event. The following code could be used in order to model an event representing the fact that a new user has been registered in our application. As explained before, the name should be a verb in the past tense, so `UserRegistered` is probably a good choice.

```
class UserRegistered implements DomainEvent
{
    private $userId;

    public function __construct(UserId $userId)
    {
        $this->userId = $userId;
        $this->occurredOn = new \DateTime();
    }

    public function userId()
    {
        return $this->userId;
    }

    public function occurredOn()
    {
        return $this->occurredOn;
    }
}
```

The minimum amount of information to notify about a new user is possibly her `UserId`. With this information, any process, command or application service, from the same Bounded Context or a different one, may act to this event.

As rule of thumb:

- `DomainEvents` are usually designed as immutable
- Constructor will initialize the full state of the `DomainEvent`
- `DomainEvents` will have getters to access its attributes
- Include the identity of the Aggregate that performs the action
- Include other Aggregate identities related with the first one
- Include parameters that caused the Event if useful

But, what happens if your Domain experts from the same BC or a different one needs more information? Let's see the same Domain Event modeled with more information, for example, the email address.

```
class UserRegistered implements DomainEvent
{
    private $userId;
    private $userEmail;

    public function __construct(UserId $userId, $userEmail)
    {
        $this->userId = $userId;
        $this->userEmail = $userEmail;
        $this->occurredOn = new \DateTime();
    }

    public function userId()
    {
        return $this->userId;
    }

    public function userEmail()
    {
        return $this->userEmail;
    }

    public function occurredOn()
```

```
    {
        return $this->occurredOn;
    }
}
```

We have added the email address. Adding more information to a `DomainEvent` can help to improve performance or simplify the integration between different Bounded Contexts. Thinking in other Bounded Context point of view could help modeling events. When a new user is created in the upstream Bounded Context, the downstream one would have to create its own user. Adding the user email, could possibly save a sync request to the upstream Bounded Context in the case the downstream one needs it. Let's see an example.

Do you remember the gamification example? In order to create the users of the gamification platform, probably called `Player`, the `UserId` from the e-commerce Bounded Context is probably enough. But, what happens if the gamification platform has to notify the users by email about being rewarded? In this case, the email address is also mandatory. So, if in the original Domain Event, the email address is included we are done. If that?s not the case, the gamification Bounded Context needs to request such information from the e-commerce one via REST or SOA integration.

Why not the whole User Entity?

Should I include the whole User Entity from my Bounded Context in the Domain Event? Our suggestion, don't. Domain Events are used to communicate a Bounded Context with itself and other Bounded Contexts. That means, what can be a `Seller` in a C2C e-commerce product catalog Bounded Context, can be an `Author` of a product review in a product feedback one. Both can share the same id or email, but `Seller` and `Author` are different concepts represented different entities from different Bounded Contexts. So, Entities from one Bounded Context have no meaning or a totally different one in the others.

6.5 Persisting Domain Events

Persisting events is always a good idea. Some readers may be thinking why not publishing Domain Events to a messaging or logging system directly, but persisting them has interesting benefits:

- You can expose your Domain Events for other BC in a REST way
- You can persist the Domain Event and the Aggregate changes in the same Database transaction before pushing it to RabbitMQ (You don?t want to notify about something that did not happen. You don?t want to miss a notification about something that did happen)
- Business Intelligence can use this data to analyse, forecast or trend
- Audit your entity changes
- For Event Sourcing, you can reconstitute Aggregates from Domain Events

6.5.1 Event Store

Where do we persist Domain Events? In an Event Store. An Event Store is a `DomainEvent` repository that lives in our Domain space as an abstraction (interface or abstract class) its responsibility is to append Domain Events and query them. A possible basic interface could be:

```
interface EventStore
{
    public function append(DomainEvent $aDomainEvent);
    public function allStoredEventsSince($anEventId);
}
```

However, depending on the usage of your `DomainEvents`, the previous interface can have more methods to query your events.

In terms of implementation, you can decide to use a Doctrine repository, a DBAL one or a plain PDO. Because `DomainEvents` are immutable using a Doctrine one adds an unnecessary performance penalty. For a small to medium application, probably Doctrine is ok. Let's see a possible implementation with Doctrine.

```
class DoctrineEventStore extends EntityRepository implements EventStore
{
    private $serializer;

    public function append(DomainEvent $aDomainEvent)
    {
        $storedEvent = new StoredEvent(
            get_class($aDomainEvent),
            $aDomainEvent->occurredOn(),
            $this->serializer()->serialize($aDomainEvent, 'json')
        );

        $this->getEntityManager()->persist($storedEvent);
    }

    public function allStoredEventsSince($anEventId)
    {
        $query = $this->createQueryBuilder('e');
        if ($anEventId) {
            $query->where('e.eventId > :eventId');
            $query->setParameters(array('eventId' => $anEventId));
        }
        $query->orderBy('e.eventId');
```

```
        return $query->getQuery()->getResult();
    }

    private function serializer()
    {
        if (null === $this->serializer) {
            /** \JMS\Serializer\Serializer\SerializerBuilder */
            $this->serializer = SerializerBuilder::create()->build();
        }

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

`StoredEvent` is the Doctrine Entity needed to map with the database. As you may have seen, when appending and after persisting the `Store`, there is no `flush` call. If this operation is inside a Doctrine transaction, this is not needed. So, let's leave it without the call and we'll go into more details when talking about Application Services. Let's see the `StoredEvent` implementation.

```
class StoredEvent implements DomainEvent
{
    private $eventId;
    private $eventBody;
    private $occurredOn;
    private $typeName;

    /**
     * @param string $aTypeName
     * @param \DateTime $anOccurredOn
     * @param string $anEventBody
     */
    public function __construct($aTypeName, \DateTime $anOccurredOn, $anEventBod\
y)
    {
        $this->eventBody = $anEventBody;
        $this->typeName = $aTypeName;
        $this->occurredOn = $anOccurredOn;
    }

    public function eventBody()
    {
```

```
        return $this->eventBody;
    }

    public function eventId()
    {
        return $this->eventId;
    }

    public function typeName()
    {
        return $this->typeName;
    }

    public function occurredOn()
    {
        return $this->occurredOn;
    }
}
```

And its mapping.

```
Ddd\Domain\Event\StoredEvent:
  type: entity
  table: event
  repositoryClass: Ddd\Infrastructure\Application\Notification\DoctrineEventStore
  id:
    eventId:
      type: integer
      column: event_id
      generator:
        strategy: AUTO
  fields:
    eventBody:
      column: event_body
      type: text
    typeName:
      column: type_name
      type: string
      length: 255
    occurredOn:
      column: occurred_on
      type: datetime
```

Because every `DomainEvent` may have different fields, we need to persist them serialized. `typeName` identifies the `DomainEvent` domain-wide. An Entity or Value Object has sense inside a BC but `DomainEvents` define a communication protocol between BC.

In distributed systems, s*** happens. You will have to deal with `DomainEvents` that are not published, lost somewhere in the chain or `DomainEvents` that are published more than once. That's why it's important to persist a `DomainEvent` with an id, so it's easy to track which `DomainEvents` have been published and which are the missing ones.

6.6 Publishing Events from the Domain Model

Domain Events should be published when the fact they represent happens. For instance, when a new user has been registered, a new `UserRegistered` event should be published.

Following the newspaper metaphor:

- **Modeling** a Domain Event is like writing a news article
- **Publishing** a Domain Event is like printing the article in the paper
- **Spreading** a Domain Event is like sending the newspaper so everyone can read the article

The recommended approach for publishing `DomainEvents` is to use a simple Listener-Observer pattern to implement a `DomainEventPublisher`.

6.6.1 Publishing a Domain Event from an Entity

Carry on with the example of a new user that has been registered in our application, let's see how the corresponding Domain Event can be published.

```
class User
{
    protected $userId;
    protected $email;
    protected $password;

    public function __construct(UserId $userId, $email, $password)
    {
        $this->setUserId($userId);
        $this->setEmail($email);
        $this->setPassword($password);

        DomainEventPublisher::instance()->publish(
            new UserRegistered(
```

```
                $this->userId
            )
        );
    }

    //...
}
```

As seen in the example, when the User is created a new `UserRegistered` event is published. It's done in the Entity constructor and not outside because, with this approach, it's easier to keep our Domain consistent, any client that creates a `new User` will publish its corresponding event. On the other hand, this makes it a bit more complex to use an infrastructure that needs to create a User Entity without using its constructor. For example, Doctrine uses `serialize` and `unserialize` technique that recreates an object without calling its constructor, however, if you have to create your own, this is not going to be as easy as in Doctrine.

In general, constructing an object from plain data such as an array is called hydratation. Let's see an easy approach to build a new User fetched from a database. First of all, let's extract the Domain Event publication in a different method applying the Factory Method pattern[5].

> The template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in a method, called template method, which defers some steps to subclasses.

```
class User
{
    protected $userId;
    protected $email;
    protected $password;

    public function __construct(UserId $userId, $email, $password)
    {
        $this->setUserId($userId);
        $this->setEmail($email);
        $this->setPassword($password);
        $this->publishEvent();

    }

    protected function publishEvent()
```

[5] http://en.wikipedia.org/wiki/Template_method_pattern

```
    {
        DomainEventPublisher::instance()->publish(
            new UserRegistered(
                $this->userId
            )
        );
    }

    //...
}
```

Now, let's extend our current `User` with a new infrastructure Entity that will do the job for us. The trick here is make `publishEvent` a no operation so the Domain Event is not published.

```
class CustomOrmUser extends User
{
    protected function publishEvent()
    {

    }

    public static function fromRawData($data)
    {
        return new self(
            new UserId($data['user_id']),
            $data['email'],
            $data['password']
        );
    }
}
```

With this approach, when using self-encapsulation as here, be careful when fetching objects from our persistence engine that are invalid because changes in the Domain rules. Another approach without using the parents constructor at all could be:

```
class CustomOrmUser extends User
{
    public function __construct()
    {
    }

    public static function fromRawData($data)
    {
        $user = new self();
        $user->userId = new UserId($data['user_id']);
        $user->email = $data['email'];
        $user->password = $data['password'];

        return $user;
    }
}
```

With this approach, parent constructor is not called and `User` attributes must be protected. Other alternatives are Reflection, passing flags in the constructor, using some proxy library such as Proxy-Manager[6] or an ORM such as Doctrine.

6.6.2 Publishing your Domain Events from Domain or Application Services

You should struggle to publish Domain Events from deeper in the chain. The nearer inside the Entity or the Value Object, the better. As we have seen in the previous section, sometimes this is not easy but the final result is simpler for the clients. We have seen developers publishing Domain Events from the Application Services or Domain Services. That seems an easier approach to implement but drives to an Anemic-Domain Model in the same way when pushing business logic to Domain Services rather than placing it into your Entities.

6.6.3 How the DomainEventPublisher works

A `DomainEventPublisher` is a Singleton class available from our Bounded Context in order to publish `DomainEvents`. It also has support to attach listeners, `DomainEventSubscriber`, that will be listening for any `DomainEvent` they are interested in. This is not quite different than when subscribing with jQuery to an event using `on` method.

[6] https://packagist.org/packages/ocramius/proxy-manager

```
class DomainEventPublisher
{
    private $subscribers;
    private static $instance = null;

    public static function instance()
    {
        if (null === static::$instance) {
            static::$instance = new static();
        }

        return static::$instance;
    }

    private function __construct()
    {
        $this->subscribers = [];
    }

    public function __clone()
    {
        throw new \BadMethodCallException('Clone is not supported');
    }

    public function subscribe(DomainEventSubscriber $aDomainEventSubscriber)
    {
        $this->subscribers[] = $aDomainEventSubscriber;
    }

    public function publish(DomainEvent $anEvent)
    {
        foreach ($this->subscribers as $aSubscriber) {
            if ($aSubscriber->isSubscribedTo($anEvent)) {
                $aSubscriber->handle($anEvent);
            }
        }
    }
}
```

The method `publish` goes through all the possible subscribers, checking if they are interested in the published Domain Event. If that's the case, the method `handle` of the subscriber is called.

The method `subscribe` adds a new `DomainEventSubscriber` that will be listening to specific Domain

Event types.

```
interface DomainEventSubscriber
{
    /**
     * @param DomainEvent $aDomainEvent
     */
    public function handle($aDomainEvent);

    /**
     * @param DomainEvent $aDomainEvent
     * @return bool
     */
    public function isSubscribedTo($aDomainEvent);
}
```

As we have already discussed, persisting all the Domain Events is a great idea. How can we easily persist all the `DomainEvents` published in our app? Using an specific subscriber for that. Let's create a `DomainEventSubscriber` that will listen to all `DomainEvents`, no matter what type, and persists them using our `EventStore`.

```
class PersistDomainEventSubscriber implements DomainEventSubscriber
{
    private $eventStore;

    public function __construct(EventStore $anEventStore)
    {
        $this->eventStore = $anEventStore;
    }

    public function handle($aDomainEvent)
    {
        $this->eventStore->append($aDomainEvent);
    }

    public function isSubscribedTo($aDomainEvent)
    {
        return true;
    }
}
```

`$eventStore` could be a custom Doctrine repository, as already seen, or any other object capable of persisting `DomainEvent` into a Database.

6.6.4 Setting up DomainEventListeners

Where is the best place to set up the subscribers to the `DomainEventPublisher`? It depends. For global subscribers that affect all the request, probably when building your `DomainEventPublisher`. If some subscribers just affect a specific Application Service, when building the Application Service. Let's see an example using Silex.

In Silex[7], the best way to register a Domain Event Publisher that will persist all Domain Events is using an Application Middleware[8]. A *before* application middleware allows you to tweak the Request before the controller is executed. It's the right place to subscribe the listener responsible for persisting those events to the database that will be send to RabbitMQ later.

```
// ...
$app['em'] = $app->share(function() {
    return (new EntityManagerFactory())->build();
});

$app['event_repository'] = $app->share(function($app) {
    return $app['em']->getRepository('Ddd\Domain\Model\Event\StoredEvent');
});

$app['event_publisher'] = $app->share(function($app) {
    return DomainEventPublisher::instance();
});

$app->before(function (Symfony\Component\HttpFoundation\Request $request) use ($\
app) {
    $app['event_publisher']->subscribe(
        new PersistDomainEventSubscriber(
            $app['event_repository']
        )
    );
});
```

With this setup, each time an Aggregate will publish a `DomainEvent`, it will get persisted into the database. Mission accomplished.

6.6.5 Unit Testing

You know already how to publish `DomainEvents`, but how we can unit test that such publishing happens? How can we really test that `UserRegistered` is really fired? The easiest way we suggest

[7] http://silex.sensiolabs.org/

[8] http://silex.sensiolabs.org/doc/middlewares.html

is to use a specific `EventListener` that will work as an Spy[9] to record if the Domain Event was published. Let's see an example of the `User` entity unit test.

```
use Ddd\Domain\DomainEventPublisher;
use Ddd\Domain\DomainEventSubscriber;

class UserTest extends \PHPUnit_Framework_TestCase
{
    //...

    /**
     * @test
     */
    public function itShouldPublishUserRegisteredEvent()
    {
        $subscriber = new SpySubscriber();
        $id = DomainEventPublisher::instance()->subscribe($subscriber);

        $userId = new UserId();
        new User($userId, 'valid@email.com', 'password');
        DomainEventPublisher::instance()->unsubscribe($id);

        $this->assertUserRegisteredEventPublished($subscriber, $userId);
    }

    private function assertUserRegisteredEventPublished($subscriber, $userId)
    {
        $this->assertInstanceOf('UserRegistered', $subscriber->domainEvent);
        $this->assertTrue($subscriber->domainEvent->userId()->equals($userId));
    }
}

class SpySubscriber implements DomainEventSubscriber
{
    public $domainEvent;

    public function handle($aDomainEvent)
    {
        $this->domainEvent = $aDomainEvent;
    }
```

[9] http://www.martinfowler.com/bliki/TestDouble.html

```
    public function isSubscribedTo($aDomainEvent)
    {
        return true;
    }
}
```

There are some alternatives. You could use a static setter for the `DomainEventPublisher` or use some reflection framework to detect the call. However, we think this approach is more natural. Last but not least, remember to clean up the spy subscription so it won't affect the rest of the unit tests execution.

6.7 Spreading the News to Remote Bounded Contexts

In order to communicate to local or remote Bounded Contexts a set of `DomainEvents`, there are two main strategies non exclusive: Messaging and REST API. The first plans to use a messaging system such as RabbitMQ to transmit them. The second plans to create a REST API for accessing the `DomainEvents` of a specific Bounded Context.

6.7.1 Messaging

With all `DomainEvents` persisted into the database, the only thing remaining to spread the news is pushing them to our favorite messaging system. We personally like RabbitMQ[10], but any other such as ActiveMQ or ZeroMQ will do the job. For integrating with RabbitMQ using PHP, there are not many options, php-amqplib[11] will do the work.

First of all, we need a service capable of sending persisted `DomainEvents` to RabbitMQ. That could be easy, what about querying `EventStore` for all the events and send each one? Not bad, however, we could push the same `DomainEvent` more than once. In general, **we need to minimize the number of `DomainEvents` republished**. If zero times, even better. In order to do that, we need some sort of component to track what `DomainEvents` have been already pushed and what are the remaining ones. Last but not least, once we know what `DomainEvents` we have to push, we send the and keep track of the last one published into our messaging system. Let's see a possible implementation for this service:

[10] https://www.rabbitmq.com

[11] https://packagist.org/packages/videlalvaro/php-amqplib

```
class NotificationService
{
    private $serializer;
    private $eventStore;
    private $publishedMessageTracker;
    private $messageProducer;

    public function __construct(
        EventStore $anEventStore,
        PublishedMessageTracker $aPublishedMessageTracker,
        MessageProducer $aMessageProducer
    )
    {
        $this->eventStore = $anEventStore;
        $this->publishedMessageTracker = $aPublishedMessageTracker;
        $this->messageProducer = $aMessageProducer;
    }

    /**
     * @return int
     */
    public function publishNotifications($exchangeName)
    {
        $publishedMessageTracker = $this->publishedMessageTracker();
        $notifications = $this->listUnpublishedNotifications(
            $publishedMessageTracker->mostRecentPublishedMessageId($exchangeName)
        );

        if (!$notifications) {
            return 0;
        }

        $messageProducer = $this->messageProducer();
        $messageProducer->open($exchangeName);
        try {
            $publishedMessages = 0;
            $lastPublishedNotification = null;
            foreach ($notifications as $notification) {
                $lastPublishedNotification = $this->publish(
                    $exchangeName,
                    $notification,
                    $messageProducer
```

```
                );
                $publishedMessages++;
            }
        } catch(\Exception $e) {
            // Log your error (trigger_error, Monolog, etc.)
        }

        $this->trackMostRecentPublishedMessage(
            $publishedMessageTracker,
            $exchangeName,
            $lastPublishedNotification
        );

        $messageProducer->close($exchangeName);

        return $publishedMessages;
    }

    protected function publishedMessageTracker()
    {
        return $this->publishedMessageTracker;
    }

    /**
     * @return StoredEvent[]
     */
    private function listUnpublishedNotifications($mostRecentPublishedMessageId)
    {
        return $this
            ->eventStore()
            ->allStoredEventsSince($mostRecentPublishedMessageId);
    }

    protected function eventStore()
    {
        return $this->eventStore;
    }

    private function messageProducer()
    {
        return $this->messageProducer;
    }
```

```
    private function publish(
        $exchangeName,
        StoredEvent $notification,
        MessageProducer $messageProducer
    )
    {
        $messageProducer->send(
            $exchangeName,
            $this->serializer()->serialize($notification, 'json'),
            $notification->typeName(),
            $notification->eventId(),
            $notification->occurredOn()
        );

        return $notification;
    }

    private function serializer()
    {
        if (null === $this->serializer) {
            $this->serializer = SerializerBuilder::create()->build();
        }

        return $this->serializer;
    }

    private function trackMostRecentPublishedMessage(
        PublishedMessageTracker $publishedMessageTracker,
        $exchangeName,
        $notification
    )
    {
        $publishedMessageTracker->trackMostRecentPublishedMessage($exchangeName,\
 $notification);
    }
}
```

`NotificationService` depends on three interfaces. We have already seen `EventStore`, responsible for appending and querying about `DomainEvents`. The second one is `PublishedMessageTracker`, responsible for keeping track of pushed messages. The third one is `MessageProducer`, an interface representing our messaging system.

```
interface PublishedMessageTracker
{
    /**
     * @param string $exchangeName
     * @return int
     */
    public function mostRecentPublishedMessageId($exchangeName);

    /**
     * @param string $exchangeName
     * @param StoredEvent $notification
     */
    public function trackMostRecentPublishedMessage($exchangeName, $notification\
);
}
```

`mostRecentPublishedMessageId` method returns the id of last `PublishedMessage`, so the process can start from the next one. `trackMostRecentPublishedMessage` is responsible for tracking what's the last message sent, in order to be able to republish messages in case you need it. `$exchangeName` represents what communication channel we are going to use to send out our `DomainEvents`. Let's see a Doctrine implementation of `PublishedMessageTracker`.

```
class DoctrinePublishedMessageTracker
    extends EntityRepository
    implements PublishedMessageTracker
{
    /**
     * @param $exchangeName
     * @return int
     */
    public function mostRecentPublishedMessageId($exchangeName)
    {
        $messageTracked = $this->findOneByExchangeName($exchangeName);
        if (!$messageTracked) {
            return null;
        }

        return $messageTracked->mostRecentPublishedMessageId();
    }

    /**
     * @param $exchangeName
```

```
     * @param StoredEvent $notification
     */
    public function trackMostRecentPublishedMessage($exchangeName, $notification)
    {
        if (!$notification) {
            return;
        }

        $maxId = $notification->eventId();

        $publishedMessage = $this->findOneByExchangeName($exchangeName);
        if (null === $publishedMessage) {
            $publishedMessage = new PublishedMessage(
                $exchangeName,
                $maxId
            );
        }

        $publishedMessage->updateMostRecentPublishedMessageId($maxId);

        $this->getEntityManager()->persist($publishedMessage);
        $this->getEntityManager()->flush($publishedMessage);
    }
}
```

This code is quite straightforward. The only edge case, we have to consider, is when no `DomainEvent` has been published already.

Why an exchange name?

We'll see this in more detail in the "Integrating Bounded Contexts" chapter. However, when a system is running and a new Bounded Context comes into play, you are interested in resending all the `DomainEvents` to the new BC. So keeping track of the last `DomainEvent` published and channel is interesting.

In order to keep track of published `DomainEvents`, we need an exchange name and a notification id. Check a possible implementation.

```
class PublishedMessage
{
    private $mostRecentPublishedMessageId;
    private $trackerId;
    private $exchangeName;

    /**
     * @param string $exchangeName
     * @param int $aMostRecentPublishedMessageId
     */
    public function __construct($exchangeName, $aMostRecentPublishedMessageId)
    {
        $this->mostRecentPublishedMessageId = $aMostRecentPublishedMessageId;
        $this->exchangeName = $exchangeName;
    }

    public function mostRecentPublishedMessageId()
    {
        return $this->mostRecentPublishedMessageId;
    }

    public function updateMostRecentPublishedMessageId($maxId)
    {
        $this->mostRecentPublishedMessageId = $maxId;
    }

    public function trackerId()
    {
        return $this->trackerId;
    }
}
```

And its corresponding mapping.

```
Ddd\Domain\Event\PublishedMessage:
  type: entity
  table: event_published_message_tracker
  repositoryClass: Ddd\Infrastructure\Application\Notification\DoctrinePublished\
MessageTracker
  id:
    trackerId:
      column: tracker_id
      type: integer
      generator:
        strategy: AUTO
  fields:
    mostRecentPublishedMessageId:
      column: most_recent_published_message_id
      type: bigint
    exchangeName:
      type: string
      column: type_name
```

Let's see now what the `MessageProducer` interface is used for and its implementation details.

```
interface MessageProducer
{
    public function open($exchangeName);

    /**
     * @param $exchangeName
     * @param string $notificationMessage
     * @param string $notificationType
     * @param int $notificationId
     * @param \DateTime $notificationOccurredOn
     * @return
     */
    public function send(
        $exchangeName,
        $notificationMessage,
        $notificationType,
        $notificationId,
        \DateTime $notificationOccurredOn);

    public function close($exchangeName);
}
```

Quite easy. The `open` and `close` methods open and close a connection with the messaging system. `send` takes a message body, message name and message id and sends them to our messaging engine whatever it is. Because we have chosen RabbitMQ, we need to implement the connection and sending process.

```
abstract class RabbitMqMessaging
{
    protected $connection;
    protected $channel;

    public function __construct(AMQPConnection $aConnection)
    {
        $this->connection = $aConnection;
        $this->channel = null;
    }

    private function connect($exchangeName)
    {
        if (null !== $this->channel) {
            return;
        }

        $channel = $this->connection->channel();
        $channel->exchange_declare($exchangeName, 'fanout', false, true, false);
        $channel->queue_declare($exchangeName, false, true, false, false);
        $channel->queue_bind($exchangeName, $exchangeName);

        $this->channel = $channel;
    }

    public function open($exchangeName)
    {

    }

    protected function channel($exchangeName)
    {
        $this->connect($exchangeName);

        return $this->channel;
    }

    public function close($exchangeName)
```

```
    {
        $this->channel->close();
        $this->connection->close();
    }
}


class RabbitMqMessageProducer extends RabbitMqMessaging implements MessageProduc\
er
{
    /**
     * @param $exchangeName
     * @param string $notificationMessage
     * @param string $notificationType
     * @param int $notificationId
     * @param \DateTime $notificationOccurredOn
     */
    public function send(
        $exchangeName,
        $notificationMessage,
        $notificationType,
        $notificationId,
        \DateTime $notificationOccurredOn
    )
    {
        $this->channel($exchangeName)->basic_publish(
            new AMQPMessage(
                $notificationMessage,
                [
                    'type' => $notificationType,
                    'timestamp' => $notificationOccurredOn->getTimestamp(),
                    'message_id' => $notificationId
                ]
            ),
            $exchangeName
        );
    }
}
```

Now that we have a `DomainService` to push `DomainEvents` into a messaging system like RabbitMQ, it's time to execute them. We need to choose a delivery mechanism to run the service. We personally suggest to create a Symfony Console[12] Command.

[12]http://symfony.com/doc/current/components/console/introduction.html

```
class PushNotificationsCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('domain:events:spread')
            ->setDescription('Notify all domain events via messaging')
            ->addArgument(
                'exchange-name',
                InputArgument::OPTIONAL,
                'Exchange name to publish events to',
                'my-bc-app'
            );
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $app = $this->getApplication()->getContainer();

        $numberOfNotifications =
            $app['notification_service']
                ->publishNotifications(
                    $input->getArgument('exchange-name')
                );

        $output->writeln(
            sprintf(
                '<comment>%d</comment> <info>notification(s) sent!</info>',
                $numberOfNotifications
            )
        );
    }
}
```

Following the Silex example, let's see the definition of the `$app['notification_service']` defined in the Silex Pimple Service Container[13].

[13] http://silex.sensiolabs.org/doc/services.html#id1

```
//...
$app['event_store'] = $app->share(function($app) {
    return $app['em']->getRepository('Ddd\Domain\Event\StoredEvent');
});

$app['message_tracker'] = $app->share(function($app) {
    return $app['em']->getRepository('Ddd\Domain\Event\PublishedMessage');
});

$app['message_producer'] = $app->share(function() {
    return new RabbitMqMessageProducer(
        new AMQPConnection('localhost', 5672, 'guest', 'guest')
    );
});

$app['notification_service'] = $app->share(function($app) {
    return new NotificationService(
        $app['event_store'],
        $app['message_tracker'],
        $app['message_producer']
    );
});
//...
```

PHP is not good for long-running processes because of memory leaking. If you need to have a command running for a long time, taking events and pushing them into RabbitMQ there are some options. You need to guarantee that your process is running and running properly. Sometimes, the process is running but the connection with RabbitMQ gets lost. The process goes into a zombie mode. We personally recommend to limit the amount of work that the worker has to do, 1000 items at a time, and finish the process. Then let tools such as Supervisor[14] rerun your job if it finds that it's not running.

6.7.2 REST

It's not the goal of this book to show you how to implement a REST API. However, with the `EventStore` already implemented in the messaging system, it should be easy to add some pagination capabilities, query for `DomainEvents`, and render a JSON or XML representation.

[14]http://supervisord.org/

6.8 Wrap-up

Now, the only thing remaining is how to listen for a notification in the messaging system, read it and execute the corresponding Application Service or Command. We'll see how to do this in the "Integrating Bounded Contexts" and "Application Services" chapters. We have seen the tricks to model a proper `DomainEvent` with a base interface. We have seen where to publish the `DomainEvent`, the nearer to the Entities the better, and what strategies to spread those `DomainEvents` to local and remote Bounded Contexts.

7. Modules

> When you place some classes together in a Module, you are telling the next developer who looks at your design to think about them together. If your model is telling a story, the Modules are chapters.
>
> Eric Evans, Domain-Driven Design

A common concern when building an application following DDD, is where do we put the code? What's the recommended way to place the code into the application? Where do we place infrastructure code? And more important, how should the different concepts inside the model be structured?

There's a tactical pattern for this: **modules**. Nowadays, everyone structures the code in modules. But DDD goes ones step further and no technical concerns are considered when using modules. Indeed, it treats *modules as a part of the model.*

> Modules should not be treated as a way to separate code but as a way to **separate meaningful concepts in the model**.

7.1 Structuring Code in Modules

If we take the example of a fictional e-commerce application, named **buy.it** it may make sense to define a module for each of the different bounded contexts that compose the e-commerce application, so each bounded context represents a self-contained and independent application

```
├── billing
│   ├── composer.json
│   ├── composer.lock
│   ├── src
│   └── tests
├── cart
│   ├── composer.json
│   ├── composer.lock
│   ├── src
│   └── tests
├── catalog
```

```
|   ├── composer.json
|   ├── composer.lock
|   ├── src
|   └── tests
└── inventory
    ├── composer.json
    ├── composer.lock
    ├── src
    └── tests
```

Each module contains an application that exposes a REST-like API. Beware that each module name represents a meaningful concept in an e-commerce system and is named in terms of *the Ubiquitous Language*:

- **Billing module** to hold all the code related to the payments, bills, waybills, order-processing systems with finite-state machine to be able to process the orders and so on.
- **Cart module** to hold all the code related to the cart system.
- **Catalog module** to hold all the code related to the product descriptions, product combinations and so on.
- **Inventory module** to hold all the code related to the management of product stocks.

Let's dig a bit further into one of those modules. Let's take for example the **Billing** context and examine the structure details. As its name suggests this module is responsible for representing all the flows that an order passes. From its creation until it's delivered to the customer who has purchased it. Furthermore, it's an independent application, so it contains a source code folder and a tests folder. The source code folder contains all the code necessary for this bounded context to work: **domain code** and **infrastructure code**.

```
├── composer.json
├── composer.lock
├── src
|   └── BuyIt
|       └── Billing
|           ├── DomainModel
|           └── Infrastructure
└── tests
```

All the code is prefixed with a vendor namespace named in terms of the organization name (**BuyIt**, in this case) and contains two subfolders: **DomainModel** holds all the domain code and **Infrastructure** holds the infrastructure layer, isolating all the domain logic from the *details* of the infrastructure layer. Following this structure we're making clear that we're going to use **Hexagonal Architecture** as a foundational architecture. An alternative structure we may have used, would be one as the following

```
├── composer.json
├── composer.lock
├── src
│   └── BuyIt
│       └── Billing
│           ├── Domain
│           │   ├── Model
│           │   └── Service
│           └── Infrastructure
└── tests
```

This style of structure uses an additional subfolder to store the services defined inside the domain model. While this organization may make sense, our preference here is to tend not to use it, since this way of separating code tends to be more focused on the architectural elements rather than the relevant concepts in the model. We believe that this style could easily lead to some sort of service layer on top of the domain model and this is not necessary a bad thing, but keep in mind that Domain Services are used to describe *things* into the domain, operations that don't belong to entities nor value objects. So, from now on we will stick with the previous code organization.

> It's possible to place code inside the **DomainModel** subfolder directly. For example, it may be quite common to place common interfaces and services in it, like the *DomainEventPublisher*, the *DomainEventSubscriber* and so on.

If we had to model a billing context, probably we would have an **Order** entity with its repository and all the state information. So our first attempt would be to directly place all those elements directly into the *DomainModel* subfolder. At a first glance, this may seem the simplest way

```
├── composer.json
├── composer.lock
├── src
│   └── BuyIt
│       └── Billing
│           ├── DomainModel
│           │   ├── Order.php
│           │   ├── OrderLine.php
│           │   ├── OrderLineWasAdded.php
│           │   ├── OrderRepository.php
│           │   └── OrderWasCreated.php
│           └── Infrastructure
└── tests
```

We've placed the *Order* and the *OrderLine* entities, the *OrderLineWasAdded* and the *OrderWasCreated* event and the *OrderRepository* into the same subfolder (*DomainModel*). **This structure may be fine, but that's because we still have a simple model**. What about the *Bill* entity plus its repository? Or the *Waybill* entity plus its respective repository? Let's add all those elements, and see how it fits into the actual code structure

```
├── composer.json
├── composer.lock
├── src
│   └── BuyIt
│       └── Billing
│           ├── DomainModel
│           │   ├── Bill.php
│           │   ├── BillLine.php
│           │   ├── BillLineWasAdded.php
│           │   ├── BillRepository.php
│           │   ├── BillWasCreated.php
│           │   ├── Order.php
│           │   ├── OrderLine.php
│           │   ├── OrderLineWasAdded.php
│           │   ├── OrderRepository.php
│           │   ├── OrderWasCreated.php
│           │   ├── Waybill.php
│           │   ├── WaybillLine.php
│           │   ├── WaybillLineWasAdded.php
│           │   ├── WaybillRepository.php
│           │   └── WaybillWasGenerated.php
│           └── Infrastructure
└── tests
```

While this style of code organization could be fine, it can become non-practical and pretty unmaintainable in the long term. Every time we iterate and add new features, the model will become even more bigger and that subfolder will be eating even more code. We're in the need to split the code in a way that give us a perspective of the model at a glance. No technical concerns, just **domain concerns**. To reach this, we can split this model using the *Ubiquitous Language*, finding meaningful concepts that help us group elements logically in terms of the *domain*. So we could try an approach as the following

```
├── composer.json
├── composer.lock
├── src
│   └── BuyIt
│       └── Billing
│           ├── DomainModel
│           │   ├── Bill
│           │   │   ├── Bill.php
│           │   │   ├── BillLine.php
│           │   │   ├── BillLineWasAdded.php
│           │   │   ├── BillRepository.php
│           │   │   └── BillWasCreated.php
│           │   ├── Order
│           │   │   ├── Order.php
│           │   │   ├── OrderLine.php
│           │   │   ├── OrderLineWasAdded.php
│           │   │   ├── OrderRepository.php
│           │   │   └── OrderWasCreated.php
│           │   └── Waybill
│           │       ├── Waybill.php
│           │       ├── WaybillLine.php
│           │       ├── WaybillLineWasAdded.php
│           │       ├── WaybillRepository.php
│           │       └── WaybillWasGenerated.php
│           └── Infrastructure
└── tests
```

This way the code is more organized, conceptually speaking. And not only that. As Evans points out *the blue book*[1], **Modules are a way to communicate** as they give us insights about how the domain model works internally, and help us increase the cohesion and decrease the coupling between the concepts. If we look at the previous example, we can see that the concepts **Order** and **OrderLine** are strongly related so they live in the same module. On the other hand, *Order* and *Waybill* although sharing the same context, they are different concepts so they live in **different modules**. Modules are not just a way to group related concepts into the model but a way to *express part of the design of the model.*

[1]http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215

Should we place *Repositories*, *Factories*, *Domain Events*, *Services* in their own subfolder?

Effectively they could be placed into their own subfolder, but it's strongly discouraged. Just because this way we would be mixing technical concerns and domain concerns, and remember that the Module's main interest is to group related concepts from the **domain** model and decouple them from the non-related. **Modules don't separate code but separate meaningful concepts**.

7.1.1 Modules in the Infrastructure Layer

So far we have been discussing how we structure and organize code in the *domain layer*, but we've almost said nothing about the *Infrastructure Layer*. And, since we're using *Hexagonal Architecture* to inverse the dependency between the domain and the infrastructure layer, we will need a place where we can put all the implementations of the interfaces defined in the domain layer. Returning to the example of the *billing* context, we need a place for the implementations of *BillRepository*, *OrderRepository* and *WaybillRepository*.

It's clear that they should be placed into the *Infrastructure* folder, but where? Suppose we decided to use *Doctrine ORM* to implement the persistence layer. How do we put the Doctrine implementations of our repositories into the *Infrastructure* folder? Let's put it directly into it and see how it looks.

```
├── composer.json
├── composer.lock
├── src
│   └── BuyIt
│       └── Billing
│           ├── DomainModel
│           │   ├── Bill
│           │   │   ├── Bill.php
│           │   │   ├── BillLine.php
│           │   │   ├── BillLineWasAdded.php
│           │   │   ├── BillRepository.php
│           │   │   └── BillWasCreated.php
│           │   ├── Order
│           │   │   ├── Order.php
│           │   │   ├── OrderLine.php
│           │   │   ├── OrderLineWasAdded.php
│           │   │   ├── OrderRepository.php
│           │   │   └── OrderWasCreated.php
│           │   └── Waybill
│           │       ├── Waybill.php
```

```
|           |       ├── WaybillLine.php
|           |       ├── WaybillLineWasAdded.php
|           |       ├── WaybillRepository.php
|           |       └── WaybillWasGenerated.php
|           └── Infrastructure
|               ├── DoctrineBillRepository.php
|               ├── DoctrineOrderRepository.php
|               └── DoctrineWaybillRepository.php
└── tests
```

We could leave it as this. But as we have seen in the *Domain Layer*, this structure and organization will rot fast and become a mess within a few model iterations. Each time the model grows, it will probably need even more infrastructure so this way we will end up mixing different technical concerns such as *persistence*, *messaging*, *logging* and a large etcetera. Our first attempt to avoid a tangled mess of infrastructure implementations is to define a module for each *technical concern* into the bounded context.

```
├── composer.json
├── composer.lock
├── src
|   └── BuyIt
|       └── Billing
|           ├── DomainModel
|           |   ├── Bill
|           |   |   ├── Bill.php
|           |   |   ├── BillLine.php
|           |   |   ├── BillLineWasAdded.php
|           |   |   ├── BillRepository.php
|           |   |   └── BillWasCreated.php
|           |   ├── Order
|           |   |   ├── Order.php
|           |   |   ├── OrderLine.php
|           |   |   ├── OrderLineWasAdded.php
|           |   |   ├── OrderRepository.php
|           |   |   └── OrderWasCreated.php
|           |   └── Waybill
|           |       ├── Waybill.php
|           |       ├── WaybillLine.php
|           |       ├── WaybillLineWasAdded.php
|           |       ├── WaybillRepository.php
|           |       └── WaybillWasGenerated.php
|           └── Infrastructure
```

```
|                   ├── Logging
|                   ├── Messaging
|                   └── Persistence
|                       ├── DoctrineBillRepository.php
|                       ├── DoctrineOrderRepository.php
|                       └── DoctrineWaybillRepository.php
└── tests
```

This looks much better and is a lot more maintainable in the long term than our first attempt. And if you know beforehand that you will always have a single persistence mechanism, you can stick with this structure and organization. It's quite simple and easy to maintain. But what about when you have to play with several persistence mechanisms? Nowadays, it's quite common to have a relational one, and some kind of shared in-memory persistence like *Redis* or *Riak*. Or to have some sort of local in-memory implementation to be able to test the code. Let's see how this fits into the actual approach.

```
├── composer.json
├── composer.lock
├── src
|   └── BuyIt
|       └── Billing
|           ├── DomainModel
|           |   ├── Bill
|           |   |   ├── Bill.php
|           |   |   ├── BillLine.php
|           |   |   ├── BillLineWasAdded.php
|           |   |   ├── BillRepository.php
|           |   |   └── BillWasCreated.php
|           |   ├── Order
|           |   |   ├── Order.php
|           |   |   ├── OrderLine.php
|           |   |   ├── OrderLineWasAdded.php
|           |   |   ├── OrderRepository.php
|           |   |   └── OrderWasCreated.php
|           |   └── Waybill
|           |       ├── Waybill.php
|           |       ├── WaybillLine.php
|           |       ├── WaybillLineWasAdded.php
|           |       ├── WaybillRepository.php
|           |       └── WaybillWasGenerated.php
|           └── Infrastructure
|               ├── Logging
```

```
│                   ├── Messaging
│                   └── Persistence
│                       ├── DoctrineBillRepository.php
│                       ├── DoctrineOrderRepository.php
│                       ├── DoctrineWaybillRepository.php
│                       ├── InMemoryBillRepository.php
│                       ├── InMemoryOrderRepository.php
│                       ├── InMemoryWaybillRepository.php
│                       ├── RedisBillRepository.php
│                       ├── RedisOrderRepository.php
│                       └── RedisWaybillRepository.php
└── tests
```

Again this now seems a bit odd. All the repository implementations are living in the same module, and this leads to a mix of several technologies. For now, we only have a few repositories. With a few more, the maintainability could start degrading considerably. So this makes the point clear that we need to create another module, *to group the related implementations by the underlying technology*.

```
├── composer.json
├── composer.lock
├── src
│   └── BuyIt
│       └── Billing
│           ├── DomainModel
│           │   ├── Bill
│           │   │   ├── Bill.php
│           │   │   ├── BillLine.php
│           │   │   ├── BillLineWasAdded.php
│           │   │   ├── BillRepository.php
│           │   │   └── BillWasCreated.php
│           │   ├── Order
│           │   │   ├── Order.php
│           │   │   ├── OrderLine.php
│           │   │   ├── OrderLineWasAdded.php
│           │   │   ├── OrderRepository.php
│           │   │   └── OrderWasCreated.php
│           │   └── Waybill
│           │       ├── Waybill.php
│           │       ├── WaybillLine.php
│           │       ├── WaybillLineWasAdded.php
│           │       ├── WaybillRepository.php
│           │       └── WaybillWasGenerated.php
```

```
|           └── Infrastructure
|               ├── Logging
|               ├── Messaging
|               └── Persistence
|                   ├── Doctrine
|                   │   ├── DoctrineBillRepository.php
|                   │   ├── DoctrineOrderRepository.php
|                   │   └── DoctrineWaybillRepository.php
|                   ├── InMemory
|                   │   ├── InMemoryBillRepository.php
|                   │   ├── InMemoryOrderRepository.php
|                   │   └── InMemoryWaybillRepository.php
|                   └── Redis
|                       ├── RedisBillRepository.php
|                       ├── RedisOrderRepository.php
|                       └── RedisWaybillRepository.php
└── tests
```

This structure and organization of the infrastructure layer is much more maintainable and easier to understand than our previous attempt. And we can have a general idea about the technologies being used in this bounded context.

7.1.1.1 Mixing Different Technologies

In large business-critical applications it's quite common to have a mix of several technologies. For example, in read-intensive web applications you usually have some sort of denormalized data source (*Solr*, *Elastic*, *Sphinx*, etc.) that provides all the reads of the application while a traditional RDBMS like MySQL or Postgres is mainly responsible to handle all the writes. When this occurs one of the concerns that normally arise is whether we can have read operations go with the search engine and the write operations go with the traditional RDBMS data source. Our general advice here, is that these kind of situations are a smell for **CQRS**, *since we are in the need to scale the reads and the writes of the application independently*. So if you can go with *CQRS*, probably that will be the best choice.

But if for any reason you cannot go with *CQRS*, an alternative approach is needed. In this situation, the use of the *Proxy pattern* from *Gang of Four* comes in handy. We can define an implementation of a repository in terms of the *Proxy pattern*.

```
namespace BuyIt\Billing\Infrastructure\FullTextSearching\Elastica;

use BuyIt\Billing\DomainModel\Order\OrderRepository;
use BuyIt\Billing\Infrastructure\Persistence\Doctrine\DoctrineOrderRepository;
use Elastica\Client;

class ElasticaOrderRepository implements OrderRepository
{
    private $client;
    private $baseOrderRepository;

    public function __construct(Client $client, DoctrineOrderRepository $baseOrd\
erRepository)
    {
        $this->client = $client;
        $this->baseOrderRepository = $baseOrderRepository;
    }

    public function find($id)
    {
        return $this->baseOrderRepository->find($id);
    }

    public function findBy(array $criteria)
    {
        $search = new \Elastica\Search($this->client);
        // ...
        return $this->toOrder($search->search());
    }

    public function add($anOrder)
    {
        // First we attempt to add it to the Elastic index
        $ordersIndex = $this->client->getIndex('orders');
        $orderType = $ordersIndex->getType('order');
        $orderType->addDocument(
            new \Elastica\Document(
                $anOrder->id(),
                $this->toArray($anOrder)
            )
        );
```

```
        $ordersIndex->refresh();

        // When it's done, we attempt to add it to the RDBMS store
        $this->baseOrderRepository->add($anOrder);
    }
}
```

This example provides a naive implementation using the `DoctrineOrderRepository` and the *Elastica* client, a client to interact with an Elastic server. Note that for some operations we are using the RDBMS datasource and for others the Elastica client. And also note that the `add` operation consists of two parts. The first one attempts to store the Order to the Elastic index and the second one attempts to store the Order into the relational database delegating the operation to the Doctrine implementation. Take into account that this is just an example and a way to do it. Probably it can be improved, for example now the whole `add` operation is synchronous. We could instead enqueue the operation to some sort of messaging middleware that stores the *Order* into Elastic, for example. There are a lot of posibilities and improvements, depending on your needs.

7.2 Leverage Modules in PHP

Until PHP 5.3, modules were not fully supported. Nowadays, since PHP 5.3, we can use PHP namespaces to implement the *Module* pattern. For historical reasons, we're going to present how namespaces were used before PHP 5.3. But you should strive to use some PHP version that supports PHP namespaces. The best choice always is going to be the latest stable version of PHP.

7.2.1 PEAR-style Namespaces

Before PHP 5.3, due to the lack of a namespace construction, PEAR-style namespaces were used. PEAR is the acronym for *PHP Extension and Application Repository* and in the good old times was a repository of reusable components. It's still active, but its use is a minority and there's a lot of unmaintained packages. Especially since composer and packagist took the stage. PEAR, as a source of reusable components, needed a way to avoid class name collisions so they started prefixing class names with namespaces. There are still projects that use this form of namespaces (PHPUnit or Zend Framework 1, to name a few). The following would be an example of PEAR-style namespaces

```
├── composer.json
├── composer.lock
└── src
    └── BuyIt
        └── Billing
            └── DomainModel
                └── Bill
                    └── Bill.php
```

The class name for the *Bill* entity, using the PEAR-style namespaces, would become `BuyIt_Billing_DomainModel_Bill_Bill`. That class name it's a bit ugly and don't follow one of the main DDD mantras: every class name should be named in terms of *Ubiquitous Language*. For this reason we strongly discourage its usage.

7.2.2 PSR-0 and PSR-4 Namespacing Conventions

Along with other important features in PHP 5.3, namespaces entered the scene. This was a major shift, a group of the most important framework collaborators emerged with PHP-FIG[2], an acronym of *PHP Framework Interop Group* in an attempt to standardize and unify common aspects of the framework and library creation. The first *PHP Standard Recommendation* (PSR, from now on) that the group released was an autoloading standard that, summing up, proposes a one to one relation between a class and a PHP file using namespaces. Nowadays PSR-4, a simplification of PSR-0 that still maintains the relation between classes and physical PHP files, is the preferred and recommended way to structure code, and we believe that this should be the one used to implement *Modules* in a project. Returning to the previous example

```
├── composer.json
├── composer.lock
└── src
    └── BuyIt
        └── Billing
            └── DomainModel
                └── Bill
                    └── Bill.php
```

The class name for the *Bill* entity, using namespaces and PSR-0/PSR-4, would become simply `Bill` and the full qualified class name would be `BuyIt\Billing\DomainModel\Bill\Bill`. As you can see, this way enables us to name domain objects in terms of the *Ubiquitous Language* and is the preferred way to structure and organize code.

[2] http://www.php-fig.org/

7.3 Wrap-up

Modules are a way of grouping and separating concepts in our domain model. Modules should be named following the Ubiquitous Language. We should not forget that Modules are a way to communicate high-level concepts, it helps us keeping coupling low and cohesion high. We've seen that we could create meaningful modules even in old versions of PHP by using prefixes. Nowadays it's easy to build our Modules following the PSR-0 and PSR-4 namespacing conventions.

8. Aggregates

8.1 Introduction

Aggregates are probably the most difficult to understand and implement building blocks of Domain-Driven Design. For properly implement them, we need to understand concepts such as transaction, locking and concurrency strategies. It's also interesting to understand their origin and how the NoSQL movement has influenced them so much.

From Vaughn Vernon's "Implementing Domain-Driven Design": Aggregates are carefully crafted consistency boundaries that cluster Entities and Value Objects. Another amazing book, you should definitely buy and read, is "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" by Pramod J. Sadalage. From this book: "In Domain-Driven Design, an aggregate is a collection of related objects that we wish to treat as a unit. In particular, it is a unit for data manipulation and management of consistency. Typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates."

8.1.1 What Martin Fowler says...

From http://martinfowler.com/bliki/DDD_Aggregate.html[1]:

Aggregate is a pattern in Domain-Driven Design. A DDD aggregate is a cluster of domain objects that can be treated as a single unit. An example may be an order and its line-items, these will be separate objects, but it's useful to treat the order (together with its line items) as a single aggregate.

An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.

Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Transactions should not cross aggregate boundaries.

DDD Aggregates are sometimes confused with collection classes (lists, maps, etc). DDD aggregates are domain concepts (order, clinic visit, playlist), while collections are generic. An aggregate will often contain multiple collections, together with simple fields. The term "aggregate" is a common one, and is used in various different contexts (e.g. UML), in which case it does not refer to the same concept as a DDD aggregate.

[1] http://martinfowler.com/bliki/DDD_Aggregate.html

8.1.2 What Wikipedia says…

From https://en.wikipedia.org/wiki/Domain-driven_design#Building_blocks_of_DDD[2]:

Aggregate: A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members.

Example: When you drive a car, you do not have to worry about moving the wheels forward, making the engine combust with spark and fuel, etc.; you are simply driving the car. In this context, the car is an aggregate of several other objects and serves as the aggregate root to all of the other systems.

8.2 A bit of history

At the beginning of the new millennium the technology world was hit by the busting of the 1990s dot-com bubble. While this saw many people questioning the economic future of the Internet, the 2000s did see several large web properties dramatically increase in scale.

This increase in scale was happening along many dimensions. Websites started tracking activity and structure in a very detailed way. Large sets of data appeared: links, social networks, activity in logs, mapping data. With this growth in data came a growth in users—as the biggest websites grew to be vast estates regularly serving huge numbers of visitors.

Coping with the increase in data and traffic required more computing resources. To handle this kind of increase, you have two choices: up or out. Scaling up implies bigger machines, more processors, disk storage, and memory. But bigger machines get more and more expensive, not to mention that there are real limits as your size increases. The alternative is to use lots of small machines in a cluster. A cluster of small machines can use commodity hardware and ends up being cheaper at these kinds of scales. It can also be more resilient—while individual machine failures are common, the overall cluster can be built to keep going despite such failures, providing high reliability.

As large properties moved towards clusters, that revealed a new problem—relational databases are not designed to be run on clusters. Clustered relational databases, such as the Oracle RAC or Microsoft SQL Server, work on the concept of a shared disk subsystem. They use a cluster-aware file system that writes to a highly available disk subsystem—but this means the cluster still has the disk subsystem as a single point of failure. Relational databases could also be run as separate servers for different sets of data, effectively sharding the database. While this separates the load, all the sharding has to be controlled by the application which has to keep track of which database server to talk to for each bit of data. Also, we lose any querying, referential integrity, transactions, or consistency controls that cross shards. A phrase we often hear in this context from people who've done this is "unnatural acts."

These technical issues are exacerbated by licensing costs. Commercial relational databases are usually priced on a single-server assumption, so running on a cluster raised prices and led to

[2] https://en.wikipedia.org/wiki/Domain-driven_design#Building_blocks_of_DDD

frustrating negotiations with purchasing departments. This mismatch between relational databases and clusters led some organization to consider an alternative route to data storage. Two companies in particular—Google and Amazon—have been very influential. Both were on the forefront of running large clusters of this kind; furthermore, they were capturing huge amounts of data. These things gave them the motive. Both were successful and growing companies with strong technical components, which gave them the means and opportunity. It was no wonder they had murder in mind for their relational databases. As the 2000s drew on, both companies produced brief but highly influential papers about their efforts: BigTable from Google and Dynamo from Amazon.

It's often said that Amazon and Google operate at scales far removed from most organizations, so the solutions they needed may not be relevant to an average organization. While it's true that most software projects don't need that level of scale, it's also true that more and more organizations are beginning to explore what they can do by capturing and processing more data—and to run into the same problems. So, as more information leaked out about what Google and Amazon had done, people began to explore making databases along similar lines—explicitly designed to live in a world of clusters. While the earlier menaces to relational dominance turned out to be phantoms, the threat from clusters was serious.

8.3 Aggregates and clusters

TBW

8.4 Anatomy of an Aggregate

An Aggregate is conformed by a root Entity that holds other Entities and Value Objects. A single Entity without any child Entities or Value Objects conforms an Aggregate by itself. That's why in some books the term Aggregates is used over the term Entity.

The main benefit or their real goal of an Aggregate is consistency in our Domain Model operations. It allows us to guarantee that changes on a hierarchy of Entities and Value Objects are performed atomically.

Consider a rough examples to introduce the idea. Imagine an e-commerce application and a typical persistence mechanism. There are Orders and Line Orders. Orders total amounts and Line Orders subtotal amounts sum must match. You will never perform changes in a Line Order amount and in the Order amount without using a transaction protecting both changes. Why? Because the UPDATE statement that updates the Line Order can work properly while the UPDATE on the Order total amount could fail due to network connectivity issues. With such situation, you would end up with a inconsistency in tables and your Domain, what would happen if you ask for the total amount in the Order and then you apply the business logic to calculate the sum of all the Line Orders? You get the idea. If you put both changes in a transaction, both will succeed or both will fail, that's consistency. So transactions help you keep this consistency.

On the other side, the negative effects are performance issues and persistence errors.

An Aggregate is fetched and persisted using its own repository, whether it holds many Entities and Value Objects or none.

In order to design an Aggregate, there are some rules or considerations to follow so we can get all the benefits minimizing the negative effects.

8.4.1 Design Aggregates based in Business True Invariants

First of all, what's an invariant? An invariant is a rule that must be true and be consistent during code execution. Let's see an example to help you. A stack[3] is a LIFO data structure where we can push elements, pop them and ask for its size. Consider a pure PHP implementation without using any specific PHP array functions such as `array_pop`.

```
class Stack
{
    private $data;

    public function __construct()
    {
        $this->data = [];
    }

    public function push($value)
    {
        $this->data[] = $value;
    }

    public function size()
    {
        return count($this->data);
    }

    /**
     * @return mixed
     */
    public function pop()
    {
        $topIndex = $this->size() - 1;
        $top = $this->data[$topIndex];
        unset($this->data[$topIndex]);
```

[3] https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

```
        return $top;
    }
}
```

Imagine that getting the size of the stack would be a really CPU intensive and high-cost call. There is an option to improve that method, introducing a private attribute to keep track of the number of elements in the internal array.

```
class Stack
{
    private $data;
    private $size;

    public function __construct()
    {
        $this->data = [];
        $this->size = 0;
    }

    public function push($value)
    {
        $this->data[] = $value;
        $this->size++;
    }

    public function size()
    {
        return $this->size;
    }

    /**
     * @return mixed
     */
    public function pop()
    {
        $topIndex = $this->size() - 1;
        $top = $this->data[$topIndex];
        unset($this->data[$topIndex]);
        $this->size--;

        return $top;
```

```
    }
}
```

With this approach, now asking for the size of the stack is a light method. What's the cost? There were some updates in `pop` and `push` methods to keep track of the new size when adding and removing elements. Let's try to find an invariant, a rule that should be true before and after any method call of the stack object. What about `$this->size === count($this->data)`? Before and after each call to any method in the stack object, the attribute size is consistent and it always holds the size of the stack.

So what's a? An invariant is a business rule that must always be true and transactionally consistent. In the previous example, the amount of an Order must match the sum of amounts of all the Line Orders is an invariant. That makes probably the Order be the root and allow us to make operations to it rather than the Order Lines that are inside. With this approach, we have a single entry point to perform operations on the cluster that will be consistent and do not break any invariant. It means, there is no chance to invoke a method to break such rule. Each time you add a Line Order or update info from the root, internally, the Order amount gets calculated.

Be careful about the "has-a"/"has-many" relations that do not necessary make two Entities become one Aggregate with one of those being the root.

8.4.2 Small Aggregates vs Big Aggregates

When designing Aggregates struggle to create small Aggregates. If there is no invariant to protect, than means all single Entities perform

As you probably have done for the 90%

8.4.3 Pushing for Eventual Consistency

8.4.4 Modify one Aggregate per transaction

"a properly designed Bounded Context modifies only one Aggregate instance per transaction in all cases. What is more, we cannot correctly reason on Aggregate design without applying transactional analysis. Limiting modification to one Aggregate instance per transaction may sound overly strict. However, it is a rule of thumb and should be the goal in most cases. It addresses the very reason to use Aggregates."”

8.4.5 Exception: UX case

When designing Aggregates following the previous rules, there is a typical exception.

8.5 Sample Application Service: User and Wishes

The best way to learn about Aggregates is seeing code. So, let's consider the following scenario: A web application where users can make wishes that would like to be granted if something happened to them. Think about it as a will. For example, I would like to send an email to my wife explaining what to do with my GitHub account if I die in a horrible accident. They way to check that I'm still alive is to answer to emails the platform is sending to me. If you want to know more about this application[4] you can visit our GitHub account[5] with this example and more.

Ok, so we have users and their wishes. Let's consider only one use case, "As a User, I want to make a Wish". How could we model these two concepts? As the good practices when designing Aggregates, let's try to push for small Aggregates, that would be, in this case, two different Aggregates of one Entity each, User and Wish. What about the relation between them? Well, we should use an identifier, for example, `UserId`. Let's see some code.

8.5.1 No invariant, two aggregates

"As a User, I want to make a Wish". We will go into Application Services in following chapters, but for now, let's check different approaches for making a Wish. We think that the first approach for the novice will be something similar to the following code.

```
class MakeWishService
{
    private $wishRepository;

    public function __construct(WishRepository $wishRepository)
    {
        $this->wishRepository = $wishRepository;
    }

    public function execute(MakeWishRequest $request)
    {
        $userId = $request->userId();
        $address = $request->address();
        $content = $request->content();

        $wish = new Wish(
            $this->wishRepository->nextIdentity(),
            new UserId($userId),
            $address,
```

[4] https://github.com/ddd-in-php/last-wishes
[5] https://github.com/ddd-in-php

```
            $content
        );

        $this->wishRepository->add($wish);
    }
}
```

How you see this code? It is probably the most performing code possible. Behind the scenes you can almost see the `INSERT` statement. It's the minimum number of operations for such a use case, one, so good job. With the current implementation you can create as many wishes as you want, following the business logic, so good job again.

However, there may be just one potential issue, you can create wishes for a user that may no exist in your Domain. That is a problem. Indeed, it doesn't really matter if you will use a Relational database or a NoSQL one, before persisting anything, with this approach, you can create a `Wish` without its corresponding `User` in memory.

Indeed, it's a broken business logic. I know, I know, you can fix that using a foreign key in the database, from `wish(user_id)` to `user(id)`. Correct, but, what happen if we are not using a database with foreign keys, even more, what happen if is a NoSQL database, such as Redis or ElasticSearch?

So, if we want to fix this issue so the same code can work properly in different infrastructures, we need to check if the user exists. Probably, the easiest approach could be in the same Application Service, couldn't be?

```
class MakeWishService
{
    //...

    public function execute(MakeWishRequest $request)
    {
        $userId = $request->userId();
        $address = $request->address();
        $content = $request->content();

        $user = $this->userRepository->ofId(new UserId($userId));
        if (null === $user) {
            throw new UserDoesNotExistException();
        }

        $wish = new Wish(
            $this->wishRepository->nextIdentity(),
            $user->id(),
            $address,
```

```
            $content
        );

        $this->wishRepository->add($wish);
    }
}
```

That could make the trick, but what's the problem about doing the check in the Application Service? There is no protection about that in our Domain, anyone, inside a Domain Service, any part of the infrastructure, etc. could do something such as the following code.

```
// Somewhere in your domain
$nonExistingUserId = new UserId('non-existing-user-id');
$wish = new Wish(
    $this->wishRepository->nextIdentity(),
    $nonExistingUserId,
    $address,
    $content
);
```

If you have already read the Factories chapter you have got the solution too. Factories help us keeping the business invariants and that's exactly what we need here. There's an implicit invariant saying that we are not allowed to make a wish without a valid user. Let's see how a factory can help us.

```
class MakeWishService
{
    private $userRepository;
    private $wishRepository;

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

    public function execute(MakeWishRequest $request)
    {
        $userId = $request->userId();
        $address = $request->address();
```

```
        $content = $request->content();

        $user = $this->userRepository->ofId(new UserId($userId));
        if (null === $user) {
            throw new UserDoesNotExistException();
        }

        $wish = $user->makeWish(
            $this->wishRepository->nextIdentity(),
            $address,
            $content
        );

        $this->wishRepository->add($wish);
    }
}
```

As you can see, `Users` make wishes, so our code does. `makeWish` is a factory method for building `Wishes`. The method returns a new wish build with the `UserId`.

```
class User
{
    // ...

    /**
     * @return Wish
     */
    public function makeWish(WishId $wishId, $address, $content)
    {
        return new Wish(
            $wishId,
            $this->id(),
            $address,
            $content
        );
    }

    // ...
}
```

Why are we returning a `Wish` and not just adding the new `Wish` to an internal collection as we would do with Doctrine probably?

To sum up, in this scenario, `User` and `Wish` do not conform an aggregate. Each Entity has its own Repository and they are linked using `UserId`. Getting all the wishes of a User can be performed by a finder in the wishes Repository.

8.5.2 No more than three Wishes per User

Our application is a huge success and now it's time to monetize. We want new users to have a maximum of three wishes available. As a User, if you want to have more wishes you'll probably have to pay for a premium account. Let's see how we could change our code to follow the new business rule about the maximum number of wishes (do not consider the premium user).

```
class MakeWishService
{
    // ...

    public function execute(MakeWishRequest $request)
    {
        $userId = $request->userId();
        $address = $request->email();
        $content = $request->content();

        $count = $this->wishRepository->numberOfWishesByUserId($userId);
        if ($count > 3) {
            throw new MaxNumberOfWishesExceededException();
        }

        $wish = new Wish(
            $this->wishRepository->nextIdentity(),
            new UserId($userId),
            $address,
            $content
        );

        $this->wishRepository->add($wish);
    }
}
```

That was easy, wasn't it? Probably too much easy. We see here different problems. First one is that Application Services should not include such business logic. They must coordinate, but not contain business logic. Probably, a better place is to put them into the `User`. We can have more control about the relation between `User` and `Wish`. However, for the problem explained here, the code works.

The second problem is about the code itself. It **does not work under race conditions**. So, it nos acceptable. Forget about Domain-Driven Design, what's the problem with this code in heavy traffic? Think for a minute. Could be possible to break the rule of a `User` to have more than three wishes? Why your QA running her freaky tests is going to be super happy?

Your QA tries two times in a calm way and ends up with a user with two wishes. Nice. Your QA wants blood. Imagine for a second that opens two tabs in her browser, fills both two forms and submits the two buttons at the same time. Suddenly, after two requests, the user ends up with four wishes in the database. What happened?

Think as a debugger, consider both requests getting at the `if ($count > 3) {` line at the same time. Both of the requests will evaluate to `false`, because the user has just two wishes. So, both requests will create the `Wish` and both of the request will add it into the database. Ouch! four wishes in a User.

We know what you're thinking. It's because we missed to put everything into a transaction. Well, imagine that the same previous code is put inside a transaction block, you will see how to do it properly in the Application Services chapter. Internally, let's check what's happening with your database.

```
START TRANSACTION;
SELECT @a:=COUNT(*) FROM wishes WHERE user_id = 'e3bb5953-5dd2-4204-a8e8-3449ea8\
88c40';
-- @a is 2
INSERT INTO wishes(id, user_id, address, message) VALUES ('7b81e576-d7dc-4736-a5\
59-340961b7102a', 'e3bb5953-5dd2-4204-a8e8-3449ea888c40', 'mom@myfamily.com', 'I\
 always love you!');
-- Ok!
SELECT @a:=COUNT(*) FROM wishes;
-- @a is 3
COMMIT;
```

If you take this SQL block and execute it line by line in two different connections (with different UUIDs for the new wishes), you will see how the `wishes` table is going to have `4` rows at the end of both executions. So, it looks like is not about the transaction.

How could we fix this issue? Probably, you may have heart about Pessimistic Concurrency and Optimistic Concurrency in persistence mechanisms let's explore them.

8.5.2.1 Pessimistic concurrency control

Widely used by relational databases, this approach assumes that conflicting changes are likely to happen and so blocks access to a resource in order to prevent conflicts. A typical example is locking a row before reading its data, ensuring that only the thread that placed the lock is able to make changes to the data in that row.

8.5.2.2 Optimistic concurrency control

This approach assumes that conflicts are unlikely to happen and doesn't block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict. For instance, it could reattempt the update, using the fresh data, or it could report the situation to the user.

Could we use any of these strategies for this use case? Mmmm, we don't think so. Because we're making a new wish,

```
class User
{
    // ...

    /**
     * @return void
     */
    public function makeWish(WishId $wishId, $address, $content)
    {
        $this->wishes[] = new Wish(
            $wishId,
            $this->id(),
            $address,
            $content
        );
    }

    // ...
}
```

8.5.3 Rendering User's Wishes

TBW

8.5.4 Updating a User's Wish

TBW

8.5.5 Granting User's Wishes

TBW

8.6 Wrap-up

TBW

9. Factories

9.1 Introduction

In the Domain, factories help in decoupling the client from knowing how to build complex objects and Aggregates. You could use them in order to create entire Aggregates as an entire piece, enforcing their invariants.

9.2 Factory Method on Aggregate Root

The Factory Method[1] pattern, as defined in the classic Gang of Four, is a creational pattern that…

> Defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.

Adding a Factory Method in the Aggregate Root hides the internal implementation details about creating aggregates from any external client. This also moves the responsibility for the integrity of the Aggregate back to the root.

In a Domain model where we have a `User` and `Wish` entity, the `User` acts as the Aggregate Root. There is no `Wish` without `User`. The `User` entity should manage its Aggregates.

The way to move the control of `Wish` back to the `User` entity is by placing a Factory Method in the Aggregate Root.

```
class User
{
    //...

    public function makeWish(WishId $wishId, $email, $content)
    {
        $wish = new WishEmail(
            $wishId,
            $this->id(),
            $email,
            $content
```

[1] http://en.wikipedia.org/wiki/Factory_method_pattern

```
        );

        DomainEventPublisher::instance()->publish(
            new WishMade($wishId)
        );

        return $wish;
    }
}
```

The client does not need to now the internal details about how the Aggregate Root handles the creation logic at all

```
$wish = $aUser->makeWish(
    $wishRepository->nextIdentity(),
    'user@example.com',
    'I want to be free!'
);
```

9.2.1 Forcing Invariants

Factory Methods in the Aggregate Root are also a good place for invariants.

In a Domain model with a `Forum` and `Article` entity, where `Article` is an Aggregate of `Forum`, publishing an Article could be something like

```
class Forum
{
    // ...

    public function publishPost(PostId $postId, $content)
    {
        $post = new Post($this->id, $postId, $content);

        DomainEventPublisher::instance()->publish(
            new PostPublished($postId)
        );

        return $post;
    }
}
```

After talking with a Domain Expert we came to the conclusion that a `Post` should not be published when the `Forum` is closed. This is an invariant and we could force it directly on `Post` creation preventing an inconsistent Domain state

```
class Forum
{
    // ...

    public function publishPost(PostId $postId, $content)
    {
        if ($this->isClosed()) {
            throw new ForumClosedException();
        }

        $post = new Post($this->id, $postId, $content);

        DomainEventPublisher::instance()->publish(
            new PostPublished($postId)
        );

        return $post;
    }
}
```

9.3 Factory on Service

Decoupling creation logic also comes very handy in our services.

9.3.1 Building Specifications

Using Specifications in our services might be the best example to illustrate how to use factories within our services.

Consider the following service example. Given a request from the outside world, we want to build a feed based on the latest Posts added to the system.

```
namespace Application\Service;

use Domain\Model\Post;
use Domain\Model\PostRepository;

class LatestPostsFeedService
{
    private $postRepository;

    public function __construct(PostRepository $postRepository)
```

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

    /**
     * @param LatestPostsFeedRequest $request
     */
    public function execute($request)
    {
        $posts = $this->postRepository->latestPosts($request->since);

        return array_map(function(Post $post) {
            return [
                'id' => $post->id()->id(),
                'content' => $post->body()->content(),
                'created_at' => $post->createdAt()
            ];
        }, $posts);
    }
}
```

Finder methods in Repositories like `latestPosts` have some limitations as they keep adding complexity to our repositories indefinitely. As we discussed in Repositories chapter, Specifications are a better approach.

Lucky for us, we have a nice `query` method in our `PostRepository` that works with Specifications.

```
class LatestPostsFeedService
{
    //...

    public function execute($request)
    {
        //...

        $posts = $this->postRepository->query($specification);

        //...
    }
}
```

Using a concrete implementation for the Specification is a bad idea

```
class LatestPostsFeedService
{
    //...

    public function execute($request)
    {
        $posts = $this->postRepository->query(
            new SqlLatestPostSpecification($request->since)
        );

        //...
    }
}
```

Coupling our high-level application service with a low-level Specification implementation mixes layers and breaks separation of concerns. In addition, its a pretty bad way of coupling our service to a concrete infrastructure implementation. There's no way you could use this service out of the SQL persistence solution. What if we want to test our service with an in-memory implementation?

The solution to this problem is to decouple Specification creation from the service itself by using the Abstract Factory pattern[2].

> Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

As we might have multiple Specification implementations we need to create an interface for the factory first.

```
namespace Domain\Model;

interface PostSpecificationFactory
{
    public function createLatestPosts(\DateTime $since);
}
```

Then we need to create factories for each `PostRepository` implementations. As an example, a Factory for the in-memory `PostRepository` implementation could be like

[2]http://en.wikipedia.org/wiki/Abstract_factory_pattern

```
namespace Infrastructure\Persistence\InMemory;

use Domain\Model\PostSpecificationFactory;

class InMemoryPostSpecificationFactory implements PostSpecificationFactory
{
    public function createLatestPosts(\DateTime $since)
    {
        return new InMemoryLatestPostSpecification($since);
    }
}
```

Once we have a centralised place for the creation logic, its easy to decouple it from the service

```
class LatestPostsFeedService
{
    private $postRepository;
    private $postSpecificationFactory;

    public function __construct(
        PostRepository $postRepository,
        PostSpecificationFactory $postSpecificationFactory
    ) {
        $this->postRepository = $postRepository;
        $this->postSpecificationFactory = $postSpecificationFactory;
    }

    public function execute($request)
    {
        $posts = $this->postRepository->query(
            $this->postSpecificationFactory->createLatestPosts($request->since)
        );

        // ...
    }
}
```

Now unit-testing our service through an in-memory `PostRepository` implementation is pretty easy

```
namespace Application\Service;

use Domain\Model\Body;
use Domain\Model\Post;
use Domain\Model\PostId;
use Infrastructure\Persistence\InMemory\InMemoryPostRepository;

class LatestPostsFeedServiceTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @var \Infrastructure\Persistence\InMemory\InMemoryPostRepository
     */
    private $postRepository;

    /**
     * @var LatestPostsFeedService
     */
    private $latestPostsFeedService;

    public function setUp()
    {
        $this->latestPostsFeedService = new LatestPostsFeedService(
            $this->postRepository = new InMemoryPostRepository()
        );
    }

    /**
     * @test
     */
    public function shouldBuildAFeedFromLatestsPosts()
    {
        $this->addPost(1, 'first', '-2 hours');
        $this->addPost(2, 'second', '-3 hours');
        $this->addPost(3, 'third', '-5 hours');

        $feed = $this->latestPostsFeedService->execute(
            new LatestPostsFeedRequest(new \DateTime('-4 hours'))
        );

        $this->assertFeedContains([
            ['id' => 1, 'content' => 'first'],
            ['id' => 2, 'content' => 'second']
```

```
        ], $feed);
    }

    private function addPost($id, $content, $createdAt)
    {
        $this->postRepository->add(new Post(
            new PostId($id),
            new Body($content),
            new \DateTime($createdAt)
        ));
    }

    private function assertFeedContains($expected, $feed)
    {
        foreach ($expected as $index => $contents) {
            $this->assertArraySubset($contents, $feed[$index]);
            $this->assertNotNull($feed[$index]['created_at']);
        }
    }
}
```

9.3.2 Building Aggregates

Entities are agnostic to the persistence mechanism. You don’t want to couple and pollute your entities with persistence details.

Take a look at the next Application Service

```
class SignInUserService
{
    private $userRepository;

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

    /**
     * @param SignInUserRequest $request
     */
    public function execute($request)
    {
        $email = $request->email();
```

```
        $password = $request->password();

        $user = $this->userRepository->userOfEmail($email);
        if (null !== $user) {
            throw new UserAlreadyExistsException();
        }

        $this->userRepository->persist(new User(
            $this->userRepository->nextIdentity(),
            $email,
            $password
        ));

        return $user;
    }
}
```

With a `User` entity like the next one

```
class User
{
    private $userId;
    private $email;
    private $password;

    public function __construct(UserId $userId, $email, $password)
    {
        //...
    }

    //...
}
```

Imagine we want to use `Doctrine` as our infrastructure persistence mechanism. `Doctrine` requires having an `id` as a plain string instance variable in order to work properly. In our entity, `$userId` is a `UserId` Value Object. Adding an additional `id` to our `User` entity just because of `Doctrine` would couple our persistence mechanism with our domain model.

We've seen in the Entities Chapter that we could solve this problem with a Surrogate Id by creating a wrapper around our `User` entity in the infrastructure layer

```
class DoctrineUser extends User
{
    private $surrogateUserId;

    public function __construct(UserId $userId, $email, $password)
    {
        parent::__construct($userId, $email, $password);
        $this->surrogateUserId = $userId->id();
    }
}
```

As, creating the `DoctrineUser` in our application service would couple again the persistence layer with our domain, we need to decouple the creation logic out of the service with an Abstract Factory. We could do this by creating an interface in our Domain.

```
interface UserFactory
{
    public function build(UserId $userId, $email, $password);
}
```

And placing the implementation of it inside our infrastructure layer.

```
class DoctrineUserFactory implements BaseUserFactory
{
    public function build(UserId $userId, $email, $password)
    {
        return new DoctrineUser($userId, $email, $password);
    }
}
```

Once decoupled, we only need to inject the Factory into our Application Service

```
class SignInUserService
{
    private $userRepository;
    private $userFactory;

    public function __construct(
        UserRepository $userRepository,
        UserFactory $userFactory
    ) {
```

```
        $this->userRepository = $userRepository;
        $this->userFactory = $userFactory;
    }

    /**
     * @param SignInUserRequest $request
     */
    public function execute($request)
    {
        //...

        $user = $this->userFactory->build(
            $this->userRepository->nextIdentity(),
            $email,
            $password
        );

        $this->userRepository->persist($user);

        return $user;
    }
}
```

9.4 Testing

You'll see a common pattern while writing your tests. Building entities and complex aggregates could be a very tedious and repetitive process. Complexity and duplication will start creeping in your test suite.

Consider the following entity

```
class Author
{
    private $username;
    private $email;
    private $fullName;

    public function __construct(
        Username $aUsername,
        FullName $aFullName,
        Email $anEmail
    ) {
```

```
        $this->username = $aUsername;
        $this->email = $anEmail;
        $this->fullName = $aFullName;
    }

    //...
}
```

Included in some test, somewhere in the system

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @test
     */
    public function itDoesSomething()
    {
        $author = new Author(
            new Username('johndoe'),
            new FullName('John', 'Doe'),
            new Email('john@doe.com')
        );

        //do something with author
    }
}
```

Services inside boundaries share concepts like entities, aggregates and value objects imagine the clutter of repeating the same building logic over and over across your tests. As we will see, extracting the building logic out of our tests comes very handy and prevents duplication.

9.4.1 Object Mother

An *Object Mother*[3] is a catchy name for a factory that creates fixed fixtures for your tests.

Following the previous example, we could extract the duplicated logic to an Object Mother so it could be reused across tests.

[3] http://martinfowler.com/bliki/ObjectMother.html

```
class AuthorObjectMother
{
    public static function createOne()
    {
        return new Author(
            new Username('johndoe'),
            new FullName('John', 'Doe'),
            new Email('john@doe.com')
        );
    }
}


class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @test
     */
    public function itDoesSomething()
    {
        $author = AuthorObjectMother::createOne();

        //do something with author
    }
}
```

You'll notice that the more tests and situations you have, the more methods the factory will have.

As Object Mothers are not very flexible, they tend to grow in complexity quickly. There is a more flexible alternative for your tests.

9.4.2 Test Data Builder

Test Data Builders are just normal Builders with default values used exclusively in your test suites so you don't have to specify irrelevant parameters on specific test cases.

```
class AuthorBuilder
{
    private $username;
    private $email;
    private $fullName;

    private function __construct()
    {
        $this->username = new Username('johndoe');
        $this->email = new Email('john@doe.com');
        $this->fullName = new FullName('John', 'Doe');
    }

    public static function anAuthor()
    {
        return new self();
    }

    public function withFullName(FullName $aFullName)
    {
        $this->fullName = $aFullName;

        return $this;
    }

    public function withUsername(Username $aUsername)
    {
        $this->username = $aUsername;

        return $this;
    }

    public function withEmail(Email $anEmail)
    {
        $this->email = $anEmail;

        return $this;
    }

    public function build()
    {
        return new Author($this->username, $this->fullName, $this->email);
```

```
    }
}


class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @test
     */
    public function itDoesSomething()
    {
        $author = AuthorBuilder::anAuthor()
            ->withEmail(new Email('other@email.com'))
            ->build();

        //do something with author
    }
}
```

We could even combine Test Data Builders to build more complicated aggregates like a *Post*

```
class Post
{
    private $id;
    private $author;
    private $body;
    private $createdAt;

    public function __construct(
        PostId $anId,
        Author $anAuthor,
        Body $aBody)
    {
        $this->id = $anId;
        $this->author = $anAuthor;
        $this->body = $aBody;
        $this->createdAt = new DateTime();
    }
}
```

And the its respective Test Data Builder. We could reuse the `AuthorBuilder` for building a default `Author`

```
class PostBuilder
{
    private $postId;
    private $author;
    private $body;

    private function __construct()
    {
        $this->postId = new PostId();
        $this->author = AuthorBuilder::anAuthor()->build();
        $this->body = new Body('Post body');
    }

    public static function aPost()
    {
        return new self();
    }

    public function withAuthor(Author $anAuthor)
    {
        $this->author = $anAuthor;

        return $this;
    }

    public function withPostId(PostId $aPostId)
    {
        $this->postId = $aPostId;

        return $this;
    }

    public function withBody(Body $body)
    {
        $this->body = $body;

        return $this;
    }

    public function build()
    {
        return new Post($this->postId, $this->author, $this->body);
```

```
    }
}
```

This solution is now flexible enough to adapt our fixtures to any kind of flow in the system under test, including the possibility of building inner entities.

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @test
     */
    public function itDoesSomething()
    {
        $post = PostBuilder::aPost()
            ->withAuthor(AuthorBuilder::anAuthor()
                ->withUsername(new Username('other'))
                ->build())
            ->withBody(new Body('Another body'))
            ->build();

        //do something with the post
    }
}
```

9.5 Wrap-up

Factories are a powerful tool for decoupling construction logic from our business logic. The Factory Method pattern not only helps moving creation responsibility to the Aggregate Root but also could force domain invariants. Using the Abstract Factory pattern in our Services allows us to separate our domain logic from infrastructure creation details. A common use case for this are Specifications and their respective persistence implementations. We've seen that factories come very handy on our test suites too. While we could extract building logic into Object Mother Factories, Test Data Builders provide more flexibility for our tests.

10. Repositories

10.1 Introduction

In order to interact with a domain object you need to hold a reference to it. One way of achieving this is by creation, alternatively you can traverse an association.

In Object-Oriented programs, objects have links (references) to other objects, which make them easily traversable. This contributes greatly to our models expressive power. The catch being, that you need a mechanism to retrieve the first object, the Aggregate Root.

Repositories act as storage locations, where a retrieved object is returned in the exact same state it was persisted in - making them very easy to reason about.

Every Aggregate type typically has a unique associated Repository, used for its persistence needs. In the case however, where it is required to share an Aggregate object hierarchy, the types might share a repository.

Once you have successfully retrieved the Aggregate from the repository, every change you make is persisted. Removing the need to go back to the repository.

10.2 Definition

Martin Fowler defines[1] a Repository as

> the mechanism between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

[1] http://martinfowler.com/eaaCatalog/repository.html

10.3 Repositories are not DAOs

Data Access Objects are a common pattern for persisting domain objects into the database. It is easy to confuse the Data Access Object pattern with a Repository. The significant difference being that Repositories represent collections, whilst DAOs are closer to the database, often being far more table-centric. Typically a DAO would contain CRUD methods for a particular domain object.

A common interface for a DAO could be

```
interface UserDAO
{
    /**
     * @param string $username
     * @return User
     */
    public function get($username);

    public function create(User $user);

    public function update(User $user);

    /**
     * @param string $username
     */
    public function delete($username);
}
```

A DAO interface could have multiple implementations which could range from ORM constructions to plain SQL queries.

The main problem with DAOs is that their responsibilities are not clearly defined. DAOs are usually perceived as a gateway to the database so it is pretty easy to greatly decrease cohesion with many specific methods to query the database.

```
interface BloatUserDAO
{
    public function get($username);

    public function create(User $user);

    public function update(User $user);

    public function delete($username);

    public function getUserByLastName($lastName);

    public function getUserByEmail($email);

    public function updateEmailAddress($username, $email);

    public function updateLastName($username, $lastName);
}
```

As you see, the DAO becomes harder to unit test as you need to implement more methods and it becomes more coupled to the `User` object. This problem will grow over-time, with many other contributors collaborating in making the ball of mud even bigger.

10.4 Collection-Oriented Repositories

Repositories mimic a collection by implementing their common interface characteristics. As a collection it should not leak any intentions of persistence behaviour, such as the notion of saving to a store.

The underlying persistence mechanism has to support for this need. You should not be required to handle changes to the objects over its lifetime. The collection references the most recent changes to the object, meaning that upon each access you get the latest object state.

Repositories implement a concrete collection type, the Set. A Set is a data-structure with the invariant that does not contain duplicate entries. If you try to add an element to a Set that is already present, it will not be added. This is useful in our use-case as each Aggregate has a unique identity that is associated with the Root Entity.

If for example we have the following Domain Model

```
namespace Domain\Model;

class Post
{
    const EXPIRE_EDIT_TIME = 120; // seconds

    private $id;
    private $body;
    private $createdAt;

    public function __construct(
        PostId $anId,
        Body $aBody,
        \DateTime $createdAt = null
    ) {
        $this->id = $anId;
        $this->body = $aBody;
        $this->createdAt = $createdAt ?: new \DateTime();
    }

    public function editBody(Body $aNewBody)
    {
        if ($this->editExpired()) {
            throw new \RuntimeException('Edit time expired');
        }

        $this->body = $aNewBody;
    }

    private function editExpired()
    {
        $expiringTime = $this->createdAt->getTimestamp() + self::EXPIRE_EDIT_TIM\
E;

        return $expiringTime < time();
    }

    public function id()
    {
        return $this->id;
    }
```

```
    public function body()
    {
        return $this->body;
    }

    public function createdAt()
    {
        return $this->createdAt;
    }
}

class Body
{
    const MIN_LENGTH = 3;
    const MAX_LENGTH = 250;

    private $content;

    public function __construct($content)
    {
        $this->setContent(trim($content));
    }

    private function setContent($content)
    {
        $this->assertNotEmpty($content);
        $this->assertFitsLength($content);

        $this->content = $content;
    }

    private function assertNotEmpty($content)
    {
        if (empty($content)) {
            throw new \DomainException('Empty body');
        }
    }

    private function assertFitsLength($content)
    {
        if (strlen($content) < self::MIN_LENGTH) {
            throw new \DomainException('Body is too sort');
```

```
        }

        if (strlen($content) > self::MAX_LENGTH) {
            throw new \DomainException('Body is too long');
        }
    }

    public function content()
    {
        return $this->content;
    }
}

class PostId
{
    private $id;

    public function __construct($id = null)
    {
        $this->id = $id ?: uniqid();
    }

    public function id()
    {
        return $this->id;
    }

    public function equals(PostId $anId)
    {
        return $this->id === $anId->id();
    }
}
```

If we wished to persist this `Post` entity, a simple in-memory `Post` Repository could be created like the following

```
class SimplePostRepository
{
    private $post = [];

    public add(Post $aPost)
    {
        $this->posts[(string) $aPost->id()] = $aPost;
    }

    public function postOfId(PostId $anId)
    {
        if (isset($this->posts[(string) $anId])) {
            return $this->posts[(string) $anId];
        }

        return null;
    }
}
```

And, as you would expect it is handled as a collection

```
$id = new PostId();
$repository = new SimplePostRepository();
$repository->add(new Post($id, 'Random content'));

// later ...
$post = $repository->postOfId($id);
$post->editBody('Updated content');

// even later ...
$post = $repository->postOfId($id);
assert('Updated content' === $post->body());
```

As you can see, from the collections point of view there is no need for a `save` method in the repository. Changes affecting the object are correctly handled by the underlying persistence layer.

The first step is to define a collection-like interface for your repository. The interface needs to define the usual collection methods, as following.

```
interface PostRepository
{
    public function add(Post $aPost);
    public function addAll(array $posts);
    public function remove(Post $aPost);
    public function removeAll(array $posts);
    // ...
}
```

The interface definition should be placed in the same module that the Aggregate uses to store.

Sometimes `remove` does not provide true Aggregate removal. There are times where you need to keep the information for legal purposes or business intelligence. In those cases, you can instead mark the Aggregate as disabled or *logically removed*. The interface could be updated accordingly, removing the removal methods or providing disable behaviour in the repository.

Another important part of repositories are the finder methods such as.

```
interface PostRepository
{
    // ...

    /**
     * @return Post
     */
    public function postOfId(PostId $anId);

    /**
     * @return Post[]
     */
    public function latestPosts(\DateTime $sinceADate);
}
```

And, to retrieve the globally unique id for a `Post`, a logical place to include it is

```
interface PostRepository
{
    // ...

    /**
     * @return PostId
     */
    public function nextIdentity();
}
```

The code responsible for building up each `Post` instance calls `nextIdentity` to get the unique identifier `PostId`.

```
$post = new Post($postRepository->nextIdentity(), $body);
```

Some developers favour placing the implementation close to the interface definition, as a sub-package of the module. However, because we want a clear separation of concerns, we recommend to place it inside the infrastructure layer instead.

10.4.1 In-Memory Implementation

As Uncle Bob wrote in Screaming Architecture[2]

> A good software architecture allows decisions about frameworks, databases, web-servers, and other environmental issues and tools, to be deferred and delayed. A good architecture makes it unnecessary to decide on Rails, or Spring, or Hibernate, or Tomcat or MySql, until much later in the project. A good architecture makes it easy to change your mind about those decisions too. A good architecture emphasizes the use-cases and decouples them from peripheral concerns.

At the early stages of your application, a fast in-memory implementation could come in handy. It is something you could use to mature other parts of your system allowing you to delay database decisions to the right moment. An in-memory repository is simple, fast and easy to implement.

For our `Post` repository an in-memory hash-map is enough to provide all the functionality we need.

[2] http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html

```
namespace Infrastructure\Persistence\InMemory;

use Domain\Model\Post;
use Domain\Model\PostId;
use Domain\Model\PostRepository;

class InMemoryPostRepository implements PostRepository
{
    private $posts = [];

    public function add(Post $aPost)
    {
        $this->posts[$aPost->id()->id()] = $aPost;
    }

    public function remove(Post $aPost)
    {
        unset($this->posts[$aPost->id()->id()]);
    }

    public function postOfId(PostId $anId)
    {
        if (isset($this->posts[$anId->id()])) {
            return $this->posts[$anId->id()];
        }

        return null;
    }

    public function latestPosts(\DateTime $sinceADate)
    {
        return $this->filterPosts(
            function(Post $post) use ($sinceADate) {
                return $post->createdAt() > $sinceADate;
            }
        );
    }

    private function filterPosts(callable $fn)
    {
        return array_values(array_filter($this->posts, $fn));
    }
```

```
    public function nextIdentity()
    {
        return new PostId();
    }
}
```

10.4.2 Doctrine ORM

Doctrine[3] is a set of libraries for database storage and object mapping. It comes bundled with the popular Symfony 2 web framework[4] by default and, among other features, it allows you to decouple your application from the persistence layer easily thanks to the Data Mapper pattern[5].

The Object Relational Mapper stands over a powerful database abstraction layer that enables database interaction through a SQL dialect called Doctrine Query Language, inspired by the famous Java Hibernate framework.

If we are going to use Doctrine ORM the first task to complete is adding the dependencies to our project through Composer[6]

```
composer require doctrine/orm:~2.4
```

10.4.2.1 Object Mapping

The mapping between your domain objects and the database can be considered an implementation detail. The domain life-cycle should not be aware of these persistence details. As such, the mapping information should be defined as part of the infrastructure layer, outside the domain and as the implementation for the repositories.

10.4.2.1.1 Doctrine Custom Mapping Types

As our `Post` entity is composed of Value Objects like `Body` or `PostId`, it is a good idea to make Custom Mapping Types for them. This will make the object mapping considerably easier.

[3] http://www.doctrine-project.org/
[4] http://symfony.com/
[5] http://martinfowler.com/eaaCatalog/dataMapper.html
[6] https://getcomposer.org/

```
namespace Infrastructure\Persistence\Doctrine\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;
use Domain\Model\Body;

class BodyType extends Type
{
    public function getSQLDeclaration(array $fieldDeclaration, AbstractPlatform \
$platform)
    {
        return $platform->getVarcharTypeDeclarationSQL($fieldDeclaration);
    }

    /**
     * @param string $value
     * @return Body
     */
    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return new Body($value);
    }

    /**
     * @param Body $value
     */
    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        return $value->content();
    }

    public function getName()
    {
        return 'body';
    }
}
```

```
namespace Infrastructure\Persistence\Doctrine\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;
use Domain\Model\PostId;

class PostIdType extends Type
{
    public function getSQLDeclaration(array $fieldDeclaration, AbstractPlatform \
$platform)
    {
        return $platform->getGuidTypeDeclarationSQL($fieldDeclaration);
    }

    /**
     * @param string $value
     * @return PostId
     */
    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return new PostId($value);
    }

    /**
     * @param PostId $value
     */
    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        return $value->id();
    }

    public function getName()
    {
        return 'post_id';
    }
}
```

Don't forget to implement the `__toString` magic method to the `PostId` Value Object, as Doctrine requires this.

```
class PostId
{
    // ...

    public function __toString()
    {
        return $this->id;
    }
}
```

10.4.2.1.2 XML Mapping

Doctrine offers multiple formats for the mapping like YAML, XML or annotations. XML is our preferred choice as it provides robust IDE auto-completion.

```
<?xml version="1.0" encoding="UTF-8"?>
<doctrine-mapping
        xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-map\
ping
                            http://raw.github.com/doctrine/doctrine2/master/doct\
rine-mapping.xsd">

    <entity name="Domain\Model\Post" table="posts">
        <id name="id" type="post_id" column="id">
            <generator strategy="NONE" />
        </id>
        <field name="body" type="body" length="250" column="body"/>
        <field name="createdAt" type="datetime" column="created_at" />
    </entity>

</doctrine-mapping>
```

10.4.2.2 Entity Manager

The `EntityManager` is the central access point for the ORM functionality, bootstraping it is easy as

```
use Doctrine\DBAL\Types\Type;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Tools;

Type::addType('post_id', 'Infrastructure\Persistence\Doctrine\Types\PostIdType');
Type::addType('body', 'Infrastructure\Persistence\Doctrine\Types\BodyType');

$entityManager = EntityManager::create(
    [
        'driver' => 'pdo_sqlite',
        'path' => __DIR__ . '/db.sqlite',
    ],
    Tools\Setup::createXMLMetadataConfiguration(
        ['/Path/To/Infrastructure/Persistence/Doctrine/Mapping'],
        $devMode = true
    )
);
```

Remember to configure as per your needs and setup.

10.4.2.3 DQL Implementation

In the case of this repository we will only need the `EntityManager` to retrieve domain objects directly from the database

```
namespace Infrastructure\Persistence\Doctrine;

use Doctrine\ORM\EntityManager;
use Domain\Model\Post;
use Domain\Model\PostId;
use Domain\Model\PostRepository;

class DoctrinePostRepository implements PostRepository
{
    protected $em;

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

    public function add(Post $aPost)
```

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

    public function remove(Post $aPost)
    {
        $this->em->remove($aPost);
    }

    public function postOfId(PostId $anId)
    {
        return $this->em->find('Domain\Model\Post', $anId);
    }

    public function latestPosts(\DateTime $sinceADate)
    {
        return $this->em->createQueryBuilder()
            ->select('p')
            ->from('Domain\Model\Post', 'p')
            ->where('p.createdAt > :since')
            ->setParameter(':since', $sinceADate)
            ->getQuery()
            ->getResult();
    }

    public function nextIdentity()
    {
        return new PostId();
    }
}
```

10.5 Persistence-Oriented

There are times when collection-oriented repositories do not fit well with our persistence mechanism. If you do not have a unit of work, keeping track of Aggregate changes is a difficult task. The only way to persist such changes is by explicitly calling `save`.

The interface definition for a persistence-oriented repository is similar to how you would define a collection-oriented equivalent.

```
interface PostRepository
{
    public function nextIdentity();
    public function postOfId(PostId $anId);
    public function save(Post $aPost);
    public function saveAll(array $posts);
    public function remove(Post $aPost);
    public function removeAll(array $posts);
}
```

In this case we now have `save` and `saveAll` methods. They provide similar functionality to the previous `add` and `addAll` methods, however, the important difference is how the client uses them. Within a collection-oriented style, you use the `add` methods just once, when the Aggregate is created. In a persistence-oriented style, you will not only use the `save` action after creating the Aggregate, but also when they are modified.

```
$post = new Post(/* ... */);
$postRepository->save($post);

// later ...
$post = $postRepository->postOfId($postId);
$post->editBody(new Body('New body!'));
$postRepository->save($post);
```

Other than this difference, the details are just in the implementation.

10.5.1 Redis Implementation

Redis[7] is an in-memory blazing-fast key-value that can be used as a cache or store.

Depending on the circumstances we could consider using Redis as a store for our Aggregates.

To get started, make sure you have a PHP client to connect to Redis. A good one is Predis[8].

```
composer require predis/predis:~1.0
```

[7] http://redis.io/

[8] https://github.com/nrk/predis

```php
namespace Infrastructure\Persistence\Redis;

use Domain\Model\Post;
use Domain\Model\PostId;
use Domain\Model\PostRepository;
use Predis\Client;

class RedisPostRepository implements PostRepository
{
    private $client;

    public function __construct(Client $client)
    {
        $this->client = $client;
    }

    public function save(Post $aPost)
    {
        $this->client->hset('posts', (string) $aPost->id(), serialize($aPost));
    }

    public function remove(Post $aPost)
    {
        $this->client->hdel('posts', (string) $aPost->id());
    }

    public function postOfId(PostId $anId)
    {
        if ($data = $this->client->hget('posts', (string) $anId)) {
            return unserialize($data);
        }

        return null;
    }

    public function latestPosts(\DateTime $sinceADate)
    {
        $latest = $this->filterPosts(
            function(Post $post) use ($sinceADate) {
                return $post->createdAt() > $sinceADate;
            }
        );
```

```
        $this->sortByCreatedAt($latest);

        return array_values($latest);
    }

    private function filterPosts(callable $fn)
    {
        return array_filter(array_map(function($data) {
            return unserialize($data);
        }, $this->client->hgetall('posts')), $fn);
    }

    private function sortByCreatedAt(&$posts)
    {
        usort($posts, function(Post $a, Post $b) {
            if ($a->createdAt() == $b->createdAt()) {
                return 0;
            }
            return ($a->createdAt() < $b->createdAt()) ? -1 : 1;
        });
    }

    public function nextIdentity()
    {
        return new PostId();
    }
}
```

10.5.2 SQL Implementation

In a classic example, we could create a simple PDO[9] implementation for our `PostRepository` just by using plain SQL queries.

[9] http://php.net/manual/en/book.pdo.php

```
namespace Infrastructure\Persistence\Sql;

use Domain\Model\Body;
use Domain\Model\Post;
use Domain\Model\PostId;
use Domain\Model\PostRepository;

class SqlPostRepository implements PostRepository
{
    const DATE_FORMAT = 'Y-m-d H:i:s';

    private $pdo;

    public function __construct(\PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function save(Post $aPost)
    {
        $sql = 'INSERT INTO posts (id, body, created_at) VALUES (:id, :body, :cr\
eated_at)';

        $this->execute($sql, [
            'id' => $aPost->id()->id(),
            'body' => $aPost->body()->content(),
            'created_at' => $aPost->createdAt()->format(self::DATE_FORMAT)
        ]);
    }

    private function execute($sql, array $parameters)
    {
        $st = $this->pdo->prepare($sql);

        $st->execute($parameters);

        return $st;
    }

    public function remove(Post $aPost)
    {
        $this->execute('DELETE FROM posts WHERE id = :id', [
```

```
            'id' => $aPost->id()->id()
        ]);
    }

    public function postOfId(PostId $anId)
    {
        $st = $this->execute('SELECT * FROM posts WHERE id = :id', [
            'id' => $anId->id()
        ]);

        if ($row = $st->fetch(\PDO::FETCH_ASSOC)) {
            return $this->buildPost($row);
        }

        return null;
    }

    private function buildPost($row)
    {
        return new Post(
            new PostId($row['id']),
            new Body($row['body']),
            new \DateTime($row['created_at'])
        );
    }

    public function latestPosts(\DateTime $sinceADate)
    {
        return $this->retrieveAll('SELECT * FROM posts WHERE created_at > :since\
_date', [
            'since_date' => $sinceADate->format(self::DATE_FORMAT)
        ]);
    }

    private function retrieveAll($sql, array $parameters = [])
    {
        $st = $this->pdo->prepare($sql);

        $st->execute($parameters);

        return array_map(function ($row) {
            return $this->buildPost($row);
```

```
        }, $st->fetchAll(\PDO::FETCH_ASSOC));
    }

    public function nextIdentity()
    {
        return new PostId();
    }

    public function size()
    {
        return $this->pdo->query('SELECT COUNT(*) FROM posts')
            ->fetchColumn();
    }
}
```

As we do not have any mapping configuration, it would be very useful to have an initialisation method for the schema within the same class. *Things that change together should remain together.*

```
class SqlPostRepository implements PostRepository
{
    // ...
    public function initSchema()
    {
        $this->pdo->exec(<<<SQL
DROP TABLE IF EXISTS posts;

CREATE TABLE posts (
    id CHAR(36) PRIMARY KEY,
    body VARCHAR(250) NOT NULL,
    created_at DATETIME NOT NULL
)
SQL
        );
    }
}
```

10.6 Extra Behaviour

Adding additional behaviour to a repository can be very beneficial, such as the ability to count all the items in a given collection. You might think to add a method with the name `count` however, as we are trying to mimic a collection, a better name would instead be `size`.

```
interface PostRepository
{
    // ...

    public function size();
}
```

And the implementation could look like

```
class DoctrinePostRepository implements PostRepository
{
    // ...

    public function size()
    {
        return $this->em->createQueryBuilder()
            ->select('count(p.id)')
            ->from('Domain\Model\Post', 'p')
            ->getQuery()
            ->getSingleScalarResult();
    }
}
```

You are able to also encapsulate calculations into the repository, along with data-storage specific and optimised queries/stored procedures. All behaviour should still however, follow the repositories collection characteristics. It is encouraged to move as much logic into domain-specific stateless Domain Services as possible, instead of simply adding these responsibilities to the repository.

In some instances you will not require the entire Aggregate for simply accessing small amounts of information. To solve this you can add repository methods to access these as shortcuts. You should make sure to only access data that could be retrieved by navigating through the Aggregate Root. As such you should not allow access to the Aggregate Roots private and internal areas, as this would violate the laid out contractual agreement.

For some use cases you will require very specific queries that are compositions of multiple Aggregate types, each returning specific information. These queries can be run and then returned as a single Value Object. It is very common for repositories to return Value Objects.

If you find yourself creating many use-case optimal finder methods, you may be introducing a common code smell. This could be an indication of a misjudged Aggregate boundary. If however, you are confident that the boundaries are correct, it could be time to explore CQRS.

10.7 Querying Repositories

Upon comparison, repositories are different than a Collection if we consider their querying ability. A Repository deals with a large set of objects that typically are not in memory when the query is performed. It is not feasible to load all the instances of a domain object in memory and perform a query over them.

A good solution is to pass a criterion and let the Repository handle the implementation details to successfully perform the operation. It might translate the criterion to SQL, ORM queries or iterate over an in-memory collection, it does not matter, the implementation deals with it.

10.7.1 Specification Pattern

A common implementation for the criterion object is the Specification Pattern. A specification is just a simple predicate that takes a domain object and returns a boolean. Given a domain object, it will return true if it *specifies* the specification and false otherwise.

```
interface PostSpecification
{
    /**
     * @return boolean
     */
    public function specifies(Post $aPost);
}
```

We just need to add a query method to our repository.

```
interface PostRepository
{
    //...

    public function query($specification);
}
```

10.7.1.1 In-Memory Implementation

As an example, if we wanted to replicate the `latestPosts` query method in our `PostRepository` by using a Specification for an in-memory implementation

```
namespace Infrastructure\Persistence\InMemory;

use Domain\Model\Post;

interface InMemoryPostSpecification
{
    /**
     * @return boolean
     */
    public function specifies(Post $aPost);
}
```

The in-memory implementation for the `latestPosts` behaviour could be as follows

```
namespace Infrastructure\Persistence\InMemory;

use Domain\Model\Post;

class InMemoryLatestPostSpecification implements InMemoryPostSpecification
{
    private $since;

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

    public function specifies(Post $aPost)
    {
        return $aPost->createdAt() > $this->since;
    }
}
```

The `query` method for our repository implementation could look as follows

```
class InMemoryPostRepository implements PostRepository
{
    // ...

    /**
     * @param InMemoryPostSpecification $specification
     *
     * @return Post[]
     */
    public function query($specification)
    {
        return $this->filterPosts(
            function(Post $post) use ($specification) {
                return $specification->specifies($post);
            }
        );
    }
}
```

Retrieving all the latest posts from the repository is as simple as creating a tailored instance of the above implementation

```
$latestPosts = $postRepository->query(
    new InMemoryLatestPostSpecification(new \DateTime('-24'))
);
```

10.7.1.2 SQL Implementation

A standard Specification works well for in-memory implementations. However, as we do not pre-load all the domain objects in-memory for a SQL implementation, we need a more specific specification for these cases.

```
namespace Infrastructure\Persistence\Sql;

interface SqlPostSpecification
{
    /**
     * @return string
     */
    public function toSqlClauses();
}
```

The SQL implementation for this specification could look like

```
namespace Infrastructure\Persistence\Sql;

class SqlLatestPostSpecification implements SqlPostSpecification
{
    private $since;

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

    public function toSqlClauses()
    {
        return "created_at > '" . $this->since->format('Y-m-d H:i:s') . "'";
    }
}
```

And how to query an SQL Post repository implementation

```
class SqlPostRepository implements PostRepository
{
    //...

    /**
     * @param SqlPostSpecification $specification
     *
     * @return Post[]
     */
    public function query($specification)
    {
        return $this->retrieveAll(
            'SELECT * FROM posts WHERE ' . $specification->toSqlClauses()
        );
    }

    private function retrieveAll($sql, array $parameters = [])
    {
        $st = $this->pdo->prepare($sql);

        $st->execute($parameters);

        return array_map(function ($row) {
```

```
            return $this->buildPost($row);
        }, $st->fetchAll(\PDO::FETCH_ASSOC));
    }
}
```

10.8 Managing Transactions

The Domain Model is not the place to manage transactions. The operations applied over the Domain Model should be agnostic of the persistence mechanism. A common approach to solving this problem is placing a Facade[10] in the Application Layer, grouping related Use Cases together. When a method of the Facade is invoked from the User Interface Layer, the business method begins a transaction. Once complete, the Facade ends the interaction by committing the transaction. If anything goes wrong, the transaction is rolled back.

```
use Doctrine\ORM\EntityManager;

class SomeApplicationServiceFacade
{
    private $em;

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

    public function doSomeUseCaseTask()
    {
        try {
            $this->em->getConnection()->beginTransaction();

            // Use domain model

            $this->em->getConnection()->commit();
        } catch(Exception $e) {
            $this->em->getConnection()->rollback();
            throw $e;
        }
    }
}
```

[10] http://en.wikipedia.org/wiki/Facade_pattern

The problem introduced with facades is that we have to repeat the same boilerplate code over and over. If we unify the way we execute use cases, we could wrap them in a transaction using the Decorator Pattern[11]

```
interface ApplicationService
{
    /**
     * @param $request
     * @return mixed
     */
    public function execute($request = null);
}

class SomeApplicationService implements ApplicationService
{
    public function execute($request = null)
    {
        // do something
    }
}
```

We do not want to couple our Application Layer with the concrete transactional procedure, so instead we can create a simple interface for it

```
interface TransactionalSession
{
    /**
     * @param  callable $operation
     * @return mixed
     */
    public function executeAtomically(callable $operation);
}
```

The implemented decorator that can make any application service transactional is as easy as the following

[11] http://en.wikipedia.org/wiki/Decorator_pattern

```
class TransactionalApplicationService implements ApplicationService
{
    private $session;
    private $service;

    public function __construct(
        ApplicationService $service,
        TransactionalSession $session
    ) {
        $this->session = $session;
        $this->service = $service;
    }

    public function execute($request = null)
    {
        $operation = function () use ($request) {
            return $this->service->execute($request);
        };

        return $this->session->executeAtomically($operation->bindTo($this));
    }
}
```

Following this, we could alternatively create a Doctrine transactional session implementation

```
class DoctrineSession implements TransactionalSession
{
    private $entityManager;

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

    public function executeAtomically(callable $operation)
    {
        return $this->entityManager->transactional($operation);
    }
}
```

Now we have everything to execute our Use Cases within a transaction

```
$useCase = new TransactionalApplicationService(
    new SomeApplicationService(
        // ...
    ),
    new DoctrineSession(
        // ...
    )
);

$response = $useCase->execute();
```

10.9 Testing Repositories

In order to be sure that the repository will work in production, you will need to test its implementation. To do this we have to test the boundaries of the system making sure that our expectations are correct.

In the case of a Doctrine test, the setup will be a little bit more sophisticated

```
use Doctrine\DBAL\Types\Type;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Tools;
use Domain\Model\Post;

class DoctrinePostRepositoryTest extends \PHPUnit_Framework_TestCase
{
    private $postRepository;

    public function setUp()
    {
        $this->postRepository = $this->createPostRepository();
    }

    private function createPostRepository()
    {
        $this->addCustomTypes();
        $em = $this->initEntityManager();
        $this->initSchema($em);

        return new PrecociousDoctrinePostRepository($em);
    }
```

```
    private function addCustomTypes()
    {
        if (!Type::hasType('post_id')) {
            Type::addType('post_id', 'Infrastructure\Persistence\Doctrine\Types\\
PostIdType');
        }

        if (!Type::hasType('body')) {
            Type::addType('body', 'Infrastructure\Persistence\Doctrine\Types\Bod\
yType');
        }
    }

    protected function initEntityManager()
    {
        return EntityManager::create(
            ['url' => 'sqlite:///:memory:'],
            Tools\Setup::createXMLMetadataConfiguration(
                ['/Path/To/Infrastructure/Persistence/Doctrine/Mapping'],
                $devMode = true
            )
        );
    }

    private function initSchema(EntityManager $em)
    {
        $tool = new Tools\SchemaTool($em);
        $tool->createSchema([
            $em->getClassMetadata('Domain\Model\Post')
        ]);
    }

    //...
}

class PrecociousDoctrinePostRepository extends DoctrinePostRepository
{
    public function persist(Post $aPost)
    {
        parent::persist($aPost);

        $this->em->flush();
```

```
    }

    public function remove(Post $aPost)
    {
        parent::remove($aPost);

        $this->em->flush();
    }
}
```

Once we have this environment setup, we can now continue to test the Repository's behaviour

```
class DoctrinePostRepositoryTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function itShouldRemovePost()
    {
        $post = $this->persistPost('irrelevant body');

        $this->postRepository->remove($post);

        $this->assertPostExist($post->id());
    }

    private function assertPostExist($id)
    {
        $result = $this->postRepository->postOfId($id);
        $this->assertNull($result);
    }

    private function persistPost($body, \DateTime $createdAt = null)
    {
        $this->postRepository->add(
            $post = new Post(
                $this->postRepository->nextIdentity(),
                new Body($body),
                $createdAt
            )
```

```
        );

        return $post;
    }
}
```

Following our assertion made earlier, if we save a `Post`, we expect to find it in the exact same state.

Now we can move on to test finding the latest posts specifying a given date

```
class DoctrinePostRepositoryTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function itShouldFetchLatestPosts()
    {
        $this->persistPost('a year ago', new \DateTime('-1 year'));
        $this->persistPost('a month ago', new \DateTime('-1 month'));
        $this->persistPost('few hours ago', new \DateTime('-3 hours'));
        $this->persistPost('few minutes ago', new \DateTime('-2 minutes'));

        $posts = $this->postRepository->latestPosts(new \DateTime('-24 hours'));

        $this->assertCount(2, $posts);
        $this->assertEquals('few hours ago', $posts[0]->body()->content());
        $this->assertEquals('few minutes ago', $posts[1]->body()->content());
    }
}
```

10.10 Testing your Services with In-Memory Implementations

Setting up a fully persistent Repository implementation can be too complex, and result in slow execution. You should care about keeping your tests fast. Going through the whole database setup and querying will slow you down enormously.

Having an in-memory implementation could help delaying persistence decisions until the end.

We could test it in the same manner we did before but this time with a full-featured fast and simple in-memory implementation.

```
class MyServiceTest extends \PHPUnit_Framework_TestCase
{
    private $service;

    public function setUp()
    {
        $this->service = new MyServiceTest(new InMemoryPostRepository());
    }

    // ...
}
```

10.11 Wrap-up

A Repository is a mechanism that acts as a storage location. The difference between a DAO and a Repository is that a DAO follows a database-first approach, decreasing cohesion with many low-level methods to query the database. Depending on the underlying persistence mechanics, we've seen different Repository approaches:

- **Collection-Oriented repositories** tend to be more pure to the domain model, even if they persist entities. From the client's point of view, it looks like a collection (Set). There's no need for explicit persistence calls on Entity updates, as the repository tracks changes on the objects. We explored Doctrine as the underlying persistence mechanism for this type of repository as it provides automatic changes monitoring on objects (Unit of Work).
- **Persistence-Oriented repositories** require explicit persistence calls as they don't track object changes. We explored Redis and plain SQL implementations.

Along the way, we discovered Specifications as a pattern that help us querying the database without sacrificing flexibility and cohesion. We also studied how to manage Transactions and how to test our services with simple and fast in-memory Repository implementations.

11. Application

11.1 Introduction

The Application layer is the area that separates the Domain Model from the clients that query or change its state. Application Services are the building blocks for such layer. As Vaughn Vernon says, "Application Services are the direct clients of the domain model". You could think about an Application Service as a point of contact between the outside world (html forms, API clients, command line, frameworks, UI, etc.) and the Domain Model itself. It might help thinking about the top level use cases that your system exposes to the world "as guest, I want to register", "as a logged user, I want to purchase a product", etc.

In this chapter, we will explore how to implement Application Services, understanding the role of the *Command Pattern* and establishing the responsibilities of an Application Service. Consider the use case of *signing up a new user*.

Conceptually, in order to register a new user we need to:

- Get an email and password from the client
- Check if the email is already in use
- Create a new user
- Add this new user to the existing user set
- Return the user we've just created

Let's go for it.

11.2 Requests

We need to send the *email* and *password* to the Application Service. There are many ways of doing such a thing from the client (HTML form, API client or even the command-line). We could just send standard parameters (email and password) through the method signature or build and send a data structure with this information. The later approach, sending a DTO[1], bring some interesting features to the table. By sending an object, it will be possible to serialise and queue it over a command bus. It will be possible to add type safety and some IDE help too.

[1] http://martinfowler.com/eaaCatalog/dataTransferObject.html

Data Transfer Object

A *DTO* is a data structure that carries information between processes. Don't mistake it with a full-featured object. A DTO does not have any behavior except for storage and retrieval of its own data (accessors and mutators). DTOs are simple objects that should not contain any business logic that would require testing.

As Vaughn Vernon says

> Application Service method signatures use only primitive types (int, strings, etc.), and possibly DTOs. As an alternative to these approaches, however, a better approach may be to design Command objects instead. There is not necessarily a right or wrong way. It mostly depends on your tastes and goals.

The implementation for a DTO that holds the data required for the Application Service could be something like

```php
namespace Lw\Application\Service\User;

class SignUpUserRequest
{
    private $email;
    private $password;

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

    public function email()
    {
        return $this->email;
    }

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

As you see, `SignUpUserRequest` does not have behaviour at all, only data. This could have come from an HTML form or an API end-point, we don't care.

11.2.1 Building Application Service Requests

Creating a request from the delivery mechanism, your favourite framework, should be pretty straightforward. On web, you could pick up parameters from the controller request and pass them down to the service inside a DTO. Same principle applies for a CLI command, read input parameters and send them down again.

With Symfony, we can extract the data we need from `Request` object from the `HttpFoundation` component.

```
//...
class UsersController extends Controller
{
    /**
     * @Route("/signup", name="signup")
     * @param Request $request
     * @return Response
     */
    public function signUpAction(Request $request)
    {
        //...
        $SignUpUserRequest = new SignUpUserRequest(
            $request->get('email'),
            $request->get('password')
        );
        //...
    }
//...
```

On a more elaborated Silex application that uses the `Form` component to capture and validate parameters

```
//...
$app->match('/signup', function (Request $request) use ($app) {
    $form = $app['sign_up_form'];
    $form->handleRequest($request);

    if ($form->isValid()) {
        $data = $form->getData();

        try {
            $app['sign_in_user_application_service']->execute(
                new SignUpUserRequest(
```

```
                $data['email'],
                $data['password']
            )
        );

        return $app->redirect($app['url_generator']->generate('login'));
    } catch (UserAlreadyExistsException $e) {
        $form
            ->get('email')
            ->addError(
                new FormError(
                    'Email is already registered by another user'
                )
            );
    } catch (\Exception $e) {
        $form
            ->addError(
                new FormError(
                    'There was an error, please get in touch with us'
                )
            );
    }
}

return $app['twig']->render('signup.html.twig', [
    'form' => $form->createView(),
]);
});
```

11.2.2 Request Design

When designing your request objects, you should always follow these principles: use primitives, design for serialization and don't include business logic inside them. This way you will be able to save unit testing dollars.

11.2.2.1 Use Primitives

We recommend using just basic types to build up your request objects. That means string, integers, booleans, and so on. We are just abstracting away input parameters. You should be able to consume Application Services independently from the delivery mechanism. Even pretty complicated HTML forms get translated into basic types all the time at the controller level. You don't want to mangle your framework and your business logic together.

On some scenarios is tempting to use Value Objects directly. Don't do it, updates on the Value Object definition will affect all clients. You'll be coupling clients with your Domain logic.

11.2.2.2 Serializable

A cool side effect of using basic types is that any request object can be easily serialized into a string and sent through the wire and stored in a messaging system or database.

11.2.2.3 No Business Logic

Avoid to put any business logic inside you request objects. Not even validation. Validation should happen inside your Domain – this is inside your Entities, Value Objects, Domain Services, etc. – as business invariants and constraints.

11.2.2.4 No Tests

Application requests are data structures not objects. Unit testing data structures is like testing getters and setters. There is no behaviour to test so there is not that much value trying to unit testing them. This structures will be covered as a side-effect of more elaborated tests like Integration or Acceptance tests.

An alternative to request objects are *Commands*. We could design an Service with multiple Application methods. Each one of them with the parameters you'd put inside the Request. It's okay for simple applications. No worries, we'll come back to this topic later.

11.3 Anatomy of an Application Service

Once we have the data encapsulated in a request, it's time for the business logic. As Vaughn Vernon says: "Keep Application Services thin, using them only to coordinate tasks on the model".

The first thing to do is to extract the necessary information from the request, this is the `email` and `password`. At a high level then we need if there is an existing user with that email already. If it's not the case, we then create and add the user to the `UserRepository`. On the special case of finding a user with the same email, we raise an exception so the client could treat it their own way, displaying an error, retrying or just ignoring it.

```
namespace Lw\Application\Service\User;

use Ddd\Application\Service\ApplicationService;
use Lw\Domain\Model\User\User;
use Lw\Domain\Model\User\UserAlreadyExistsException;
use Lw\Domain\Model\User\UserRepository;

class SignUpUserService
{
    private $userRepository;

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

    public function execute(SignUpUserRequest $request)
    {
        $email = $request->email();
        $password = $request->password();

        $user = $this->userRepository->ofEmail($email);
        if (!$user) {
            throw new UserAlreadyExistsException();
        }

        $this->userRepository->add(
            new User(
               $this->userRepository->nextIdentity(),
               $email,
               $password
            )
        );
    }
}
```

Nice! If you are wondering what is this `UserRepository` thing doing in the constructor, we'll see next.

Handling Exceptions

Exceptions raised by application services are a way of communicating unusual cases and flows to the client. Exceptions on this layer are related with business logic (like not finding a user), not with implementation details like `PDOException`, `PredisException` or `DoctrineException`.

11.3.1 Dependency Inversion

Handling users is not responsibility of the service. As we've seen in the repositories chapter, there is a specialised class that deals with `User` collections, the `User` repository. This is a dependency from the Application Service to the repository. We don't want to couple the Application Service with a concrete implementation of the Repository as then, we would be coupling our service with infrastructure details. So we depend on the contract (interface) that concrete implementations depend on, the `UserRepository`.

This dependency will be fulfilled and injected at runtime with a concrete implementation like a `DoctrineUserRepository` or even a `InMemoryUserRepository` on test environment.

Application services could depend on Domain services like `GetBadgesByUser` too. At runtime the implementation for such a service could be quite elaborated. Imagine a `HttpGetBadgesByUser` for integrating a bounded context through HTTP protocol.

Depending on abstractions will make our Application Service immune to low-level infrastructure changes.

11.3.2 Instantiating Application Services

Instantiating just your Application Service is easy. Building the dependency tree might be tricky depending on how complicated are the dependencies to build. For such a purpose, most frameworks come with a Dependency Injection container. Without one, you'll end up with something like the following code somewhere in your controller.

```
$redisClient = new Predis\Client([
    'scheme' => 'tcp',
    'host'   => '10.0.0.1',
    'port'   => 6379
]);

$userRepository = new RedisUserRepository($redisClient);
$signUp = new SignUpUserService($userRepository);
$signUp->execute(new SignUpUserRequest(
    'user@example.com',
```

```
    'password'
));
```

We decided to use the Redis[2] implementation for the `UserRepository`. There are all the details we need to build that repository right there, like Predis[3] configuration. This is not only inefficient it also spreads duplication across controllers.

You could refactor the construction logic into a Factory or you could use a Dependency Injection Container. Most of modern frameworks come with it.

Is it bad to use a Dependency Injection Container?

Not at all. Dependency Injection Containers are just a tool. They help abstracting away the complexities of building your dependencies. They come very handy for building infrastructure artifacts. Symfony offers a complete solution[4].

Take into account that passing the entire container as a whole to one of services is a bad practice. That would be like coupling the entire context of your application with the domain. If you fetch an object for a specific service, do it. Don't make that service aware of the entire context.

Let's see how would we do it with Silex

```
// ...

$app = new \Silex\Application();
$app['redis_parameters'] = [
    'scheme' => 'tcp',
    'host'   => '127.0.0.1',
    'port'   => 6379
];

$app['redis'] = $app->share(function ($app) {
    return new Predis\Client($app['redis_parameters']);
});

$app['user_repository'] = $app->share(function ($app) {
```

[2] http://redis.io/
[3] https://github.com/nrk/predis
[4] http://symfony.com/doc/current/book/service_container.html

```
    return new RedisUserRepository(
        $app['redis']
    );
});

$app['sign_up_user_application_service'] = $app->share(function ($app) {
    return new SignUpUserService(
        $app['user_repository']
    );
});

//...

$app->match('/signup', function (Request $request) use ($app) {
    //...
    $app['sign_up_user_application_service']->execute(
        new SignUpUserRequest(
            $request->get('email'),
            $request->get('password')
        )
    );
    //...
});
```

As you can see, `$app` is used as the Service Container. We register all the components needed and their dependencies. `sign_up_user_application_service` depends on the definitions made above. Changing the implementation for the `user_repository` is a easy as returning something else (MySQL, MongoDB, etc.), we don't need to change the service code at all.

The equivalent for a Symfony application

```
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services
           http://symfony.com/schema/dic/services/services-1.0.xsd">
    <services>
        <service
            id="sign_up_user_application_service"
            class="SignUpUserService">
            <argument type="service" id="user_repository" />
        </service>
```

```
        <service
            id="user_repository"
            class="RedisUserRepository">
            <argument type="service">
                <service class="Predis\Client" />
            </argument>
        </service>
    </services>
</container>
```

Now that you have the definition of your Application Service in the Symfony Service container, getting the service at any context is pretty straightforward. All delivery mechanisms share the same definition, Web Controllers, REST controllers or even Console Commands. The Service is available on any class implementing the `ContainerAware` interface. Getting the service is as easy as calling `$this->get('sign_up_user_application_service')`.

To sum up, how do you build your services (ad-hoc, using Service Container, Factories, etc.) does not matter. It's important to keep it out to the Infrastructure boundary.

11.3.3 Execution

There are two different approaches for invoking Application Services. A dedicated class per use case with a single execution method or a multiple application services – or use case execution methods – inside the same class.

11.3.3.1 One Class per Application Service

Our preferred approach, probably the one that fits all scenarios.

```
class SignUpUserService
{
    // ...

    public function execute(SignUpUserRequest $request)
    {
        // ...
    }
}
```

Using a dedicated class per Application Service makes it more robust to external changes (Single Responsibility Principle). There are fewer reasons to change the class as the service does one and only one thing. The Application Service will be easier to test as it does less things. It's easier to implement a common Application Service contract, making class decoration easier (check out transactions in repositories chapter). There is greater cohesion with the dependencies as all dependencies are dedicated the Application Service.

The execution method could have a more expressive name, like `signUp`. However, the `execute` Command Pattern[5] format standardises a common contract across Application Services enabling easy decoration. Handy for transactions.

11.3.3.2 Multiple Application Service Methods per Class

There are cases where might be a good idea to group of cohesive Application Services under the same class

```
class UserService
{
    // ...

    public function signUp(SignUpUserRequest $request)
    {
        // ...
    }

    public function signIn(SignUpUserRequest $request)
    {
        // ...
    }

    public function logOut(LogOutUserRequest $request)
    {
        // ...
    }
}
```

We don't recommend such approach as not all Application Services are 100% cohesive. Some services will require different dependencies and you'll end up with Application Services depending on things that they don't need. Another issue is that this kind of class grows fast. As it violates the *Single Responsibility Principle*, there will be multiple reasons to change and maybe break it.

11.3.4 Returning Values

After signing up, we might be thinking about redirecting the user to a profile page. The most immediate reflection for doing that on the controller could be returning the `User` entity directly from the service.

[5] http://martinfowler.com/bliki/DecoratedCommand.html

```
class SignUpUserService
{
    // ...

    public function execute(SignUpUserRequest $request)
    {
        // ...

        $user = new User(
           $this->userRepository->nextIdentity(),
           $email,
           $password
        );

        $this->userRepository->add($user);

        return $user;
    }
}
```

Then, from the controller, we would pick up the `id` field and redirect to some other place.

However, think twice about what we've just done. We've returned a full-featured entity to the controller. This will allow the delivery mechanism to bypass the Application Layer and interact directly with the the domain.

Imagine the `User` entity offers an `updateEmailAddress` method. You could try to prevent it, but at some point in the future, somebody might think about using it.

```
$app->match('/signup', function (Request $request) use ($app) {
    //...
    $user = $app['sign_up_user_application_service']->execute(
        new SignUpUserRequest(
            $request->get('email'),
            $request->get('password')
        )
    );

    $user->updateEmailAddress('shouldnotupdate@email.com');
    //...
});
```

Not only that, the data that the presentation layer needs is not the same that the Domain manages. We don't want to evolve and couple the domain layer around the presentation layer. We want to evolve them freely.

We need a way a flexible way of decoupling both layers.

11.3.4.1 DTO from Aggregate Instances

We could return sterile data structures with the information the presentation layer needs. As we've seen before, DTOs fit with this scenario. We just need to compose them in the Application Service and return them to the client.

```
class UserDTO
{
    private $email;
    //...

    public function __construct(User $user)
    {
        $this->email = $user->email();
        //...
    }

    public function email()
    {
        return $this->email;
    }
}
```

The `UserDTO` will expose whatever read-only data we need from the `User` entity on the presentation layer avoiding exposing behaviour.

```
class SignUpUserService
{
    public function execute(SignUpUserRequest $request)
    {
        // ...

        $user = // ...

        return new UserDTO($user);
    }
}
```

Mission accomplished. Now we could pass parameters to the template engine, transform them into widgets, tags, subtemplates or do whatever we want with the data on the presentation side.

```
$app->match('/signup', function (Request $request) use ($app) {
    /**
     * @var UserDTO $user
     */
    $userDto = $app['sign_up_user_application_service']->execute(
        new SignUpUserRequest(
            $request->get('email'),
            $request->get('password')
        )
    );

    //...
});
```

However, letting the Application Service decide how to build the DTO reveals another limitation. As building the DTO depends exclusively on the Application Service, adapting the DTO to different clients will be very difficult. Consider the data needed for a redirect on a Web Controller and the data needed for a REST response for the same use case. Not the same data at all.

Let's allow the client to define how to built the DTO by passing a specific DTO assembler.

```
class SignUpUserService
{
    private $userDtoAssembler;

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

    public function execute(SignUpUserRequest $request)
    {
        // ...

        $user = // ...

        return $this->userDtoAssembler->assemble($user);
    }
}
```

Now the client can customise the response by passing a specific `UserDTOAssembler`.

11.3.4.2 Data Transformers

There are some cases where generating intermediate DTOs for more complex responses like JSON, XML, CSV, iCAL Contact could be seen as an unnecessary overhead. We could output the representation in a buffer and ask for it later on in the delivery side.

Transformers transform high-level domain concepts into low-level client details. Let's see an example

```
interface UserDataTransformer
{
    public function write(User $user);

    /**
     * @return mixed
     */
    public function read();
}
```

Consider the case of generating different data representations for a given product. Usually the product information is served through a web interface (HTML) but we might be interested in offer other formats like XML, JSON or CSV. This might enable integrations with other services.

Similar case for a Blog. We might expose to the world our potential as writers in HTML but some people will be interested in consuming our articles through RSS. The use cases – Application Services – remain the same, the representation does not.

DTO's are a clean and simple solution that could be passed to template engines for different representations but it might complicate the logic of this last step of data transformation. The logic for such templates could become a problem to maintain, test and understand.

Data Transformers might be a better approach on specific cases. These are just black boxes with Domain Concepts as inputs (Aggregates, Entities, etc.) and read-only representations (XML, JSON, CSV, etc.) as outputs. This transformers could be really easy to test.

```
class JsonUserDataTransformer implements UserDataTransformer
{
    private $data;

    public function write(User $user)
    {
        // More complex logic could be placed here
        // As using JMSSerializer, native json, etc.
        $this->data = json_encode($user);
    }

    /**
     * @return string
     */
    public function read()
    {
        return $this->data;
    }
}
```

That was easy. Wondering how would look like the XML or CSV one?

Let's see how to integrate the Data Transformer with our Application Service.

```
class SignUpUserService
{
    private $userRepository;
    private $userDataTransformer;

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

    public function execute(SignUpUserRequest $request)
    {
        // ...

        $user = // ...
```

```
        $this->userDataTransformer->write($user);
    }

    /**
     * @return UserDataTransformer
     */
    public function userDataTransformer()
    {
        return $this->userDataTransformer;
    }
}
```

Similar to the DTO Assembler approach but this time without returning a concrete value. The Data Transfomer is being used to hold and interact with the data.

The main issue with DTO's is the overhead of writing them. Most of the time your Domain concepts and DTO representations will present the same structure. Most of the time you'll feel is not worthy.

> One case where it is useful to use something like a DTO is **when you have a significant mismatch between the model in your presentation layer and the underlying domain model**. In this case it makes sense to make presentation specific facade/gateway that maps from the domain model and presents an interface that's convenient for the presentation. It fits in nicely with Presentation Model. This is worth doing, but it's only worth doing for screens that have this mismatch (in this case it isn't extra work, since you'd have to do it in the screen anyway.) – Martin Fowler, PoEAA

We think the long-term vision will worth the investment. On medium to big projects, interface representations and Domain concepts change at very different rhythms. You might want to decouple them from each other to lower the friction for updates. Using DTO's or Data Transformers allow you to evolve your model freely without having to think about breaking the layout all the time.

11.3.5 Multiple Application Services on Compound Layouts

Most of the time, no layout is as simple as a single Application Service. Our projects have pretty complicated interfaces.

Consider the homepage of a specific project, how do we do for rendering so many pieces and Use Cases? There are a few options, let's check them out

11.3.5.1 AJAX Content Integration

You could let the browser ask to different endpoints directly and combine the data in the layout right after through AJAX or HIJAX[6]. You will avoid mixing a lot of Application Services in your Controllers but it might have a performance penalty depending on the number of requests triggered.

11.3.5.2 ESI Content Integration

Edge Side Includes or ESI[7] is a tiny markup language similar to the previous approach but this time on the server side. It requires some extra effort configuring extra middlewares like Nginx or Varnish to make it work.

11.3.5.3 Symfony Sub Requests

If you use Symfony, Sub Requests[8] could be an interesting option. Extracted from the Symfony site:

> In addition to the main request that's sent into HttpKernel::handle, you can also send so-called sub request. A sub request looks and acts like any other request, but typically serves to render just one small portion of a page instead of a full page. You'll most commonly make sub-requests from your controller (or perhaps from inside a template, that's being rendered by your controller). This creates another full request-response cycle where this new Request is transformed into a Response. The only difference internally is that some listeners (e.g. security) may only act upon the master request. Each listener is passed some sub-class of KernelEvent, whose isMasterRequest() can be used to check if the current request is a master or sub request.

This is great as you'll get the benefits of invoking separate Application Services without AJAX penalty nor complicated ESI configurations.

11.3.5.4 One Controller, Multiple Application Services

One last option could be managing multiple Application Services withing the same controller. The controller logic could get a little bit dirty. It will handle and merge the responses to pass to the view.

11.4 Testing

As you are interested in testing the *behaviour* of the Application Service itself, there is no need to turn it into an *Integration Test* with complicated setups going against a real database. You are not interested in testing the low-level details. Most of the time a *Unit Test* will be enough.

[6] https://en.wikipedia.org/wiki/Hijax
[7] https://en.wikipedia.org/wiki/Edge_Side_Includes
[8] http://symfony.com/doc/current/components/http_kernel/introduction.html#sub-requests

```
class SignUpUserServiceTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @var \Lw\Domain\Model\User\UserRepository
     */
    private $userRepository;

    /**
     * @var SignUpUserService
     */
    private $SignUpUserService;

    public function setUp()
    {
        $this->userRepository = new InMemoryUserRepository();
        $this->SignUpUserService = new SignUpUserService($this->userRepository);
    }

    /**
     * @test
     * @expectedException \Lw\Domain\Model\User\UserAlreadyExistsException
     */
    public function alreadyExistingEmailShouldThrowAnException()
    {
        $this->executeSignUp();
        $this->executeSignUp();
    }

    private function executeSignUp()
    {
        return $this->SignUpUserService->execute(
            new SignUpUserRequest(
                'user@example.com',
                'password'
            )
        );
    }

    /**
     * @test
     */
    public function afterUserSignUpItShouldBeInTheRepository()
```

```
    {
        $user = $this->executeSignUp();

        $this->assertSame(
            $user,
            $this->userRepository->ofId($user->id())
        );
    }
}
```

We've used an in-memory implementation for the User Repository. This is what is called a Fake, a fully functional implementation for the repository that will make our test work as a unit. We don't need to go to the database to test the behaviour of this class. That would make our test slow and fragile.

Checking for Domain Events submission might be interesting too. If creating a user fires a `user registered` event, assuring it has been triggered might be a good idea.

```
class SignUpUserServiceTest extends \PHPUnit_Framework_TestCase
{
    //...

    /**
     * @test
     */
    public function itShouldPublishUserRegisteredEvent()
    {
        $subscriber = new SpySubscriber();
        $id = DomainEventPublisher::instance()->subscribe($subscriber);

        $user = $this->executeSignUp();
        $userId = $user->id();

        DomainEventPublisher::instance()->unsubscribe($id);
        $this->assertUserRegisteredEventPublished($subscriber, $userId);
    }

    private function assertUserRegisteredEventPublished($subscriber, $userId)
    {
        $this->assertInstanceOf('UserRegistered', $subscriber->domainEvent);
        $this->assertTrue($subscriber->domainEvent->userId()->equals($userId));
    }
}
```

```
class SpySubscriber implements DomainEventSubscriber
{
    public $domainEvent;

    public function handle($aDomainEvent)
    {
        $this->domainEvent = $aDomainEvent;
    }

    public function isSubscribedTo($aDomainEvent)
    {
        return true;
    }
}
```

11.5 Transactions

Transactions are detail related with the persistence mechanism. Domain layer should not be aware of this low-level implementation detail. Thinking about beginning, committing or rolling back a transaction at this level is a big smell. This level of detail belongs to the infrastructure layer.

The best way of handling transactions is to not handle them at all (explicitly). We could wrap our Application Services with a Decorator implementation for handling the transaction session automatically.

We've implemented a solution to this problem in one of our repos, check it out[9].

```
interface TransactionalSession
{
    /**
     * @return mixed
     */
    public function executeAtomically(callable $operation);
}
```

This contract takes a piece of code and executes it atomically. Depending on your persistence mechanism, you'll end up with different implementations.

Let's see how we could do it with *Doctrine ORM*

[9] https://github.com/dddinphp/ddd

```
class DoctrineSession implements TransactionalSession
{
    private $entityManager;

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

    public function executeAtomically(callable $operation)
    {
        return $this->entityManager->transactional($operation);
    }
}
```

On a sneak peek of how we would make this work on the client for this code.

```
/** @var EntityManager $em */

$nonTxApplicationService = new SignUpUserService(
    $em->getRepository('BoundedContext\Domain\Model\User\User')
);

$txApplicationService = new TransactionalApplicationService(
    $nonTxApplicationService,
    new DoctrineSession($em)
);

$response = $txApplicationService->execute(
    new SignUpUserRequest(
        'user@example.com',
        'password'
    )
);
```

Now that we have the Doctrine implementation for transactional sessions, it would be great to create a decorator for our Application Services. With this approach we make transactional requests transparent to the Domain.

```
class TransactionalApplicationService implements ApplicationService
{
    private $session;
    private $service;

    public function __construct(
        ApplicationService $service,
        TransactionalSession $session
    ) {
        $this->session = $session;
        $this->service = $service;
    }

    public function execute($request = null)
    {
        if (empty($this->service)) {
            throw new \LogicException('A use case must be specified');
        }

        $operation = function () use ($request) {
            return $this->service->execute($request);
        };

        return $this->session->executeAtomically($operation);
    }
}
```

A nice side-effect of using Doctrine Session is that it automatically manages the `flush` method. You don't need to add the flush inside your Domain or Infrastructure.

11.6 Security

In case you are wondering how to manage and handle user credentials and security in general, unless is responsibility of your domain, we recommend to let the Framework handle it. User session is a concern of the delivery mechanism. Polluting the Domain with these concepts will make it harder to develop.

11.7 Domain Events

Domain Event listeners have to be configured before the Application Service gets executed or nobody will be noticed. There are situations where you'll have to be explicitly and configure the listener before executing the Application Service.

```
//...
$subscriber = new SpySubscriber();
DomainEventPublisher::instance()->subscribe($subscriber);

$applicationService = //...
$applicationService->execute(...);
```

Most of the time this will be done by configuring the Dependency Injection Container.

11.8 Command Handlers

An interesting way of executing Application Services is through a Command Bus library. A good one is Tactician[10]

> What is a Command Bus? The term is mostly used when we combine the Command pattern[11] with a service layer[12]. Its job is to take a Command object (which describes what the user wants to do) and match it to a Handler (which executes it). This can help structure your code neatly.

Fair enough, our Application Services are the Service Layer and our Request objects looks pretty much like Commands.

Wouldn't be great if we had a mechanism to link all the Application Services and then based on the Request execute the correct one? Well, that is actually what a Comment Bus is.

11.8.1 Tactician Library and Other Options

Tactician is a Command Bus library. It allows you to use the Command Pattern for your Application Services. It's specially convenient for Application Services but you could use any kind of input tho.

Let's see an example from the Tactician website

[10]https://tactician.thephpleague.com/
[11]https://en.wikipedia.org/wiki/Command_pattern
[12]http://martinfowler.com/eaaCatalog/serviceLayer.html

```php
// You build a simple message object like this:
class PurchaseProductCommand
{
    protected $productId;
    protected $userId;

    // ...and constructor to assign those properties...
}

// And a Handler class that expects it:
class PurchaseProductHandler
{
    public function handle(PurchaseProductCommand $command)
    {
        // use command to update your models, etc
    }
}

// And then in your controllers, you can fill in the command using your favorite
// form or serializer library, then drop it in the CommandBus and you're done!
$command = new PurchaseProductCommand(42, 29);
$commandBus->handle($command);
```

That's it. Tactician is the `$commandBus` service. It does all the plumbing for finding the right handler and method. This can avoid a lot of boilerplate code. Here Commands and Handlers are just normal classes but you can configure whatever fits better your app.

Summarising, we can conclude that Commands are just Request objects and Command Handlers are just Application Services.

A cool thing about Tactician (and Command Buses in general) is that they are really easy to extend. Tactician provides plugins for common tasks like logging and database transactions. That way you can forget about setting up the wiring on every handler.

Another interesting plugin for Tactician is Bernard[13] integration. Bernard is an asynchronous job queue that allows you to leave some tasks for later processing. Heavy processes block the user response and most of the time could be delayed for later processing. Answer the user fast and let the know once the job is done.

Mathias Noback developed a cool alternative to Tactician called SimpleBus check it out[14].

[13]http://bernard.readthedocs.org/
[14]http://simplebus.github.io/MessageBus/

11.9 Wrap-up

Application Services represent the Application layer of your Bounded Context. These high-level use cases should be pretty simple and skinny as their purpose evolves around Domain coordination. Application Services are the entry point for Domain logic interaction. We've seen that Requests and Commands keep things organised, DTO's and Data Transformers allow us to decouple data representation from Domain conceptualisation, that building Application Services is pretty straightforward with Dependency Injection Containers and that we have plenty of options for combining Application Services in complex layouts.

12. Integrating Bounded Contexts

Every enterpise application is typically composed of several areas in which the company operates. Areas such as *billing*, *inventory*, *shipping management*, *catalog* and so on are common examples. The easiest manner in which to manage all these concerns may seem to lean towards a *monolithic* system. You might wonder, does it have to be this way? What if any friction garnered between teams working on these seperate areas could be reduced by splitting this big monolithic application into smaller, independent chunks. We will now be exploring how to do this, so get prepared for insights and heuristics around **strategical design**.

> **Dealing With Distributed Systems**
>
> Dealing with distributed systems is **hard**. Breaking a system into independent autonomous parts has benefits, but it also increases complexity. For example, the coordination and synchronization of those systems is not trivial and as a result should be considered carefully. As *Martin Fowler* said in the *PoEAA* book, the first law of distributed systems is always: ***Don't distribute***.

12.1 Integration Through the Data Store

One of the most commonly used techniques to integrate different parts of an application has always been to share the same data store, along with the same code base. This is usually known as a *monolithic application*, often ending up with a single data-store that hosts the data related to all the concerns within the application.

Consider an e-commerce application, a shared data-store would contain all concerns (eg: tables within a relational database) surrounding the catalog, billing, inventory, and so on. There is nothing bad with this approach per se, for example in small linear applications were the complexity is not too high. However, within complex domains, some issues can arise. If you share data across many tables touching multiple application concerns, transactions will have a big impact on performance.

Another less technical problem that could develop will be in-regard to the *Ubiquitous Language*. The main advantage to the seperation of *Bounded Contexts* is having **a single Ubiquitous Language for each one**. In doing so, models will be separated into their own contexts. Mixing all models together within the same context can lead to ambiguity and confusion.

Going back to the e-commerce system, imagine we want to introduce the concept of a t-shirt. Within the catalogue context, a t-shirt would be a *product* with properties like *color*, *size*, *material* and maybe some fancy *pictures*. In the *inventory* system however, we do not really wish to concern ourselves with these. A *product* here has a different meaning, were we care about different properties like *weight*, *location in the warehouse* or *dimensions*. Mixing both contexts together will tangle concepts and will complicate the design. In DDD terms, mixing concepts in this manner is what is called a *Shared Kernel*.

> **Shared Kernel**
>
> *Designate some subset of the domain model that the teams agree to share. Of course this includes, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team. Integrate a functional system frequently, but somewhat less often than the pace of CONTINUOUS INTEGRATION within the teams. At these integrations, run the tests of both teams.*
>
> *Eric Evans* - Domain-Driven Design, Tackling complexity in the heart of software. Chapter 14 - Shared Kernel

We do not recommend using a Shared Kernel. As multiple teams can collide within the development of it, ending up having maintenance issues and becoming a friction point. Changes in the Shared Kernel should be agreed upon beforehand, between all parties involved. Conceptually it has other problems, such as people seeing it as a bag to place 'stuff' that does not belong anywhere else, growing indefinitely.

A better way of dealing with the ever growing monolithic complexity is to break it up in different autonomous *pieces*. Such as communicating through REST, RPC or messaging systems. This requires drawing clear boundaries, with each context likely ending up with their own infrastructure – data stores, servers, messaging middleware, etc. – and even its own team. As you may foresee, this could lead to some degree of duplication. That is a trade-off that we are willing to make in order to reduce complexity. These autonomous pieces receive the name of ***Bounded Contexts***.

12.2 Integration Relationships

12.2.1 Customer / Supplier

When there is a unidirectional integration between two Bounded Contexts, where one acts as a provider (**upstream**) and the other as a client of it (**downstream**) we will end up with ***Customer - Supplier Development Teams***.

> *Establish a clear customer/supplier relationship between the two teams. In planning sessions, make the downstream team play the customer role to the upstream team. Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule.*
>
> *Jointly develop automated acceptance tests that will validate the interface expected. Add these tests to the upstream team's test suite, to be run as part of its' continuous integration. This testing will free the upstream team to make changes without fear of side effects downstream.*
>
> *Eric Evans* - Domain-Driven Design, Tackling complexity in the heart of software.

Customer / Supplier Development Teams is the most common way of integrating Bounded Contexts. It usually represents a *win - win* situation when teams work closely.

12.2.2 Separate Ways

Following on with the e-commerce example, think about reporting revenue to an old legacy retailer financial system. The integration could be so expensive resulting in it not being worth the effort to implement. This is called in DDD strategic terms ***Separate Ways***.

> *Integration is always expensive. Sometimes the benefit is small. So Declare a BOUNDED CONTEXT to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope.*
>
> *Eric Evans* - Domain-Driven Design, Tackling complexity in the heart of software.

12.2.3 Conformist

Consider again the e-commerce example and integration with a third party *shipping service*. Your domain and theirs differ in models, teams and infrastructure. They will not participate in your product plannings or provide any solutions to the e-commerce system. These teams do not have a close relationship. We could choose to accept and *conform* to their domain model. This is what we call in strategic design a ***Conformist Integration***

> *Eliminate the complexity of translation between BOUNDED CONTEXTS by slavishly adhering to the model of the upstream team. Although this cramps the style of the downstream designers and probably does not yield the ideal model for the application, choosing CONFORMITY enormously simplifies integration. Also, you will share a UBIQUITOUS LANGUAGE with your supplier team. The supplier is in the driver's seat, so it is good to make communication easy for them. Altruism may be sufficient to get them to share information with you.*
>
> *Eric Evans* - Domain-Driven Design, Tackling complexity in the heart of software.

12.3 Implementing Bounded Context Integrations

To make things easier, we will assume Bounded Contexts have a relationship of *Customer - Supplier*.

12.3.1 Modern RPC

With *modern RPC* we refer to RPC through RESTful resources. A Bounded Context reveals to the outside world a clear interface to interact with. It exposes *resources* that could be manipulated through HTTP verbs. We could say that the Bounded Context offers a set of services and operations. In strategical terms, this is what is called an ***Open Host Service***.

> **Open Host Service**
>
> *Define a protocol that gives access to your subsystem as a set of SERVICES. Open the protocol so that all who need to integrate with you can use it. Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs. Then, use a one-off translator to augment the protocol for that special case so that the shared protocol can stay simple and coherent.*
>
> *Eric Evans* - Domain-Driven Design, Tackling complexity in the heart of software.

Lets explore an example provided within the Last Wishes application[1] that comes with the books' Github organization.

The application is a web platform whose purpose is letting people save their last wills before they die. There are two contexts, one responsible in handling wills – the **Will** Bounded Context – and the Gamification Context[2] in charge of giving points to the users of the system. In the Will Context, the user could have badges that are related to the number of points the user made on the Gamification Context. This means that we need to integrate both contexts together in order to show the badges a user has on the Will Context.

The Gamification Context is a full-fledged event-driven application powered by a custom eventsourcing engine. It is a full-stack Symfony application that uses **FOSRestBundle**[3], **BazingaHateoasBundle**[4], **JMSSerializerBundle**[5], **NelmioApiDocBundle**[6] and **OngrElasticsearchBundle**[7] to provide a level 3 and up REST API (commonly known as *the Glory of REST*) according to the **Richardson Maturity Model**[8]. All the events triggered within this Context are projected against an Elasticsearch server in order to produce the data needed for the views. We will expose the number of points made

[1]https://github.com/dddinphp/last-wishes
[2]https://github.com/dddinphp/last-wishes-gamify
[3]http://symfony.com/doc/current/bundles/FOSRestBundle/index.html
[4]https://github.com/willdurand/BazingaHateoasBundle
[5]https://github.com/schmittjoh/JMSSerializerBundle
[6]https://github.com/nelmio/NelmioApiDocBundle/
[7]https://github.com/ongr-io/ElasticsearchBundle
[8]http://martinfowler.com/articles/richardsonMaturityModel.html

for a given user through an endpoint like `http://gamification.context.host/api/users/{id}`. We will fetch the user projection from Elasticsearch and serialise it to a format previously negotiated with the client.

```
namespace AppBundle\Controller;

use FOS\RestBundle\Controller\Annotations as Rest;
use FOS\RestBundle\Controller\FOSRestController;
use Nelmio\ApiDocBundle\Annotation\ApiDoc;

class UsersController extends FOSRestController
{
    /**
     * @ApiDoc(
     *  resource=true,
     *  description="Finds a user given a user ID",
     *  statusCodes={
     *      200 = "Returned when the user have been found",
     *      404 = "Returned when the user could not be found"
     *  }
     * )
     *
     * @Rest\View(
     *  statusCode = 200
     * )
     */
    public function getUserAction($id)
    {
        $repo = $this->get('es.manager.default.user');

        $user = $repo->find($id);

        if (!$user) {
            throw $this->createNotFoundException(
                sprintf(
                    'A user with an ID of %s does not exist',
                    $id
                )
            );
        }

        return $user;
```

```
    }
}
```

As we explained in the **architecture** chapter, *reads* are treated as an infrastructure concern. There is no need to wrap them inside a Command / Command Handler flow.

The resulting JSON+HAL representation of a user will be

```
{
  "id": "c3c587c6-610a-42df-90d3-8e9a181d65d0",
  "points": 0,
  "_links": {
    "self": {
      "href": "http://gamification.context/api/users/c3c587c6-610a-42df-90d3-8e9\
a181d65d0"
    }
  }
}
```

Now we are in a good position for integrating both contexts. We just need to write the client in the Will Context for consuming the endpoint we have just created. Should we mix both domain models? Digesting the Gamification Context directly will mean adapting the Will Context to the Gamification one, resulting in a **Conformist** integration. However, separating these concerns seems worth the investment. We need a layer for guaranteeing the integrity and the consistency of the Domain Model within the Will Context. We need to translate *points* (Gamification) to *badges* (Will). This translation mechanism is what in DDD is called an ***Anti-corruption Layer***.

> **Anti-corruption Layer**
>
> *Create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer translates in both directions as necessary between the two models.*
>
> *Eric Evans* - Domain-Driven Design, Tackling complexity in the heart of software.

So, what does the Anti-corruption layer look like? Most of the time Service will be interacting with a combination of *Adapters* and *Facades*. The Services encapsulate and hide the complexities behind these complex transformations. Facades aid in hiding and encapsulating access details required in fetching data from the Gamification model. Adapters translate between models, often using specialised *Translators*.

Lets see how to define a *User Service* within the Will's model, that will be responsible to retrieve the badges earned by a given user.

```
namespace Lw\Domain\Model\User;

interface UserService
{
    public function badgesFrom(UserId $id);
}
```

Now, the implementation in the Infrastructure side. We will use an adapter for the transformation process.

```
namespace Lw\Infrastructure\Service;

use Lw\Domain\Model\User\UserId;
use Lw\Domain\Model\User\UserService;

class TranslatingUserService implements UserService
{
    private $userAdapter;

    public function __construct(UserAdapter $userAdapter)
    {
        $this->userAdapter = $userAdapter;
    }

    public function badgesFrom(UserId $id)
    {
        return $this->userAdapter->toBadges($id);
    }
}
```

The Adapter for the transformation

```
namespace Lw\Infrastructure\Service;

use GuzzleHttp\Client;

class HttpUserAdapter implements UserAdapter
{
    private $client;

    public function __construct(Client $client)
    {
```

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

    public function toBadges($id)
    {
        $response = $this->client->get(sprintf('/users/%s', $id), [
            'allow_redirects' => true,
            'headers' => [
                'Accept' => 'application/hal+json'
            ]
        ]);

        $badges = [];

        if (200 === $response->getStatusCode()) {
            $badges =
                (new UserTranslator())
                    ->toBadgesFromRepresentation(
                        json_decode(
                            $response->getBody(),
                            true
                        )
                    )
            ;
        }

        return $badges;
    }
}
```

As you can see, the Adapter acts as a **Facade to the Gamification Context** to. We did it this way as fetching the User resource in the Gamification side is pretty straightforward. The Adapter uses the **UserTranslator** to perform the translation.

```
namespace Lw\Infrastructure\Service;

use Lw\Infrastructure\Domain\Model\User\FirstWillMadeBadge;
use Symfony\Component\PropertyAccess\PropertyAccess;

class UserTranslator
{
    public function toBadgesFromRepresentation($representation)
    {
        $accessor = PropertyAccess::createPropertyAccessor();

        $points = $accessor->getValue($representation, 'points');

        $badges = [];
        if ($points > 3) {
            $badges[] = new FirstWillMadeBadge();
        }

        return $badges;
    }
}
```

The *Translator* specialises in transforming the points coming from the Gamification Context into badges.

We have shown how to integrate two Bounded Contexts where respective teams share a **Customer / Supplier** relationship. The Gamification Context exposes the integration through an **Open Host Service** implemented by a RESTful protocol. On the other side, the Will Context consumes the service through an **Anti-corruption Layer** responsible in translating the model from one domain to the other, ensuring the Will Contexts' integrity.

12.3.2 Message Queues

RESTful resources is not the only way of enabling integrations between Bounded Contexts. As we will see, messaging middleware enables decoupled integrations between different contexts.

Continuing with the Last Wishes application, we have just implemented a unidirectional relationship between two teams to manage *points* and *badges* within their respective contexts. We left important functionality out of scope on purpose: **rewarding the user every time they make a wish**.

We could go for another **Open Host Service** with a pull strategy. The Will context will be pulling the Gamification context periodically to get badges on sync (eg: through an scheduler like Cron). This solution will impact on the users experience and it will waste a lot of unnecessary resources.

A better approach is to use a **messaging middleware**. With this solution Contexts could push messages to a middleware (often a message queue). Interested parties will be able to subscribe, inspect and consume information on-demand in a decoupled fashion. In order to do this, we need a **specialised, shared and common communication language** so all the parties can understand the information transmitted. This what is called the **Published Language**.

Published Language

Use a well-documented shared language that can express the necessary domain information as a common medium of communication, translating as necessary into and out of that language.

Eric Evans - Domain-Driven Design, Tackling complexity in the heart of software.

Thinking about the format of these messages, looking closer at our Domain Model we realise we already have it! **Domain Events**. There is no need for defining a new communication protocol between Bounded Contexts. We can use Domain Events to define a common language across contexts. Their definition of **something that Domain Experts care about just happened** just fits perfect with what we are looking after a **Published Language**.

In our example, we could use **RabbitMQ**[9] as a messaging middleware. This is probably one of the most reliable and robust messaging **AMQP**[10] protocol out there. We will incorparate the widely used PHP libraries **php-amqplib**[11] and **RabbitMQBundle**[12].

Lets start with the Will context as it is the one which triggers events when the user signs up or when making a wish. As we have already seen in the **domain events** chapter **it is a good idea to store domain events into a persistent mechanism**, so we will take it for granted. We need a *message publisher* to fetch and publish stored domain events from the event store to the messaging middleware. We already did the integration with RabbitMQ in the **domain events** chapterm so we just need to implement the code in the Gamification Context. We will listen for events triggered by the Will Context. As we are using Symfony on the Gamification side, we can take advantage already with the RabbitMQBundle to make things easier.

We define two message consumers for the *User Signed Up* and *Wish Was Made* events.

[9]https://www.rabbitmq.com/
[10]https://www.amqp.org/
[11]https://github.com/videlalvaro/php-amqplib
[12]https://github.com/videlalvaro/RabbitMqBundle

```
namespace AppBundle\Infrastructure\Messaging\PhpAmqpLib;

use Lw\Gamification\Command\SignupCommand;
use OldSound\RabbitMqBundle\RabbitMq\ConsumerInterface;
use PhpAmqpLib\Message\AMQPMessage;

class PhpAmqpLibLastWillUserRegisteredConsumer implements ConsumerInterface
{
    private $commandBus;

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

    public function execute(AMQPMessage $message)
    {
        $type = $message->get('type');

        if ('Lw\Domain\Model\User\UserRegistered' === $type) {
            $event = json_decode($message->body);
            $eventBody = json_decode($event->event_body);

            $this->commandBus->handle(
                new SignupCommand($eventBody->user_id->id)
            );

            return true;
        }

        return false;
    }
}
```

Note that in this case we are only processing messages whose type is `Lw\Domain\Model\User\UserRegistered`.

And the consumer for the *User Signed Up* event.

```
namespace AppBundle\Infrastructure\Messaging\PhpAmqpLib;

use Lw\Gamification\Command\RewardUserCommand;
use Lw\Gamification\DomainModel\AggregateDoesNotExist;
use OldSound\RabbitMqBundle\RabbitMq\ConsumerInterface;
use PhpAmqpLib\Message\AMQPMessage;

class PhpAmqpLibLastWillWishWasMadeConsumer implements ConsumerInterface
{
    private $commandBus;

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

    public function execute(AMQPMessage $message)
    {
        $type = $message->get('type');

        if ('Lw\Domain\Model\Wish\WishWasMade' === $type) {
            $event = json_decode($message->body);
            $eventBody = json_decode($event->event_body);

            try {
                $points = 5;
                $this->commandBus->handle(
                    new RewardUserCommand(
                        $eventBody->user_id->id,
                        $points
                    )
                );
            } catch (AggregateDoesNotExist $e) {
                // Noop
            }

            return true;
        }

        return false;
    }
}
```

Again, we are only interested in tracking `Lw\Domain\Model\Wish\WishWasMade` events.

In both cases we use a Command Bus, which is out of the scope of this chapter. We can summarise it as a highway that decouples the Command and Receiver. The **when** and **how** a Command is executed is independent from **who** triggered it.

The Gamification Context uses **Tactician**[13] (and **TacticianBundle**[14]), a simple command bus that can be extended and adapted to your system.

So now we are almost ready to start consuming events from the Will Context. The only missing piece is to define the RabbitMQBundle configuration in Symfony's `config.yml` file

```
services:
    last_will_user_registered_consumer:
        class: AppBundle\Infrastructure\Messaging\PhpAmqpLib\PhpAmqpLibLastWillU\
serRegisteredConsumer
        arguments:
            - @tactician.commandbus

    last_will_wish_was_made_consumer:
        class: AppBundle\Infrastructure\Messaging\PhpAmqpLib\PhpAmqpLibLastWillW\
ishWasMadeConsumer
        arguments:
            - @tactician.commandbus

old_sound_rabbit_mq:
    connections:
        default:
            host:       "%rabbitmq_host%"
            port:       "%rabbitmq_port%"
            user:       "%rabbitmq_user%"
            password:   "%rabbitmq_password%"
            vhost:      "%rabbitmq_vhost%"
            lazy:       true

    consumers:
        last_will_user_registered:
            connection: default
            callback: last_will_user_registered_consumer

            exchange_options:
                name: last-will
```

[13] http://tactician.thephpleague.com/

[14] https://github.com/thephpleague/tactician-bundle

```
                type: fanout

            queue_options:
                name: last-will

        last_will_wish_was_made:
            connection: default
            callback: last_will_wish_was_made_consumer

            exchange_options:
                name: last-will
                type: fanout

            queue_options:
                name: last-will
```

Probably, the most convenient RabbitMQ configuration is the *Publish / Subscribe*[15]** *pattern. All messages published by the Will Context will be delivered to all connected consumers. This is called* **fanout* in the RabbitMQ exchange configuration. The exchange consists of an agent being in charge of delivering messages to the corresponding queues.

```
> php app/console rabbitmq:consumer --messages=1000 last_will_user_registered
> php app/console rabbitmq:consumer --messages=1000 last_will_wish_was_made
```

With those two commands Symfony will execute both consumers and they will start listening for Domain Events. We have specified a limit of 1000 messages to consume as PHP is not the best platform to execute long-running processes. It also might be a good idea to use something like **Supervisor**[16] to monitor and restart processes periodically.

12.4 Wrap-up

Although we have only seen a small part, strategical design is at the heart and soul of Domain-Driven Design. It is an essential part that helps you in developing better and more semantic models. We recommend to use messaging middleware to integrate bounded contexts as that naturally leads to simpler, decoupled and event-driven architectures.

[15] https://www.rabbitmq.com/tutorials/tutorial-three-php.html
[16] http://supervisord.org/

13. Bibliography

- Implementing Domain-Driven Design[1] by Vaughn Vernon
- Patterns of Enterprise Application Architecture[2] by Martin Fowler
- Enterprise Integration Patterns[3] by Gregor Hohpe and Bobby Woolf
- Domain-Driven Design[4] by Eric Evans
- NoSQL Distilled[5] by Martin Fowler and Pramod J. Sadalage
- Building Microservices[6] by Sam Newman

[1] http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8
[2] http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin-ebook/dp/B000OZ0NAI
[3] http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E
[4] http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software-ebook/dp/B00794TAUG
[5] http://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620
[6] http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358

Appendix A: Hexagonal Architecture with PHP

Article published in the php|architect magazine. June 2014. Carlos Buenosvinos (@buenosvinos).

Introduction

With the rise of Domain-Driven Design (DDD), architectures promoting domain centric designs are becoming more popular. This is the case with **Hexagonal Architecture**, also known as **Ports and Adapters**, that seems to have being rediscovered just now by PHP developers. Invented in 2005 by Alistair Cockburn, one of the Agile Manifesto authors, the Hexagonal Architecture allows an application to be equally driven by users, programs, automated tests or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases. This results into agnostic infrastructure web applications that are easier to test, write and maintain. Let's see how to apply it using real PHP examples.

Your company is building a brainstorming system called *Idy*. Users add and rate ideas so the most interesting ones can be implemented in a company. It's Monday morning, another sprint is starting and you are reviewing some user stories with your team and your Product Owner. **"As a not logged in user**, **I want to rate an idea and the author should be notified by email"**, that's a really important one, isn't it?

First Approach

As a good developer, you decide to divide and conquer the user story, so you'll start with the first part, "I want to rate an idea". After that, you will face "the author should be notified by email". That sounds like a plan.

In terms of business rules, rating an idea is as easy as finding the idea by its identifier in the ideas repository, where all the ideas live, add the rating, recalculate the average and save the idea back. If the idea does not exist or the repository is not available we should throw an exception so we can show an error message, redirect the user or do whatever the business asks us for.

In order to *execute* this *UseCase*, we just need the idea identifier and the rating from the user. Two integers that would come from the user request.

Your company web application is dealing with a Zend Framework 1 legacy application. As most of companies, probably some parts of your app may be newer, more SOLID, and others may just be a big ball of mud. However, you know that it does not matter at all which framework you are using, it's all about writing clean code that makes maintenance a low cost task for your company.

You're trying to apply some Agile principles you remember from your last conference, how it was, yeah, I remember "make it work, make it right, make it fast". After some time working you get something like Listing 1.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        // Getting parameters from the request
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        // Building database connection
        $db = new Zend_Db_Adapter_Pdo_Mysql(array(
            'host'     => 'localhost',
            'username' => 'idy',
            'password' => '',
            'dbname'   => 'idy'
        ));

        // Finding the idea in the database
        $sql = 'SELECT * FROM ideas WHERE idea_id = ?';
        $row = $db->fetchRow($sql, $ideaId);
        if (!$row) {
            throw new Exception('Idea does not exist');
        }

        // Building the idea from the database
        $idea = new Idea();
        $idea->setId($row['id']);
        $idea->setTitle($row['title']);
        $idea->setDescription($row['description']);
        $idea->setRating($row['rating']);
        $idea->setVotes($row['votes']);
        $idea->setAuthor($row['email']);

        // Add user rating
        $idea->addRating($rating);

        // Update the idea and save it to the database
```

```
        $data = array(
            'votes' => $idea->getVotes(),
            'rating' => $idea->getRating()
        );
        $where['idea_id = ?'] = $ideaId;
        $db->update('ideas', $data, $where);

        // Redirect to view idea page
        $this->redirect('/idea/'.$ideaId);
    }
}
```

I know what readers are thinking: "Who is going to access data directly from the controller? This is a 90's example!", ok, ok, you're right. If you are already using a framework, it's likely that you are also using an ORM. Maybe done by yourself or any of the existing ones such as Doctrine, Eloquent, ZendDB, etc. If this is the case, you are one step further from those who have some Database connection object but don't count your chickens before they're hatched.

For newbies, Listing 1 code just works. However, if you take a closer look at the Controller, you'll see more than business rules, you'll also see how your web framework routes a request into your business rules, references to the database or how to connect to it. So close, you see references to your **infrastructure**.

Infrastructure is the **detail that makes your business rules work**. Obviously, we need some way to get to them (API, web, console apps, etc.) and effectively we need some physical place to store our ideas (memory, database, NoSQL, etc.). However, we should be able to exchange any of these pieces with another that behaves in the same way but with different implementations. What about starting with the Database access?

All those `Zend_DB_Adapter` connections (or straight MySQL commands if that's your case) are asking to be promoted to some sort of object that encapsulates fetching and persisting Idea objects. They are begging for being a Repository.

Repositories and the Persistence Edge

Whether there is a change in the business rules or in the infrastructure, we must edit the same piece of code. Believe me, in CS, you don't want many people touching the same piece of code for different reasons. Try to make your functions do one and just one thing so it's less probable having people messing around with the same piece of code. You can learn more about this by having a look at the Single Responsibility Principle (SRP). For more information about this principle: http://www.objectmentor.com/resources/articles/srp.pdf

Listing 1 is clearly this case. If we want to move to Redis or add the author notification feature, you'll have to update the `rateAction` method. Chances to affect aspects of the `rateAction` not related with the one updating are high. Listing 1 code is fragile. If it is common in your team to hear "If it works, don't touch it", SRP is missing.

So, we must decouple our code and encapsulate the responsibility for dealing with fetching and persisting ideas into another object. The best way, as explained before, is using a Repository. Challenged accepted! Let's see the results in Listing 2.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new IdeaRepository();
        $idea = $ideaRepository->find($ideaId);
        if (!$idea) {
            throw new Exception('Idea does not exist');
        }

        $idea->addRating($rating);
        $ideaRepository->update($idea);

        $this->redirect('/idea/'.$ideaId);
    }
}

class IdeaRepository
{
    private $client;

    public function __construct()
    {
        $this->client = new Zend_Db_Adapter_Pdo_Mysql(array(
            'host'     => 'localhost',
            'username' => 'idy',
            'password' => '',
            'dbname'   => 'idy'
        ));
    }

    public function find($id)
    {
```

```
        $sql = 'SELECT * FROM ideas WHERE idea_id = ?';
        $row = $this->client->fetchRow($sql, $id);
        if (!$row) {
            return null;
        }

        $idea = new Idea();
        $idea->setId($row['id']);
        $idea->setTitle($row['title']);
        $idea->setDescription($row['description']);
        $idea->setRating($row['rating']);
        $idea->setVotes($row['votes']);
        $idea->setAuthor($row['email']);

        return $idea;
    }

    public function update(Idea $idea)
    {
        $data = array(
            'title' => $idea->getTitle(),
            'description' => $idea->getDescription(),
            'rating' => $idea->getRating(),
            'votes' => $idea->getVotes(),
            'email' => $idea->getAuthor(),
        );

        $where = array('idea_id = ?' => $idea->getId());
        $this->client->update('ideas', $data, $where);
    }
}
```

The result is nicer. The `rateAction` of the `IdeaController` is more understandable. When read, it talks about business rules. `IdeaRepository` is a **business concept**. When talking with business guys, they understand what an `IdeaRepository` is: A place where I put Ideas and get them.

A Repository "mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects." as found in Martin Fowler's pattern catalog.

If you are already using an ORM such as Doctrine, your current repositories extend from an `EntityRepository`. If you need to get one of those repositories, you ask Doctrine `EntityManager` to do the job. The resulting code would be almost the same, with an extra access to the `EntityManager` in the controller action to get the `IdeaRepository`.

At this point, we can see in the landscape one of the edges of our hexagon, the *persistence* edge. However, this side is not well drawn, there is still some relationship between what an `IdeaRepository` is and how it's implemented.

In order to make an effective separation between our *application boundary* and the *infrastructure boundary* we need an additional step. We need to explicitly decouple behavior from implementation using some sort of interface.

Decoupling Business and Persistence

Have you ever experienced the situation when you start talking to your Product Owner, Business Analyst or Project Manager about your issues with the Database? Can you remember their faces when explaining how to persist and fetch an object? They had no idea what you were talking about.

The truth is that they don't care, but that's ok. If you decide to store the ideas in a MySQL server, Redis or SQLite it is your problem, not theirs. Remember, from a business standpoint, **your infrastructure is a detail**. Business rules are not going to change whether you use Symfony or Zend Framework, MySQL or PostgreSQL, REST or SOAP, etc.

That's why it's important to decouple our IdeaRepository from its implementation. The easiest way is to use a proper interface. How can we achieve that? Let's take a look at Listing 3.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new MySQLIdeaRepository();
        $idea = $ideaRepository->find($ideaId);
        if (!$idea) {
            throw new Exception('Idea does not exist');
        }

        $idea->addRating($rating);
        $ideaRepository->update($idea);

        $this->redirect('/idea/'.$ideaId);
    }
}

interface IdeaRepository
```

```
{
    /**
     * @param int $id
     * @return null|Idea
     */
    public function find($id);

    /**
     * @param Idea $idea
     */
    public function update(Idea $idea);
}

class MySQLIdeaRepository implements IdeaRepository
{
    // ...
}
```

Easy, isn't it? We have extracted the `IdeaRepository` behaviour into an interface, renamed the IdeaRepository into `MySQLIdeaRepository` and updated the `rateAction` to use our `MySQLIdeaRepository`. But what's the benefit?

We can now exchange the repository used in the controller with any implementing the same interface. So, let's try a different implementation.

Migrating our Persistence to Redis

During the sprint and after talking to some mates, you realize that using a NoSQL strategy could improve the performance of your feature. Redis is one of your best friends. Go for it and show me your Listing 4.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new RedisIdeaRepository();
        $idea = $ideaRepository->find($ideaId);
        if (!$idea) {
            throw new Exception('Idea does not exist');
```

```
        }

        $idea->addRating($rating);
        $ideaRepository->update($idea);

        $this->redirect('/idea/'.$ideaId);
    }
}

interface IdeaRepository
{
    // ...
}

class RedisIdeaRepository implements IdeaRepository
{
    private $client;

    public function __construct()
    {
        $this->client = new \Predis\Client();
    }

    public function find($id)
    {
        $idea = $this->client->get($this->getKey($id));
        if (!$idea) {
            return null;
        }

        return unserialize($idea);
    }

    public function update(Idea $idea)
    {
        $this->client->set(
            $this->getKey($idea->getId()),
            serialize($idea)
        );
    }

    private function getKey($id)
```

```
    {
        return 'idea:' . $id;
    }
}
```

Easy again. You've created a `RedisIdeaRepository` that implements `IdeaRepository` interface and we have decided to use Predis as a connection manager. Code looks smaller, easier and faster. But what about the controller? It remains the same, we have just changed which repository to use, but it was just one line of code.

As an exercise for the reader, try to create the IdeaRepository for SQLite, a file or an in-memory implementation using arrays. Extra points if you think about how ORM Repositories fit with Domain Repositories and how ORM *@annotations* affect this architecture.

Decouple Business and Web Framework

We have already seen how easy it can be to changing from one persistence strategy to another. However, the persistence is not the only edge from our Hexagon. What about how the user interacts with the application?

Your CTO has set up in the roadmap that your team is moving to Symfony2, so when developing new features in you current ZF1 application, we would like to make the incoming migration easier. That's tricky, show me your Listing 5.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute($ideaId, $rating);

        $this->redirect('/idea/'.$ideaId);
    }
}

interface IdeaRepository
{
    // ...
```

```
}

class RateIdeaUseCase
{
    private $ideaRepository;

    public function __construct(IdeaRepository $ideaRepository)
    {
        $this->ideaRepository = $ideaRepository;
    }

    public function execute($ideaId, $rating)
    {
        try {
            $idea = $this->ideaRepository->find($ideaId);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }

        if (!$idea) {
            throw new IdeaDoesNotExistException();
        }

        try {
            $idea->addRating($rating);
            $this->ideaRepository->update($idea);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }

        return $idea;
    }
}
```

Let's review the changes. Our controller is not having any business rules at all. We have pushed all the logic inside a new object called `RateIdeaUseCase` that encapsulates it. This object is also known as Controller, Interactor or Application Service.

The magic is done by the `execute` method. All the dependencies such as the `RedisIdeaRepository` are passed as an argument to the constructor. All the references to an `IdeaRepository` inside our UseCase are pointing to the interface instead of any concrete implementation.

That's really cool. If you take a look inside `RateIdeaUseCase`, there is nothing talking about MySQL or Zend Framework. No references, no instances, no annotations, nothing. It is like your infrastructure doesn't mind. It just talks about business logic.

Additionally, we have also tuned the Exceptions we throw. Business processes also have exceptions. NotAvailableRepository and IdeaDoesNotExist are two of them. Based on the one being thrown we can react in different ways in the framework boundary.

Sometimes, the number of parameters that a UseCase receives can be too many. In order to organize them, it's quite common to build a *UseCase request* using a Data Transfer Object (DTO) to pass them together. Let's see how you could solve this in Listing 6.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $this->redirect('/idea/'.$response->idea->getId());
    }
}

class RateIdeaRequest
{
    public $ideaId;
    public $rating;

    public function __construct($ideaId, $rating)
    {
        $this->ideaId = $ideaId;
        $this->rating = $rating;
    }
}

class RateIdeaResponse
{
    public $idea;

    public function __construct(Idea $idea)
```

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

class RateIdeaUseCase
{
    // ...

    public function execute($request)
    {
        $ideaId = $request->ideaId;
        $rating = $request->rating;

        // ...

        return new RateIdeaResponse($idea);
    }
}
```

The main changes here are introducing two new objects, a Request and a Response. They are not mandatory, maybe a UseCase has no request or response. Another important detail is how you build this request. In this case, we are building it getting the parameters from ZF request object.

Ok, but wait, what's the real benefit? It's easier to change from one framework to other, or execute our UseCase from another *delivery mechanism*. Let's see this point.

Rating an idea using the API

During the day, your Product Owner comes to you and says: "by the way, a user should be able to rate an idea using our mobile app. I think we will need to update the API, could you do it for this sprint?". Here's the PO again. "No problem!". Business is impressed with your commitment.

As Robert C. Martin says: "The Web is a delivery mechanism [...] Your system architecture should be as ignorant as possible about how it is to be delivered. You should be able to deliver it as a console app, a web app, or even a web service app, without undue complication or any change to the fundamental architecture".

Your current API is built using Silex, the PHP micro-framework based on the Symfony2 Components. Let's go for it in Listing 7.

```
require_once __DIR__.'/../vendor/autoload.php';

$app = new \Silex\Application();

// ... more routes

$app->get(
    '/api/rate/idea/{ideaId}/rating/{rating}',
    function($ideaId, $rating) use ($app) {
        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        return $app->json($response->idea);
    }
);

$app->run();
```

Is there anything familiar to you? Can you identify some code that you have seen before? I'll give you a clue.

```
$ideaRepository = new RedisIdeaRepository();
$useCase = new RateIdeaUseCase($ideaRepository);
$response = $useCase->execute(
    new RateIdeaRequest($ideaId, $rating)
);
```

"Man! I remember those 3 lines of code. They look exactly the same as the web application". That's right, because the UseCase encapsulates the business rules you need to prepare the request, get the response and act accordingly.

We are providing our users with another way for rating an idea; another *delivery mechanism*.

The main difference is where we created the `RateIdeaRequest` from. In the first example, it was from a ZF request and now it is from a Silex request using the parameters matched in the route.

Console app rating

Sometimes, a UseCase is going to be executed from a Cron job or the command line. As examples, batch processing or some testing command lines to accelerate the development.

While testing this feature using the web or the API, you realize that it would be nice to have a command line to do it, so you don't have to go through the browser.

If you are using shell scripts files, I suggest you to check the Symfony Console component. What would the code look like?

```php
namespace Idy\Console\Command;

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

class VoteIdeaCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('idea:rate')
            ->setDescription('Rate an idea')
            ->addArgument('id', InputArgument::REQUIRED)
            ->addArgument('rating', InputArgument::REQUIRED)
        ;
    }

    protected function execute(
        InputInterface $input,
        OutputInterface $output
    ) {
        $ideaId = $input->getArgument('id');
        $rating = $input->getArgument('rating');

        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $output->writeln('Done!');
    }
}
```

Again those 3 lines of code. As before, the UseCase and its business logic remain untouched, we are just providing a new *delivery mechanism*. Congratulations, you've discovered the *user side* hexagon edge.

There is still a lot to do. As you may have heard, a real craftsman does TDD. We have already started our story so we must be ok with just testing after.

Testing Rating an Idea UseCase

Michael Feathers introduced a definition of legacy code as *code without tests*. You don't want your code to be legacy just born, do you?

In order to unit test this UseCase object, you decide to start with the easiest part, what happens if the repository is not available? How can we generate such behavior? Do we stop our Redis server while running the unit tests? No. We need to have an object that has such behavior. Let's use a *mock* object in Listing 9.

```
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @test
     */
    public function whenRepositoryNotAvailableAnExceptionShouldBeThrown()
    {
        $this->setExpectedException('NotAvailableRepositoryException');
        $ideaRepository = new NotAvailableRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $useCase->execute(
            new RateIdeaRequest(1, 5)
        );
    }
}

class NotAvailableRepository implements IdeaRepository
{
    public function find($id)
    {
        throw new NotAvailableException();
    }

    public function update(Idea $idea)
    {
        throw new NotAvailableException();
```

```
    }
}
```

Nice. `NotAvailableRepository` has the behavior that we need and we can use it with `RateIdeaUseCase` because it implements `IdeaRepository` interface.

Next case to test is what happens if the idea is not in the repository. Listing 10 shows the code.

```
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function whenIdeaDoesNotExistAnExceptionShouldBeThrown()
    {
        $this->setExpectedException('IdeaDoesNotExistException');
        $ideaRepository = new EmptyIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $useCase->execute(
            new RateIdeaRequest(1, 5)
        );
    }
}

class EmptyIdeaRepository implements IdeaRepository
{
    public function find($id)
    {
        return null;
    }

    public function update(Idea $idea)
    {

    }
}
```

Here, we use the same strategy but with an `EmptyIdeaRepository`. It also implements the same interface but the implementation always returns `null` regardless which identifier the `find` method receives.

Why are we testing these cases?, remember Kent Beck's words: "Test everything that could possibly break".

Let's carry on with the rest of the feature. We need to check a special case that is related with having a read available repository where we cannot write to. Solution can be found in Listing 11.

```
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function whenUpdatingInReadOnlyAnIdeaAnExceptionShouldBeThrown()
    {
        $this->setExpectedException('NotAvailableRepositoryException');
        $ideaRepository = new WriteNotAvailableRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest(1, 5)
        );
    }
}

class WriteNotAvailableRepository implements IdeaRepository
{
    public function find($id)
    {
        $idea = new Idea();
        $idea->setId(1);
        $idea->setTitle('Subscribe to php[architect]');
        $idea->setDescription('Just buy it!');
        $idea->setRating(5);
        $idea->setVotes(10);
        $idea->setAuthor('hi@carlos.io');

        return $idea;
    }

    public function update(Idea $idea)
    {
        throw new NotAvailableException();
    }
```

```
}
```

Ok, now the key part of the feature is still remaining. We have different ways of testing this, we can write our own mock or use a mocking framework such as Mockery or Prophecy. Let's choose the first one. Another interesting exercise would be to write this example and the previous ones using one of these frameworks.

```
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function whenRatingAnIdeaNewRatingShouldBeAddedAnIdeaUpdated()
    {
        $ideaRepository = new OneIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest(1, 5)
        );

        $this->assertSame(5, $response->idea->getRating());
        $this->assertTrue($ideaRepository->updateCalled);
    }
}

class OneIdeaRepository implements IdeaRepository
{
    public $updateCalled = false;

    public function find($id)
    {
        $idea = new Idea();
        $idea->setId(1);
        $idea->setTitle('Subscribe to php[architect]');
        $idea->setDescription('Just buy it!');
        $idea->setRating(5);
        $idea->setVotes(10);
        $idea->setAuthor('hi@carlos.io');

        return $idea;
```

```
    }

    public function update(Idea $idea)
    {
        $this->updateCalled = true;
    }
}
```

Bam! 100% Coverage for the UseCase. Maybe, next time we can do it using TDD so the test will come first. However, testing this feature was really easy because of the way decoupling is promoted in this architecture.

Maybe you are wondering about this:

```
$this->updateCalled = true;
```

We need a way to guarantee that the update method has been called during the UseCase execution. This does the trick. This *test double* object is called a *spy*, *mocks* cousin.

When to use mocks? As a general rule, use mocks when crossing boundaries. In this case, we need mocks because we are crossing from the domain to the persistence boundary.

What about testing the infrastructure?

Testing Infrastructure

If you want to achieve 100% coverage for your whole application you will also have to test your infrastructure. Before doing that, you need to know that those unit tests will be more coupled to your implementation than the business ones. That means that the probability to be broken with implementation details changes is higher. So it's a trade-off you will have to consider.

So, if you want to continue, we need to do some modifications. We need to decouple even more. Let's see the code in Listing 13.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $useCase = new RateIdeaUseCase(
            new RedisIdeaRepository(
```

```
                new \Predis\Client()
            )
        );

        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $this->redirect('/idea/'.$response->idea->getId());
    }
}

class RedisIdeaRepository implements IdeaRepository
{
    private $client;

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

    // ...

    public function find($id)
    {
        $idea = $this->client->get($this->getKey($id));
        if (!$idea) {
            return null;
        }

        return $idea;
    }
}
```

If we want to 100% unit test `RedisIdeaRepository` we need to be able to pass the `Predis\Client` as a parameter to the repository without specifying TypeHinting so we can pass a mock to force the code flow necessary to cover all the cases.

This forces us to update the Controller to build the Redis connection, pass it to the repository and pass the result to the UseCase.

Now, it's all about creating mocks, test cases and having fun doing asserts.

Arggg, So Many Dependencies!

Is it normal that I have to create so many dependencies by hand? No. It's common to use a Dependency Injection component or a Service Container with such capabilities. Again, Symfony comes to the rescue, however, you can also check PHP-DI 4 http://php-di.org/.

Let's see the resulting code in Listing 14 after applying Symfony Service Container component to our application.

```
class IdeaController extends ContainerAwareController
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $useCase = $this->get('rate_idea_use_case');
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $this->redirect('/idea/'.$response->idea->getId());
    }
}
```

The controller has been modified to have access to the container, that's why it is inheriting from a new base controller `ContainerAwareController` that has a `get` method to retrieve each of the services contained.

```
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services
           http://symfony.com/schema/dic/services/services-1.0.xsd">
    <services>
        <service
            id="rate_idea_use_case"
            class="RateIdeaUseCase">
            <argument type="service" id="idea_repository" />
        </service>

        <service
```

```
            id="idea_repository"
            class="RedisIdeaRepository">
            <argument type="service">
                <service class="Predis\Client" />
            </argument>
        </service>
    </services>
</container>
```

In Listing 15, you can also find the XML file used to configure the Service Container. It's really easy to understand but if you need more information, take a look to the Symfony Service Container Component site in http://symfony.com/doc/current/book/service_container.html

Domain Services and Notification Hexagon Edge

Are we forgetting something? "the author should be notified by email", yeah! That's true. Let's see in Listing 16 how we have updated the UseCase for doing the job.

```
class RateIdeaUseCase
{
    private $ideaRepository;
    private $authorNotifier;

    public function __construct(
        IdeaRepository $ideaRepository,
        AuthorNotifier $authorNotifier
    )
    {
        $this->ideaRepository = $ideaRepository;
        $this->authorNotifier = $authorNotifier;
    }

    public function execute(RateIdeaRequest $request)
    {
        $ideaId = $request->ideaId;
        $rating = $request->rating;

        try {
            $idea = $this->ideaRepository->find($ideaId);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
```

```
        }

        if (!$idea) {
            throw new IdeaDoesNotExistException();
        }

        try {
            $idea->addRating($rating);
            $this->ideaRepository->update($idea);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }

        try {
            $this->authorNotifier->notify(
                $idea->getAuthor()
            );
        } catch(Exception $e) {
            throw new NotificationNotSentException();
        }

        return $idea;
    }
}
```

As you realize, we have added a new parameter for passing `AuthorNotifier` Service that will send the email to the author. This is the *port* in the "Ports and Adapters" naming. We have also updated the business rules in the `execute` method.

Repositories are not the only objects that may access your infrastructure and should be decoupled using interfaces or abstract classes. Domain Services can too. When there is a behavior not clearly owned by just one Entity in your domain, you should create a Domain Service. A typical pattern is to write an abstract Domain Service that has some concrete implementation and some other abstract methods that the *adapter* will implement.

As an exercise, define the implementation details for the AuthorNotifier abstract service. Options are SwiftMailer or just plain `mail` calls. It's up to you.

Let's Recap

In order to have a *clean architecture* that helps you create easy to write and test applications, we can use Hexagonal Architecture. To achieve that, we encapsulate user story business rules inside a UseCase or Interactor object. We build the UseCase request from our framework request, instantiate the UseCase and all its dependencies and then execute it. We get the response and act accordingly based on it. If our framework has a Dependency Injection component you can use it to simplify the code.

The same UseCase objects can be used from different *delivery mechanisms* in order to allow users access the features from different clients (web, API, console, etc.)

For testing, play with mocks that behave like all the interfaces defined so special cases or error flows can also be covered. Enjoy the good job done.

Hexagonal Architecture

In almost all the blogs and books you will find drawings about concentric circles representing different areas of software. As Robert C. Martin explains in his “Clean Architecture” post, the outer circle is where your infrastructure resides. The inner circle is where your Entities live. The overriding rule that makes this architecture work is **The Dependency Rule**. This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle.

Key Points

Use this approach if 100% unit test code coverage is important to your application. Also, if you want to be able to switch your storage strategy, web framework or any other type of third-party code. The architecture is especially useful for long-lasting applications that need to keep up with changing requirements.

What’s Next?

If you are interested in learning more about Hexagonal Architecture and other near concepts you should review the related URLs provided at the beginning of the article, take a look at CQRS and Event Sourcing. Also, don’t forget to subscribe to google groups and RSS about DDD such as http://dddinphp.org and follow on Twitter people like @VaughnVernon, and @ericevans0.