



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

Carlos Buenosvinos, Christian Soronellas and Keyvan Akbary

This book is for sale at <http://leanpub.com/ddd-in-php>

This version was published on 2016-06-28

ISBN 978-0-9946084-1-3



Leanpub

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

© 2014 - 2016 Carlos Buenosvinos, Christian Soronellas and Keyvan Akbary

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>

Contents

1.	Foreword by Vaughn Vernon	1
2.	Foreword by Matthias Noback	2
3.	Preface	4
3.1	Who Should Read This Book	5
3.2	DDD and PHP Community	6
3.3	Summary of Chapters	6
3.3.1	Chapter 1: Getting Started with Domain-Driven Design	6
3.3.2	Chapter 2: Architectural Styles	6
3.3.3	Chapter 3: Value Objects	6
3.3.4	Chapter 4: Entities	7
3.3.5	Chapter 5: Domain Services	7
3.3.6	Chapter 6: Domain Events	7
3.3.7	Chapter 7: Modules	7
3.3.8	Chapter 8: Aggregates	7
3.3.9	Chapter 9: Factories	7
3.3.10	Chapter 10: Repositories	8
3.3.11	Chapter 11: Application	8
3.3.12	Chapter 12: Integrating Bounded Contexts	8
3.3.13	Appendix A: Hexagonal Architecture with PHP	8
3.4	Code and Examples	8
4.	Acknowledgements	9
5.	About the Authors	11
5.1	Carlos Buenosvinos	11
5.2	Christian Soronellas	11
5.3	Keyvan Akbary	12
6.	Value Objects	13
6.1	Definition	13
6.2	Value Object vs. Entity	14
6.3	Currency and Money Example	14
6.4	Characteristics	16

CONTENTS

6.4.1	Measures, Quantifies, or Describes	17
6.4.2	Immutability	17
6.4.3	Conceptual Whole	19
6.4.4	Value Equality	20
6.4.5	Replaceability	22
6.4.6	Side-Effect-Free Behavior	22
6.5	Basic Types	24
6.6	Testing Value Objects	25
6.7	Persisting Value Objects	26
6.7.1	Persisting Single Value Objects	27
6.7.1.1	Embedded Value with an Ad Hoc ORM	28
6.7.1.2	Embedded Value (Embeddables) with Doctrine >= 2.5.*	30
6.7.1.3	Embedded Value with Doctrine <= 2.4.*	33
6.7.1.4	Serialized LOB and Ad Hoc ORM	36
6.7.1.4.1	Improved Serialization with JMS Serializer	37
6.7.1.5	Serialized LOB with Doctrine	38
6.7.1.5.1	Doctrine Object Mapping Type	38
6.7.1.5.2	Doctrine Custom Types	41
6.7.2	Persisting a Collection of Value Objects	44
6.7.2.1	Collection Serialized into a Single Column	45
6.7.2.2	Collection Backed by a Join Table	45
6.7.2.2.1	Collection Backed by a Join Table with Doctrine	46
6.7.2.2.2	Collection Backed by a Join Table with an Ad Hoc ORM	50
6.7.2.3	Collection Backed by a Database Entity	50
6.7.3	NoSQL	51
6.7.3.1	PostgreSQL JSONB and MySQL JSON Type	51
6.8	Security	51
6.9	Wrap-Up	52
7.	Bibliography	53
8.	Appendix A: Hexagonal Architecture with PHP	54
8.1	Introduction	54
8.2	First Approach	54
8.3	Repositories and the Persistence Edge	56
8.4	Decoupling Business and Persistence	59
8.5	Migrating our Persistence to Redis	60
8.6	Decouple Business and Web Framework	62
8.7	Rating an idea using the API	65
8.8	Console app rating	66
8.9	Testing Rating an Idea UseCase	68
8.10	Testing Infrastructure	72
8.11	Arggg, So Many Dependencies!	74

CONTENTS

8.12	Domain Services and Notification Hexagon Edge	75
8.13	Let's Recap	76
8.14	Hexagonal Architecture	77
8.15	Key Points	77
8.16	What's Next?	77

1. Foreword by Vaughn Vernon

TBW

Vaughn Vernon

Author of *Implementing Domain-Driven Design*¹ and *Domain-Driven Design Distilled*²

¹<http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8>

²<http://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420>

2. Foreword by Matthias Noback

I must admit that when I first heard of the *Domain-Driven Design in PHP* initiative, I was a bit worried. The danger was twofold: first of all, when glancing over the table of contents, the subject matter looked like it was a rehash of content that was already available in several other Domain-Driven Design books. Second, writing a book on Domain-Driven Design targeted specifically toward the PHP community seemed needlessly narrowing, particularly as Domain-Driven Design itself is not language specific. As such, this book might inhibit PHP developers from looking past the boundaries of their own community, especially when considering that there's a lot going on beyond the scope of PHP. In fact, even Domain-Driven Design is one of those things, as it did not originate in the PHP community.

After reading the book, I'm happy to inform you that my worries have been invalidated!

With regard to my first concern: of course there is some overlap with previously published Domain-Driven Design books. Yet the authors have restrained themselves. The theoretical parts are exactly what you need to be able to understand what's going on in the code samples. Besides, if you never read another Domain-Driven Design book, this book gives you what you need to start applying some of its principles and patterns in your code, as it's practical by nature.

My second concern — about the PHP aspect of this book — has been addressed very well. It turns out there are a lot of things to say about Domain-Driven Design *in a PHP world*. This book is specifically targeted at an audience consisting of PHP developers. The code samples resemble real-world PHP projects, and they use a programming style we know from projects using Symfony or Silex. For persisting domain objects, Doctrine ORM — which is the *de facto* standard data mapper for PHP — is used.

This book also fulfills a need I've often recognized in the PHP community: the need for concrete examples. For authors, it's challenging to come up with proper examples of how to apply certain ideas that have a low risk of being misinterpreted or abused in real-world projects — even more so in Domain-Driven Design, which is philosophical by nature.

In the case of this book, the authors haven't been afraid to show many useful examples, along with some interesting alternative solutions. They aren't just handwaving at the solution; they take the time to provide a full disclosure on the subject — for example, when they talk about saving snapshots for aggregates with a large number of domain events, or when they discuss integrating bounded contexts using RabbitMQ. I can't recall having previously seen an implementation of these things in a book or article on Domain-Driven Design.

For me personally, Domain-Driven Design is one the most interesting subjects in software development today. There is so much to discover, and there are many subjects related to it: agile software development, TDD, and BDD, but also living documentation, visualization, and knowledge crunching techniques. Once you start looking into all of this, you'll realize that Domain-Driven

Design is an area of expertise worth investigating, as it enables you to add much more to your own worth as a software developer.

So, I guess what I want to say is this: dive into this book, learn from it, and then pick up another book (see the list of references at the end of this book for suggestions of future reading). Continuous learning is a fundamental part of keeping up to date in the software industry, so don't stop here.

Oh, and by the way: if you get a chance to go to Barcelona, make sure you take part in one of the many PHP or Symfony events. The community is big, friendly, and full of interesting ideas. You'll find the authors of this book there too. They are all invested in the local PHP community and are happy to share their insights and experiences with you!

Matthias Noback

Author of *A Year with Symfony*¹

¹<https://leanpub.com/a-year-with-symfony>

3. Preface

In 2014, after two years of reading about and working with Domain-Driven Design, Christian and Carlos, friends and workmates, traveled to Berlin to participate in Vaughn Vernon’s *Implementing Domain-Driven Design* 3-Day Workshop. The training was fantastic, and all the concepts that were swirling around in their minds suddenly became very real during the trip. However, they were the only two PHP developers in a room full of Java and .NET ones.

Around the same time, php[tek], an annual PHP conference, opened its call for papers, and Carlos sent one about Hexagonal Architecture. His talk was rejected, but Eli White — of musketeers.me and php[architect] fame — got in touch with him a month later wondering if he was interested in writing an article about Hexagonal Architecture for the magazine php[architect]. So in June 2014, “Hexagonal Architecture with PHP” was published. That article was the origin of this book. You’ll find the article included in the appendix.

Carlos has been leading agile teams of between 20 and 100 people since 2006. He’s been a Certified ScrumMaster since 2010 and has helped many different companies and teams facing the challenge of writing code that is easy and cheap to maintain. Domain-Driven Design has played a significant role in his experience of keeping the speed high when dealing with big teams and companies running multiple products. Alongside Carlos, Christian has worked as Lead Architect for six years at Emagister and Atrapalo, sharing the same experience of applying and teaching Domain-Driven Design.

In late 2015, Carlos and Christian talked about extending the article and sharing all their knowledge and experience of applying Domain-Driven Design in production. They were very excited about the idea behind the book: helping the PHP community delve into Domain-Driven Design from a practical approach. At that time, concepts such as Rich Domain Models and Applications that were framework agnostic were not so common in the PHP community. In December 2015, the first commit to the GitHub book repository was pushed.

Around the same time, in a parallel universe, Keyvan co-founded Funddy, a crowdfunding platform for the masses built on top of the concepts and building blocks of Domain-Driven Design. Domain-Driven Design proved itself effective in the exploratory process and modeling of building an early-stage startup like Funddy. It also helped handling the complexity of the company, with its constantly changing environment and requirements. So after connecting and discussing with Carlos and Christian, Keyvan proudly signed on as the third writer.

Together we’ve written the book we wanted to have when we started with Domain-Driven Design. It’s full of examples, production-ready code, shortcuts, and our recommendations based on our experiences of what worked and what didn’t for our respective teams. We arrived at Domain-Driven Design via its building blocks — Tactical Patterns — which is why this book is mainly about them. Reading it will help you learn them, write them, and go deep into their implementations. You’ll

also discover how to integrate bounded contexts using synchronous and asynchronous approaches, which will open your world to strategic design — though the latter is a road you'll have to discover on your own.

This book is heavily inspired by *Implementing Domain-Driven Design*¹ by Vaughn Vernon (aka the *Red Book*), and *Domain-Driven Design: Tackling Complexity in the Heart of Software*² by Eric Evans (aka the *Blue Book*). You should buy both books. You should read them carefully. You should love them.

3.1 Who Should Read This Book

This book is highly recommended to PHP Developers, Architects, and Tech Leads. It will help you become a better professional. It will give you a new overview of and approach to the applications you're developing. If you're a Junior profile, getting into Value Objects, Entities, Repositories, and Domain Events is really important in order to model any Domain you will face in the future. For an average profile, understanding the benefits of Hexagonal Architecture and the boundaries between your framework and your Application is key for writing code that's easier to maintain in the real world (framework migrations, testing, etc.). More advanced readers will have fun both exploring how to use Domain Events in order to integrate applications and getting deeper into Aggregate design.

Although Domain-Driven Design is not about technology, you still need it to make HTTP requests to access your Domain. We recommend using specific PHP frameworks and libraries — such as Symfony, Silex, and Doctrine — throughout the book. For some examples, we also use specific technologies such as MySQL, RabbitMQ, Redis, and Elasticsearch. However, most important are the concepts behind the scenes — concepts that are applicable regardless of the technology used to implement them. We will also encourage you to properly use Domain-Driven Design building blocks to build PHP applications that can work with different stacks — even at the same time.

The book is also loaded with tons of details and examples, such as how to properly design and implement all the building blocks of Domain-Driven Design — including Value Objects, Entities, Services, Domain Events, Aggregates, Factories, Repositories, and Application Services — with PHP. They explain what the role of the main PHP libraries and frameworks used today (Doctrine, Symfony, Silex, etc.) in Domain-Driven Design are. They teach how to apply Hexagonal Architecture within your application, regardless of whether you use an open source framework or your own one. They show how to integrate Bounded Contexts using REST frameworks and messaging mechanisms. If you're interested in any of these subjects, this book is for you.

¹<http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577>

²<http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215>

3.2 DDD and PHP Community

In 2016, Christian and Carlos went to the first official Domain-Driven Design conference, [DDD Europe](#)³. They were really happy to see some PHP open source leaders, such as Marco Pivetta (Doctrine) and Sebastian Bergmann (PHPUnit), attending the conference.

Two years before that, Domain-Driven Design arrived in the PHP community. However, there is still a lack of documentation and real code examples. Why? We think not many people have worked with this kind of approach in production yet — even people in other more established communities such as Java. Maybe this is because their project complexity is low, or maybe it's because they don't know how to do it. Whatever the reason, this book is written for the community. One of our goals is to teach you how you can write an Application that solves your Domain issues without getting coupled to specific frameworks or technologies.

3.3 Summary of Chapters

The book is arranged with each chapter exploring a separate tactical building block of Domain-Driven Design. It also includes an introduction to Domain-Driven Design, information on how to integrate different Bounded Contexts or Applications, and some interesting appendixes.

3.3.1 Chapter 1: Getting Started with Domain-Driven Design

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?

3.3.2 Chapter 2: Architectural Styles

Bounded Contexts can be implemented in different ways and using different approaches. However, two styles are getting more popular, and they are Hexagonal Architecture and CQRS + ES. In this chapter, we'll see these two main Architectural Styles, understand what their main strengths are, and discover when to use them.

3.3.3 Chapter 3: Value Objects

Value Objects are the basic pieces for rich modeling. We'll learn what their properties are and what makes them so important. We'll check how to persist them using Doctrine and custom ORMs. We'll show how to properly validate and unit test them. And finally, we'll see what a test case of testing immutability looks like.

³<http://dddeurope.com/>

3.3.4 Chapter 4: Entities

Entities are the Domain-Driven Design building blocks that are uniquely identified and mutable. We'll see how to create and validate them and how to properly map them using a custom ORM and Doctrine. We'll also access whether or not annotations are the best mapping approach for Entities and look at the different strategies for generating an identity.

3.3.5 Chapter 5: Domain Services

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

3.3.6 Chapter 6: Domain Events

Domain Events are a great Inversion of Control (IoC) mechanism. In Domain-Driven Design, they are important for communicating different Bounded Contexts asynchronously, improving your Application performance using eventual consistency, and decoupling your Application from its infrastructure.

3.3.7 Chapter 7: Modules

With so many tactical building blocks, it's a bit difficult to know where to place them in code, especially if you are dealing with a framework like Symfony. We'll review how PHP namespaces can be used for implementing Modules. We'll also check different hierarchies of folders for organizing Domain Model code, Application Code, and Infrastructure code.

3.3.8 Chapter 8: Aggregates

Aggregates are probably the most difficult part of tactical Domain-Driven Design. We'll look at the key concepts when dealing with them and discover how to design them. We'll also propose a practical scenario where two aggregates become one when adding a business rule and demonstrate how the rest of the objects must be refactored.

3.3.9 Chapter 9: Factories

Factory methods and objects help us keep business invariants, which is why they're so important in Domain-Driven Design. Here, we'll also check the relation between Factories and Aggregates.

3.3.10 Chapter 10: Repositories

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

3.3.11 Chapter 11: Application

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 that they're easy to test and keep thin. We'll also review how to prepare request objects, define dependencies, and return results.

3.3.12 Chapter 12: Integrating Bounded Contexts

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 is our suggestion for asynchronous communication.

3.3.13 Appendix A: Hexagonal Architecture with PHP

Here is where you'll find the original article written by Carlos and published by php|architect in June 2014.

3.4 Code and Examples

The authors have created an organization at GitHub called *Domain-Driven Design in PHP*⁴, which is where all the code examples from this book, additional snippets, and some complete sample projects are available. For example, you can find “Last Wishes”⁵, a simple Domain-Driven Design-style application showing different examples explained in this book. Additionally, you'll find our “CQRS Blog”⁶, along with “Gamify”⁷, a Bounded Context that adds gamification capabilities to “Last Wishes.” Finally, if you find any issue or fix or have a suggestion or comment while reading this book, you can create an issue in the “DDD in PHP Book Issues”⁸ repository. We fix them as they come in. If you're interested, we also urge you to watch those projects and provide feedback.

⁴<https://github.com/dddinphp>

⁵<https://github.com/dddinphp/last-wishes>

⁶<https://github.com/dddinphp/blog-cqrs>

⁷<https://github.com/dddinphp/last-wishes-gamify>

⁸<https://github.com/dddinphp/ddd-in-php-book-issues>

4. Acknowledgements

First of all, we would like to thank all our friends and family. Without their support, writing this book would have been an even more difficult task. Thanks for accommodating our schedules and taking care of our children in order to free up time for us to focus on writing. You are wonderful, and part of this book is also yours.

We are three Spaniards who wrote a book in English, so if you'd guess our English is far from perfect, you'd be correct. Luckily for us, Edd Mann has been supporting us with the language since the beginning. He's not just a great collaborator but also a great friend, and we owe him a huge thanks.

A group of PHP developers in Barcelona defends what we call *el camino del rigor*, or *the path of rigor*. It existed before the *craftsmanship* movement, and it means to struggle with everything stacked against us in order to build exceptional things in an exceptional way. Two particular developers and friends from that group are Albert Casademont and Ricard Clau, both of whom are extraordinary people committed to the community. Thank you so much for helping with the revision process. Your contributions have been incredibly valuable.

We would like to thank every developer who has worked with us in the companies where we've applied Domain-Driven Design. We know you have been struggling when learning and applying these concepts. Some of you were not so open-minded at the beginning, but after using the basic building blocks for a while, you became evangelists. Thanks for your faith.

Our book was for sell from the moment we put the first chapters on [Leanpub](https://leanpub.com/ddd-in-php)¹. Early adopters who bought the book at the beginning gave us the much needed love and support to keep pushing to get this done. Writing our first book was difficult, but even more difficult was choosing a subject such as Domain-Driven Design — particularly since all three of us are extremely detail oriented. So thanks to all the early buyers for the motivation to keep going.

Thanks also to Matthias Noback for his foreword and feedback on the book. The end result is better because of his contributions.

A special mention to [Vaughn Vernon](https://vaughnvernon.co/)² — not just because his work was an incredible source of information and inspiration for us, but also because he has helped us find a good publisher, given us valuable advice, and shared ideas with us. Thanks so much for your help.

Last but not least, we would like to express our gratitude to all the people who have reported issues, made suggestions, and otherwise contributed to our [GitHub repository](https://github.com/dddinphp/ddd-in-php-book-issues)³. To all of you, thank you. You have helped us make this book better. More importantly, you've helped the community grow

¹<https://leanpub.com/ddd-in-php>

²<https://vaughnvernon.co/>

³<https://github.com/dddinphp/ddd-in-php-book-issues>

and helped other developers be better developers. As [Robert C. Martin](https://twitter.com/unclebobmartin)⁴ wrote in his book, *Clean Code: A Handbook of Agile Software Craftsmanship*⁵, “You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.” 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 Roviroso, Luis Cordova, Raúl Ramos, Juan Maturana, Nil Portugués, Nikolay Zujev, Fernando Pradas, Raúl Araya, Neal Brooks, Hubert Béague, Aleksander Reksć, and Marc Aube.

⁴<https://twitter.com/unclebobmartin>

⁵<http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>

5. About the Authors

5.1 Carlos Buenosvinos

Carlos is a PHP Extreme Programmer with more than 15 years of experience developing web applications and more than 10 years experience as a Tech Lead and CTO leading teams of between 20 and 100 people. He is a Certified ScrumMaster (CSM) and has coached and trained close to two dozen different companies in Agile practices, both as an employee and as a consultant. On the technical side, he is a Zend PHP Engineer, a Zend Framework Engineer, and MySQL certified. He is also a board member of the PHP Barcelona User Group. He has worked in e-commerce (Atrapalo and eBay), payment processing (Vendo), classifieds (Emagister), and B2B recruiting tools (XING). He is interested in JavaScript, DevOps, and Scala. He likes developing for mobile, Raspberry Pi, and games.

- Twitter: [@buenosvinos](#)¹
- Web: <https://carlosbuenosvinos.com>²
- GitHub: <https://github.com/carlosbuenosvinos>³

5.2 Christian Soronellas

Christian is a passionate Software Developer, Software Journeyman, and Craftsman Apprentice. He's an Extreme Programmer soul with more than 10 years of experience in web development. He's also a Zend PHP 5.3 Certified Engineer, a Zend Framework Certified Engineer, and a SensioLabs Certified Symfony Developer. He has worked as a freelancer, as well as at Privalia, Emagister, Atrapalo, and Enalquiler as a Software Architect.

- Twitter: [@theUniC](#)⁴
- GitHub: <https://github.com/theUniC>⁵

¹<https://twitter.com/buenosvinos>

²<https://carlosbuenosvinos.com>

³<https://github.com/carlosbuenosvinos>

⁴<https://twitter.com/theUniC>

⁵<https://github.com/theUniC>

5.3 Keyvan Akbary

Keyvan is a polyglot Software Developer who loves Software fundamentals, the Craftsmanship movement, Extreme Programming, SOLID principles, Clean Code, Design Patterns, and Testing. He's also a sporadic Functional Programmer. He understands technology as a medium for providing value. He has worked on countless projects as a freelancer, on video streaming (Youzee), and on an online marketplace (MyBuilder) – all in addition to founding a crowdfunding company (Funddy). Currently, Keyvan is working in FinTech as a Lead Developer at TransferWise London.

- Twitter: [@keyvanakbary](https://twitter.com/keyvanakbary)⁶
- Web: <http://keyvanakbary.com>⁷
- GitHub: <https://github.com/keyvanakbary>⁸

⁶<https://twitter.com/keyvanakbary>

⁷<http://keyvanakbary.com>

⁸<https://github.com/keyvanakbary>

6. Value Objects

Value Objects are a fundamental building block of Domain-Driven Design, and they're 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. Value Objects 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 modeled using a Value Object. In this case, it's 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 there's very little overhead. As a result, new instance creation is favored over reference reuse, even when being used to represent the same value. Equality is then checked based on the comparability of the fields of both instances.

6.1 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 name, middle name, last name, and title), currencies, colors, phone numbers, and postal addresses.



Exercise

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

6.2 Value Object vs. Entity

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

Value Object:

When people exchange dollar bills, they generally do not distinguish between each unique bill; they only are concerned about the face value of the dollar bill. In this context, dollar bills are value objects. However, the Federal Reserve may be concerned about each unique bill; in this context each bill would be an entity.

Entity:

Most airlines distinguish each seat uniquely on every flight. Each seat is an entity in this context. However, Southwest Airlines, EasyJet and Ryanair do not distinguish between every seat; all seats are the same. In this context, a seat is actually a value object.



Exercise

Think about the concept of an address (street, number, zip code, etc.). What is a possible context where an address could be modeled as an Entity and not as a Value Object? Discuss your findings with a peer.

6.3 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 a floating-point rounding issue, which in turn allows 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-letter uppercase ISO code:

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

⁴<http://martinfowler.com/eaCatalog/money.html>

```
1 class Currency
2 {
3     private $isoCode;
4
5     public function __construct($anIsoCode)
6     {
7         $this->setIsoCode($anIsoCode);
8     }
9
10    private function setIsoCode($anIsoCode)
11    {
12        if (!preg_match('/^[A-Z]{3}$/', $anIsoCode)) {
13            throw new \InvalidArgumentException();
14        }
15
16        $this->isoCode = $anIsoCode;
17    }
18
19    public function isoCode()
20    {
21        return $this->isoCode;
22    }
23 }
```

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 behavior for a given concept.



Extra Validations for Currency

In the previous code example, we can build a *Currency* with an *AAA* currency ISO code. That is not valid at all. Write a more specific rule that will check if the ISO Code is valid. A full list of valid currency ISO codes can be found [here](http://www.xe.com/iso4217.php)⁵. If you need help, take a look at the *Money*⁶ packagist library.

Money is used to measure a specific amount of currency. It's modeled 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 — e.g. in the case of USD or EUR, cents.

As a bonus, you might also notice that we're using [self encapsulation](http://martinfowler.com/bliki/SelfEncapsulation.html)⁷ to set the ISO code, which centralizes changes in the Value Object itself:

⁵<http://www.xe.com/iso4217.php>

⁶<https://github.com/moneyphp/money>

⁷<http://martinfowler.com/bliki/SelfEncapsulation.html>

```
1 class Money
2 {
3     private $amount;
4     private $currency;
5
6     public function __construct($anAmount, Currency $aCurrency)
7     {
8         $this->setAmount($anAmount);
9         $this->setCurrency($aCurrency);
10    }
11
12    private function setAmount($anAmount)
13    {
14        $this->amount = (int) $anAmount;
15    }
16
17    private function setCurrency(Currency $aCurrency)
18    {
19        $this->currency = $aCurrency;
20    }
21
22    public function amount()
23    {
24        return $this->amount;
25    }
26
27    public function currency()
28    {
29        return $this->currency;
30    }
31 }
```

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

6.4 Characteristics

While modeling an Ubiquitous Languages concept in code, you should always favor Value Objects over Entities. Value Objects are easier to create, test, use, and maintain.

Keeping this in mind, you can decide whether or not the concept in question can be modeled 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 behavior.

6.4.1 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 money it is. The `Money` object measures or quantifies units of a given `Currency`.

6.4.2 Immutability

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

As such, Value Objects should be created through their constructors. 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's 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](#)⁸, which is considered an anti-pattern.

It's 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](#)⁹, 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](#)¹⁰. These specific factory methods are also known as semantic constructors. The main goal of `fromMoney` is to provide more contextual meaning than the plain constructor. More radical approaches propose to make the `_construct` method private and build every instance using a semantic constructor.

In our `Money` object, we could add some useful factory methods like the following:

⁸<http://www.martinfowler.com/bliki/AnemicDomainModel.html>

⁹http://en.wikipedia.org/wiki/Function_overloading

¹⁰http://en.wikipedia.org/wiki/Factory_method_pattern

```
1 class Money
2 {
3     // ...
4
5     public static function fromMoney(Money $aMoney)
6     {
7         return new self(
8             $aMoney->amount(),
9             $aMoney->currency()
10        );
11    }
12
13    public static function ofCurrency(Currency $aCurrency)
14    {
15        return new self(0, $aCurrency);
16    }
17 }
```

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 affect these factory methods. This small implementation detail helps 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 that are commonplace in a stateful context. If we require a state change, we now have to 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 instead return a new `Money` instance with the desired modifications. Fortunately, it is relatively simple to abide by this rule, as shown in the example below:


```
1 class Money
2 {
3     // ...
4
5     public function increaseAmountBy($anAmount)
6     {
7         return new self(
8             $this->amount() + $anAmount,
9             $this->currency()
10        );
11    }
12 }
```

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:

```
1 $aMoney = new Money(100, new Currency('USD'));
2 $otherMoney = $aMoney->increaseAmountBy(100);
3
4 var_dump($aMoney === $otherMoney); // bool(false)
5
6 $aMoney = $aMoney->increaseAmountBy(100);
7 var_dump($aMoney === $otherMoney); // bool(false)
```

6.4.3 Conceptual Whole

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

```
1 class Product
2 {
3     private $id;
4     private $name;
5
6     /**
7      * @var int
8      */
9     private $amount;
10
11    /**
12     * @var string
13     */
```

```
14     private $currency;  
15  
16     // ...  
17 }
```

This approach has some noticeable flaws, if say, for example, you want to validate the ISO. It doesn't really make sense for the `Product` to be responsible for the currency's ISO validation (thus violating 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 for being 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?

6.4.4 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, conceptualize two `Money` objects representing 1 USD. Can we consider them equal? In the "real world," are two bills of 1 USD valued the same? Of course they are. Directing our attention back to the code, the Value Objects in question refer to separate instances of `Money`. However, we can consider them to both represent the same value, so in turn they are *equal*.

In regards to PHP, it's commonplace to compare two Value Objects using the `==` operator. Examining the [PHP documentation](#)¹¹ definition of the operator highlights an interesting behavior:

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 behavior works in agreement with 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:

¹¹<http://php.net/manual/en/language.oop5.object-comparison.php>

```
1 $a = new Currency('USD');
2 $b = new Currency('USD');
3
4 var_dump($a == $b); // bool(true)
5 var_dump($a === $b); // bool(false)
6
7 $c = new Currency('EUR');
8
9 var_dump($a == $c); // bool(false)
10 var_dump($a === $c); // bool(false)
```

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 confirm that both their associated ISO codes are the same. The `===` operator does the job pretty well in this case:

```
1 class Currency
2 {
3     // ...
4
5     public function equals(Currency $currency)
6     {
7         return $currency->isoCode() === $this->isoCode();
8     }
9 }
```

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

```
1 class Money
2 {
3     // ...
4
5     public function equals(Money $money)
6     {
7         return
8             $money->currency()->equals($this->currency()) &&
9             $money->amount() === $this->amount();
10    }
11 }
```

6.4.5 Replaceability

Consider a Product Entity that contains a Money Value Object used to quantify its price. Consider also two Product Entities with an identical price — for example 100 USD. This scenario could be modeled 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 behavior can be considered an unexpected side effect. For example, if Carlos was hired on February 20, 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 of 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's recommended instead to replace the object as a whole:

```
1 $this->price = new Money(100, new Currency('USD'));
2 // ...
3 $this->price = $this->price->increaseAmountBy(200);
```

This kind of behavior 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; instead, a new value is returned.

6.4.6 Side-Effect-Free Behavior

If we want to include some additional behavior — like an `add` method — in our Money class, 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:

```
1 class Money
2 {
3     // ...
4
5     public function add(Money $money)
6     {
7         if ($money->currency() !== $this->currency()) {
8             throw new \InvalidArgumentException();
9         }
10
11         $this->amount += $money->amount();
12     }
13 }
```

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

```
1 class Banking
2 {
3     public function aMethod()
4     {
5         $aMoney = new Money(100, new Currency('USD'));
6         $this->otherMethod($aMoney);
7         // ...
8     }
9 }
```

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 behavior.

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

```
1 class Money
2 {
3     // ...
4
5     public function add(Money $money)
6     {
7         if (!$money->currency()->equals($this->currency())) {
8             throw new \InvalidArgumentException();
9         }
10
11         // start-of-new-code-to-take-a-look
12         return new self(
13             $money->amount() + $this->amount(),
14             $this->currency()
15         );
16         // end-of-new-code-to-take-a-look
17     }
18 }
```

With this simple change, immutability is guaranteed. Each time two instances of 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.

6.5 Basic Types

Consider the following code snippet:

```
1 $a = 10;
2 $b = 10;
3 var_dump($a == $b);
4 // bool(true)
5 var_dump($a === $b);
6 // bool(true)
7 $a = 20;
8 var_dump($a);
9 // integer(20)
10 $a = $a + 30;
11 var_dump($a);
12 // integer(50)
```

Although `$a` and `$b` are different variables stored in different memory locations, when compared, they are the same. They hold the same value, so we consider them equal. You can change the value of `$a` from 10 to 20 at any time that you want, making the new value 20 and eliminating the 10. You can replace integer values as much as you want without consideration of the previous value, because you are not modifying it; you are just replacing it. If you apply any operation on them, such as addition (i.e. `$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 when explicitly passed by reference, you are passing a value. It doesn't 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.

6.6 Testing Value Objects

Value Objects are tested in the same way normal objects are. However, the immutability and side-effect-free behavior 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:

```
1 class MoneyTest extends \PHPUnit_Test_TestCase
2 {
3     /**
4      * @test
5      */
6     public function copiedMoneyShouldRepresentSameValue()
7     {
8         $aMoney = new Money(100, new Currency('USD'));
9
10        $copiedMoney = Money::fromMoney($aMoney);
11
12        $this->assertTrue($aMoney->equals($copiedMoney));
13    }
14
15    /**
16     * @test
17     */
18    public function originalMoneyShouldNotBeModifiedOnAddition()
19    {
20        $aMoney = new Money(100, new Currency('USD'));
21
22        $aMoney->add(new Money(20, new Currency('USD')));
```

```
23
24     $this->assertEquals(100, $aMoney->amount());
25 }
26
27 /**
28  * @test
29  */
30 public function moneysShouldBeAdded()
31 {
32     $aMoney = new Money(100, new Currency('USD'));
33
34     $newMoney = $aMoney->add(new Money(20, new Currency('USD')));
35
36     $this->assertEquals(120, $newMoney->amount());
37 }
38
39 // ...
40 }
```

6.7 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 that's an option in some cases. Instead, it's 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 provides 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 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:

```
1 class Product
2 {
3     private $productId;
4     private $name;
5     private $price;
6
7     public function __construct(
8         $aProductId,
9         $aName,
```



```
10     Money $aPrice
11   ) {
12     $this->setProductId($aProductId);
13     $this->setName($aName);
14     $this->setPrice($aPrice);
15   }
16
17   // ...
18 }
```

Assuming you have a [Repository](#) for persisting Product Entities, an implementation to create and persist a new Product could look like this:

```
1 $product = new Product(
2     $productRepository->nextIdentity(),
3     'Domain-Driven Design in PHP',
4     new Money(999, new Currency('USD'))
5 );
6
7 $productRepository->persist($product);
```

Now let's look at both the ad hoc ORM and the Doctrine implementations that could be used to persist a Product Entity containing Value Objects. We will highlight the application of the Embedded Value and Serialized LOB patterns, along with the differences between persisting a single Value Object and a collection of them.



Why Doctrine?

[Doctrine](#)¹² is a great ORM. It solves 80 percent of the requirements a PHP application faces. It has a great community. With a correctly tuned setup, 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.

6.7.1 Persisting Single Value Objects

Many different options are available to persist a single Value Object. These range from using Serialized LOB or Embedded Value as mapping strategies, to using 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

¹²<http://www.doctrine-project.org/projects/orm.html>

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 [DBAL¹³](#) 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.

6.7.1.1 Embedded Value with an Ad Hoc ORM

If we're 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 one for its currency ISO code:

```

1 CREATE TABLE `products` (
2     id INT NOT NULL,
3     name VARCHAR(255) NOT NULL,
4     price_amount INT NOT NULL,
5     price_currency VARCHAR(3) NOT NULL
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

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`, you should take care to create the INSERT statement, bind the parameters, and execute it:

```

1 class DbalProductRepository extends DbalRepository implements ProductRepository
2 {
3     public function add(Product $aProduct)
4     {
5         $sql = 'INSERT INTO products VALUES (?, ?, ?, ?)';
6         $stmt = $this->connection()->prepare($sql);
7         $stmt->bindValue(1, $aProduct->id());
8         $stmt->bindValue(2, $aProduct->name());
9         $stmt->bindValue(3, $aProduct->price()->amount());
10        $stmt->bindValue(4, $aProduct->price()->currency()->isoCode());
11        $stmt->execute();
12
13        // ...
14    }
15 }
```

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

¹³<http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/>

```

1 mysql> select * from products \G
2 ***** 1. row *****
3         id: 1
4         name: Domain-Driven Design in PHP
5     price_amount: 999
6     price_currency: USD
7 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 in order 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:

```

1 class DbalProductRepository extends DbalRepository implements ProductRepository
2 {
3     public function productOfId($anId)
4     {
5         $sql = 'SELECT * FROM products WHERE id = ?';
6         $stmt = $this->connection()->prepare($sql);
7         $stmt->bindValue(1, $anId);
8         $res = $stmt->execute();
9         // ...
10
11        return new Product(
12            $row['id'],
13            $row['name'],
14            new Money(
15                $row['price_amount'],
16                new Currency(
17                    $row['price_currency']
18                )
19            )
20        );
21    }
22 }

```

There are some benefits to this approach. First, you can easily read step-by-step how the persistence and subsequent creation occur. 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 and 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.

This is one of the reasons why Doctrine uses internal proxies and `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 don't 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 to roll out your own ORM, be ready to solve some fundamental problems such as events, different constructors, Value Objects, lazy load relations, etc. That's why we recommend giving Doctrine a try for Domain-Driven Design 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, instead using a static factory method.

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

Doctrine stable release is currently 2.5 and it comes with support for mapping Value Objects, thereby removing the need to do this yourself as in Doctrine 2.4. Since December 2015, Doctrine has support for nested embeddables. The support is not 100 percent, but it's high enough to give it a try. In case it doesn't work for your scenario, take a look at the next section. For official documentation, check the [Doctrine Embeddables reference](#)¹⁴. This option, if implemented correctly, is definitely the one that we most recommend. It would be the simplest, most elegant solution, which also provides search support in your DQL queries.

Because `Product`, `Money`, and `Currency` classes have already been shown, the only remaining thing for this alternative is to show the Doctrine mapping files:

¹⁴<http://doctrine-orm.readthedocs.org/en/latest/tutorials/embeddables.html>

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping
3     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6     https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8     <entity
9         name="Product"
10        table="product">
11        <id
12            name="id"
13            column="id"
14            type="string"
15            length="255">
16            <generator
17                strategy="NONE">
18            </generator>
19        </id>
20
21        <field
22            name="name"
23            type="string"
24            length="255"
25        />
26
27        <embedded
28            name="price"
29            class="Ddd\Domain\Model\Money"
30        />
31    </entity>
32 </doctrine-mapping>
```

In the product mapping, we are defining price as an instance variable that will hold a Money instance. At the same time, Money is designed with an amount and a Currency instance:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping
3     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6     https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8     <embeddable
9         name="Ddd\Domain\Model\Money">
10
11         <field
12             name="amount"
13             type="integer"
14         />
15
16         <embedded
17             name="currency"
18             class="Ddd\Domain\Model\Currency"
19         />
20     </embeddable>
21 </doctrine-mapping>
```

Finally, it's time to show the Doctrine mapping for our Currency Value Object:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping
3     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6     https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8     <embeddable
9         name="Ddd\Domain\Model\Currency">
10
11         <field
12             name="iso"
13             type="string"
14             length="3"
15         />
16     </embeddable>
17 </doctrine-mapping>
```

As you can see, it's a standard embeddable definition with just one string field that holds the ISO code. This approach is the easiest way to do this and is much more effective. By default, Doctrine names your columns by prefixing them using the Value Object name. You can change this behavior to meet your needs by changing the `column-prefix` attribute in the XML notation.

6.7.1.3 Embedded Value with Doctrine <= 2.4.*

However, if you're stuck in Doctrine 2.4, what is an acceptable solution for using embedded values with Doctrine < 2.5? We need to now surrogate all the Value Object attributes in the Product Entity, which means creating 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:

```
1 class Product
2 {
3     private $productId;
4     private $name;
5     private $price;
6
7     // start-of-new-code-to-take-a-look
8     private $surrogateCurrencyIsoCode;
9     private $surrogateAmount;
10    // end-of-new-code-to-take-a-look
11
12    public function __construct($aProductId, $aName, Money $aPrice)
13    {
14        $this->setProductId($aProductId);
15        $this->setName($aName);
16        $this->setPrice($aPrice);
17    }
18
19    private function setPrice(Money $aMoney)
20    {
21        $this->price = $aMoney;
22        // start-of-new-code-to-take-a-look
23        $this->surrogateAmount = $aMoney->amount();
24        $this->surrogateCurrencyIsoCode = $aMoney->currency()->isoCode();
25        // end-of-new-code-to-take-a-look
26    }
27
28    // start-of-new-code-to-take-a-look
29    private function price()
```

```
30     {
31         if (null === $this->price) {
32             $this->price = new Money(
33                 $this->surrogateAmount,
34                 new Currency($this->surrogateCurrency)
35             );
36         }
37
38         return $this->price;
39     }
40     // end-of-new-code-to-take-a-look
41
42     // ...
43 }
```

As you can see, there are two new attributes: one for the amount, and another for the ISO code of the currency. We've also updated the `setPrice` method in order to keep attribute consistency when setting it. On top of this, we 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:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping
3     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6     https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8     <entity
9         name="Product"
10        table="product">
11
12        <id
13            name="id"
14            column="id"
15            type="string"
16            length="255">
17            <generator
18                strategy="NONE">
19            </generator>
20        </id>
21
22        <field
```



```
23         name="name"
24         type="string"
25         length="255"
26     />
27     // start-of-new-code-to-take-a-look
28     <field
29         name="surrogateAmount"
30         type="integer"
31         column="price_amount"
32     />
33     <field
34         name="surrogateCurrencyIsoCode"
35         type="string"
36         column="price_currency"
37     />
38     // end-of-new-code-to-take-a-look
39 </entity>
40 </doctrine-mapping>
```



Surrogate Attributes

These two new fields don't strictly belong to the Domain, as they don't refer to infrastructure details, rather are a 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 tradeoff, acceptable. There is another use of surrogate attributes in this book; you can find it when [surrogating Entity identities](#).

If we wanted to push these two attributes outside of the Domain, this could be achieved through the use of an [Abstract Factory](#)¹⁵. First, we need to create a new Entity, `DoctrineProduct`, in our Infrastructure folder. This Entity will 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'll 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're required to call `new Product`, but because we need to deal with `DoctrineProduct` and we don't want our Application Services to know about infrastructure details, we'll need to use Factories to create `Product` Entities. So, in every instance where Entity creation occurs with `new`, you will instead call `createProduct` on `ProductFactory`.

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

¹⁵http://en.wikipedia.org/wiki/Abstract_factory_pattern

6.7.1.4 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 easily be persisted and fetched. 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:

```

1 CREATE TABLE `products` (
2     id INT NOT NULL,
3     name VARCHAR(255) NOT NULL,
4     // start-of-new-code-to-take-a-look
5     price TEXT NOT NULL
6     // end-of-new-code-to-take-a-look
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

```

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:

```

1 class DbalProductRepository extends DbalRepository implements ProductRepository
2 {
3     public function add(Product $aProduct)
4     {
5         $sql = 'INSERT INTO products VALUES (?, ?, ?)';
6         $stmt = $this->connection()->prepare($sql);
7         $stmt->bindValue(1, $aProduct->id());
8         $stmt->bindValue(2, $aProduct->name());
9         // start-of-new-code-to-take-a-look
10        $stmt->bindValue(
11            3,
12            $this->serialize(
13                $aProduct->price()
14            )
15        );
16        // end-of-new-code-to-take-a-look
17
18        // ...
19    }
20
21    private function serialize($object)
22    {
23        return serialize($object);

```

```
24     }
25 }
```

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:

```
1  mysql> select * from products \G
2  ***** 1. row *****
3     id: 1
4     name: Domain-Driven Design in PHP
5     price: 0:22:"Ddd\Domain\Model\Money":2:{s:30:" Ddd\Domain\Model\Money amount";i:\
6 999;s:32:" Ddd\Domain\Model\Money currency";0:25:"Ddd\Domain\Model\Currency":1:{\
7 s:34:" Ddd\Domain\Model\Currency isoCode";s:3:"USD";}}
8 1 row in set (0.00 sec)
```

This approach does the job. However, it's not recommended due to problems occurring when refactoring classes in your code. Could you imagine the problems if we decided to rename our `Money` class? Could you imagine the changes that would be required in our database representation when moving the `Money` class from one namespace to another? Another tradeoff, as explained before, is the lack of querying capabilities. It doesn't matter whether you use Doctrine or not; writing a query to get the products cheaper than, say, 200 USD is almost impossible while 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 specialized library for handling serialization processes.

6.7.1.4.1 Improved Serialization with JMS Serializer

`serialize/unserialize` native PHP strategies have a problem when dealing with class and namespace refactoring. One alternative is to 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 favored approach: using an open source serializer library such as [JMS Serializer](http://jmsyst.com/libs/serializer)¹⁶. Let's see an example of applying it for serializing a `Money` object:

¹⁶<http://jmsyst.com/libs/serializer>

```
1 $myMoney = new Money(  
2     999,  
3     new Currency('USD')  
4 );  
5  
6 $serializer = JMS\Serializer\SerializerBuilder::create()->build();  
7 $jsonData = $serializer->serialize($myMoney, 'json');
```

In order to unserialize the object, the process is straightforward:

```
1 $serializer = JMS\Serializer\SerializerBuilder::create()->build();  
2 // ...  
3 $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](#)¹⁷ and the [Cookbook](#)¹⁸ for more information. JMS Serializer is a must in any Domain-Driven Design project.

6.7.1.5 Serialized LOB with Doctrine

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

6.7.1.5.1 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, which maps an SQL CLOB to a PHP object using `serialize()` and `unserialize()`.

According to the [Doctrine DBAL 2 documentation](#)¹⁹:

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.

¹⁷http://jmsyst.com/libs/serializer/master/reference/xml_reference

¹⁸<http://jmsyst.com/libs/serializer/master/cookbook>

¹⁹<http://doctrine-orm.readthedocs.io/projects/doctrine-dbal/en/latest/reference/types.html#object>

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 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 look at a possible XML mapping for the Product Entity by using the object type:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <doctrine-mapping
3      xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6      https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8      <entity
9          name="Product"
10         table="products">
11
12         <id
13             name="id"
14             column="id"
15             type="string"
16             length="255">
17             <generator
18                 strategy="NONE">
19             </generator>
20         </id>
21
22         <field
23             name="name"
24             type="string"
25             length="255"
26         />
27         // start-of-new-code-to-take-a-look
28         <field
29             name="price"
```

```
30         type="object"
31     />
32     // end-of-new-code-to-take-a-look
33 </entity>
34 </doctrine-mapping>
```

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

```
1 // ...
2 $em->persist($product);
3 $em->flush($product);
```

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

```
1 // ...
2 $repository = $em->getRepository('Ddd\Domain\Model\Product');
3 $item = $repository->find(1);
4 var_dump($item);
5
6 /*
7 class Ddd\Domain\Model\Product#177 (3) {
8     private $productId =>
9         int(1)
10    private $name =>
11        string(41) "Domain-Driven Design in PHP"
12    private $money =>
13        class Ddd\Domain\Model\Money#174 (2) {
14            private $amount =>
15                string(3) "100"
16            private $currency =>
17                class Ddd\Domain\Model\Currency#175 (1) {
18                    private $isoCode =>
19                        string(3) "USD"
20                }
21            }
22    }
23 */
```

Last but not least, the Doctrine documentation states that:

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](#)²⁰ for more information.

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

6.7.1.5.2 Doctrine Custom Types

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

As the [Doctrine DBAL 2 documentation](#)²¹ 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:

```

1 use Ddd\Domain\Model\Currency;
2 use Ddd\Domain\Model\Money;
3 use Doctrine\DBAL\Types\TextType;
4 use Doctrine\DBAL\Platforms\AbstractPlatform;
5
6 class MoneyType extends TextType
7 {
8     const MONEY = 'money';
9
10    public function convertToPHPValue($value, AbstractPlatform $platform)
11    {
12        $value = parent::convertToPHPValue($value, $platform);
13
14        $value = explode('|', $value);
15        return new Money(

```

²⁰<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#doctrine-mapping-types>

²¹<http://doctrine-orm.readthedocs.io/projects/doctrine-dbal/en/latest/reference/types.html#custom-mapping-types>

```
16         $value[0],
17         new Currency($value[1])
18     );
19 }
20
21 public function convertToDatabaseValue($value, AbstractPlatform $platform)
22 {
23     return implode(
24         '|',
25         [
26             $value->amount(),
27             $value->currency()->isoCode()
28         ]
29     );
30 }
31
32 public function getName()
33 {
34     return self::MONEY;
35 }
36 }
```

Using Doctrine, you're required to register all Custom Types. It's common to use an EntityManagerFactory that centralizes this EntityManager creation. You could alternatively perform this step by bootstrapping your application:

```
1 use Doctrine\DBAL\Types\Type;
2 use Doctrine\ORM\EntityManager;
3 use Doctrine\ORM\Tools\Setup;
4
5 class EntityManagerFactory
6 {
7     public function build()
8     {
9         Type::addType(
10             'money',
11             'Ddd\Infrastructure\Persistence\Doctrine\Type\MoneyType'
12         );
13
14         return EntityManager::create(
15             [
16                 'driver' => 'pdo_mysql',
```



```
17         'user'      => 'root',
18         'password' => '',
19         'dbname'   => 'ddd',
20     ],
21     Setup::createXMLMetadataConfiguration(
22         [__DIR__.' /config'],
23         true
24     )
25 );
26 }
27 }
```

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

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping>
3
4     <entity
5         name="Product"
6         table="product">
7
8         <!-- ... -->
9         // start-of-new-code-to-take-a-look
10        <field
11            name="price"
12            type="money"
13        />
14        // end-of-new-code-to-take-a-look
15    </entity>
16 </doctrine-mapping>
```

Why Use XML mapping?

Thanks to the XSD schema validation in the headers of the XML mapping file, most integrated development environments (IDEs) provide auto-complete functionality for all the elements and attributes present in the mapping definition. However, in other parts of the book, we use YAML to show a different syntax.

Let's check the database to see how the price was persisted using this approach:

```

1 mysql> select * from products \G
2 ***** 1. row *****
3     id: 1
4     name: Domain-Driven Design in PHP
5     price: 999|USD
6 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 it's still not the best option for building your DQL queries. Check the [Doctrine Custom Mapping Types reference](#)²² for more information.



Time to Discuss

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

6.7.2 Persisting a Collection of Value Objects

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

```

1 class HistoricalProduct extends Product
2 {
3     /**
4      * @var Money[]
5      */
6     protected $prices;
7
8     public function __construct($aProductId, $aName, Money $aPrice, array $someP\
9 rices)
10    {
11        parent::__construct($aProductId, $aName, $aPrice);
12        $this->setPrices($somePrices);
13    }
14
15    private function setPrices(array $somePrices)
16    {

```

²²<http://doctrine-orm.readthedocs.org/en/latest/cookbook/custom-mapping-types.html>

```
17     $this->prices = $somePrices;
18 }
19
20 public function prices()
21 {
22     return $this->prices;
23 }
24 }
```

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

As in the previous sections, serialization is a plausible approach if you don't 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 an undetermined collection of historical prices?

6.7.2.1 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, but if you wish to persist a large collection, be sure to consider the maximum column length and the maximum 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.

6.7.2.2 Collection Backed by a Join Table

In case you want to persist and query an Entity by its related Value Objects, 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 set them up with a “one-to-many”/”one-to-one” relation with the owner, a real Entity. To summarize, 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:

```

1 CREATE TABLE `historical_products` (
2   `id` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
3   `name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
4   `price_amount` int(11) NOT NULL,
5   `price_currency` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
6   PRIMARY KEY (`id`)
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

```

The `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:

```

1 CREATE TABLE `prices` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `amount` int(11) NOT NULL,
4   `currency` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
5   PRIMARY KEY (`id`)
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

```

Finally, a table that relates products and prices is needed:

```

1 CREATE TABLE `products_prices` (
2   `product_id` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
3   `price_id` int(11) NOT NULL,
4   PRIMARY KEY (`product_id`, `price_id`),
5   UNIQUE KEY `UNI_Q_62F8E673D614C7E7` (`price_id`),
6   KEY `IDX_62F8E6734584665A` (`product_id`),
7   CONSTRAINT `FK_62F8E6734584665A` FOREIGN KEY (`product_id`) REFERENCES `histor\
8 ical_products` (`id`),
9   CONSTRAINT `FK_62F8E673D614C7E7` FOREIGN KEY (`price_id`) REFERENCES `prices` \
10  (`id`)
11 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

```

6.7.2.2.1 Collection Backed by a Join Table with Doctrine

Doctrine requires that all database Entities 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 alterations 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 using the first option. Let's review the changes required to implement it in the Money Value Object:

```
1  class Money
2  {
3      private $amount;
4      private $currency;
5
6      // start-of-new-code-to-take-a-look
7      private $surrogateId;
8      private $surrogateCurrencyIsoCode;
9      // end-of-new-code-to-take-a-look
10
11     public function __construct($amount, Currency $currency)
12     {
13         $this->setAmount($amount);
14         $this->setCurrency($currency);
15     }
16
17     private function setAmount($amount)
18     {
19         $this->amount = $amount;
20     }
21
22     private function setCurrency(Currency $currency)
23     {
24         $this->currency = $currency;
25         // start-of-new-code-to-take-a-look
26         $this->surrogateCurrencyIsoCode = $currency->isoCode();
27         // end-of-new-code-to-take-a-look
28     }
29
30     public function currency()
31     {
32         // start-of-new-code-to-take-a-look
33         if (null === $this->currency) {
34             $this->currency = new Currency($this->surrogateCurrencyIsoCode);
```

```
35     }
36     // end-of-new-code-to-take-a-look
37
38     return $this->currency;
39 }
40
41 public function amount()
42 {
43     return $this->amount;
44 }
45
46 public function equals(Money $aMoney)
47 {
48     return
49         $this->amount() === $aMoney->amount()
50         && $this->currency()->equals($this->currency());
51 }
52 }
```

As seen above, two new attributes have been added. The first one, `surrogateId`, is not used by our Domain, but it's 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's really easy to map our Value Object with Doctrine.

The Money mapping is quite straightforward:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping
3     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6     https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8     <entity
9         name="Ddd\Domain\Model\Money"
10        table="prices">
11
12        <id
13            name="surrogateId"
14            type="integer"
15            column="id">
16            <generator
17                strategy="AUTO">
```

```
18         </generator>
19     </id>
20
21     <field
22         name="amount"
23         type="integer"
24         column="amount"
25     />
26     <field
27         name="surrogateCurrencyIsoCode"
28         type="string"
29         column="currency"
30     />
31 </entity>
32 </doctrine-mapping>
```

Using Doctrine, the HistoricalProduct Entity would have following mapping:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping
3     xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
6     https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
7
8     <entity
9         name="Ddd\Domain\Model\HistoricalProduct"
10        table="historical_products"
11        repository-class="Ddd\Infrastructure\Domain\Model\DoctrineHistoricalProd\
12    uctRepository">
13
14        <many-to-many
15            field="prices"
16            target-entity="Ddd\Domain\Model\Money">
17
18            <cascade>
19                <cascade-all/>
20            </cascade>
21
22            <join-table
23                name="products_prices">
24
```

```
25         <join-columns>
26             <join-column
27                 name="product_id"
28                 referenced-column-name="id"
29             />
30         </join-columns>
31
32         <inverse-join-columns>
33             <join-column
34                 name="price_id"
35                 referenced-column-name="id"
36                 unique="true"
37             />
38         </inverse-join-columns>
39     </join-table>
40 </many-to-many>
41 </entity>
42 </doctrine-mapping>
```

6.7.2.2.2 Collection Backed by a Join Table with an Ad Hoc ORM

It's 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 the removal of Value Objects is handled, in order to not leave orphan Money Value Objects.



Exercise

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

6.7.2.3 Collection Backed by a Database Entity

“Database Entity” is the same strategy as “Join Table,” with the addition of the Value Object that 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 overly 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 is used.

6.7.3 NoSQL

What about NoSQL mechanisms such as Redis, MongoDB, or CouchDB? Unfortunately you don't 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'll be required to again rebuild the Value Object during Redis retrieval.

6.7.3.1 PostgreSQL JSONB and MySQL JSON Type

If our database engine would allow us to not only use the Serialized LOB strategy but also search based on its value, we would have the best of both approaches. Well, good news: now you *can* do this. As of PostgreSQL version 9.4, support for **JSONB**²³ has been added. Value Objects can be persisted as JSON serializations and subsequently queried within this JSON serialization.

MySQL has done the same. As of MySQL 5.7.8, MySQL supports a native JSON data type that enables efficient access to data in JSON (JavaScript Object Notation) documents. The JSON data type provides these advantages over storing JSON-format strings in a string column:

- Automatic validation of JSON documents stored in JSON columns. Invalid documents produce an error.
- Optimized storage format. JSON documents stored in JSON columns are converted to an internal format that permits **quick read access to document elements**. When the server later must read a JSON value stored in this binary format, the value need not be parsed from a text representation. The binary format is structured to enable the server to **look up subobjects or nested values directly** by key or array index without reading all values before or after them in the document.

If Relational Databases add support for document and nested document searches with high performance and with all the benefits of an ACID (Atomicity, Consistency, Isolation, Durability) philosophy, it could save a lot of complexity in many projects.

6.8 Security

Another interesting detail of modeling your Domain concepts using Value Objects is regarding its security benefits. Consider an application within the context of selling flight tickets. If you deal with International Air Transport Association airport codes, also known as **IATA codes**²⁴, you can decide to use a string or model the 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

²³<http://www.postgresql.org/docs/9.4/static/functions-json.html>

²⁴https://en.wikipedia.org/wiki/International_Air_Transport_Association_airport_code

chance you forget anywhere important? On the other hand, think about trying to instantiate new `IATA("BCN"; DROP TABLE users; --")`. If you centralize the [guards](#)²⁵ in the constructor and then pass an IATA Value Object into your model, avoiding SQL Injections or similar attacks gets easier.

If you want to know more about the security side of Domain-Driven Design, you can follow [Dan Bergh Johnsson](#)²⁶ or read his [blog](#)²⁷.

6.9 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 Domain-Driven Design application, using an ORM is a must. However, in order to persist Value Objects using Doctrine, the preferred way is using embeddables. In case you are stuck in version 2.4, 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).

²⁵[https://en.wikipedia.org/wiki/Guard_\(computer_science\)](https://en.wikipedia.org/wiki/Guard_(computer_science))

²⁶<https://twitter.com/danbjson>

²⁷<http://dearjunior.blogspot.com.es/search/label/domain%20driven%20security>

7. Bibliography

- Brandolini, Alberto. 2016 *Introducing EventStorming*¹. Leanpub.
- Vernon, Vaughn. 2016 *Domain-Driven Design Distilled*². Addison-Wesley Professional.
- Newman, Sam. 2015. *Building Microservices*³. O'Reilly Media.
- Evans, Eric. 2014. *Domain-Driven Design Reference*⁴. Dog Ear Publishing.
- Vernon, Vaughn. 2013. *Implementing Domain-Driven Design*⁵. Addison-Wesley Professional.
- Hohpe, Gregor and Bobby Woolf. 2012. *Enterprise Integration Patterns*⁶. Addison-Wesley Professional.
- Fowler, Martin and Pramod J. Sadalage. 2012. *NoSQL Distilled*⁷. Addison-Wesley Professional.
- C. Martin, Robert. 2008. *Clean Code*⁸. Prentice Hall.
- Meszaros, Gerard. 2007. *xUnit Test Patterns: Refactoring Test Code*⁹. Addison-Wesley Professional.
- Nilson, Jimmy. 2006. *Applying Domain-Driven Design*¹⁰. Addison-Wesley Professional.
- Evans, Eric. 2003. *Domain-Driven Design*¹¹. Addison-Wesley Professional.
- Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*¹². Addison-Wesley Professional.
- C. Martin, Robert. 2002. *Agile Software Development*¹³. Pearson.
- Beck, Kent. 2002. *Test Driven Development by Example*¹⁴. Addison-Wesley Professional.

¹https://leanpub.com/introducing_eventstorming

²<http://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420>

³<http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358>

⁴<http://www.amazon.com/Domain-Driven-Design-Reference-Definitions-Summaries/dp/1457501198>

⁵<http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8>

⁶<http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E>

⁷<http://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620>

⁸<http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>

⁹<https://www.amazon.com/xUnit-Test-Patterns-Refactoring-Code/dp/0131495054>

¹⁰<http://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202>

¹¹<http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215>

¹²<http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin-ebook/dp/B000OZ0NAI>

¹³<http://www.amazon.com/Software-Development-Principles-Patterns-Practices/dp/0135974445>

¹⁴<http://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530>

8. Appendix A: Hexagonal Architecture with PHP

Article published in the php|architect magazine. June 2014. Carlos Buenosvinos (@buenosvinos).

8.1 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 is 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?

8.2 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 is 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.

```
1 class IdeaController extends Zend_Controller_Action
2 {
3     public function rateAction()
4     {
5         // Getting parameters from the request
6         $ideaId = $this->request->getParam('id');
7         $rating = $this->request->getParam('rating');
8
9         // Building database connection
10        $db = new Zend_Db_Adapter_Pdo_Mysql([
11            'host'      => 'localhost',
12            'username' => 'idy',
13            'password' => '',
14            'dbname'   => 'idy'
15        ]);
16
17        // Finding the idea in the database
18        $sql = 'SELECT * FROM ideas WHERE idea_id = ?';
19        $row = $db->fetchRow($sql, $ideaId);
20        if (!$row) {
21            throw new Exception('Idea does not exist');
22        }
23
24        // Building the idea from the database
25        $idea = new Idea();
26        $idea->setId($row['id']);
27        $idea->setTitle($row['title']);
28        $idea->setDescription($row['description']);
29        $idea->setRating($row['rating']);
30        $idea->setVotes($row['votes']);
31        $idea->setAuthor($row['email']);
32
33        // Add user rating
34        $idea->addRating($rating);
35
36        // Update the idea and save it to the database
```

```
37     $data = [  
38         'votes' => $idea->getVotes(),  
39         'rating' => $idea->getRating()  
40     ];  
41     $where['idea_id = ?'] = $ideaId;  
42     $db->update('ideas', $data, $where);  
43  
44     // Redirect to view idea page  
45     $this->redirect('/idea/'.$ideaId);  
46 }  
47 }
```

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 is 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.

8.3 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 is 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.

```
1 class IdeaController extends Zend_Controller_Action
2 {
3     public function rateAction()
4     {
5         $ideaId = $this->request->getParam('id');
6         $rating = $this->request->getParam('rating');
7
8         $ideaRepository = new IdeaRepository();
9         $idea = $ideaRepository->find($ideaId);
10        if (!$idea) {
11            throw new Exception('Idea does not exist');
12        }
13
14        $idea->addRating($rating);
15        $ideaRepository->update($idea);
16
17        $this->redirect('/idea/' . $ideaId);
18    }
19 }
20
21 class IdeaRepository
22 {
23     private $client;
24
25     public function __construct()
26     {
27         $this->client = new Zend_Db_Adapter_Pdo_Mysql([
28             'host'      => 'localhost',
29             'username' => 'idy',
30             'password' => '',
31             'dbname'   => 'idy'
32         ]);
33     }
34
35     public function find($id)
36     {
```

```
37     $sql = 'SELECT * FROM ideas WHERE idea_id = ?';
38     $row = $this->client->fetchRow($sql, $id);
39     if (!$row) {
40         return null;
41     }
42
43     $idea = new Idea();
44     $idea->setId($row['id']);
45     $idea->setTitle($row['title']);
46     $idea->setDescription($row['description']);
47     $idea->setRating($row['rating']);
48     $idea->setVotes($row['votes']);
49     $idea->setAuthor($row['email']);
50
51     return $idea;
52 }
53
54 public function update(Idea $idea)
55 {
56     $data = [
57         'title' => $idea->getTitle(),
58         'description' => $idea->getDescription(),
59         'rating' => $idea->getRating(),
60         'votes' => $idea->getVotes(),
61         'email' => $idea->getAuthor(),
62     ];
63
64     $where = ['idea_id = ?' => $idea->getId()];
65     $this->client->update('ideas', $data, $where);
66 }
67 }
```

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 is 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.

8.4 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 is 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.

```
1 class IdeaController extends Zend_Controller_Action
2 {
3     public function rateAction()
4     {
5         $ideaId = $this->request->getParam('id');
6         $rating = $this->request->getParam('rating');
7
8         $ideaRepository = new MySQLIdeaRepository();
9         $idea = $ideaRepository->find($ideaId);
10        if (!$idea) {
11            throw new Exception('Idea does not exist');
12        }
13
14        $idea->addRating($rating);
15        $ideaRepository->update($idea);
16
17        $this->redirect('/idea/' . $ideaId);
18    }
19 }
20
21 interface IdeaRepository
```

```
22 {
23     /**
24      * @param int $id
25      * @return null|Idea
26      */
27     public function find($id);
28
29     /**
30      * @param Idea $idea
31      */
32     public function update(Idea $idea);
33 }
34
35 class MySQLIdeaRepository implements IdeaRepository
36 {
37     // ...
38 }
```

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.

8.5 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.

```
1 class IdeaController extends Zend_Controller_Action
2 {
3     public function rateAction()
4     {
5         $ideaId = $this->request->getParam('id');
6         $rating = $this->request->getParam('rating');
7
8         $ideaRepository = new RedisIdeaRepository();
9         $idea = $ideaRepository->find($ideaId);
10        if (!$idea) {
11            throw new Exception('Idea does not exist');
```

```
12     }
13
14     $idea->addRating($rating);
15     $ideaRepository->update($idea);
16
17     $this->redirect('/idea/'. $ideaId);
18 }
19 }
20
21 interface IdeaRepository
22 {
23     // ...
24 }
25
26 class RedisIdeaRepository implements IdeaRepository
27 {
28     private $client;
29
30     public function __construct()
31     {
32         $this->client = new \Predis\Client();
33     }
34
35     public function find($id)
36     {
37         $idea = $this->client->get($this->getKey($id));
38         if (!$idea) {
39             return null;
40         }
41
42         return unserialize($idea);
43     }
44
45     public function update(Idea $idea)
46     {
47         $this->client->set(
48             $this->getKey($idea->getId()),
49             serialize($idea)
50         );
51     }
52
53     private function getKey($id)
```

```
54     {
55         return 'idea:' . $id;
56     }
57 }
```

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.

8.6 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.

```
1  class IdeaController extends Zend_Controller_Action
2  {
3      public function rateAction()
4      {
5          $ideaId = $this->request->getParam('id');
6          $rating = $this->request->getParam('rating');
7
8          $ideaRepository = new RedisIdeaRepository();
9          $useCase = new RateIdeaUseCase($ideaRepository);
10         $response = $useCase->execute($ideaId, $rating);
11
12         $this->redirect('/idea/' . $ideaId);
13     }
14 }
15
16 interface IdeaRepository
17 {
18     // ...
```

```
19 }
20
21 class RateIdeaUseCase
22 {
23     private $ideaRepository;
24
25     public function __construct(IdeaRepository $ideaRepository)
26     {
27         $this->ideaRepository = $ideaRepository;
28     }
29
30     public function execute($ideaId, $rating)
31     {
32         try {
33             $idea = $this->ideaRepository->find($ideaId);
34         } catch(Exception $e) {
35             throw new RepositoryNotAvailableException();
36         }
37
38         if (!$idea) {
39             throw new IdeaDoesNotExistException();
40         }
41
42         try {
43             $idea->addRating($rating);
44             $this->ideaRepository->update($idea);
45         } catch(Exception $e) {
46             throw new RepositoryNotAvailableException();
47         }
48
49         return $idea;
50     }
51 }
```

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 does not 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 is 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.

```
1 class IdeaController extends Zend_Controller_Action
2 {
3     public function rateAction()
4     {
5         $ideaId = $this->request->getParam('id');
6         $rating = $this->request->getParam('rating');
7
8         $ideaRepository = new RedisIdeaRepository();
9         $useCase = new RateIdeaUseCase($ideaRepository);
10        $response = $useCase->execute(
11            new RateIdeaRequest($ideaId, $rating)
12        );
13
14        $this->redirect('/idea/' . $response->idea->getId());
15    }
16 }
17
18 class RateIdeaRequest
19 {
20     public $ideaId;
21     public $rating;
22
23     public function __construct($ideaId, $rating)
24     {
25         $this->ideaId = $ideaId;
26         $this->rating = $rating;
27     }
28 }
29
30 class RateIdeaResponse
31 {
32     public $idea;
33
34     public function __construct(Idea $idea)
```

```
35     {
36         $this->idea = $idea;
37     }
38 }
39
40 class RateIdeaUseCase
41 {
42     // ...
43
44     public function execute($request)
45     {
46         $ideaId = $request->ideaId;
47         $rating = $request->rating;
48
49         // ...
50
51         return new RateIdeaResponse($idea);
52     }
53 }
```

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 is easier to change from one framework to other, or execute our UseCase from another *delivery mechanism*. Let's see this point.

8.7 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.

```
1 require_once __DIR__ . '/../vendor/autoload.php';
2
3 $app = new \Silex\Application();
4
5 // ... more routes
6
7 $app->get(
8     '/api/rate/idea/{ideaId}/rating/{rating}',
9     function ($ideaId, $rating) use ($app) {
10         $ideaRepository = new RedisIdeaRepository();
11         $useCase = new RateIdeaUseCase($ideaRepository);
12         $response = $useCase->execute(
13             new RateIdeaRequest($ideaId, $rating)
14         );
15
16         return $app->json($response->idea);
17     }
18 );
19
20 $app->run();
```

Is there anything familiar to you? Can you identify some code that you have seen before? I'll give you a clue.

```
1 $ideaRepository = new RedisIdeaRepository();
2 $useCase = new RateIdeaUseCase($ideaRepository);
3 $response = $useCase->execute(
4     new RateIdeaRequest($ideaId, $rating)
5 );
```

“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.

8.8 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?

```
1 namespace Idy\Console\Command;
2
3 use Symfony\Component\Console\Command\Command;
4 use Symfony\Component\Console\Input\InputArgument;
5 use Symfony\Component\Console\Input\InputInterface;
6 use Symfony\Component\Console\Output\OutputInterface;
7
8 class VoteIdeaCommand extends Command
9 {
10     protected function configure()
11     {
12         $this
13             ->setName('idea:rate')
14             ->setDescription('Rate an idea')
15             ->addArgument('id', InputArgument::REQUIRED)
16             ->addArgument('rating', InputArgument::REQUIRED)
17     };
18 }
19
20 protected function execute(
21     InputInterface $input,
22     OutputInterface $output
23 ) {
24     $ideaId = $input->getArgument('id');
25     $rating = $input->getArgument('rating');
26
27     $ideaRepository = new RedisIdeaRepository();
28     $useCase = new RateIdeaUseCase($ideaRepository);
29     $response = $useCase->execute(
30         new RateIdeaRequest($ideaId, $rating)
31     );
32
33     $output->writeln('Done!');
34 }
35 }
```

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.

8.9 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.

```
1 class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
2 {
3     /**
4      * @test
5      */
6     public function whenRepositoryNotAvailableAnExceptionShouldBeThrown()
7     {
8         $this->setExpectedException('NotAvailableRepositoryException');
9         $ideaRepository = new NotAvailableRepository();
10        $useCase = new RateIdeaUseCase($ideaRepository);
11        $useCase->execute(
12            new RateIdeaRequest(1, 5)
13        );
14    }
15 }
16
17 class NotAvailableRepository implements IdeaRepository
18 {
19     public function find($id)
20     {
21         throw new NotAvailableException();
22     }
23
24     public function update(Idea $idea)
25     {
26         throw new NotAvailableException();
27     }
28 }
```

```
27     }
28 }
```

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.

```
1  class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
2  {
3      // ...
4
5      /**
6       * @test
7       */
8      public function whenIdeaDoesNotExistAnExceptionShouldBeThrown()
9      {
10         $this->setExpectedException('IdeaDoesNotExistException');
11         $ideaRepository = new EmptyIdeaRepository();
12         $useCase = new RateIdeaUseCase($ideaRepository);
13         $useCase->execute(
14             new RateIdeaRequest(1, 5)
15         );
16     }
17 }
18
19 class EmptyIdeaRepository implements IdeaRepository
20 {
21     public function find($id)
22     {
23         return null;
24     }
25
26     public function update(Idea $idea)
27     {
28
29     }
30 }
```

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.

```
1 class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
2 {
3     // ...
4
5     /**
6      * @test
7      */
8     public function whenUpdatingInReadOnlyAnIdeaAnExceptionShouldBeThrown()
9     {
10         $this->setExpectedException('NotAvailableRepositoryException');
11         $ideaRepository = new WriteNotAvailableRepository();
12         $useCase = new RateIdeaUseCase($ideaRepository);
13         $response = $useCase->execute(
14             new RateIdeaRequest(1, 5)
15         );
16     }
17 }
18
19 class WriteNotAvailableRepository implements IdeaRepository
20 {
21     public function find($id)
22     {
23         $idea = new Idea();
24         $idea->setId(1);
25         $idea->setTitle('Subscribe to php[architect]');
26         $idea->setDescription('Just buy it!');
27         $idea->setRating(5);
28         $idea->setVotes(10);
29         $idea->setAuthor('hi@carlos.io');
30
31         return $idea;
32     }
33
34     public function update(Idea $idea)
35     {
36         throw new NotAvailableException();
37     }
38 }
```

38 }

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.

```
1 class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
2 {
3     // ...
4
5     /**
6      * @test
7      */
8     public function whenRatingAnIdeaNewRatingShouldBeAddedAndIdeaUpdated()
9     {
10         $ideaRepository = new OneIdeaRepository();
11         $useCase = new RateIdeaUseCase($ideaRepository);
12         $response = $useCase->execute(
13             new RateIdeaRequest(1, 5)
14         );
15
16         $this->assertSame(5, $response->idea->getRating());
17         $this->assertTrue($ideaRepository->updateCalled);
18     }
19 }
20
21 class OneIdeaRepository implements IdeaRepository
22 {
23     public $updateCalled = false;
24
25     public function find($id)
26     {
27         $idea = new Idea();
28         $idea->setId(1);
29         $idea->setTitle('Subscribe to php[architect]');
30         $idea->setDescription('Just buy it!');
31         $idea->setRating(5);
32         $idea->setVotes(10);
33         $idea->setAuthor('hi@carlos.io');
34
35         return $idea;
```

```
36     }
37
38     public function update(Idea $idea)
39     {
40         $this->updateCalled = true;
41     }
42 }
```

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:

```
1 $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?

8.10 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 is 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.

```
1 class IdeaController extends Zend_Controller_Action
2 {
3     public function rateAction()
4     {
5         $ideaId = $this->request->getParam('id');
6         $rating = $this->request->getParam('rating');
7
8         $useCase = new RateIdeaUseCase(
9             new RedisIdeaRepository(
```

```
10         new \Predis\Client()
11     )
12 );
13
14 $response = $useCase->execute(
15     new RateIdeaRequest($ideaId, $rating)
16 );
17
18 $this->redirect('/idea/'.$response->idea->getId());
19 }
20 }
21
22 class RedisIdeaRepository implements IdeaRepository
23 {
24     private $client;
25
26     public function __construct($client)
27     {
28         $this->client = $client;
29     }
30
31     // ...
32
33     public function find($id)
34     {
35         $idea = $this->client->get($this->getKey($id));
36         if (!$idea) {
37             return null;
38         }
39
40         return $idea;
41     }
42 }
```

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 is all about creating mocks, test cases and having fun doing asserts.

8.11 Arggg, So Many Dependencies!

Is it normal that I have to create so many dependencies by hand? No. It is 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.

```
1 class IdeaController extends ContainerAwareController
2 {
3     public function rateAction()
4     {
5         $ideaId = $this->request->getParam('id');
6         $rating = $this->request->getParam('rating');
7
8         $useCase = $this->get('rate_idea_use_case');
9         $response = $useCase->execute(
10             new RateIdeaRequest($ideaId, $rating)
11         );
12
13         $this->redirect('/idea/' . $response->idea->getId());
14     }
15 }
```

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.

```
1 <?xml version="1.0" ?>
2 <container xmlns="http://symfony.com/schema/dic/services"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://symfony.com/schema/dic/services
5     http://symfony.com/schema/dic/services/services-1.0.xsd">
6     <services>
7         <service
8             id="rate_idea_use_case"
9             class="RateIdeaUseCase">
10             <argument type="service" id="idea_repository" />
11         </service>
12
13         <service
```



```
14         id="idea_repository"
15         class="RedisIdeaRepository">
16         <argument type="service">
17             <service class="Predis\Client" />
18         </argument>
19     </service>
20 </services>
21 </container>
```

In Listing 15, you can also find the XML file used to configure the Service Container. It is 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

8.12 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.

```
1  class RateIdeaUseCase
2  {
3      private $ideaRepository;
4      private $authorNotifier;
5
6      public function __construct(
7          IdeaRepository $ideaRepository,
8          AuthorNotifier $authorNotifier
9      )
10     {
11         $this->ideaRepository = $ideaRepository;
12         $this->authorNotifier = $authorNotifier;
13     }
14
15     public function execute(RateIdeaRequest $request)
16     {
17         $ideaId = $request->ideaId;
18         $rating = $request->rating;
19
20         try {
21             $idea = $this->ideaRepository->find($ideaId);
22         } catch(Exception $e) {
23             throw new RepositoryNotAvailableException();
```

```
24     }
25
26     if (!$idea) {
27         throw new IdeaDoesNotExistException();
28     }
29
30     try {
31         $idea->addRating($rating);
32         $this->ideaRepository->update($idea);
33     } catch(Exception $e) {
34         throw new RepositoryNotAvailableException();
35     }
36
37     try {
38         $this->authorNotifier->notify(
39             $idea->getAuthor()
40         );
41     } catch(Exception $e) {
42         throw new NotificationNotSentException();
43     }
44
45     return $idea;
46 }
47 }
```

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 is up to you.

8.13 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.

8.14 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.

8.15 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.

8.16 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.