DOMAIN-DRIVEN DESIGN

IN PHP

Carlos Buenosvinos
Christian Soronellas
Keyvan Akbary

Domain-Driven Design in PHP - 2n Edition

Discover DDD, Architectural Styles, Tactical Design Implementations, and Bounded Context Integration with PHP 8.5 examples

Carlos Buenosvinos, Christian Soronellas, and Keyvan Akbary

This book is available at https://leanpub.com/ddd-in-php

This version was published on 2026-04-06

ISBN 978-0-9946084-1-3

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” by @buenosvinos, @theUniC and @keyvanakbary https://leanpub.com/ddd-in-php?utm_source=social+utm_medium=twitter+utm_campaign=book_buy @dddbook #DDDesign

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:

#DDDinPHP

Also By These Authors

Books by Carlos Buenosvinos

CQRS by Example

Books by Christian Soronellas

CQRS by Example

Books by Keyvan Akbary

El Manual del Manager

CQRS by Example

Contents

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

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

To my parents, John and Mercedes, who raised me free of constraints. This will be the first book of many. To my love, Clara, for your unconditional support and infinite patience. – **Keyvan**

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 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 didn't 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 one gives you what you need to start applying some Domain-Driven Design 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 the authors 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 seen in the PHP community: the need for concrete examples. It's not always easy for authors to come up with proper illustrations of how to apply certain ideas that have a low risk of being misinterpreted or abused in real-world projects. And in Domain-Driven Design, which is philosophical by nature, this is even more challenging.

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 solutions either; they take the time to provide detailed explanations

– such as 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 of 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*

Preface

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

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, which you'll find in the appendix, was the origin of this book.

In late 2014, Carlos and Christian talked about extending the article and sharing all their knowledge of and experience in 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 framework-agnostic applications weren't so common in the PHP community. So in December 2014, 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 handle the complexity of the company, with its constantly changing environment and requirements. And after connecting with Carlos and Christian and discussing the book, 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 implement them. 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* by Vaughn Vernon (aka *the Red Book*), and *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (aka *the Blue Book*). You should buy both books. You should read them carefully. You should love them.

Who Should Read This Book

If you're a PHP Developer, Architect, or Tech Lead, we highly recommend this book. 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 important in order to model any Domain you'll 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 delving deeper into Aggregate design.

Although Domain-Driven Design is not about technology, you still need it to make HTTP requests to access your Domain. Throughout the book, we recommend using specific PHP frameworks and libraries, such as Symfony and Doctrine. Some examples reference Silex, a micro-framework that has since been discontinued — the same patterns apply to any modern Symfony application. For some examples, we also use specific technologies, such as MySQL, RabbitMQ, Redis, and Elasticsearch. However, most important are the behind-the-scenes concepts — concepts that are applicable regardless of the technology used to implement them.

Additionally, the book is 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. It explains what the role of the main PHP libraries and frameworks used in Domain-Driven Design are. The book also teaches how to apply Hexagonal Architecture within your application, regardless of whether you use an open source framework or your own one. Finally, it shows 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.

DDD and PHP Community

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

Domain-Driven Design arrived in the PHP community two years prior to that conference. However, there's 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 being coupled to specific frameworks or technologies.

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 an appendix.

Chapter 1: Getting Started with Domain-Driven Design

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

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.

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 figure out 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.

Chapter 4: Entities

Entities are 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 assess whether or not annotations are the best mapping approach for Entities and look at the different strategies for generating an Identity.

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.

Chapter 6: Domain Events

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

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're dealing with a framework like Symfony. We'll review how PHP namespaces can be used for implementing Modules. We'll also discover different hierarchies of folders for organizing Domain Model code, Application Code, and Infrastructure Code.

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 we'll demonstrate how the rest of the objects must be refactored.

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 explore the relationship between Factories and Aggregates.

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.

Chapter 11: Application

An Application is the thin layer that connects outside clients 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.

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.

Appendix: Hexagonal Architecture with PHP

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

Code and Examples

The authors have created an organization at GitHub called *Domain-Driven Design in PHP*, 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*, a simple Domain-Driven Design-style application showing different examples explained in this book. Additionally, you'll find our *CQRS Blog*, along with *Gamify*, 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* repository. We fix them as they come in. If you're interested, we also urge you to watch our projects and provide feedback.

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're 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 supported 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. The final review was done by the professional copy editor Natalye Childress. She has done a great work rewriting our words to make them understandable. Thank you so much. Our book is easier and more enjoyable to read.

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've been struggling when learning and applying these concepts. Some of you weren't 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 sale from the moment we put the first chapters on Leanpub. Early adopters who bought the book in its beginning stages gave us the much needed love and support to get this done. Thanks 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 — not just because his work was an incredible source of information and inspiration for us, but also because he helped us find a good publisher, gave us valuable advice, and shared ideas with us. Thanks so much for your help.

Last but not least, we'd like to express our gratitude to all the people who have reported issues, made suggestions, and otherwise contributed to our GitHub repository. To all of you, thank you. You've helped us make this book better. More importantly, you've helped the community grow and helped other developers be better developers. As Robert C. Martin wrote in his book, *Clean Code: A Handbook of Agile Software Craftsmanship*, "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, Victor Guardiola, Oriol González, Henry Snoek, Tom Jowitt, Sascha Schimke, Sven Herrmann, Daniel Abad, Luis Rovirosa, Luis Cordova, Raúl Ramos, Juan Maturana, Nil Portugués, Nikolai Zujev, Fernando Pradas, Raúl Araya, Neal Brooks, Hubert Béague, Aleksander Rekść, Sebastian Machuca, Nicolas Oelgart, Sebastiaan Stok, Vladimir Hraban, Vladas Dirzys, Max Gulturyan, Ivan Đurđevac, Marc Verney, Matthias Gutjahr, Dennis König, Jordi Puig, David Fernández, Vladimir Shadyan and Marc Aube.

About the Authors

Carlos Buenosvinos

Carlos is an Extreme Programmer (XP) and DevOps with more than 20 years of experience in developing Web and Mobile Applications. For the last ten years, he has played various leading roles such as Tech Lead, VP of Engineering, and CTO. He has mentored engineering and product teams of up to 150 members in multiple different markets such as E-commerce, E-Learning, Payment Processing, Classifieds, and Recruiting Market.

As an employee and consultant, he has contributed to the success of start-ups and well-established brands. Some examples are SEAT, NewWork/XING, Atr□□palo, GMV, PCComponentes, Cash Converters, Emagister, O2O, Opositatest, Techpump, Packlink, eBay, Lowpost, Vendo, Riplife, and many more.

He is the happy creator of Ansistrano, the most starred Ansible Galaxy role. He is also the author of the book Domain-Driven Design in PHP. He is also a conference speaker and organizer of the DevOps Barcelona Conference and the PHP Barcelona Conference.

His main areas of expertise are Agile Team Management (Scrum and Kanban), Best Development Practices (Extreme Programming, Domain-Driven Design, and Microservice Architectures), and Digital Transformation (Agile, XP, and DevOps).

You can follow him at Twitter, at his blog or at GitHub.

Christian Soronellas

Christian is an Extreme Programmer and has over 15 years of experience helping tech companies succeed from a broad variety of roles, from Software Engineer to CTO. He has helped companies such as Privalia, Emagister, Atr□□palo, Enalquiler, PlanetaHuerto, PcComponentes or Opositatest. He is the author of the book Domain-Driven Design in PHP as well as a conference co-organizer of DevOps Barcelona Conference and PHP Barcelona Conference

You can follow him at Twitter or at GitHub.

Keyvan Akbary

Keyvan is an Engineering Leader and programmer with more than 15 years of experience crafting products customers love and helping teams succeed. He understands technology as a medium for providing value, not the end itself. He has a passion for Distributed Systems, Software fundamentals, SOLID principles, Clean Code, Design Patterns, Domain-Driven Design and, Testing; as well as being a sporadic Functional Programmer. For the last 7 years he has also focused on growing teams in high scaleup product companies, advocating for customer-centric product development, Extreme Programming, DevOps, Lean, and Kanban.

He has worked on countless projects as a freelancer, on video streaming at Youzee, tradesman marketplace at MyBuilder, in addition to founding his own crowdfunding startup Funddy, and leading FinTech teams at Wise. Currently, he is leading engineering in the ride-hailing space as Head of Engineering at Cabify.

He is also the author of "Domain-Driven Design in PHP" and "The Manager's Manual".

You can follow him at Twitter, at LinkedIn, at his blog or at GitHub.

Getting Started with Domain-Driven Design

So what is all the fuss about? If you've already read books on this topic by Vaughn Vernon and Eric Evans, you're probably familiar with what we're about to say, as we borrow heavily from their definitions and explanations. Domain-Driven Design, or DDD, is an approach that helps us succeed in understanding and building software model designs. It provides us with *strategic* and *tactical* modeling tools to aid designing high-quality software that meets our business goals.

The main goal of this book is to show you PHP code examples of the Domain-Driven Design tactical patterns. If you want to learn more about the strategic patterns and the main Domain-Driven Design, you should read *Domain-Driven Design Distilled*[1] by Vaughn Vernon or *Domain-Driven Design Reference: Definitions and Pattern Summaries*[2] by Eric Evans.

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

Why Domain-Driven Design Matters

Software is not just about code. If you think about it, code is rarely the end goal of our profession. Code is just the medium to solve business problems. So why does it have to talk a different language? Domain-Driven Design emphasizes making sure businesses and software speak the same language. Once broken the barrier, there is no need for translations or tedious syncing, information doesn't get lost. Everyone contributes to discovering the Business Domain, not just coders. The resulting software is the only truth for the common language.

[1] http://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420

[2] http://www.amazon.com/Domain-Driven-Design-Reference-Definitions-Summaries/dp/1457501198

Domain-Driven Design also provides a framework for strategic and tactical design – strategic to pinpoint the most important areas to develop based on business value, and tactical to build a working Domain Model of battle-tested building blocks and patterns.

The Three Pillars of Domain-Driven Design

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

1. **Ubiquitous Language**: Domain Experts and software developers work together to build a common language for the business areas being developed. There's no *us versus them*; it's always *us*. Developing software is a business investment and not just a cost. The effort involved in building the Ubiquitous Language helps spread deep Domain insight among all team members.
2. **Strategic Design**: Domain-Driven Design addresses the strategy behind the direction of the business and not just the technical aspects. It helps define the internal relationships and early warning feedback systems. On the technical side, strategic design protects each business service by providing the motivation for how service-oriented architecture should be achieved.
3. **Tactical Design**: Domain-Driven Design provides the tools and the building blocks for iterative software deliverables. Tactical design tools produce software that is not only correct, but that is also testable and less error prone.

Ubiquitous Language

Along with Bounded Contexts, Ubiquitous Language is one of the main strengths of Domain-Driven Design.

In Terms of Context

For now, consider that a Bounded Context is a conceptual boundary around a system. The Ubiquitous Language inside a boundary has a specific contextual meaning. Concepts outside of this context can have different meanings.

So, how to find, explore and capture this very special language?

- Identify key business processes, their inputs, and their outputs.
- Create a glossary of terms and definitions.

- Capture important software concepts with some kind of documentation.
- Share and expand upon the collected knowledge with the rest of the team (Developers and Domain Experts).

Since Domain-Driven Design was born, new techniques for improving the process of building the Ubiquitous Language have emerged. The most important one, which is now widely adopted across the industry, is Event Storming.

Event Storming

Alberto Brandolini explains Event Storming and its advantages in a blog post[3], and he does it far more succinctly than we could:

> Event Storming is a workshop format for quickly exploring complex business domains. - It is **powerful**: it has allowed me and many practitioners to come up with a comprehensive model of a complete business flow in hours instead of weeks. - It is **engaging**: the whole idea is to bring people with the questions and people who know the answer in the same room and to build a model together. - It is **efficient**: the resulting model is perfectly aligned with a Domain-Driven Design implementation style (particularly fitting an Event Sourcing approach), and allows for a quick determination of Context and Aggregate boundaries. - It is **easy**: the notation is ultra-simple. No complex UML that might cut off participants from the heart of the discussion. - It is **fun**: I always had a great time leading the workshops, people are energised and deliver more than they expected. The right questions arise, and the atmosphere is the right one.

Since the first edition of this book, Event Storming has matured significantly. Brandolini published his definitive guide, *Introducing EventStorming*[4], and the technique has become a standard practice in organizations adopting Domain-Driven Design. Variants such as Big Picture Event Storming (for high-level exploration) and Design-Level Event Storming (for drilling into specific Bounded Contexts) are now commonly used. If you haven't tried Event Storming yet, we strongly recommend it as a starting point for any Domain-Driven Design initiative.

Considering Domain-Driven Design

Domain-Driven Design is not a silver bullet; as with everything in software, it depends on the context. As a rule of thumb, use it to simplify your Domain, but

[3] http://ziobrando.blogspot.com.es/2013/11/introducing-event-storming.html
[4] https://leanpub.com/introducing_eventstorming

never to add more complexity.

If your application is data-centric and your use cases mainly manipulate rows in a database and perform CRUD operations – that is, Create, Read, Update, and Delete – you don't need Domain-Driven Design. Instead, the only thing your company needs is a fancy face in front of your database.

If your application has a small number of use cases and straightforward CRUD-like behavior, it might be simpler to use a framework like Symfony or Laravel to handle your business logic directly. There's no magic number – the old "30 use cases" rule of thumb is less about counting and more about recognizing when your business rules start to outgrow simple procedural code.

However, when your system starts accumulating conditional logic, business rules that span multiple concepts, or workflows that are hard to explain in a single sentence, you may be moving toward the dreaded "Big Ball of Mud[5]." If you know for sure your system will grow in complexity, you should consider using Domain-Driven Design to fight that complexity.

If you know your application is going to grow and is likely to change often, Domain-Driven Design will definitely help in managing the complexity and refactoring your model over time. This is especially true in modern PHP development, where frameworks like Symfony already provide infrastructure that aligns well with Domain-Driven Design principles – such as Symfony Messenger for command and event buses, and Doctrine ORM for persistence that respects your Domain Model.

If you don't understand the Domain you're working on because it's new and nobody has invested in a solution before, this might mean it's complex enough for you to start applying Domain-Driven Design. In this case, you'll need to work closely with Domain Experts to get the models right.

The Tricky Parts

Applying Domain-Driven Design is not easy. It requires time and effort to get around the Business Domain, terminology, research, and collaboration with Domain Experts rather than coding jargon. You'll need to have the commitment of Domain Experts for getting involved in the process too. This will require an open and healthy continuous conversation to model their spoken language into software. On top of that, we'll have to make an effort to avoid thinking technically, to think seriously about the behaviour of objects and the Ubiquitous Language first.

[5] https://en.wikipedia.org/wiki/Big_ball_of_mud

Strategic Overview

In order to provide a general overview of the strategic side of Domain-Driven Design, we'll use an approach from Jimmy Nilsson's book, *Applying Domain-Driven Design and Patterns*[6]. Consider two different spaces: the problem space and the solution space.

In the problem space, Domain-Driven Design uses Domains and Subdomains to group and organize what companies want to solve. In the case of an online travel agency (OTA), the problem is about dealing with things like flight tickets and booking hotels. Such a Domain can be organized into different Subdomains such as Pricing, Inventory, User Management, etc.

In the solution space, Domain-Driven Design provides two patterns: Bounded Contexts and Context Maps. The goal is to define how to provide an implementation to all the identified Subdomains by defining their interactions and the details of those interactions. Continuing with the OTA example, each of the Subdomains will be solved with a Bounded Context implementation – for example, consider a custom Web Application developed by a team for the Pricing Management Subdomain, and an off-the-shelf solution for the User Management Subdomain. The Context Map will show how each Bounded Context is related to the rest. Inside the Context Map, we can see what type of relation two Bounded Contexts have (e.g. customer-supplier, partners). The ideal approach is to have each Subdomain implemented by one Bounded Context, but that's not always possible.

In terms of implementation, when following Domain-Driven Design, you'll end up defining clear boundaries between different parts of your system. These boundaries may be implemented as separate services in a distributed architecture, or as separate modules within a Modular Monolith – the key is the boundary itself, not how it's deployed. As you may already know, distributed architectures are more complex than monolithic ones, so why is this approach interesting, especially for big and complex companies? Is it really worth it? Well, it is. Well-defined boundaries – whether deployed independently or not – are proven to increase overall company productivity because they can be developed and maintained by focused teams. If your Domain – the problem you need to solve – is not complex, applying the strategic part of Domain-Driven Design can add unnecessary overhead and slow down your development speed.

If you want to know more about the strategic part of Domain-Driven Design, you should take a look at the first three chapters of Vaughn Vernon's book, *Im-*

[6] http://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202

plementing Domain-Driven Design[7], or the book *Domain-Driven Design: Tackling Complexity in the Heart of Software*[8] by Eric Evans, both of which specifically focus on this aspect.

Related Movements: Microservices, Self-Contained Systems, and Modular Monoliths

There are other movements promoting architectures that follow the same principles Domain-Driven Design is promoting. Microservices and Self-Contained Systems are well-established examples, and more recently, the Modular Monolith has gained traction as a pragmatic middle ground. James Lewis and Martin Fowler define Microservices in the *Microservices Resource Guide*[9]:

> The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are also independently deployable using fully automated machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and also use different data storage technologies.

If you want to know more about Microservices, their guide is a good place to start.

How is this related to Domain-Driven Design? As explained in Sam Newman's book, *Building Microservices*[10], Microservices are implementations of Domain-Driven Design Bounded Contexts.

In addition to Microservices, another related movement is Self-Contained Systems. According to the Self-Contained Systems website[11]:

> The Self-contained System (SCS) approach is an architecture that focuses on a separation of the functionality into many independent systems, making the complete logical system a collaboration of many

[7] http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon-ebook/dp/B00BCLEBN8

[8] http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215

[9] http://martinfowler.com/microservices/

[10] http://www.amazon.com/Building-Microservices-Sam-Newman/dp/1491950358

[11] http://scs-architecture.org

> smaller software systems. This avoids the problem of large monoliths that grow constantly and eventually become unmaintainable. Over the past few years, we have seen its benefits in many mid-sized and large-scale projects.
>
> The idea is to break a large system apart into several smaller self-contained system, or SCSs, that follow certain rules.

The website also spells out seven characteristics of SCS:

> Each SCS is an autonomous web application. For the SCS's domain all data, the logic to process that data and all code to render the web interface is contained within the SCS. An SCS can fulfill its primary use cases on its own, without having to rely on other systems being available.
>
> Each SCS is owned by one team. This does not necessarily mean that only one team might change the code, but the owning team has the final say on what goes into the code base, for example by merging pull-requests.
>
> Communication with other SCSs or 3rd party systems is asynchronous wherever possible. Specifically, other SCSs or external systems should not be accessed synchronously within the SCS's own request/response cycle. This decouples the systems, reduces the effects of failure, and thus supports autonomy. The goal is decoupling concerning time: An SCS should work even if other SCSs are temporarily offline. This can be achieved even if the communication on the technical level is synchronous, e.g. by replicating data or buffering requests.
>
> An SCS can have an optional service API. Because the SCS has its own web UI it can interact with the user – without going through a UI service. However, an API for mobile clients or for other SCSs might still be useful.
>
> Each SCS must include data and logic. To really implement any meaningful features both are needed. An SCS should implement features by itself and must therefore include both.
>
> An SCS should make its features usable to end-users by its own UI. Therefore the SCS should have no shared UI with other SCSs. SCSs might still have links to each other. However, asynchronous integration means that the SCS should still work even if the UI of another SCS is not available.

> To avoid tight coupling an SCS should share no business code with other SCSs. It might be fine to create a pull-request for an SCS or use common libraries, e.g. database drivers or oAuth clients.

Modular Monoliths

Since the first edition of this book, many teams that adopted Microservices early have learned that distributed systems introduce significant operational complexity – network failures, eventual consistency, distributed tracing, and deployment coordination. This has led to a renewed interest in the Modular Monolith approach: a single deployable application where Bounded Contexts are implemented as well-separated modules with clear boundaries, but without the overhead of network communication between them.

A Modular Monolith gives you many of the benefits of Domain-Driven Design's strategic design – explicit boundaries, independent models, and team autonomy – while keeping the simplicity of a single deployment. If a module later needs to be extracted into a separate service, the clean boundaries make that migration straightforward. For many PHP applications, this is the sweet spot: you get the modeling discipline of Domain-Driven Design without prematurely committing to a distributed architecture.

Exercise

Discuss the pros and cons of such distributed architectures with your workmates. Think about using different languages, deployment processes, infrastructure responsibilities, etc. Also consider: could a Modular Monolith give you the same benefits with less operational complexity?

Wrap-Up

During this chapter you've learned:

- Domain-Driven Design is not about technology; it's actually about providing value in the field you're working in by focusing on the model. Everyone takes part in the process of discovering the Domain, and developers and Domain Experts team up to build the knowledge base by sharing the same language, the Ubiquitous Language.
- Domain-Driven Design provides tactical and strategic modeling tools to design high-quality software. Strategic design targets the business direction,

helps in defining the internal relationships, and technically protects each business service by defining strong boundaries. Tactical design provides useful building blocks for iterative design.

- Domain-Driven Design only makes sense in certain contexts. It's not a silver bullet for every problem in software, so whether or not you use it highly depends on the amount of complexity you're dealing with.
- Domain-Driven Design is a long-term investment; it requires active effort. Domain Experts will be required to collaborate closely with developers, and developers will have to think in terms of the business. In the end, the business customer is the one who has to be pleased.

Implementing Domain-Driven Design requires effort. If it were easy, everybody would be writing high-quality code. Get ready, because you'll soon learn how to write code that, when read, will perfectly describe the business your company operates on. Enjoy this journey!

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 rather 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.

Definition

Ward Cunningham defines[12] 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[13] 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.

[12] http://c2.com/cgi/wiki?ValueObject
[13] http://martinfowler.com/bliki/ValueObject.html

> 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.

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.

Value Objects in Modern PHP

PHP 8.1 introduced `readonly` properties, and PHP 8.2 added `readonly` classes. These language features make Value Objects much more natural to implement in PHP: immutability is now enforced at the language level, eliminating the need for private setter workarounds. A modern Value Object can be as concise as:

```
readonly class Money { public function __-
construct(private int $amount, private Currency
$currency) {} }
```

Throughout this chapter, we'll show the evolution of Value Object implementations. The code examples in the repository have been updated to use these modern PHP features.

Exercise

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

Value Object vs. Entity

Consider the following examples from Wikipedia[14], in order to better understand the difference between Value Objects and Entities.

Value Object:

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

> 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.

Currency and Money Example

`Currency` and `Money` Value Objects are probably the most used examples for explaining Value Objects, thanks to the Money pattern[15]. This design pattern provides a solution for modeling a problem that avoids 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 that meters and yards describe distance units. Each currency is represented with a three-letter uppercase ISO code:

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

```php
class Currency
{
    private string $isoCode;

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

    private function setIsoCode(string $anIsoCode): void
    {
        if (!preg_match('/^[A-Z]{3}$/', $anIsoCode)) {
            throw new \InvalidArgumentException(
                sprintf('"%s" is not a valid ISO code', $anIsoCode)
            );
        }

        $this->isoCode = $anIsoCode;
    }

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

One of the main goals of Value Objects is also the holy grail of Object-Oriented design: encapsulation. By following this pattern, you'll 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 isn't 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[16]. If you need help, take a look at the Money[17] packagist library.

[16] http://www.xe.com/iso4217.php
[17] https://github.com/moneyphp/money

Currency as a Backed Enum?

PHP 8.1 introduced backed enums, which might seem like a natural fit for currencies: `enum Currency: string { case USD = 'USD'; case EUR = 'EUR'; ... }`. This works well when your application only supports a fixed, known set of currencies. However, if your Domain needs to handle any valid ISO 4217 currency – including those added in the future – a class with validation logic remains more flexible. Choose the approach that matches your Domain's constraints.

`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[18] to set the ISO code, which centralizes changes in the Value Object itself:

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

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

```
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.

Characteristics

While modeling an Ubiquitous Language 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 determine whether 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 can be compared with others through value equality.
- It is completely replaceable when the measurement or description changes.
- It supplies its collaborators with side-effect-free behavior.

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`.

Immutability

This is one of the most important aspects to grasp. Object values shouldn't 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. In modern PHP, declaring properties as `readonly` enforces this at the language level — any attempt to modify a readonly property after construction will result in an `Error`.

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[19], which is considered an anti-pattern.

It's also good to point out that it's not recommended to hold references to Entities in your Value Objects. Entities are mutable, and holding references to them could lead to undesirable side effects occurring in the Value Object.

In languages with method overloading[20], 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're able to provide a similar capability by way of factory methods[21]. 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` and `Currency` objects, we could add some useful factory methods like the following:

```
1  class Currency
2  {
3      private string $isoCode;
4
5      public static function fromValue(string $anIsoCode): self
6      {
7          return new self($anIsoCode);
8      }
9
10     private function __construct(string $anIsoCode)
11     {
12         $this->setIsoCode($anIsoCode);
13     }
14
15     // ...
16 }
```

[19] http://www.martinfowler.com/bliki/AnemicDomainModel.html
[20] http://en.wikipedia.org/wiki/Function_overloading
[21] http://en.wikipedia.org/wiki/Factory_method_pattern

```php
class Money
{
    // ...

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

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

    public static function fromAmountAndCurrency(
        int $anAmount,
        Currency $aCurrency
    ): self {
        return new self(
            $anAmount,
            $aCurrency
        );
    }
}
```

By using the `self` keyword, we don't couple the code with the class name. As such, a change to the class name or namespace won't affect these factory methods. Be also aware of making the constructor private so using your factory methods is the only way to instantiate new objects. These small implementation details 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, for example, a `Money` Value Object, we're required to instead return a new `Money` instance with the desired modifications. Fortunately, it's relativity simple to abide by this rule, as shown in the example below:

```php
class Money
{
    // ...

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

The `Money` object returned by `increaseAmountBy` is different from the `Money` client object that received the method call. This can be observed in the example comparability checks below:

```php
$aMoney = Money::fromAmountAndCurrency(100, Currency::fromValue('USD'));
$otherMoney = $aMoney->increaseAmountBy(100);

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

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

Conceptual Whole

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

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

    private int $amount;
    private string $currency;

    // ...
}
```

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 it also allows you to create higher-order concepts and a more concrete Ubiquitous Language.

Exercise

Discuss with a peer whether or not an email could be considered a Value Object. Does the context it's used in matter?

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, imagine 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, they both represent the same value, which makes them equal.

In regards to PHP, it's commonplace to compare two Value Objects using the `==` operator. Examining the PHP Documentation[22] definition of the operator highlights an interesting behavior:

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

> 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 subtyped Value Objects.

Keeping this in mind, the even stricter `===` operator doesn't help us, unfortunately:

> 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:

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

```php
class Currency
{
    // ...

    public function equals(self $currency): bool
    {
        return $currency->isoCode() === $this->isoCode();

        // You could also access directly
        // the $isoCode field if necessary
        // return $currency->isoCode === $this->isoCode;
    }
}
```

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

```php
class Money
{
    // ...

    public function equals(self $aMoney): bool
    {
        return
            $aMoney->currency()->equals($this->currency()) &&
            $aMoney->amount() === $this->amount();
    }
}
```

Replaceability

Consider a `Product` Entity that contains a `Money` Value Object used to quantify its price. Additionally, consider 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's correct or not, it's not what people expect.

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

```
$this->price = Money::fromAmountAndCurrency(100, Currency::fromValue('USD'));
// ...
$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.

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:

```
class Money
{
    // ...

    public function add(self $aMoney): self
    {
        if (!$aMoney->currency()->equals($this->currency())) {
            throw new \InvalidArgumentException(
                'Currencies do not match'
            );
        }

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

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 a mysterious method call to `otherMethod` in our code:

```
class Banking
{
    public function doSomething(): void
    {
        $aMoney = Money::fromAmountAndCurrency(100, Currency::fromValue('USD'));

        $this->otherMethod($aMoney); //mysterious call
        // ...
    }
}
```

Everything is fine until, for some reason, we start seeing unexpected results when we're returning or finished with `otherMethod`. Suddenly, `$aMoney` no longer contains 100 USD. What happened? And what happens if `otherMethod` internally uses our previously defined `add` method? Maybe you're unaware that `add` mutates the state of the `Money` instance. This is what we call a side effect. You must avoid generating side effects. You must not mutate your arguments. If you do, the developer using your objects may experience strange behaviors. They'll complain, and they'll be correct.

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:

```
class Money
{
    // ...

    public function add(self $aMoney): self
    {
        $this->guardSameCurrencies($aMoney);

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

    private function guardSameCurrencies(self $aMoney): void
    {
        if (!$aMoney->currency()->equals($this->currency())) {
            throw new \InvalidArgumentException(
                'Currencies do not match'
            );
        }
```

```
22      }
23  }
```

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.

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  // int(20)
10 $a = $a + 30;
11 var_dump($a);
12 // int(50)
```

Although `$a` and `$b` are different variables stored in different memory locations, when compared, they're the same. They hold the same value, so we consider them equal. You can change the value of `$a` from `10` to `20` at any time that you want, making the new value `20` and eliminating the `10`. You can replace integer values as much as you want without consideration of the previous value because you're not modifying it; you're just replacing it. If you apply any operation – such as addition (i.e. `$a + $b`) – to these variables, you get another new value that can be assigned to another variable or a previously defined one. When you pass `$a` to another function, except when explicitly passed by reference, you're passing a value. It doesn't matter if `$a` gets modified within that function, because in your current code, you'll still have the original copy. Value Objects behave as basic types.

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're testing before performing any modifications. Assert both are equal using the implemented equality check. Perform the actions you want to test and assert the results. Finally, assert that the original object and copy are still equal.

Let's put this into practice and test the side-effect-free implementation of our `add` method in the `Money` class:

```
class MoneyTest extends TestCase
{
    /**
     * @test
     */
    public function copiedMoneyShouldRepresentSameValue()
    {
        $aMoney = Money::fromAmountAndCurrency(100, Currency::fromValue('USD'));

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

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

    /**
     * @test
     */
    public function originalMoneyShouldNotBeModifiedOnAddition()
    {
        $aMoney = Money::fromAmountAndCurrency(100, Currency::fromValue('USD'));
        $otherMoney = Money::fromAmountAndCurrency(20, Currency::fromValue('USD')\
);

        $aMoney->add($otherMoney);

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

    /**
     * @test
     */
    public function moniesShouldBeAdded()
    {
        $aMoney = Money::fromAmountAndCurrency(100, Currency::fromValue('USD'));
        $otherMoney = Money::fromAmountAndCurrency(20, Currency::fromValue('USD')\
);

        $newMoney = $aMoney->add($otherMoney);

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

```
42
43      // ...
44  }
```

Persisting Value Objects

Value Objects are not persisted on their own; they're typically persisted within an Aggregate. Value Objects shouldn't 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 isn't 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've intentionally decided to simplify this example, with the id being a string and not a Value Object:

```
1   class Product
2   {
3       private string $productId;
4       private string $name;
5       private Money $price;
6
7       public static function create(
8           string $aProductId,
9           string $aName,
10          Money $aPrice
11      ): self {
12          return new self(
13              $aProductId,
14              $aName,
15              $aPrice
16          );
17      }
18
19      protected function __construct(
20          string $aProductId,
21          string $aName,
22          Money $aPrice
23      ) {
24          $this->setProductId($aProductId);
25          $this->setName($aName);
```

```
26          $this->setPrice($aPrice);
27      }
28
29      // ...
30  }
```

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

```
1  $product = Product::create(
2      $productRepository->nextIdentity(),
3      'Domain-Driven Design in PHP',
4      Money::fromAmountAndCurrency(999, Currency::fromValue('USD'))
5  );
6
7  $productRepository->add($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'll 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[23] 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.

Since the first edition, Doctrine has evolved to version 3.x. The most notable change for Value Objects is that PHP attributes have replaced XML and annotation-based mapping. Value Objects can now be mapped as embeddables using `#[ORM\Embeddable]` on the class and `#[ORM\Embedded]` on the owning Entity's property. This is more concise and keeps the mapping close to the code.

[23] https://www.doctrine-project.org/projects/orm.html

Persisting Single Value Objects

Many different options are available for persisting a single Value Object. These range from using Serialize 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 company may have developed in order to persist Entities in a database. In our scenario, the ad hoc ORM code is going to be implemented using the DBAL[24] library. According to the official documentation[25], "The Doctrine database abstraction & access layer (DBAL) offers a lightweight and thin runtime layer around a PDO-like API and a lot of additional, horizontal features like database schema introspection and manipulation through an OO API."

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:

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

For persisting the Entity in the database, our Repository has to map each of the fields of the Entity and the ones from the `Money` Value Object. If you're using an ad hoc ORM Repository based on DBAL – let's call it `DbalProductRepository` – you must take care of creating the `INSERT` statement, binding the parameters, and executing the statement:

[24] http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/

[25] http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/introduction.html

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

        $stmt->execute();
        // ...
    }
}
```

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

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

As you can see, you can map your Value Objects and query parameters in an ad hoc manner 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:

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

        return Product::create(
            $row['id'],
            $row['name'],
            Money::fromAmountAndCurrency(
                $row['price_amount'],
                Currency::fromValue(
                    $row['price_currency']
                )
            )
        );
    }
}
```

There are some benefits to this approach. First, you can easily read, step by step, how the persistence and subsequent creations 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 Aggregate's creation. If you use the `new` operator, you'll 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 only be created with the `new` operator 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.

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

The latest stable Doctrine release is currently version 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 also 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[26]. This option, if implemented correctly, is definitely the one we recommend most. It would be the simplest, most elegant solution, that also provides search support in your DQL queries.

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

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

    <entity
        name="Product"
        table="product">
        <id
            name="id"
            column="id"
            type="string"
            length="255">
            <generator
                strategy="NONE">
            </generator>
        </id>

        <field
```

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

```
            name="name"
            type="string"
            length="255"
        />

        <embedded
            name="price"
            class="Ddd\Domain\Model\Money"
        />
    </entity>
</doctrine-mapping>
```

In the product mapping, we're 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:

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

    <embeddable
        name="Ddd\Domain\Model\Money">

        <field
            name="amount"
            type="integer"
        />

        <embedded
            name="currency"
            class="Ddd\Domain\Model\Currency"
        />
    </embeddable>
</doctrine-mapping>
```

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

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

    <embeddable
        name="Ddd\Domain\Model\Currency">

        <field
            name="iso"
            type="string"
            length="3"
        />
    </embeddable>
</doctrine-mapping>
```

As you can see, the above code has a standard embeddable definition with just one string field that holds the ISO code. This approach is the easiest way to use embeddables 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.

Embedded Value with Doctrine <= 2.4.*

If you're still stuck in Doctrine 2.4 probably it's time to consider an upgrade since this version of Doctrine is currently too old (6 years old as of day of writing). Anyway you may wonder what an acceptable solution for using Embedded Values with Doctrine < 2.5 is. 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:

```php
class Product
{
    private string $productId;
    private string $name;
    private Money $price;

    private string $surrogateCurrencyIsoCode;
    private int $surrogateAmount;

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

    private function setPrice(Money $aMoney): void
    {
        $this->price = $aMoney;

        $this->surrogateAmount = $aMoney->amount();
        $this->surrogateCurrencyIsoCode =
            $aMoney->currency()->isoCode();
    }

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

        return $this->price;
    }

    // ...
}
```

As you can see, there are two new attributes: one for the amount, and another for the ISO code of the currency. We'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:

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

    <entity
        name="Product"
        table="product">

        <id
            name="id"
            column="id"
            type="string"
            length="255">
            <generator
                strategy="NONE">
            </generator>
        </id>

        <field
            name="name"
            type="string"
            length="255"
        />

        <field
            name="surrogateAmount"
            type="integer"
            column="price_amount"
        />
        <field
            name="surrogateCurrencyIsoCode"
            type="string"
            column="price_currency"
        />
    </entity>
</doctrine-mapping>
```

Surrogate Attributes

These two new fields don't strictly belong to the Domain, as they don't refer to Infrastructure details. Rather, they're 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's 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[27]. 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're 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'll 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.

Serialized LOB and Ad Hoc ORM

If the addition of searching capabilities to the Value Objects attributes is not important, there's 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 are reduced to a single column:

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

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

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

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

        $stmt->bindValue(
            3,
            serialize(
                $aProduct->price()
            )
        );

        // ...
    }
}
```

Let's see how our `Product` is now represented in the database. The table column `price` is a `TEXT` type column that contains a serialization of a `Money` object representing 9.99 USD:

```
mysql> select * from products \G
*************************** 1. row ***************************
   id: 1
 name: Domain-Driven Design in PHP
price: O:22:"Ddd\Domain\Model\Money":2:{s:30:" Ddd\Domain\Model\Money amount";i:9\
99;s:32:" Ddd\Domain\Model\Money currency";O:25:"Ddd\Domain\Model\Currency":1:{s:\
34:" Ddd\Domain\Model\Currency isoCode";s:3:"USD";}}
1 row in set (0.00 sec)
```

This approach does the job. However, it'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.

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's another favored approach: using an open source serializer library such as Zumba JsonSerializer[28]. Let's see an example of applying it when serializing a `Money` object:

```
$myMoney = Money::fromAmountAndCurrency(
    999,
    Currency::fromValue('USD')
);

$serializer = new \Zumba\JsonSerializer\JsonSerializer();

$json = $serializer->serialize($myMoney);
```

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

```
$serializer = new \Zumba\JsonSerializer\JsonSerializer();
// ...
$myMoney = $serializer->unserialize($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 which attributes of an object should be omitted in the serialization process – for example, a password.

Check out the Mapping Reference[29] and the Cookbook[30] for more information. JMS Serializer is a must in any Domain-Driven Design project.

Serialized LOB with Doctrine

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

[28] https://github.com/zumba/json-serializer
[29] http://jmsyst.com/libs/serializer/master/reference/xml_reference
[30] http://jmsyst.com/libs/serializer/master/cookbook

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[31], `object` type:

> maps and converts object data based on PHP serialization. If you need to store an exact representation of your object data, you should consider using this type as it uses serialization to represent an exact copy of your object as string in the database. Values retrieved from the database are always converted to PHP's object type using unserialization or null if no data is present.
>
> This type will always be mapped to the database vendor's `text` type internally as there is no way of storing a PHP object representation natively in the database. Furthermore this type requires a 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:

[31] http://doctrine-orm.readthedocs.io/projects/doctrine-dbal/en/latest/reference/types.html#object

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

    <entity
        name="Product"
        table="products">

        <id
            name="id"
            column="id"
            type="string"
            length="255">
            <generator
                strategy="NONE">
            </generator>
        </id>

        <field
            name="name"
            type="string"
            length="255"
        />

        <field
            name="price"
            type="object"
        />
    </entity>
</doctrine-mapping>
```

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

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

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

```
// ...
$repository = $em->getRepository(Product::class);
$item = $repository->find(1);
var_dump($item);

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

Last but not least, the Doctrine DBAL 2 Documentation[32] 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.

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?

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, in order to persist the former.

As the Doctrine DBAL 2 Documentation[33] explains:

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

[33] http://doctrine-orm.readthedocs.io/projects/doctrine-dbal/en/latest/reference/types.html#custom-mapping-types

> 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, that 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(
11         $value,
12         AbstractPlatform $platform
13     )
14     {
15         $value = parent::convertToPHPValue($value, $platform);
16
17         $value = explode('|', $value);
18         return Money::fromAmountAndCurrency(
19             $value[0],
20             Currency::fromValue($value[1])
21         );
22     }
23
24     public function convertToDatabaseValue(
25         $value,
26         AbstractPlatform $platform
27     )
28     {
29         return implode(
30             '|',
31             [
32                 $value->amount(),
33                 $value->currency()->isoCode()
34             ]
35         );
36     }
37
```

```
38      public function getName()
39      {
40          return self::MONEY;
41      }
42  }
```

Using Doctrine, you're required to register all Custom Types. It's common to use an `EntityManagerFactory` that centralizes this `EntityManager` creation. Alternatively, you could 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   use Ddd\Infrastructure\Persistence\Doctrine\Type\MoneyType;
6
7   class EntityManagerFactory
8   {
9       public function build(): EntityManager
10      {
11          Type::addType('money', MoneyType::class);
12
13          return EntityManager::create(
14              [
15                  'driver'   => 'pdo_mysql',
16                  'user'     => 'root',
17                  'password' => '',
18                  'dbname'   => 'ddd',
19              ],
20              Setup::createXMLMetadataConfiguration(
21                  [__DIR__.'/config'],
22                  true
23              )
24          );
25      }
26  }
```

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

```xml
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping>

    <entity
        name="Product"
        table="product">

        <!-- ... -->

        <field
            name="price"
            type="money"
        />
    </entity>
</doctrine-mapping>
```

Why Use XML Mapping?

Thanks to the XSD schema validation in the headers of the XML mapping file, most integrated development environments (IDEs) provide auto-complete functionality for all the elements and attributes present in the mapping definition. And the other widely used mapping format, the YAML mapping format, has been deprecated in the 2.7 version.

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

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

This approach is an improvement on the one before in terms of future refactoring. However, searching capabilities remain limited due to the format of the column. With the Doctrine Custom types, you can improve the situation a little, but it's still not the best option for building your DQL queries. See Doctrine Custom Mapping Types[34] 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.

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

Persisting a Collection of Value Objects

Imagine that we'd now 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:

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

    public static function create(
        string $aProductId,
        string $aName,
        Money $aPrice,
        array $somePrices
    ): self {
        return new self(
            $aProductId,
            $aName,
            $aPrice,
            $somePrices
        );
    }

    private function __construct(
        string $aProductId,
        string $aName,
        Money $aPrice,
        array $somePrices
    )
    {
        parent::__construct($aProductId, $aName, $aPrice);
        $this->setPrices($somePrices);
    }

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

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

`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 are 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?

Collection Serialized into a Single Column

Serializing a collection of Value Objects into a single column is most likely the easiest solution. Everything that was previously explained in the section about 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

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

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'll 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're still treated as Value Objects.

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

```
CREATE TABLE historical_products (
    id CHAR(36) NOT NULL,
    name VARCHAR(255) NOT NULL,
    price_amount INT(11) NOT NULL,
    price_currency CHAR(3) NOT NULL,
    PRIMARY KEY (id)
);
```

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 a collection. A new `prices` table is now required in order to persist all the different `Money` Value Objects that a `Product` Entity can handle:

```
CREATE TABLE prices (
    id INT(11) NOT NULL AUTO_INCREMENT,
    amount INT(11) NOT NULL,
    currency CHAR(3) COLLATE NOT NULL,
    PRIMARY KEY (id)
);
```

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

```
CREATE TABLE products_prices (
    product_id CHAR(36) NOT NULL,
    price_id INT(11) NOT NULL,
    PRIMARY KEY (product_id, price_id),
    UNIQUE KEY uniq_price_id (price_id),
    KEY idx_product_id (product_id),
    CONSTRAINT fk_product_id FOREIGN KEY (product_id) REFERENCES historical_produ\
cts (id),
    CONSTRAINT fk_price_id FOREIGN KEY (price_id) REFERENCES prices (id)
);
```

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 required in order to avoid this so-called **boundary leak**. With a class extension, creating `new` instances

of the `Money` Value Object class from any Domain Object isn't 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:

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

    private string $surrogateId;
    private string $surrogateCurrencyIsoCode;

    public static function fromAmountAndCurrency(
        int $anAmount,
        Currency $aCurrency
    ): self {
        return new self($anAmount, $aCurrency);
    }

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

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

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

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

```

```
42          return $this->currency;
43      }
44
45      public function amount(): int
46      {
47          return $this->amount;
48      }
49
50      public function equals(self $aMoney): bool
51      {
52          return
53              $this->amount() === $aMoney->amount() &&
54              $this->currency()->equals($aMoney->currency());
55      }
56  }
```

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.github.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"
```

```
        />
        <field
            name="surrogateCurrencyIsoCode"
            type="string"
            column="currency"
        />
    </entity>
</doctrine-mapping>
```

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

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

    <entity
        name="Ddd\Domain\Model\HistoricalProduct"
        table="historical_products"
        repository-class="Ddd\Infrastructure\Domain\Model\DoctrineHistoricalProdu\
ctRepository">

        <many-to-many
            field="prices"
            target-entity="Ddd\Domain\Model\Money">

            <cascade>
                <cascade-all/>
            </cascade>

            <join-table
                name="products_prices">

                <join-columns>
                    <join-column
                        name="product_id"
                        referenced-column-name="id"
                    />
                </join-columns>

                <inverse-join-columns>
                    <join-column
                        name="price_id"
                        referenced-column-name="id"
                        unique="true"
                    />
```

```
38                </inverse-join-columns>
39            </join-table>
40        </many-to-many>
41    </entity>
42 </doctrine-mapping>
```

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. It's important to be careful about 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.

Collection Backed by a Database Entity

Database Entity is the same solution as Join Table, with the addition of the Value Object that's 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.

NoSQL

What about NoSQL mechanisms such as Redis, MongoDB, or CouchDB? Unfortunately, you can't escape 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'll 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.

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[35] 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. According to the MySQL 5.7 Reference Manual[36], 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.

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[37], 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'll be checking that the string is a valid IATA code. What's the chance you forget somewhere important? On the other hand, think about trying to instantiate `new IATA("BCN'; DROP TABLE users;--")`. If you centralize the guards[38] in

[35] http://www.postgresql.org/docs/9.4/static/functions-json.html
[36] https://dev.mysql.com/doc/refman/5.7/en/json.html
[37] https://en.wikipedia.org/wiki/International_Air_Transport_Association_airport_code
[38] https://en.wikipedia.org/wiki/Guard_(computer_science)

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[39] or read his blog[40].

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're 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).

[39] https://twitter.com/danbjson
[40] http://dearjunior.blogspot.com.es/search/label/domain%20driven%20security

Appendix: Hexagonal Architecture with PHP

The following article was posted in php|architect magazine in June 2014 by Carlos Buenosvinos.

Introduction

With the rise of Domain-Driven Design (DDD), architectures promoting domain centric designs are becoming more popular. This is the case with **Hexagonal Architecture**, also known as **Ports and Adapters**, that seems to have being rediscovered just now by PHP developers. Invented in 2005 by Alistair Cockburn, one of the Agile Manifesto authors, the Hexagonal Architecture allows an application to be equally driven by users, programs, automated tests or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases. This results into agnostic infrastructure web applications that are easier to test, write and maintain. Let's see how to apply it using real PHP examples.

Your company is building a brainstorming system called *Idy*. Users add and rate ideas so the most interesting ones can be implemented in a company. It 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?

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.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        // Getting parameters from the request
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        // Building database connection
        $db = new Zend_Db_Adapter_Pdo_Mysql([
            'host'     => 'localhost',
            'username' => 'idy',
            'password' => '',
            'dbname'   => 'idy'
        ]);

        // Finding the idea in the database
        $sql = 'SELECT * FROM ideas WHERE idea_id = ?';
        $row = $db->fetchRow($sql, $ideaId);
        if (!$row) {
            throw new Exception('Idea does not exist');
        }

        // Building the idea from the database
        $idea = new Idea();
        $idea->setId($row['id']);
        $idea->setTitle($row['title']);
        $idea->setDescription($row['description']);
        $idea->setRating($row['rating']);
        $idea->setVotes($row['votes']);
        $idea->setAuthor($row['email']);

        // Add user rating
        $idea->addRating($rating);

        // Update the idea and save it to the database
```

```
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.

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.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new IdeaRepository();
        $idea = $ideaRepository->find($ideaId);
        if (!$idea) {
            throw new Exception('Idea does not exist');
        }

        $idea->addRating($rating);
        $ideaRepository->update($idea);

        $this->redirect('/idea/'.$ideaId);
    }
}

class IdeaRepository
{
    private $client;

    public function __construct()
    {
        $this->client = new Zend_Db_Adapter_Pdo_Mysql([
            'host'     => 'localhost',
            'username' => 'idy',
            'password' => '',
            'dbname'   => 'idy'
        ]);
    }

    public function find($id)
    {
```

```
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.

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` behavior into an interface, renamed the IdeaRepository into `MySQLIdeaRepository` and updated the `rateAction` to use our `MySQLIdeaRepository`. But what's the benefit?

We can now exchange the repository used in the controller with any implementing the same interface. So, let's try a different implementation.

Migrating our Persistence to Redis

During the sprint and after talking to some mates, you realize that using a NoSQL strategy could improve the performance of your feature. Redis is one of your best friends. Go for it and show me your Listing 4.

```
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      }
```

```php
}

interface IdeaRepository
{
    // ...
}

class RedisIdeaRepository implements IdeaRepository
{
    private $client;

    public function __construct()
    {
        $this->client = new \Predis\Client();
    }

    public function find($id)
    {
        $idea = $this->client->get($this->getKey($id));
        if (!$idea) {
            return null;
        }

        return unserialize($idea);
    }

    public function update(Idea $idea)
    {
        $this->client->set(
            $this->getKey($idea->getId()),
            serialize($idea)
        );
    }

    private function getKey($id)
    {
        return 'idea:' . $id;
    }
}
```

Easy again. You've created a `RedisIdeaRepository` that implements `IdeaRepository` interface and we have decided to use Predis as a connection manager. Code looks smaller, easier and faster. But what about the controller? It remains the same, we have just changed which repository to use, but it was just one line of code.

As an exercise for the reader, try to create the IdeaRepository for SQLite, a file or an in-memory implementation using arrays. Extra points if you think about how ORM Repositories fit with Domain Repositories and how ORM *@annotations* affect this architecture.

Decouple Business and Web Framework

We have already seen how easy it can be to changing from one persistence strategy to another. However, the persistence is not the only edge from our Hexagon. What about how the user interacts with the application?

Your CTO has set up in the roadmap that your team is moving to Symfony2, so when developing new features in you current ZF1 application, we would like to make the incoming migration easier. That's tricky, show me your Listing 5.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute($ideaId, $rating);

        $this->redirect('/idea/'.$ideaId);
    }
}

interface IdeaRepository
{
    // ...
}

class RateIdeaUseCase
{
    private $ideaRepository;

    public function __construct(IdeaRepository $ideaRepository)
    {
        $this->ideaRepository = $ideaRepository;
    }

    public function execute($ideaId, $rating)
    {
        try {
            $idea = $this->ideaRepository->find($ideaId);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }

        if (!$idea) {
            throw new IdeaDoesNotExistException();
```

```
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.

```php
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $this->redirect('/idea/'.$response->idea->getId());
    }
}

class RateIdeaRequest
{
    public $ideaId;
    public $rating;

    public function __construct($ideaId, $rating)
    {
        $this->ideaId = $ideaId;
        $this->rating = $rating;
    }
}

class RateIdeaResponse
{
    public $idea;

    public function __construct(Idea $idea)
    {
        $this->idea = $idea;
    }
}

class RateIdeaUseCase
{
    // ...

    public function execute($request)
    {
        $ideaId = $request->ideaId;
        $rating = $request->rating;

        // ...

```

```
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.

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.

Console app rating

Sometimes, a UseCase is going to be executed from a Cron job or the command line. As examples, batch processing or some testing command lines to accelerate the development.

While testing this feature using the web or the API, you realize that it would be nice to have a command line to do it, so you don't have to go through the browser.

If you are using shell scripts files, I suggest you to check the Symfony Console component. What would the code look like?

```php
namespace Idy\Console\Command;

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

class VoteIdeaCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('idea:rate')
            ->setDescription('Rate an idea')
            ->addArgument('id', InputArgument::REQUIRED)
            ->addArgument('rating', InputArgument::REQUIRED)
        ;
    }

    protected function execute(
        InputInterface $input,
        OutputInterface $output
    ) {
        $ideaId = $input->getArgument('id');
        $rating = $input->getArgument('rating');

        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $output->writeln('Done!');
    }
}
```

Again those 3 lines of code. As before, the UseCase and its business logic remain untouched, we are just providing a new *delivery mechanism*. Congratulations, you've discovered the *user side* hexagon edge.

There is still a lot to do. As you may have heard, a real craftsman does TDD. We have already started our story so we must be ok with just testing after.

Testing Rating an Idea UseCase

Michael Feathers introduced a definition of legacy code as *code without tests*. You don't want your code to be legacy just born, do you?

In order to unit test this UseCase object, you decide to start with the easiest part, what happens if the repository is not available? How can we generate such behavior? Do we stop our Redis server while running the unit tests? No. We need to have an object that has such behavior. Let's use a *mock* object in Listing 9.

```
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 }
```

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.

```php
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function whenIdeaDoesNotExistAnExceptionShouldBeThrown()
    {
        $this->setExpectedException('IdeaDoesNotExistException');
        $ideaRepository = new EmptyIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $useCase->execute(
            new RateIdeaRequest(1, 5)
        );
    }
}

class EmptyIdeaRepository implements IdeaRepository
{
    public function find($id)
    {
        return null;
    }

    public function update(Idea $idea)
    {

    }
}
```

Here, we use the same strategy but with an `EmptyIdeaRepository`. It also implements the same interface but the implementation always returns `null` regardless which identifier the `find` method receives.

Why are we testing these cases?, remember Kent Beck's words: "Test everything that could possibly break".

Let's carry on with the rest of the feature. We need to check a special case that is related with having a read available repository where we cannot write to. Solution can be found in Listing 11.

```php
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function whenUpdatingInReadOnlyAnIdeaAnExceptionShouldBeThrown()
    {
        $this->setExpectedException('NotAvailableRepositoryException');
        $ideaRepository = new WriteNotAvailableRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest(1, 5)
        );
    }
}

class WriteNotAvailableRepository implements IdeaRepository
{
    public function find($id)
    {
        $idea = new Idea();
        $idea->setId(1);
        $idea->setTitle('Subscribe to php[architect]');
        $idea->setDescription('Just buy it!');
        $idea->setRating(5);
        $idea->setVotes(10);
        $idea->setAuthor('hi@carlos.io');

        return $idea;
    }

    public function update(Idea $idea)
    {
        throw new NotAvailableException();
    }
}
```

Ok, now the key part of the feature is still remaining. We have different ways of testing this, we can write our own mock or use a mocking framework such as Mockery or Prophecy. Let's choose the first one. Another interesting exercise would be to write this example and the previous ones using one of these frameworks.

```php
class RateIdeaUseCaseTest extends \PHPUnit_Framework_TestCase
{
    // ...

    /**
     * @test
     */
    public function whenRatingAnIdeaNewRatingShouldBeAddedAndIdeaUpdated()
    {
        $ideaRepository = new OneIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest(1, 5)
        );

        $this->assertSame(5, $response->idea->getRating());
        $this->assertTrue($ideaRepository->updateCalled);
    }
}

class OneIdeaRepository implements IdeaRepository
{
    public $updateCalled = false;

    public function find($id)
    {
        $idea = new Idea();
        $idea->setId(1);
        $idea->setTitle('Subscribe to php[architect]');
        $idea->setDescription('Just buy it!');
        $idea->setRating(5);
        $idea->setVotes(10);
        $idea->setAuthor('hi@carlos.io');

        return $idea;
    }

    public function update(Idea $idea)
    {
        $this->updateCalled = true;
    }
}
```

Bam! 100% Coverage for the UseCase. Maybe, next time we can do it using TDD so the test will come first. However, testing this feature was really easy because of the way decoupling is promoted in this architecture.

Maybe you are wondering about this:

```
$this->updateCalled = true;
```

We need a way to guarantee that the update method has been called during the UseCase execution. This does the trick. This *test double* object is called a *spy*, *mocks* cousin.

When to use mocks? As a general rule, use mocks when crossing boundaries. In this case, we need mocks because we are crossing from the domain to the persistence boundary.

What about testing the infrastructure?

Testing Infrastructure

If you want to achieve 100% coverage for your whole application you will also have to test your infrastructure. Before doing that, you need to know that those unit tests will be more coupled to your implementation than the business ones. That means that the probability to be broken with implementation details changes is higher. So it 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.

```
class IdeaController extends Zend_Controller_Action
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $useCase = new RateIdeaUseCase(
            new RedisIdeaRepository(
                new \Predis\Client()
            )
        );

        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $this->redirect('/idea/'.$response->idea->getId());
    }
}

class RedisIdeaRepository implements IdeaRepository
{
```

```
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.

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.

```php
class IdeaController extends ContainerAwareController
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $useCase = $this->get('rate_idea_use_case');
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );

        $this->redirect('/idea/'.$response->idea->getId());
    }
}
```

The controller has been modified to have access to the container, that's why it is inheriting from a new base controller `ContainerAwareController` that has a `get` method to retrieve each of the services contained.

```xml
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services
           http://symfony.com/schema/dic/services/services-1.0.xsd">
    <services>
        <service
            id="rate_idea_use_case"
            class="RateIdeaUseCase">
            <argument type="service" id="idea_repository" />
        </service>

        <service
            id="idea_repository"
            class="RedisIdeaRepository">
            <argument type="service">
                <service class="Predis\Client" />
            </argument>
        </service>
    </services>
</container>
```

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

Domain Services and Notification Hexagon Edge

Are we forgetting something? “the author should be notified by email”, yeah! That’s true. Let’s see in Listing 16 how we have updated the UseCase for doing the job.

```
class RateIdeaUseCase
{
    private $ideaRepository;
    private $authorNotifier;

    public function __construct(
        IdeaRepository $ideaRepository,
        AuthorNotifier $authorNotifier
    )
    {
        $this->ideaRepository = $ideaRepository;
        $this->authorNotifier = $authorNotifier;
    }

    public function execute(RateIdeaRequest $request)
    {
        $ideaId = $request->ideaId;
        $rating = $request->rating;

        try {
            $idea = $this->ideaRepository->find($ideaId);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }

        if (!$idea) {
            throw new IdeaDoesNotExistException();
        }

        try {
            $idea->addRating($rating);
            $this->ideaRepository->update($idea);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }

        try {
            $this->authorNotifier->notify(
                $idea->getAuthor()
            );
        } catch(Exception $e) {
            throw new NotificationNotSentException();
        }

```

```
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.

Let's Recap

In order to have a *clean architecture* that helps you create easy to write and test applications, we can use Hexagonal Architecture. To achieve that, we encapsulate user story business rules inside a UseCase or Interactor object. We build the UseCase request from our framework request, instantiate the UseCase and all its dependencies and then execute it. We get the response and act accordingly based on it. If our framework has a Dependency Injection component you can use it to simplify the code.

The same UseCase objects can be used from different *delivery mechanisms* in order to allow users access the features from different clients (web, API, console, etc.)

For testing, play with mocks that behave like all the interfaces defined so special cases or error flows can also be covered. Enjoy the good job done.

Hexagonal Architecture

In almost all the blogs and books you will find drawings about concentric circles representing different areas of software. As Robert C. Martin explains in his "Clean Architecture" post, the outer circle is where your infrastructure resides. The inner circle is where your Entities live. The overriding rule that makes this architecture work is **The Dependency Rule**. This rule says that source code dependencies can

only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle.

Key Points

Use this approach if 100% unit test code coverage is important to your application. Also, if you want to be able to switch your storage strategy, web framework or any other type of third-party code. The architecture is especially useful for long-lasting applications that need to keep up with changing requirements.

What's Next?

If you are interested in learning more about Hexagonal Architecture and other near concepts you should review the related URLs provided at the beginning of the article, take a look at CQRS and Event Sourcing. Also, don't forget to follow the DDD community online and follow people like @VaughnVernon and @ericevans0.

Bibliography

Beck, Kent. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2002.

Brandolini, Alberto. *Introducing EventStorming*. Leanpub, 2016.

Evans, Eric. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014.

Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2012.

Martin, Robert C. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.

Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

Meszaros, Gerard. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007.

Newman, Sam. *Building Microservices*. O'Reilly Media, 2015.

Nilsson, Jimmy. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*. Addison-Wesley Professional, 2006.

Sadalage, Pramod J., and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

Vernon, Vaughn. *Domain-Driven Design Distilled*. Addison-Wesley Professional, 2016.

Vernon, Vaughn. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.