CHAPTER 2

Introduction

Object-Oriented Programming (OOP) and the Domain Model

PHP developers nowadays generally think and code in an object-oriented way. Application functionality is built using classes, objects, methods, inheritance, and other object-oriented techniques. In the beginning, OOP was used primarily for the general technical aspects of applications, such as MVC frameworks or logging and mailing libraries. All these components can be used more or less "as is" in other applications, regardless of the domains those applications inhabit: for example, e-commerce, portal, or community site. For complex systems, or if aspects such as maintainability and extensibility are important, OOP is also an advantage in the domain-specific code. Basically, every application consists of two types of code: general technical code and domain-specific code. General technical code is often reusable when built as a library or framework; domain-specific code is often too customized to be reused.

Object-oriented domain-specific code is characterized by the existence of a so-called *domain model*. A domain model includes:

- Classes and objects representing the main concepts of a domain, the so-called *entities*. These elements can also be *value objects*, to be precise, but compared to entities, value objects don't have a persistent identity. In an online shop, the main elements would be "Customer," "Order," "Product," "Cart," and so forth.
- Associations between domain-specific classes and objects. In our online shop example, an `Order` would have at least one `Customer` that it references as well as one or more references to the `Product(s)` ordered.
- Domain-specific functions implemented as a part of an entity. In our online shop, a `Cart` could have a `calculateTotalPrice()` method to calculate the final price based on the items in the `Cart` and their quantities.

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_2

- Functions that span multiple entities usually implemented in so-called *services*, simply because they cannot clearly be assigned to one single entity. In our online shop, the "Checkout" service may take care of lowering the inventory, invoicing, updating the order history, etc. The service deals with several entities at once.
- Domain-specific objects used instead of generic data containers such as PHP arrays whenever possible (exceptions prove the rule here).
- Business logic (such as business rules) implemented within the domain objects of an application whenever possible, not in controllers, for example (controllers are used in MVC–based frameworks to handle user input).

The main advantage of the domain model lies in having the domain-specific code centrally defined in classes and objects, an approach which facilitates maintenance and changes. The possibility of incidentally breaking a function, when changing or extending code, drops. By isolating domain-specific code from general technical code, portability is supported. That's helpful, for example, when migrating from one application framework to another.

Besides the advantages mentioned above, the domain model also supports teamwork. Often, when building and launching a new software product, programmers work together with business and marketing folks. A domain model can bridge the mental gap between business and IT by unifying the terminology used. This alone makes a domain model invaluable.

Demo Application

Concrete examples make things easier to understand. Throughout this book, a demo application called "Talking" will help to put theory into practice. `Talking` is a (simple) web application allowing users to publish content online. Figure 2-1 shows the application's domain model:

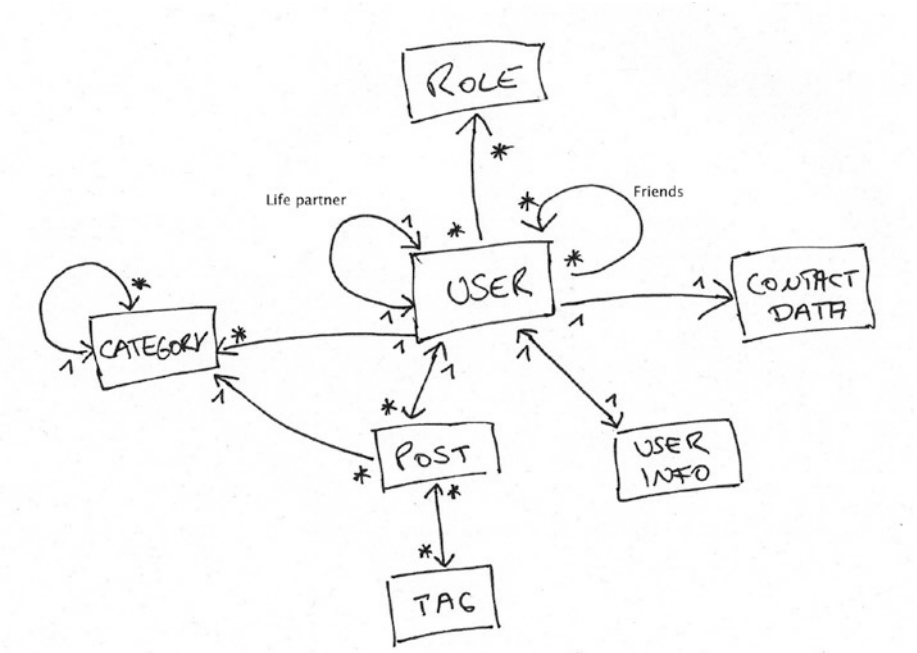

Figure 2-1. *Demo application "Talking" - Domain Model*

In the `Talking` demo application, a `User` can write `Posts`. A `Post` always has only one author (the `User`), and the two entities reference each other. A `User` can act in one or more `Roles`. A `User` references its `Roles`, but from a given `Role`, one cannot access the `Users` who reference the Role. A `User` references `UserInfo` holding the date of registration and the date of de-registration, if available. `UserInfo` references back to a `User`. A `User` references `ContactData` where the `User's` email and phone number are saved. `ContactData` does not reference back to its `User`. A `User` may reference another `User` as its life partner. The `User`'s life partner references back if known. A `User` may have an unlimited number of friends. Given a `User`, one can identify its friends, but there is no reference back. The `Post` of a `User` can have an unlimited number of `Tags`. A `Tag` can be reused in several `Posts`. There is a bidirectional association between a `Post` and its `Tags`. A `Post` references its `Category`, however, there is no reference from a `Category` to its `Posts`. A `Category` can have subcategories which it references, as well as a parent `Category`, if given. Categories are user-specific. A `User` references its categories, but there is no reference back.

Our demo domain model is designed to use as many different association types as possible, so that we can see most of the features of Doctrine 2 in action in the scope of the demo application. In a real application, the domain model would probably look a bit different. As you can see, not every association between entities is bidirectional. And as we will see later in this book, this is an essential feature of an object-relational mapping (ORM) system such as Doctrine 2 living in the object-oriented world. In relational databases, there are no unidirectional associations; they are always bidirectional by design.