Practical PHP Patterns: Data Transfer Object

Information can travel in very different ways between the parts of an applications, which could be tricky when these parts are distributed on different tiers, or times. The Data Transfer Object pattern prescribes to use a first-class citizen like a class to model the data used in the communication.

This pattern was originally meant to aid logic distribution: coarse grained interfaces, which are ideal for remote interaction, must return more data in each call to reduce the number of messages sent over the network and their overhead. This kind of objects is built with the goal of be easy to serialize and send over the wire.

In the PHP world, the communication is usually either from server to server, or even targets the same server: PHP is a lot different from other languages which are used for application distribution. Commonly the serialization and unserialization of objects are accomplished by the same codebase, which has a short life (the time of an HTTP request) and stores data with the serialization mechanism to maintain state between different requests. As a result, the usual dependencies discourse, which prescribes to use a Data Mapper between the Domain Model and the DTOs, do not apply; to the point that sometimes even domain model objects are serialized directly (an ORM like Doctrine 2 allows you to do so). There are more differences with classic Java DTOs which we'll see in this article.

Implementation

The definition of Data Transfer Object talks about communication between different processes: subsequent executions of the same PHP script are indeed different processes (or, when the same process is reused, it does not share any variable space with the previous executions).
PHP is peculiar also from the implementation point of view: a DTO may not even be an object, since an ordinary or multidimensional array will do the same job in many cases.

However, an object implementation is necessary to take advantage of object handlers, where the structure of data is not hierarchical but involves a connected graph of objects or recursive data structures. This implementation choice is much more clear than using arrays and variable references, which can be tricky in PHP.
Basically, a DTO is the object which we have been told to never write: it has getters and setters to expose allof its properties, while providing nearly no encapsulation nor business logic. It's a data structure, like a Value Object (which was its name in certain Java literature). But it is not immutable nor it carries the semantic meaning of Value Objects.

Use cases

Data Transfer Objects come handy in many use cases. In their simplest implementation, they are used for returning multiple values from a method.

Their further evolution involves then serialization mechanisms; of course both the provenience and the destination of a Data Transfer Object need to be a PHP environment, a fact which opens different scenarios from classical Java distribution ones, that involve clients well. For example a Data Transfer object, easily serializable, can be stored in caches (memcache) or sessions ($_SESSION).

Other use cases regard databases: according to Fowler, Record Sets like PDOStatement can be defined as the Data Transfer Objects of relational databases.
There are even more scenarios for Data Transfer Object, like clients which return a serialized data structure, or their usage for breaking dependencies between different layers: a Controller may populate a DTO and pass it to the View.

Serialization

As you may have experience while using var_dump() in PHP, the objects of a Domain Model are usually interconnected in a complex graph; DTOs isolate a subset of the graph and make it easy to serialize, even by omitting part of the information.

Thus, serialization of domain object needs some complex infrastructure (lazy loading proxies which are discarded on serialization and must be re-initialized during the merge with a current object graph); sometimes is by far easier to extract data in a DTO, especially when you do not have particular libraries at your disposal.

Once you have an isolated object graph, PHP will handle serialization by itself, even without marker interfaces; it will simply include all the public, protected and private properties (this behavior can be modified by specifying a __sleep() method).