Software development : A personal experience using PHP

David Ramalho <dramalho@blackorange.pt>

Abstract

The purpose of this project is to explain some methods used to organize and plan the development of a software program. The methods shown arise from certain needs a moderately complex project has, mainly those of file organization, coding clearness, logical structure and so forth. None of the techniques mentioned are to be taken as rules of thumb but more as guidelines, built to provide the programmer a system that helps maintain logic and readability to the project's code.

Table of Contents

Introduction

Programming, like so much in life, is mostly about managing order while creating chaos, making sense of the complexity or to put it simply, connecting the dots. Engineering teaches us to divide and conquer, by encouraging modularity, component reuse and logical structure, and in software development this is particularly useful and programming techniques like OOP (Object oriented programming) were developed to address exactly that, by offering ways to create small, meaningful "black boxes" that perform very specific functions that can be put together to form greater pieces, or rather small systems which, when put to work together, work as a large system that, hopefully, will meet the project's requirements. Also, experience tells us that projects tend to change with time, that bugs creep in, that technology evolves, programming languages change, operating systems are upgraded, etc. , and all these "random" events will probably require changes to the code, and time degrades one's memory so we better hope for helpful information on where to find things.

From my personal experience I saw specific needs emerge, needs that somehow had to be addressed and that I'll try to lay out and explain, not wanting to pass them as rules, but merely as ideas that may be useful to some and, with some luck, as a topic to be discussed with others more or less experienced. As with everything systematic we, as individuals, need to be tamed and disciplined, and sometimes the amount of work makes it very hard to follow all the little rules we set up for ourselves, and although I consider this somewhat normal it must be avoided at all cost because later, it will just be harder to reconcile things.

In the next chapters will try to address coding standards, code comments that are actually useful and finally the way a Web application, although we can use the same strategy to other types of applications, can be built and how our file structure will reflect that strategy.

Programming with PHP

PHP, a recursive acronym for "PHP: Hypertext Preprocessor", is an Open Source general-purpose scripting language that has enjoyed a wide spread and that is actively improved. It's been around since 1995 when it was known as PHP/FI and was more Perl than the PHP we now have, and has evolved since then. It's current version, PHP 5, has seen it's object model greatly improved to allow for features known in other languages (interface, abstract, final, etc.), and continues to improve the language that was awarded "Programming Language of 2004" by TIOBE's Programming Community Index [TIOBE].

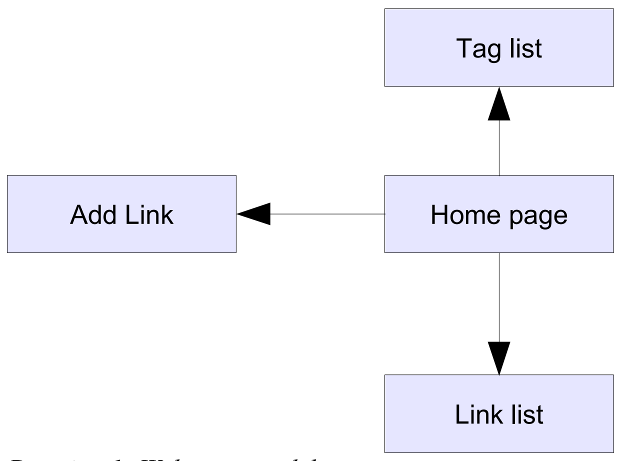
I have used PHP for some time now on a few professional and personal projects, most of them web based, but have also used it to build scripts that perform many helpful functions. For those familiar with more strict languages like C, C++ or JAVA, just to mention a few, you may find some of PHP's features very different, like the fact that variables don't need to be declared and don't have a particular type, but the language is conventional enough for people with experience to learn it quickly, and powerful enough to consider using instead of the aforementioned languages.

Example project

For the purposes of this project, I'll work on an example project that, being relatively simple, tries to encompass enough situations to provide a useful model. We'll design a bookmarking system, much like [DEL.ICIO.US] that uses tags to create a flat hierarchy. We'll use a simple data model and resume a few details of the implementation for clarity and for simplicity.

Page model

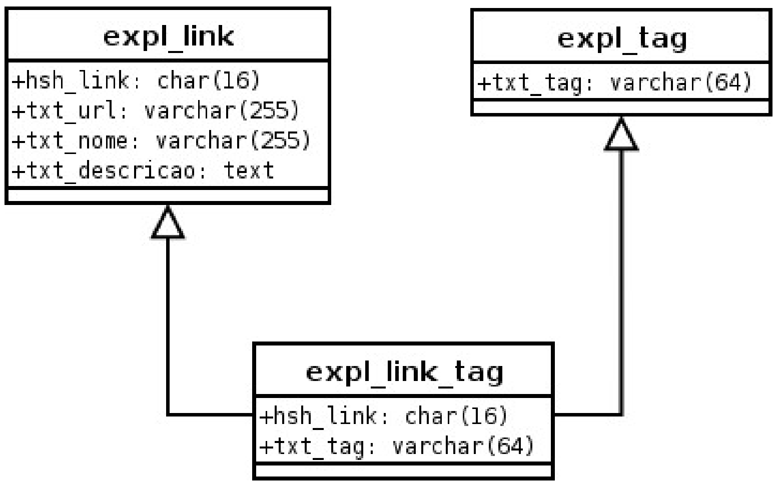
Our project will use four pages to manage and display our bookmarks.



Drawing 1: Web page model

- Tag list : This page shows all the tags we have so far.
- Add link : this is the page we use to add a link to our list.
- Home page : The main page, where we see our latest bookmarks.
- Link list : this page is where we use to see our bookmarks.

Data model



Coding standards

Whether you're working alone or with a large team, it's nice to be able to look at code from time to time and have your eyes surf through the textual structure with ease, finding small blocks of code quickly, seeing exactly what goes inside a loop, know that function getName() will, in fact get a Name instead of an address and so forth. We train the eye and the brain for this sort of mechanical work by looking at different examples of code that looks similar in structure, by looking at a variable name and being able to tell it's scope, type and purpose, by figuring out blocks by it's indentation, in essence, by finding familiar patterns. But we all know that humans are not entirely prone to following rules, especially if they're not even defined, and we also know humans are, for the most part, creative creatures that take pride in doing things in many different ways, even if only so slightly, and so it's very important, from time to time, to assign rules to certain areas, and computer code is definitely one of them.

But what exactly should we standardize? Well, quite a lot actually, from the way you name your variables to the place where you position your block opening character, but let's look at some examples of items we should apply standards to.

Indentation

Indenting your code means giving visual definition to your otherwise defined blocks of code. For example, when you define an `if` clause you write it's code a bit to the right to show that those commands depend on the outcome of the `if condition`'s result. This is one of those guidelines that's very very close to being a rule.

Example :

```
If ( 1 == 1 ) {
doThis();
doThat();
} else {
doNothing();
}
```

is more confusing to read than

```
If ( 1 == 1 ) {
   doThis();
   doThat();
} else {
   doNothing();
}
```

Variable naming

The purpose of including the variable naming in the standard is to motivate programmers to put as much information about the variable in it's name as possible, information like scope and/or type can easily be inserted in the variable name by applying a prefix to the variable name in the following way : `prefix + variable type + variable name`, where the prefix might be : 'l' for local variable, 'g' for global variables, 'p' for function parameters, and so on, and the variable type might be the short name for each type, like int or char. Another point worth mentioning about naming conventions is the use of case and/or characters to join the words that make the variable's name. I for one prefer the use of mixed case with the first letter in lower-case (for variables) but there are many options, all you have to do is choose one and stick to it.

As for the name of the variable itself, while you can't standardize it, you should stress out that it should be as explicit as it can about the content of the variable, avoid using numbers and try to keep the name as short as possible without using acronyms. Finally, the noble exception for all these rules usually resides on loop variables, that tradition states should be $i, $j, $k and so on, and the great argument we have for this is, their scope is very very small and usually they're numbers, also, everyone knows $i, so it's basically a standard that goes without saying.

Example :

```
$lIntAge;
$pStrName;
```

```
$gStrAddress;
$l_int_age;
$p_str_name;
$g_str_address;

$i;
$j;
$k;
```

Function naming

Functions are pieces of code that actually perform actions, so the name should always be a verb that show what the function's purpose is. Case wise, I prefer to use mixed case with the first letter in lower-case, but other examples exist, just remember to choose one way and apply it all-round .

Example:

```
getName();
setName();
sumAge();
goAddress();
shutdownComputer();
launch_nuclear_devices();
cross_road();
```

Class naming

Classes represent entities, so nouns are preferred here. The rest you already know, except that I prefer to use mixed case with the first letter upper-cased.

Example :

```
class Link;
class LinkManager;
class Tag;
class TagDAO;
```

File naming

This greatly depends on the programming language, but as an example, and being coherent with the language we've been using so far, I'll talk about PHP a bit more. PHP files, should end in .php to help the operating system or web server understand what interpreter to call, but we should standardize file naming to help understand what sort of code the file contains. Although Object Oriented Programming should be used in 99% of cases, you sometimes need to write a small script or program and just stick to Procedural Programming, but even so, you should get used to put each class in it's own file, every function in it's own file or at least group functions similar in context in one file, and use the correct naming standard to identify each one.

If you're writing a class in one file, it should be named after the class name followed by a postfix indicating the file contains a class, for example, if you had class Link, the file could be named `link.class.php` . Same for functions, except you should use *function.php* instead of *class.php*, for instance, `math.functions.php` .

Control Structures

As with everything else so far, we're aiming at readability so here are a few guidelines I use :

- Leave space between the parenthesis and braces, it's easier to read

    -
      ```
      if ( $lAge < 25 ) { }
      if($lAge<25){ }
      ```
- Always use braces regardless of the number of statements it has.
- Always indent the statements inside the control structures
- Separate non nested control structures using line(s)
- Chose a brace style (I actually prefer the first)

    -
      ```
      if ( $lAge < 25 ) {
          holdAgainstHim();
      }
      ```

    -
      ```
      if($lAge<25)
      {
          holdAgainstHim();
      }
      ```

Comments

Comments are one of those things everyone believes in, it's the actual writing that's harder, but as with all things, practice makes you write better comments without worrying too much and without losing too much time with them. You should comment as much as possible without writing a novel for each function, write often, write line by line when coding an algorithm, explain what the variables are, what the function does, what the class represents and what the file contains, later, when you're going through the code to implement that new exciting change to the function, you'll be thankful you took the time.

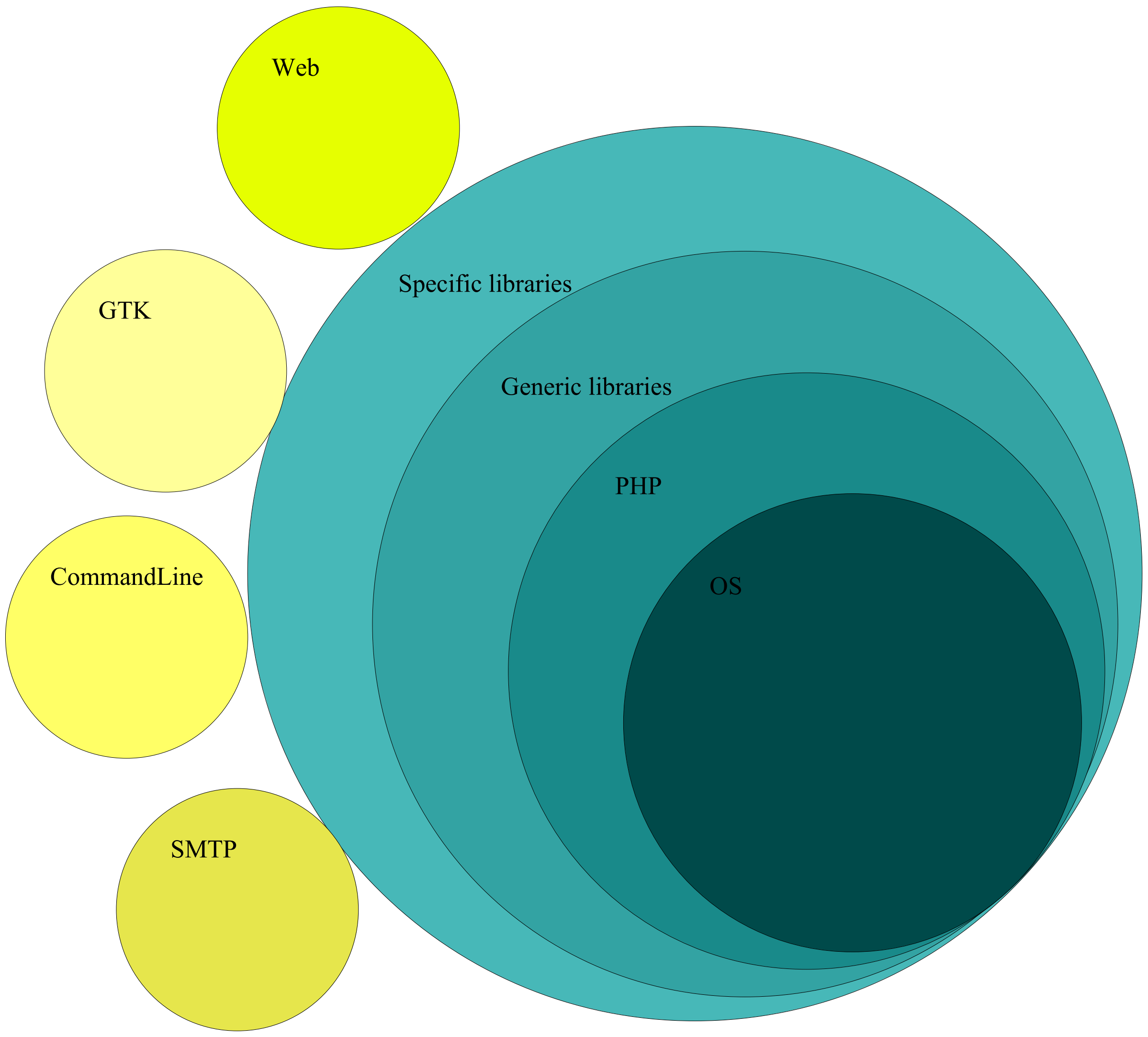
One way to make comments really useful is to adhere to JavaDoc and similar standards like [PHPDOC]. These specifications provide a few rules to write up comments that can be parsed by special software to automatically build documentation for you, documentation that express your classes, it's members and methods, small descriptions of how they work, and so on. These are priceless applications that take some time getting used to, but that are well worth the trouble.

Separation of logic and platform, and then some more

What is there to separate?

A web application, like our example project SimpleBookmarker is meant to be accessed from the browser, giving us access to specific pages where we can see, edit and create bookmarks, but these pages are merely an interface to the data (and logic) that represent our true system, and, as the project evolves, other interfaces might start to make sense, think SMS, SMTP (e-mail), Web Services (SOAP), a desktop version using [PHP-GTK] and so on. Ideas evolve, small projects start to grow and extend functionalities beyond their initial scopes, so it actually makes little sense to build something with one type of interface (or platform) in mind, and separating data, logic and presentation is one way to help projects have a more flexible future, or at least, spear us some work down the road.

The blue print



Drawing 2: Build in layers

The layers :

- OS – This represents the Operating System we're working on. [PHP] works on top of various operating systems and some of it's libraries are, sadly, OS specific, so you might want to be careful and check the documentation to avoid using any OS specific functions to avoid having problems in the future.

- PHP – This represents the PHP interpreter and all it's libraries, always remember that everything evolves, and programming languages do just that, so keep an open mind about the future and try to keep your code up to date with the language.
- Generic Libraries – This layer represents one very important asset, your previous works. Sometime you develop code to handle generic situations, like Database Abstraction, Configuration classes, and so on. This code can, and should, be used in multiple projects, for it will save you time and bring familiarity of processes between different projects.
- Specific Libraries – This layer represents the specific code you write for each project, the classes and functions that make sense solely on a specific project. This is a very important layer because this is where all your program's logic should be, this layer will enable the construction of simple interfaces that need only know how to push the right buttons. Project wise this, is the most important and time consuming layer on our structure. Also, take your time to identify specific classes that might descend to the "Generic Layer", sometimes we write code that's very generic and can perform similar tasks in different projects, so look carefully, you have only to gain by moving libraries to the previous layer.
- Interfaces – Finally, the layer that shows your program to the world, the only layer common users will ever know, and the one that represents all that's good or bad about your software. Your interfaces can only be as good as your program logic is, and that is on another layer, so be careful and don't overrate the time you spend on designing interfaces, the best software is the one that works the best, at least most of the times.

Conclusion

This project's intention is to talk a little bit about structure, standards, the importance of organization when designing software. I tried to share some ideas that might help software construction, but by any means, are these ideas final or even the best for you, each person has his style and each person has it's own personal view of organization, and the best this text can do is to try to get you to think about these sort of things and hopefully, be of any use to someone.

Apendix A : references

[PHP] – http://www.php.net

[PHP-GTK] – http://gtk.php.net

[TIOBE] – http://www.tiobe.com/tpci.htm

[PHPDOC] – http://phpdoc.org/index.php

[DEL.ICIO.US] - http://del.icio.us/