Johan Persson (johanp@aditus.nu)
Aditus Consulting,
S−128 46 Emågatan 16, Stockholm
Sweden

1 JpGraph and DDDA

This paper is divided into two parts, in the first part we discuss the JpGraph PHP library and in the second part the DDDA, Database Driven Documentation Architecture. For more detailed information see http://www.aditus.nu/jpgraph/

JpGraph is a graph drawing library for PHP (4.0.2 and higher) and is implemented in fully OO−PHP. The library supports a large variety of plots to cater for both business graphics as well as scientific plots (both linear and logarithmic plots are supported). In addition to all standard plot types (line, bar, pie, 3D−pie, error, radar, scatter etc.) JpGraph also supports the generation of Gantt−charts and supports the use of CSIM (Client Side Image Maps) for defining hotspots within the graphs. JpGraph has also a built in cache system to reduce load of the server, anti−aliasing of lines and full support for both bit mapped and TTF fonts.

An important feature of JpGraph is the low learning curve for what is a rather complex library. This is achieved while still allowing the developer access to a fine grained control of all aspects of the graphs when needed. The complexity of this library is managed through a carefully crafted orthogonal OO design as well as context sensitive default values. Features will be available when needed and only then.

The second part; the DDDA, refers both to the architecture and application. This is a generic system for producing class and method documentation for OO PHP script code. The distinguishing feature of DDDA, as compared to other documentation system (e.g. PHPDoc, Doxygen etc), is the fact that no user level comments are stored in the PHP script file, instead all code information (signatures, properties and class hierarchies) is extracted into a database and then augmented with suitable documentation and examples. The architecture then allows for the generation of statistics and documentation (in various formats) directly from the database. This makes it possible to objectively track the progress and status of the documentation. The printing/output sub−system is designed around a plugin framework which allows for the generation of various document formats by simply choosing a appropriate formatter plugin. By default DDDA 1.0 is shipped with a HTML formatter plugin.

1.1 Software License

JpGraph is released under a dual license. QPL 1.0 for non commercial and educational use and JpGraph Professional License for commercial use. DDDA is released under QPL 1.0.

2 JpGraph

In this section we will give a high level overview of the architecture and design of JpGraph. Due to the space limitations it is impossible to do more than just touch upon all features of this library. For full details the reader is referred to the JpGraph documentation and class reference available from http://www.aditus.nu/jpgraph/

2.1 Introductory example

We would like to start by showing how easy it is to generate a dynamic graph with PHP and JpGraph by the means of a simple example. We will on purpose keep this example very simple and not discuss all details in this paper..

The first principle to remember is technically simple but often conceptually confusing. A JpGraph PHP script *is an image*. From the browser perspective it will just see an *image stream* returned back when the image script is called. Hence each graph must be contained in it's own script and can **not** return any ordinary HTML[1]. To include the image in a HTML page the standard IMG tag is used, e.g.

```
<IMG SRC="myimage.php" BORDER=0>
```

2.2 The example graph

To be able to generate any graphs we must first include the JpGraph library files so that the necessary classes are accessible in our script.

```
01: include 'jpgraph.php';
02: include 'jpgraph_line.php';
```

The first line includes the core library which is needed for all graphs. The second line includes the *line−plot* extension. Each plot type is implemented in it's own extension library as one step to manage the complexity of the library.

```
03: $dataArray = array(12,7,23,1,9);
```

The next line (#03) is the data array to be plotted.

```
04: $graph = new Graph(300,200);
05: $graph−>SetScale('textlin');
```

The lines above creates the graph to hold the plot. On line #04 we create a new graph with size 300x200 (W x H) pixels and on line #05 we specify that we want to use

[1]Failure to adhere to this rule will result in a browser "Headers already sent" error since each HTTP stream can only have one MIME type.

a text scale for the X−axis and a linear scale for the Y−axis. W can now create the line plot and add it to this graph by using the lines

```
06: $lineplot = new LinePlot($dataArray);
07: $graph->Add($lineplot);
```

In line #06 w create a new line plot based on the data in the array and on the follow−ing line (#07) we add this plot to our graph. The only thing that now remains is to send the generated image back to the browser.

```
08: $graph->Stroke();
```

This final line sends back *an image* to the browser. We have thus managed to create a simple graph in only 6 lines (excluding include directives) of code by relying on the context sensitive default values in the library. For production use we probably want to decorate the graph a bit more. Perhaps by adding a graph title, add legends, make the colors a little bit more exciting, change the fonts and so on. All these modifications (and many more) is straightforward but will of course increase the size of the script.

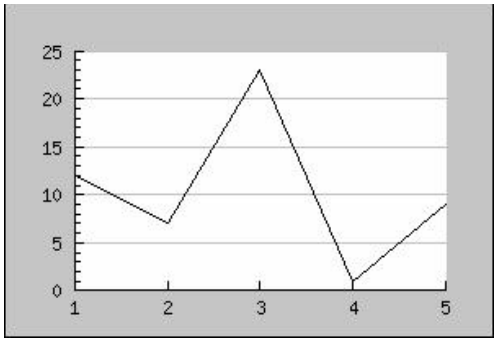


Figure 1 : The simplest possible graph is generated by 6 lines of code, (this first example has on purpose been kept simple).

The same basic structure is used to create all graphs. For example, if we instead had wanted to display the same data with a bar plot instead we would just have had to change line #06 and #07 to

```
06: $barplot = new BarPlot($dataArray);
07: $graph->Add($barplot);
```

We conclude this introductory example by giving an example on how we can enhance the graph in figure 1 to give it a more attractive design. For reasons of space we don't show the complete code for this graph (source is available at http://www.aditus.nu/phpconf2002/fig2.phps).

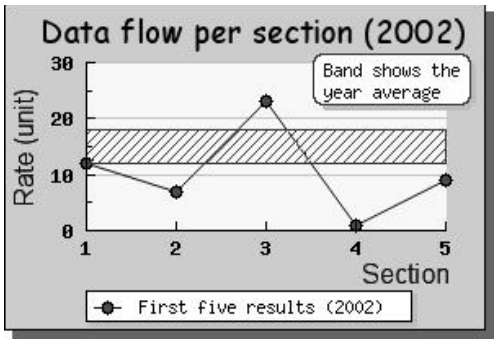


Figure 2 : Enhancing our example to make it more attractive and more informative

2.3 JpGraph Architecture and design

JpGraph is based on the low level image functions available in PHP through the GD−library which is used to provide the basic pixel−on/off functionality as well as the interface for the image encoding. Depending on what support libraries are available JpGraph supports both JPEG, PNG and GIF encoding of images. JpGraph works with both GD 1.8.x and GD 2.x. A high level view of the architecture is shown in figure 3.

One of the cornerstones in the design of this library was to make it an extendible framework rather than a fixed functionality library. Hence the library is divided into a core engine (in *jpgraph.php*) and plot extensions. The core engine provides all the necessary infrastructure for graph drawing and the extensions provides the actual logic to produce the specific plot. These extensions are logically named after the plot type they provide such as *jpgraph_pie.php* (used to draw 2D pie plots) or *jpgraph_radar.php* (used to produce radar plots). Version 1.8 provides 11 such core extensions. This modularity also helps in minimizing the amount of code that needs to be included in a PHP script which uses JpGraph. This design has proven itself in practice. In version 1.0 of JpGraph (released Feb 2001) only 2 basic extensions were provided. All other extension has been added with a bare minimum of changes to the core architecture in later versions.

The other main goal of the design was to provide a complete and clean, as far as PHP allows, OO design and implementation of the library[2]. By keeping the API consistent (and orthogonal) between the different graphical objects in the graph it is often only necessary to learn one method which behaves in the same way on all other objects where the action/operation makes sense. A prime example of this is the *SetFont()* method which is used in exactly the same way in all objects which uses texts.

[2]During the implementation of JpGraph several "discoveries" about specific difficulties in implementing a non−trivial PHP OO script was made. Some of the lessons learnt can be found at http://www.aditus.nu/jpgraph/jpg_phpoo.php

When it comes to designing an OO library there is always a balance between a pure OO design with no redundancy and practical use of the library. In JpGraph many objects have additional properties which in turn has properties and so on. This could yield quite uncomfortable long method calls if no non−pure shortcuts is introduced. For example, the pure way of accessing one specific method in JpGraph would be

```
$graph->xaxis->scale->ticks->Set(....);
```

Although the hierarchy is quite obvious we have in this (and similar cases) allowed short forms to be introduced. In this particular case the short form would be

```
$graph->xaxis->SetTicks(...)
```

Attention has been focused on making the commonly used methods more easily accessible.

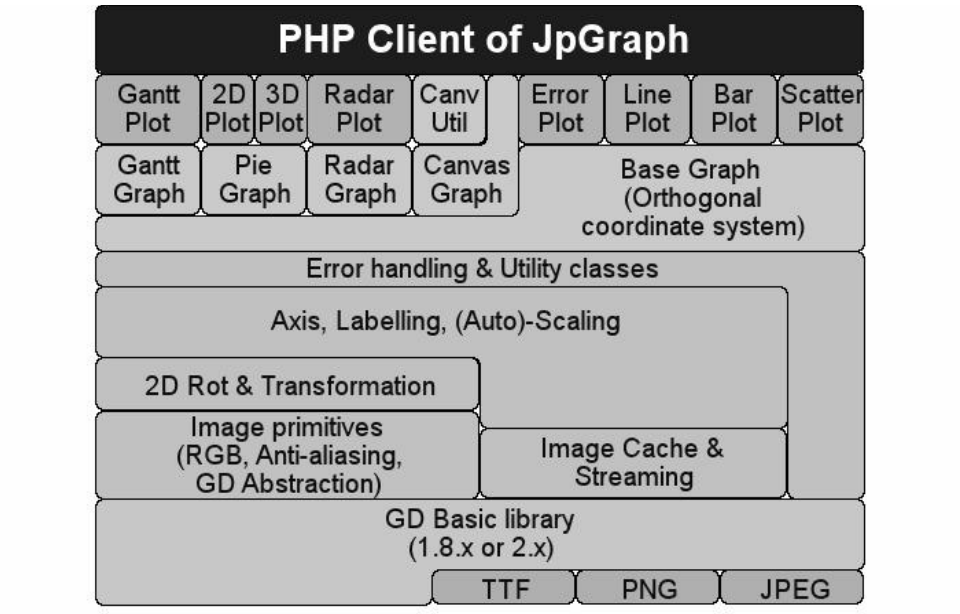


Figure 3 : Overview of the JpGraph Architecture (Image itself made with JpGraph).

2.4 JpGraph performance

Being completely written in PHP script the performance can't rival native C (or C++ libraries). A typical medium complex line graph takes in the order of 40−50ms to generate on a 800 MHz PIII server. In the case where caching can be used and no actual image construction needs to take place the overhead compared to just including the image directly in a HTML <IMG> tag is often less than 10ms. In the case where the

PHP installation also makes use of some of the available PHP caching system (not to be confused with the caching system used in JpGraph) a substantial speedup will be achieved since this will avoid the byte compilation of the core graphic engine for each image script.

Switching from GD 1.x to GD 2.x will also increase performance since JpGraph can take advantage of the additional functionality available and can replace some of the internal methods in JpGraph by the equivalent methods in GD 2.x. There is also the usual trade off between quality and speed. For example, enabling the built in anti−aliasing in JpGraph will increase the CPU usage.

2.5 JpGraph image and color quality

The quality of the images generated with JpGraph is dependent upon the GD library. To enhance image quality you can enable an internal anti−aliasing algorithm which is applied to all line drawing in the graph. This will have some performance impact but will greatly increase the fidelity of lines. Due to performance considerations JpGraph only applies anti−aliasing to single lines (not circles or arcs). For the next version of JpGraph a new anti−aliasing scheme is devised which will rely on an image resampling (post processing) method which will be able to enhance all aspects of the image.

In terms of color fidelity there is a limitation in GD 1.x since it only allows paletted images (not true color) This means that enabling anti−aliasing (which may use many colors) or using very colorful backgrounds (or gradient fillings) might exhaust the available palette. In this case JpGraph can be told to try to find the closest match among the available colors when a new color can't be allocated. Switching to GD 2.x will solve this problem since JpGraph is then able to make full use of the true color support in GD 2.x[3].

2.6 Planned major future enhancements

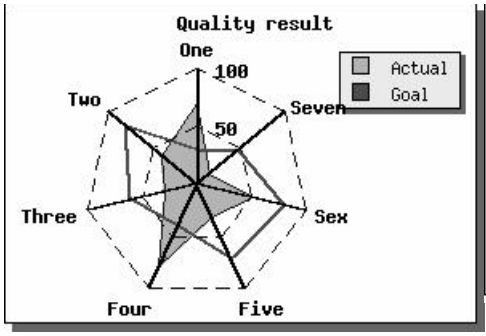
More information can be found at the JpGraph site.

- From version 1.9 (Jan 2003) a new anti−aliasing filter will be available.
- From version 2.0 (Summer 2003) JpGraph will require the new built−in GD library in PHP 4.3 as well as making it possible to have multiple graphs (not just plots) on the same image. Relying on the builtin library will make it possible to support, among other things, alpha−blending. This will result in much enhanced quality of the graphs as well as performance improvements.

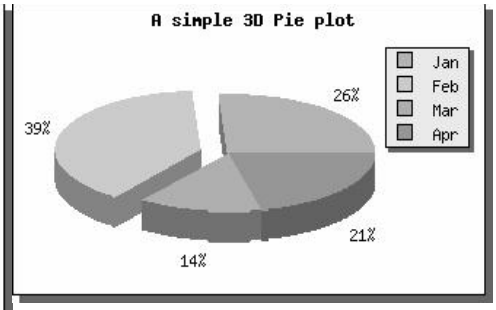
[3]JpGraph must be told in the configuration to use GD2.x and TrueColor images. Note that there are known issues with using TTF fonts in true−color images in GD 2.x

2.7 JpGraph samples

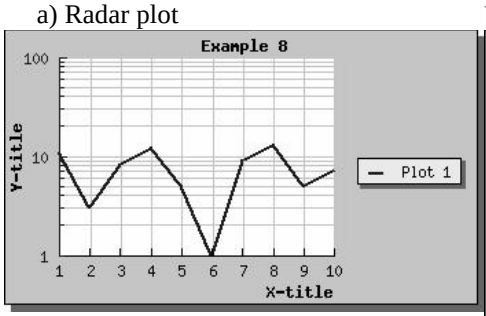
We conclude this introduction to JpGraph by giving a few number of example images[4] . For a complete image−gallery (with 200+ images) the reader is referred to the JpGraph site (http://www.aditus.nu/jpgraph/)



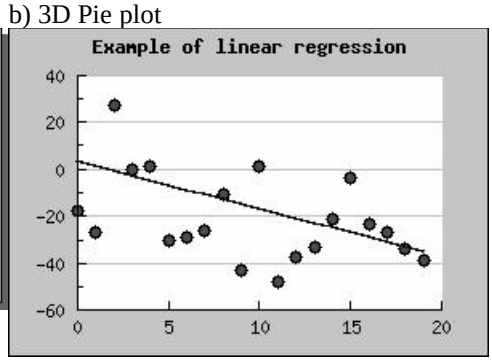
a) Radar plot



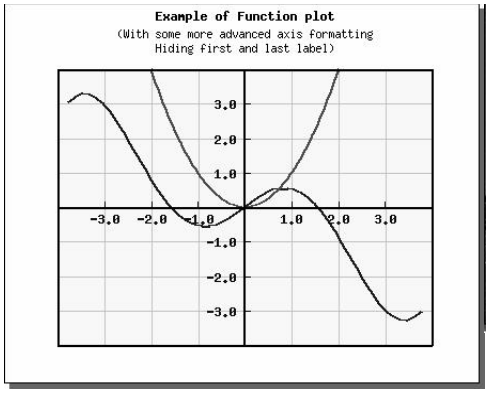
b) 3D Pie plot



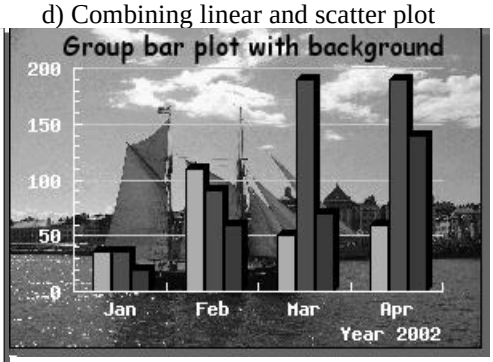
c) Text−logarithmic linear plot



d) Combining linear and scatter plot



e) Scientific style plot



f) Using a background image

Figure 4 : A few examples of images generated by JpGraph 1.8

[4]Due to reproduction limitations the images in these proceedings may have been scaled and therefore suffered quality loss.

3 DDDA – Database Driven Document Architecture

In this section we give a very short introduction to a new documentation system especially built to handle large OO PHP code. As an example of this architecture/application the JpGraph class reference may be studied. A longer and more in depth system description may be found at http://www.aditus.nu/jpgraph/

3.1 Rationale

During the development of JpGraph it became apparent that no tool existed that fulfilled the requirements to help documenting a complex PHP OO library. The existing documentation tools (e.g. PHPDoc) have some limitations that, in our mind, made them less than suitable for the task at hand. Some issues we see with these tools:

1. They use special markup tags to include all documentation in the source. No documentation can exist outside the script code. This has (at least) three major limitations:
   a) The source itself will contain both the system level documentation which is aimed at a maintainer of the code as well as user level documentation aimed at the final user of the library. This, in our mind, makes the code difficult to navigate in since it will significantly increase the script size. Especially the practice of including example code in the library can be very confounding for a maintainer.
   b) For an interpretating language this increase in file size will have a performance impact. It is our belief that code should contain system level documentation (technical class and method comments) which together with a suitable architecture overview helps in maintaining the code. This is often **not** documentation suitable or even helpful for an end user of the library.
   c) Adding documentation will in affect change the source and this will make it necessary to release new versions of the baseline even though no code changes has been made. It will also increase the risk that a document maintainer by mistake corrupts the source which can lead to time consuming code restores. This confusion between core development and documentation effort makes it more complicated to split the responsibility between these two tasks.
2. There is no simple way of achieving objective automatic tracking of the progress and status of the documentation to direct the work where it is most needed.
3. The existing PHP documentation tools does not handle class hierarchies very well.

Based on these observations it was decided that a new tool was needed. This new architecture tries to combine the best of both automatic and manual system documentation by first parsing and extracting all information about the classes from the project files and then storing all the information in a database. The developer is then

presented with a WEB−interface which lets him[5] augment the extracted information with parameter and method description as well as example code. To help the developer focus on less documented areas of the system the DDDA uses a weighted point system to determine the degree of documentation for both classes and methods. The end result of this process is an easy to understand percentage which indicates the level of documentation for a specific class or method. Whenever the code base is modified the tool is re−run and the statistics will clearly indicate where modifications have been made and additional documentation is required.

3.2 DDDA architecture and usage

Using the DDDA with an OO PHP application/library involves one initial setup and three phases repeated throughout the lifetime of the project. These phases are done using the WEB−based interface to the DDDA architecture.

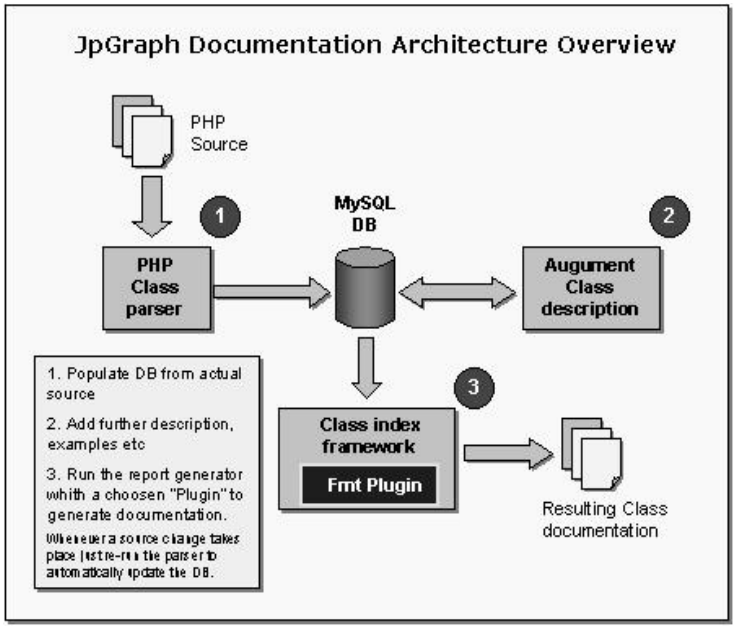


Figure 5 : Principle overview of the DDDA architecture

1. Project setup phase. Only needed once for each project. This involves specifying files that belong to the application/library as well as defining project parameters.
2. Parsing phase. Needs to be done whenever a file in the project has changed. The DDDA architecture keeps automatic track on which files need to be re−parsed to avoid parsing non changed files. This phase collects all information on available

[5]We use the masculine genus, as is practice, meaning both male and female developers.

classes, hierarchies, properties and methods and stores them in the DB[6]. In the case the class or method already exists in the DB a further comparison is made to check if any methods have changed and updates them appropriately.

3. Augmenting phase. This means using the DDDA WEB−based interface and adding comments, parameter description and examples to the methods and classes.
4. Document generating phase. Using a suitable format plugin to generate the class reference. By default a standard HTML formatter plugin is supplied in version 1.0 of the DDDA.

3.3 Example of the DDDA Web−interface

In order to make the architecture easy to work with and platform independent a basic WEB−based interface is provided, figure 6 illustrates the overview for a project. This is the main interface to select methods/classes to augment.

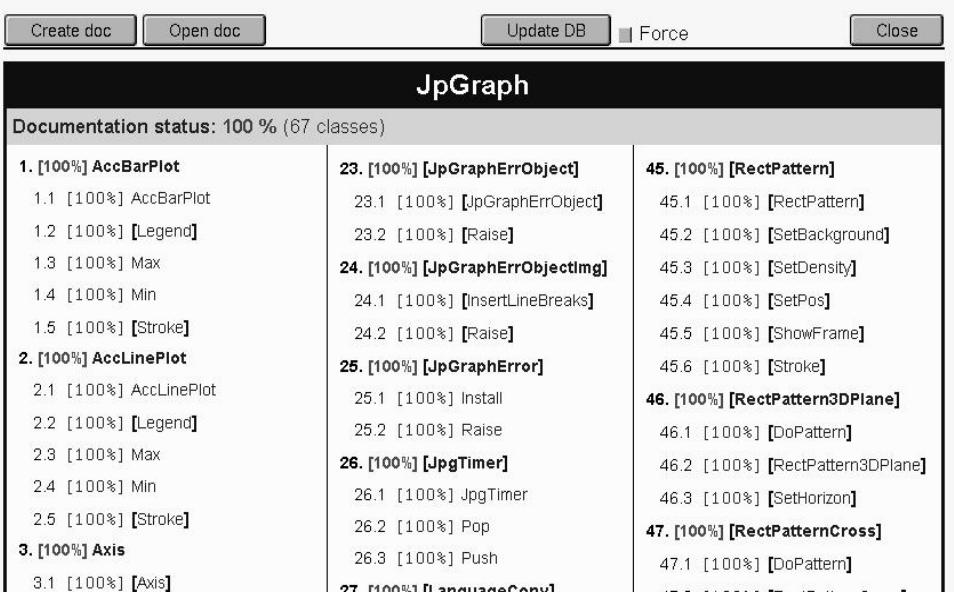


Figure 6 : The project overview view with Classes and Methods together with percentage figures illustrating the documentation status for each class and method.

About the author

Johan Persson, (johanp@aditus.nu) is the founder of Aditus consulting and has more than 12 years of experience in VLSP[7], system design, and software management. He is known to be fighting for realistic project plans and has been able to avoid "death−march" projects.

[6]DDDA currently uses a MySQL DB accessed through a DB abstraction layer.

[7]VLSP=Very Large Software Projects