Version: $Id: index.html,v 1.2 2002/09/17 21:52:45 aditus Exp $

PHP Database Driven Documentation Architecture (PHP DDDA)

Johan Persson (johanp@aditus.nu)
Aditus Software

Abstract

Here we present a new MySQL based documentation system for PHP. The proposed architecture, DDDA, combines the best features from automatic source code documentation system like doxygen with traditional hand crafted documentation. The benefit of this system is that it will clearly separate user level documentation from system level documentation and thereby keeping the code base clean. The system also allows easy generation of weighted documentation statistics which might be used to generate objective progess indication and help focus the documentation effort on underdocumented parts of the system.

1. Introduction and motivation

The first question anybody reading this will ask is probably:

"Why another documentation system when there are already automatic system in existence?"

To answer that question let's first think of who are the target of the class documentation. At one hand you have the user who wants user level description and examples. On the other hand maintainers of the class also needs a further level of internal system documentation which is of little use for the end user.

For the maintainer of the class the code itself should contain necessary code level comments together with suitable architectural overviews. For the end user the documentation should be provided entirely external to the code. Current automatic system insists on keeping all documentation in the source itself from where it later can be extracted to produce documentation.

The problem with these system is that the code gets cluttered with a lot of user level documentation which makes the code difficult to navigate in.

On the other hand an automatic system has the advantage that it can extract class information automatically (like class hierarchies and function prototypes) which is always tedious and error prone to copy&paste manually.

The proposed system combines the two methods by first automatically extract all class information storing it in a database and then let the developer augment that database with suitable user level description of classes and methods.

Now, whenever the code is modified the automatic extraction is just re-run and the class hierarchy gets automatically adjusted. Existing description and examples in the DB are preserved.

The documentation can then be automatically generated from that DB. By applying different formatting modules different types of documents can of course be generated.

An overview of the proposed system is shown in Figure 1.

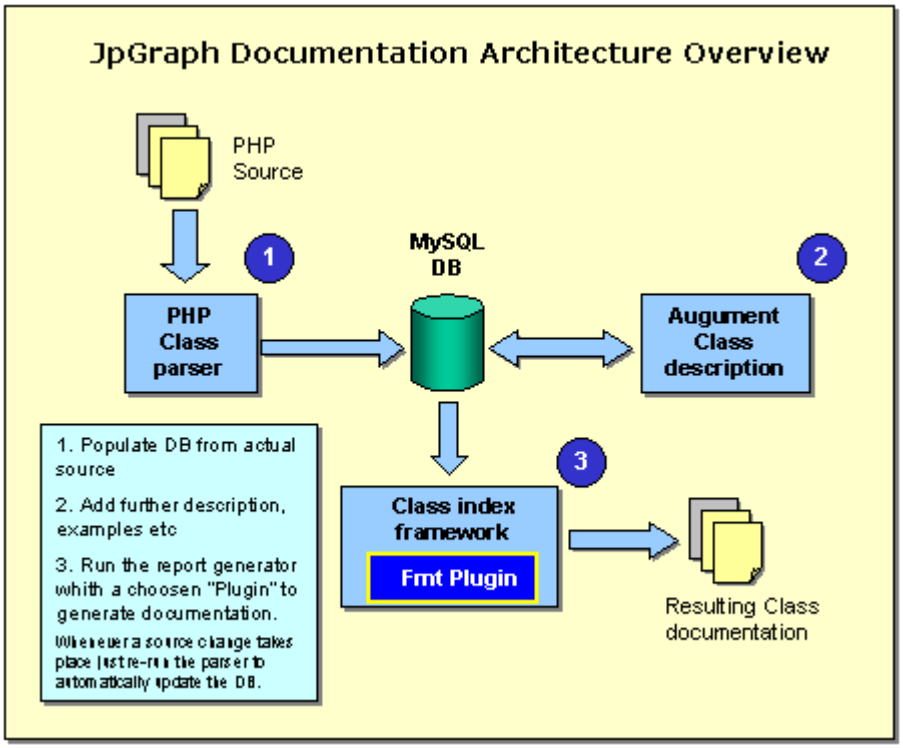

Fig 1. Schematic overview of the DDDA architecture.

An additional (almost accidental) feature that has proven useful in practice is the fact that having all the information in a database makes it easy to generate static's on the state of documentation. For example, we can easily see how many percent of the public APIs that are documented. This is very useful for tracking the progress of documentation and making sure that the effort spend documenting is directed to the most needing classes.

2. Overview PHP DDDA

In this section we will give an overview of each of the three different subsystems, the parser subsystem, the augmenting subsystem and the report generating subsystem. These three systems together makes up the whole of the DDDA architecture. Where appropriate we will also give some further details.

2.1 Parser subsystem

The heart of the system is the PHP parser implemented in class Parser. It scans a given source file and extract all classes, instance variables and functions declaration (including arguments). In addition to this information gathering it also performs some rudimentary static analysis of the code. Specifically it checks for unused instance variables and warns for the case where the programmer might have intended to access an instance variable but forgot to add '$this->'. (Which is a mistake that has caused this author some less than well-spend debugging time.)

Since the class Parser is a generic parser class it doesn't know anything about databases so the actual work is done by the subclass DBParser which extends Parser and provides appropriate factory functions to handle the interface to the DB. The way this works is that the Parser class provides "virtual" functions which can easily be overridden by the appropriate sub class. By "plugging" in to the Parser framework it is very easy to adapt it to specific needs in a very clean way without having to be intimate familiar with the inner workings of the Parser.

All database access is done through a DB server object which is responsible for all aspects of DB access. Class DBServer hence which provides functionality to access and query a MySQL DB in various ways. In addition to the server object each query is returned as an instance of class DBResult which is then used by a client to manipulate and extract information from the result of a query. The relationship between the

classes involved with the parsing aspect of the DDDA system is illustrated in Figure 2. below.

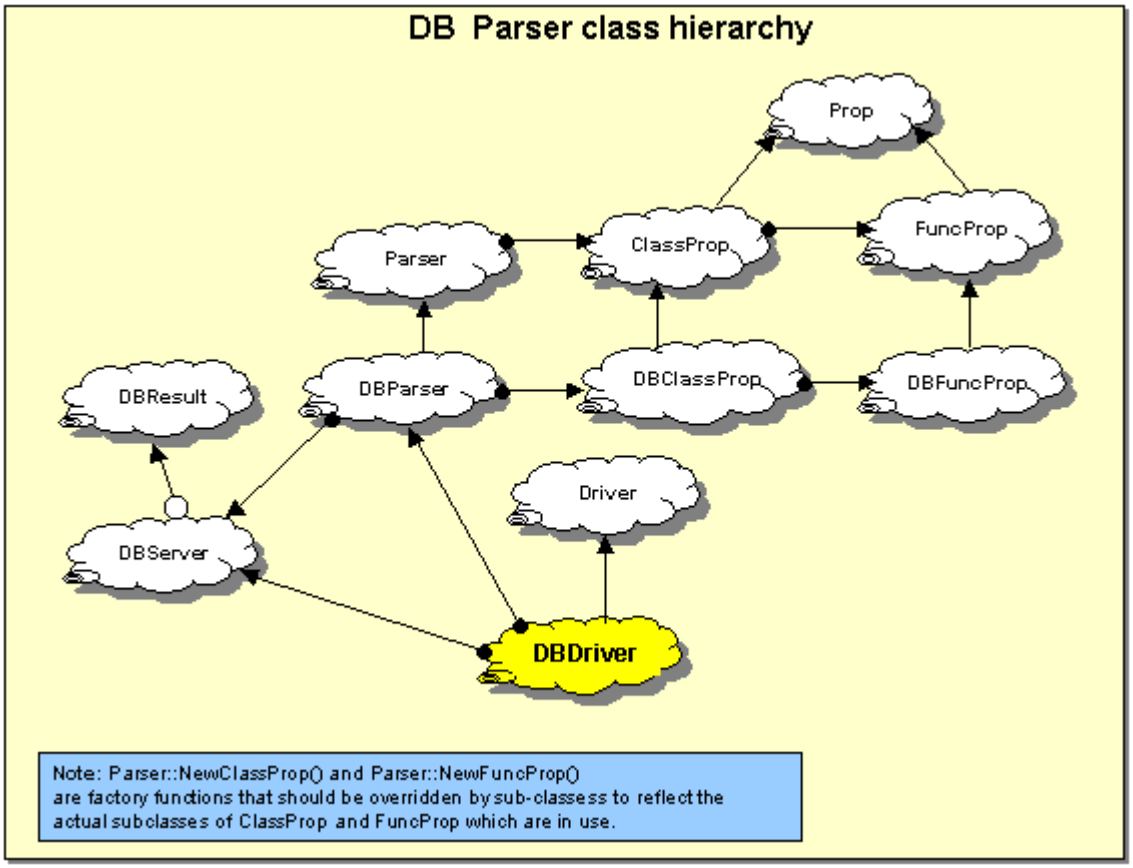

Fig 2. Class relationship for the parsing functionality of the DDDA architecture.

Since all this functionality is implemented in a PHP script the script needs an entry point. To handle this each script has it's own driver object. This object is responsible for initialization and extracting possible script variables (such as arguments in a URL specification). By convention all script execution is kicked off by the predefined Run() method in the driver.

2.2 Database augmenting subsystem

Once all class data has been gathered and stored in the DB it is time for the programmer to add his intelligent description of all the classes and methods as well as adding suitable examples. This is all handled by the Database augmenting subsystem.

This consists of a number of forms to modify/edit data for classes and methods as well as an index page to navigate and chose the class/method to work on.

The interface is completely WEB based and starts with an overview of all stored classes and methods in the BD from where you drill down to the class/method you want to edit. In version 1.0 of DDDA this interface is fairly basic and some more creative use of JavaScript and DHTML is probably not a bad thing. However, due to time constrains, DDDA 1.0 mainly focus on functionality rather than a lot of "fancy" WEB programming. This is also a thing that very easily can be added as a front end cosmetic "thingy" later on.

Once the programmer/documenter has chosen the class/method to edit he is presented with a standard WEB-form with all the static information about the class/method. He can then just add suitable description and examples.

The overall class hierarchy for the augment subsystem is shown in Figure 3. below.

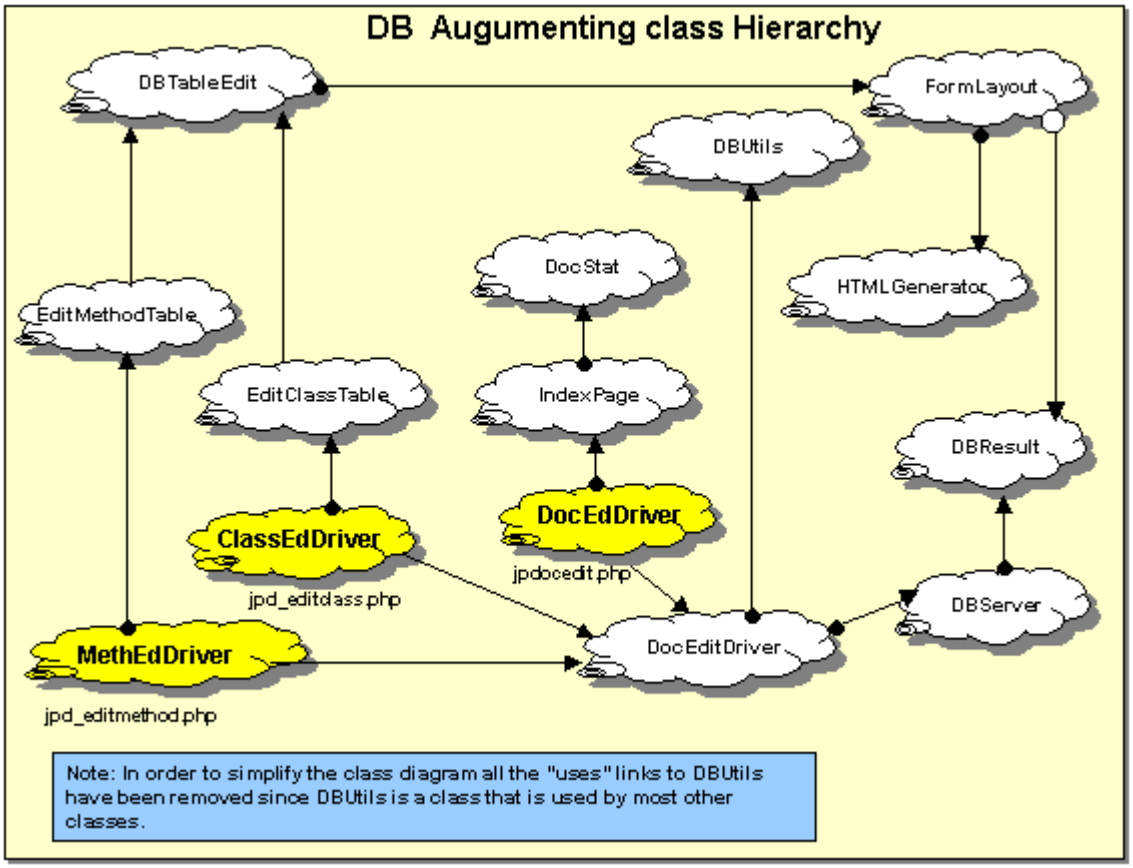

Fig 3. Class relationship for the class augmenting framework.

As was previously mentioned it was found that the ability to generate statistics is quite useful to direct attention to less than well documented areas. It has also proven as a very good way of actually keeping track of the documentation status in a very measurable well.

All statistical processing is centralized in the class DocStat. The static's is based on a point system where each class (and method) receives a certain number of points based on the level of documentation. The points system is weighted so that a class overview is considered much more important than a forgotten comment for a single method parameter. The actual points are than compared to the maximum achievable for that class (or method) to generated a percent figure.

2.2.1 The form layout engine

During developing of augmenting subsystem of the system it was realized that a generic system for generating form layout code together with the necessary DB queries based on the specified fields and DB table would be very useful and later on save time when modifying the forms. Hence the creation of a specific form layout class. This now form the basis of all form handling. A quick overview is given in this section.

The DB edit and layout enginge (class DBTableEdit and class FormLayout) automatically generates code to edit a specified table, row by row. It takes care of both generating the actual HTML as well as the necessary logic to handle the form submitting and DB handling. The layout of the form is specified by giving the specified row and column for each field that should be in the form together with the type for the field and some additional layout parameters. There are 8 compulsory and 2 optional parameters necessary to specify one input cell. The parameters are as follows

Parameter	Explanation
dbfield-name	The name of the database field that corresponds to this form field. Note that by convention the actual field in the DB has the name 'fld_' as prefix to this name.
row	Which row in the form this field should appear in.

col	Which column in the form this field should appear in.	
span	How many columns this field should span	
label-text	The text label to be shown close to this field in a way specified by the next parameter	
label-align	Specifies if the label should be shown above the field or to the left of the field.	
fld-align	Specifies if the field should be aligned to left or right in the cell	
fld-type	What type of field should this be. Possible types are:	
	FLDTYPE_TEXTINPUT	Normal textinput. Field arguments are field length and maxlength
	FLDTYPE_TEXTAREA	Textarea input. Field arguments are size in columns and rows
	FLDTYPE_DROPDOWN	A dropdown listbox. Fieldargument is first a vector of the possible values to display. The value retuned is the same as the displayed text.
	FLDTYPE_DROPDOWNCODE	Similair to FLDTYPE_DROPDOWN but with the difference that the supplied vector must both have a display value and code value for each entry. This means that the vealue stored in the DB can be different from the displayed value.
	FLDTYPE_STATICTEXT	Display non-edible static text
	FLDTYPE_STATICTEXTCODE	Similair to FLDTYPE_STATICTEXT but with the difference that the DB stores a code value which is translated to a displayed text by the vector passed as the first field argument.
	FLDTYPE_NONDBTEXT	A field that has nothing to do with the DB. Can display arbitrary text either as a static supplied value as the first fieldargument or in the call to display form.
	FLDTYPE_TIMESTAMP	Type is a MySQL timestamp which gets translated to a nice human readable string.
	FLDTYPE_RADIO	Display as a radiobuttob. The values will be the same as the displayed text and should be supplied as an array in the first fieldtype specific argument. To limit the number of radiobuttons per row a maximum number of columns can be given as the second argument.
	FLDTYPE_RADIOCODE	Same as FLDTYPE_RADIO but with the change that for each radiobutton the array should supply both a displaty value and code value.
	FLDTYPE_CHECK	A single check box. The value to be used can be specified as the first argument. If not explicitly set the value '1' will be used.
fld-arg1	The first fieldtype specific argument.	
fld-arg2	The second fieldtype specific argument.	

For reasons of brievity we don't give a full description of all the details on how to use this as a standalone system instead we just give an example. Study the following specification (which happens to be a slightly modified class editing form from the DDDA system)

```
 $formSpec = array(
  array('name',1,1,3,'',LBLPOS_LEFT,FLDPOS_LEFT,FLDTYPE_NONDBTEXT),
array('public',2,1,1,'',LBLPOS_LEFT,FLDPOS_LEFT,FLDTYPE_DROPDOWNCODE,$yn),
array('file',2,2,1,'File:',LBLPOS_LEFT,FLDPOS_LEFT,FLDTYPE_STATICTEXT),
array('linenbr',2,3,1,'#',LBLPOS_LEFT,FLDPOS_LEFT,FLDTYPE_STATICTEXT),
array('ref1',4,1,1,'Ref1:',LBLPOS_TOP,FLDPOS_LEFT,FLDTYPE_DROPDOWN,$cl),
```

```
array('ref2',4,2,1,'Ref2:',LBLPOS_TOP,FLDPOS_LEFT,FLDTYPE_DROPDOWN,$cl),
array('ref3',4,3,1,'Ref3:',LBLPOS_TOP,FLDPOS_LEFT,FLDTYPE_DROPDOWN,$cl),
array('ref4',4,4,1,'Ref4:',LBLPOS_TOP,FLDPOS_LEFT,FLDTYPE_DROPDOWN,$cl),
array('desc',3,1,4,'',LBLPOS_LEFT,FLDPOS_LEFT,FLDTYPE_TEXTAREA,70,5),
array('timestamp',1,4,1,'',LBLPOS_LEFT,FLDPOS_LEFT,FLDTYPE_TIMESTAMP));
```

when given as specification it will generate the form shown in Figure 4 below.

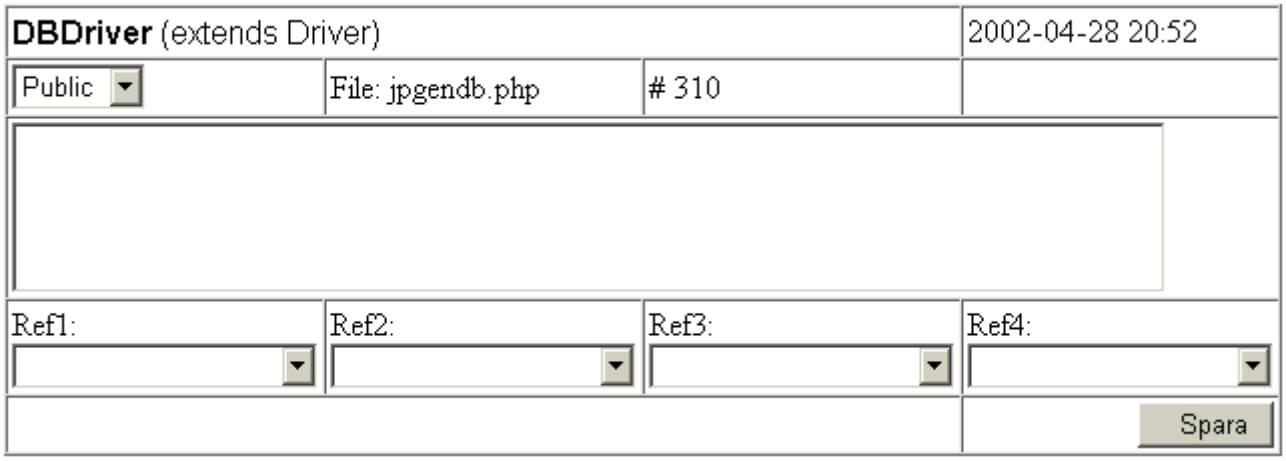

Fig 4. Example of form generated by the previous given specification ("Spara" means "Save" in Swedish)

2.3 Printing and formatting subsystem

Once all data is entered in the DB it is time to produce some documents. The extraction of class and method data is all handled by a framework which then uses a specific formatting plug in which is responsible for producing the actual output.

In version 1.0 of DDDA we supply a basic HTML formatter plug in which generates (surprise!) HTML in the form of a class reference.

In the future it is anticipated that 3:rd parties might supply PDF or RTF formatting plugins or even alternate HTML formatters.

The formatter plug in consist of a base class ClassFormatter which have a number of virtual methods which the actual plug in must implement in order to generate the output. These virtual methods will be automatically called by the framework with specific information according to the basic assumption of the layout of the code. These hooks will make it possible to generate very different layout should you not be happy with the supplied HTML formatter.

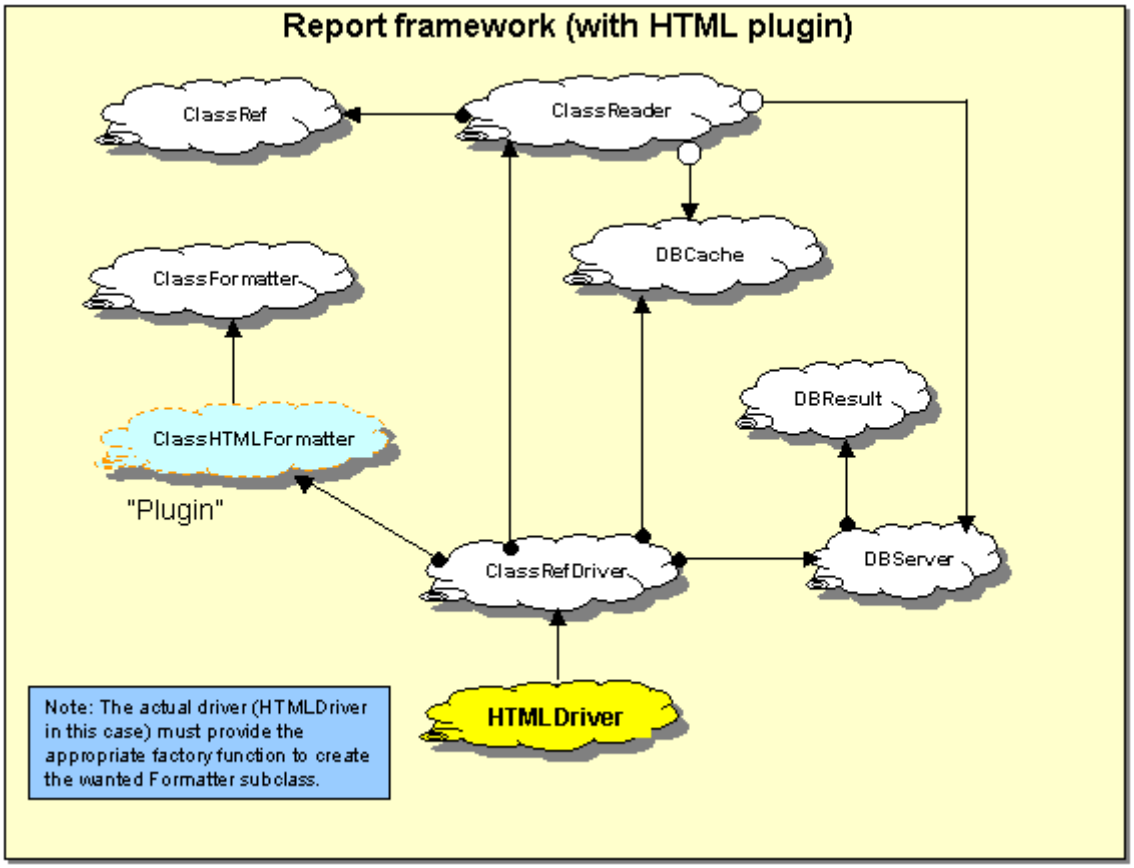

Fig 5. Class hierarchy for the report framework

Since the report generator is responsible to generate a lot of cross references which potentially could lead to unnecessary SQL queries and performance problems the report generator makes use of a DB Cache (class DBCache) which pre fetches the class and method information which dramatically increases performance.

2.3.1 Visualizing inheritance in classes

It is worth mentioning the way class hierarchy is visualized. For each class you will in the beginning of the class description see a list of all methods that are part of this class in the form of a short table. If the class inherits then this table will also show all the methods of the parent in another column. and so on. This way we can solve the problem of having an overview of available methods in classes that has long inheritance chains. When formatting the method we use a special indication for methods that override a method in a super class. The overridden method in the super class is "grayed out" to better visualize that the method is actually implemented by the subclass.

An example of this formatting style is given in Figure 6 below.

RectPatternCross	**RectPattern**
DoPattern() RectPatternCross()& nbsp; SetDensity() SetOrder() SetPos()	RectPattern() SetBackground() SetDensity() SetPos() ShowFrame() Stroke()

Fig 6. Example how methods in a class are visualized, class RectPatternCross inherits from RectPattern. Observe that an overridden method is "grayd-out". All methods are hyper linked which makes it easy to quickly get a complete overview of the capabilities for a class. (Note that the links in the above example does NOT work since this table is taken directly from the real index)

2.4 Database scheme

The database scheme doesn't reveal anything surprising. Each documentation project consists of three tables. One table for all methods, one for all classes and finally one for all class variables. Global functions are considered members of the "GLOBAL" class. In addition to these project specific tables there are two project tables used to keep track of all projects and the files associated to each project.

Some purists might argue that storing both a foreign key and a foreign name is duplicate and that the tables aren't normalized. However, this is by design to avoid table lookups and slightly improve performance.

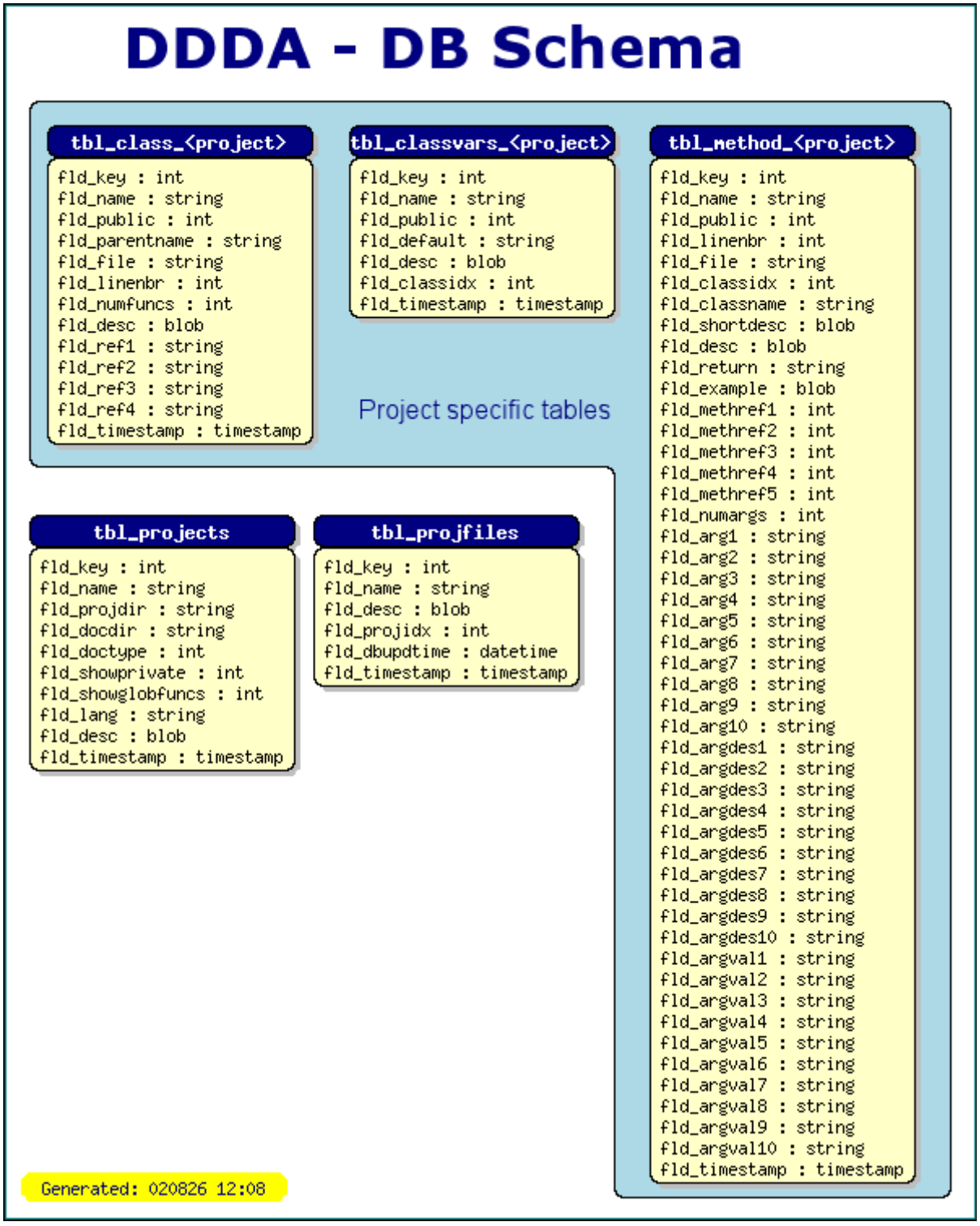

Fig 7. DDDA Database schema.

Each table has a primary key "fld_key" and a timestamp "fld_timestamp" to keep track of when the table row last was edited. We don't give detailed explanation of each field here but instead refer the interested

reader to the system documentation. Most of the fields should be self-explanatory anyway.

3. Working with the system from a user perspective

The user interface is meny and form based making it relatively straightford to work with the system.

When first started the user is presented with the DDDA main meny as shown below.

Fig 8. DDDA Main meny.

From this meny the user can choose to either create a new project or work on an existing project. The first alternative is used to get an overview of an existing project, the second to update the DB from the script files that makes up a project and the last alternative to modify or create a project.

3.1 Creating and modifying projects

After chosing the third meny alternative the user is presented with the form to create a new project as illustrated in figure 9 below.

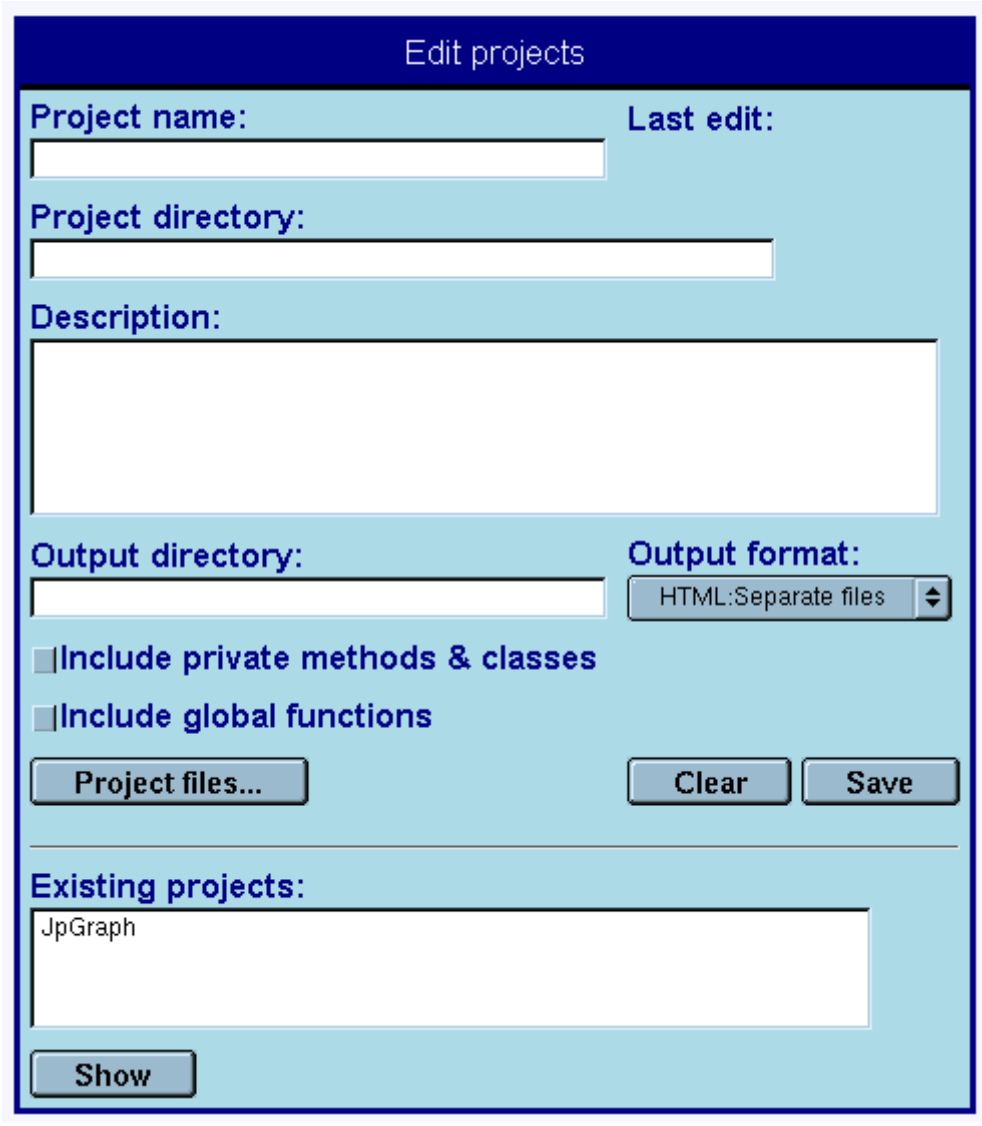

Fig 9. The form to edit and create new projects.

From the above form the user may eiter modify an existing project or create a new one.

3.2 Working on existing projects

After choosing an existing project the user is presented with an overview of all existing classes and methods in the project together with the percentage figures on the documentation as partly shown in figure 10 below.

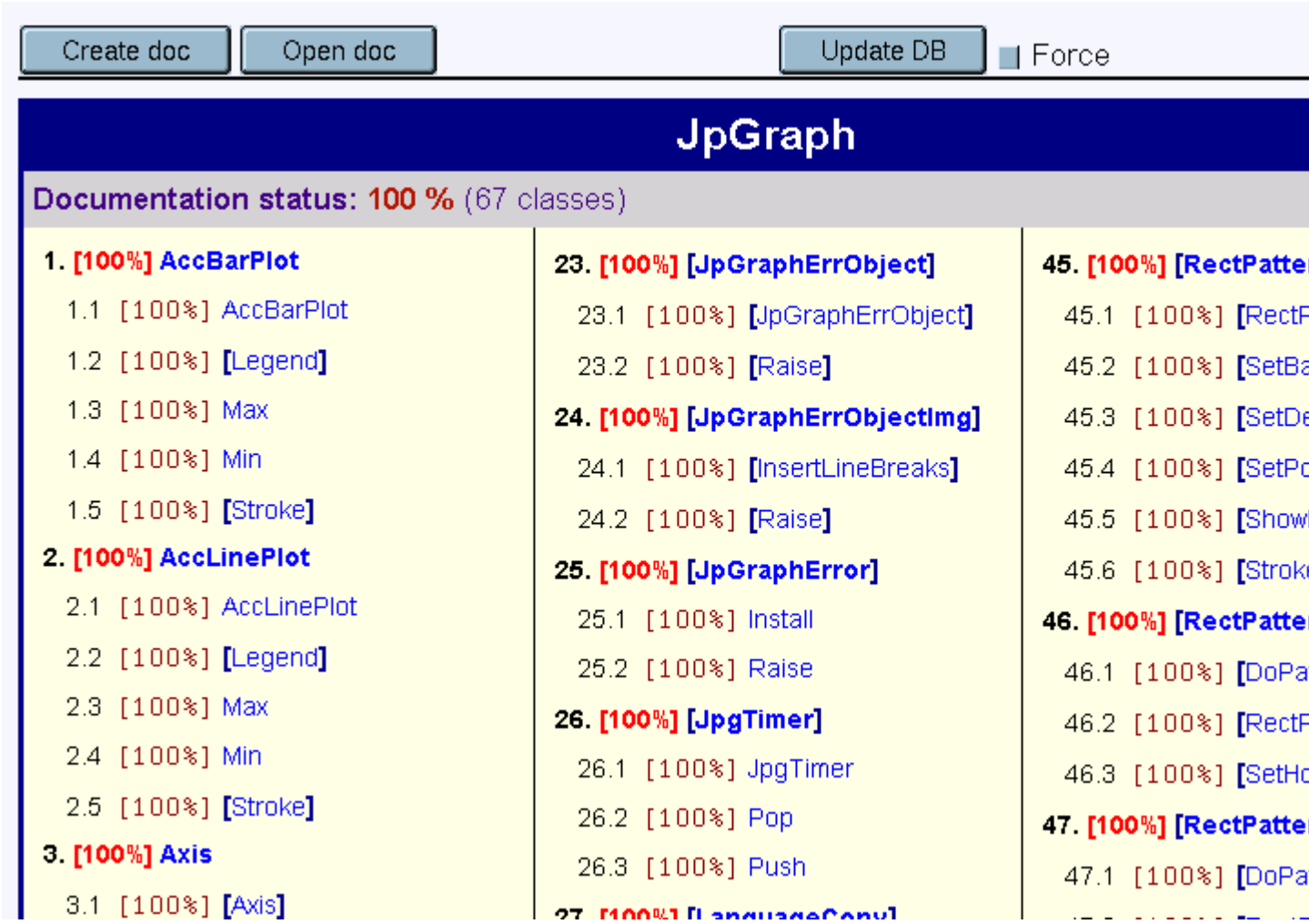

Fig 10. Existing project. All the classes together with the both the overall documentation status as well as the status of the individual classes.

From this view the user may now choose to edit a specific class or method by clicking on its name. The corresponding forms are displyed below in figure 12

LinearScale::MatchMin3()

Summary:

Determine the minimum of three values with a weight for last value

Argument:	Description:
$a	First value
$b	Second value
$c	Third value
$weight	Weight for third value

Returns:

Description:

```
Determine the minimum of three values. The third value
is weighted with '$weight' in a positive since so the
other values have to be '$weight' times smaller to be
choosen.
<p>
This helper method is used when we decide on waht scale
```

See also:

Example:

```
// For a or b to be choosen as min they have to be less
than
// 80% of c.
// If a=12, b=13, c=14 then b will be choosen since a
```

Scope: Public

Last edit: 2002-07-07 23:28

Save

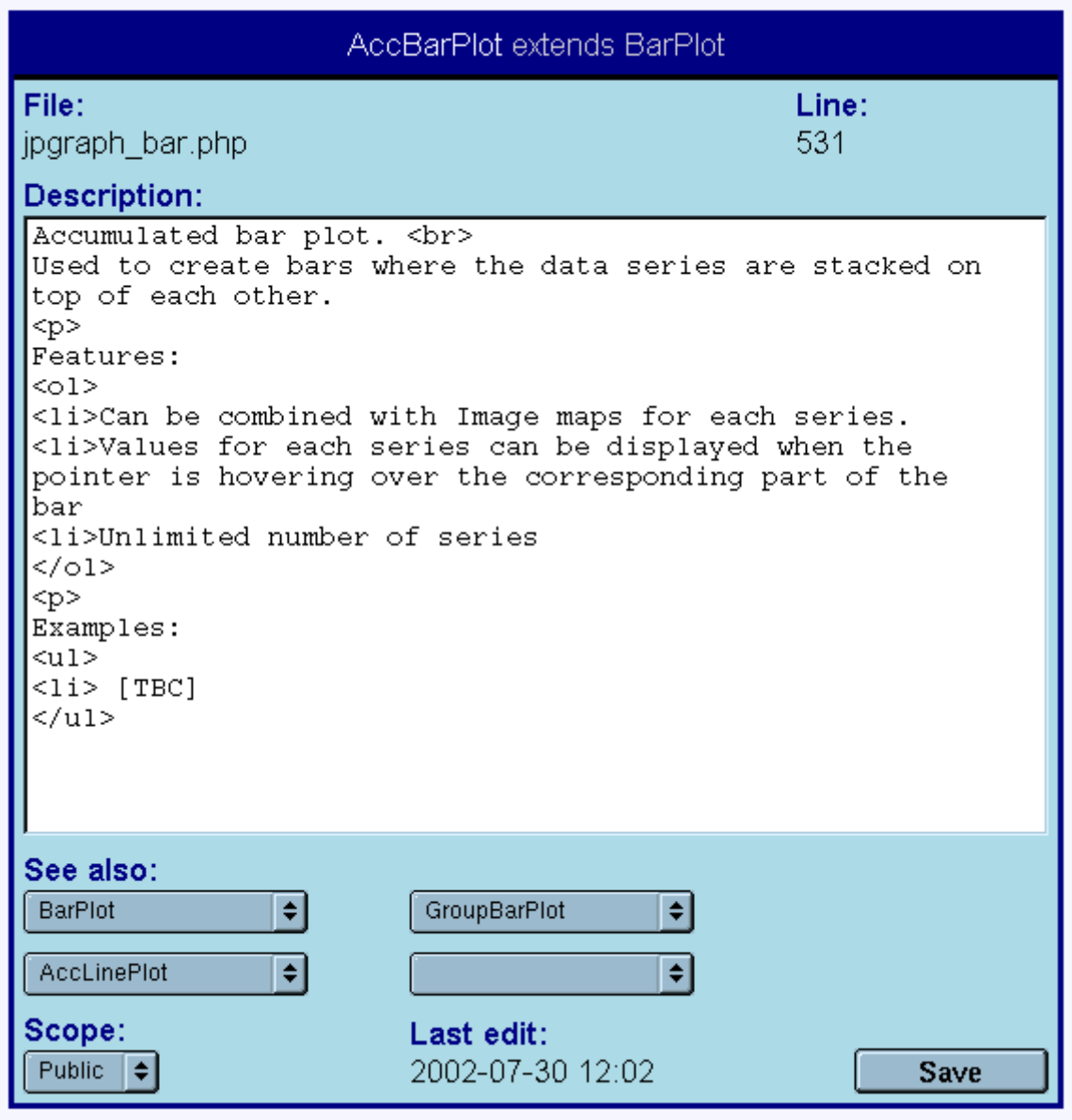

Fig 12. Editing classes and methods.

4. Conclusions and further enhancements

The system was first put to test to document JpGraph which is a medium complex php library consisting of around approx 10,000 loc. It has proven to work well in practice. The obvious improvements is mainly WEB technicalities where the user interface could be improved. However that doesn't involve any functionality changes but rather visual and navigational improvements.

In terms of future functional enhancements for the next version the plans are to:

- Enhance the parser to automatically recognize private and public methods based on naming conventions.
- Add graphical statistics
- Adapt the UI to work with other than the Opera browser