Documenting unit interfaces using fpdoc

Michaël Van Canneyt

June 24, 2007

1 Introduction

Documentation is an integral part of development. This is not only true for Component or package developers, but also for small development teams in companies that develop end-user software: If one developer needs to understand the code of another, some minimal documentation helps understanding how something was developed, or why a particular road was taken.

Obviously for people who develop units or components or complete packages that are distributed to other developers, the API must be documented, especially if there are a lot of classes, functions or methods. Not every developer wants to browse through many lines of code in order to understand how a particular method or group of methods functions.

Many tools exist to automate the process of documenting the interface of units. Lots of these tools are commercial, some are free of charge. In this article, the documenting tool of the Free Pascal Compiler is presented. It can be used to document pascal units: Units for Free Pascal, Delphi or Turbo Pascal. As of version 1.0.10 of Free Pascal, the fpdoc tool is available and distributed with the Free Pascal Compiler. However, it can be compiled and used independent of the Free Pascal compiler.

2 fpdoc Architecture

Obviously, any documenting tool will need to parse the unit or program which must be documented. This is true for fpdoc as well. Some tools require the documentation to be included in comments in the unit file. By contrast, fpdoc works by scanning one or more unit files, and combines these with the contents of one or more documentation (description) files. The documentation files are totally separate from the unit files: i.e. the units do not need any hooks or anchors to which the documentation must be 'attached'.

The documentation files are a set of XML files, which have a clearly defined, mostly hierarchical structure. The XML is mainly used to structure the contents of the documentation file, so that the documentation of a particular identifier in a unit can easily be located. The actual descriptions are formatted in a subset of XHTML, so anyone with experience in HTML will have no problem understanding the format of the documentation files. The use of XML also makes it easy to maintain or manipulate the documentation files with readily available tools: if so desired, the XML files could be converted to some completely other format.

From these files, two large trees or lists are built in memory: one with the identifiers of the units that must be documented, and the other with the descriptions found in the XML files. Writing the documentation is handled by browsing the identifier tree, and writing out the documentation, along the way inserting any documentation found in the XML description tree.

fpdoc is written using classes: A basic documentation writer class handles all logic of cross-referencing and converting XHTML to the requested output format, while descendent classes handle the specifics of the chosen output format: At this point descendents for HTML, XHTML and LaTeXhave been implemented. Plain text and RTF (plain RTF and WinHelp) are also planned. It is obvious that the needs for printed documentation ae different from the needs for browseable documentation, so the back-ends take care of almost any aspect of writing documentation: the order in which things are written, generating overviews etc.

The whole fpdoc is created using classes available in the FCL (Free Component Library) of the Free Pascal compiler: both XML handling and Pascal unit parsing are achieved with existing classes. The GUI editing tool is created using fpGTK - a Pascal object-oriented wrapper around GTK.

3 Document description files

The documentation file format is 'package' oriented, in that Each XML file can contain documentation for various packages. A package can loosely be defined as a collection of units which work together. Component suites such as Indy or Turbopower's Orpheus are good examples. It could also be a Delphi package, but it is broader than this. A package is represented by the XML **package** tag. The reverse is also possible: documentation for a package may also be spread over multiple XML files: the documentation of the various XML files will be merged in one big tree.

A package consists of one or more modules: a module corresponds with a single pascal unit, and is represented by the **module** tag. Each module consists of one or more elements: one element for each identifier in the unit, represented by the **element** tag.

Putting all this together, this means that a documentation file should have the following minimal structure:

```
<?xml version="1.0" encoding="ISO8859-1"?>
<fpdoc-descriptions>
<package name="PackageName">
<module name="UnitName">
<element name="IdentifierName">

... description tags...

</element>
</module>
</package>
</fpdoc-descriptions>
```

As can be seen from the tags, each tag has a *name* atribute: The name is used to identify the package, unit or identifier. Names can be case insensitive: when the documenting engine looks for the documentation of a certain identifier, it will search for the name in a case insensitive manner. When inserting cross-reference links (discussed later on), the names are also used.

The names are formed in a structured way: the name is the name of the pascal identifier. For identifiers that are part of another structure or function, such as record fields, method names or function arguments, the name is the name of the parent structure, followed by a dot and the field name. For instance, given the following definitions:

```
Type
```

```
  MyRecord = Record
   FieldA : Integer;
   FieldB : String;
  end;

Function MyFunction(Arg1 : Integer) : Boolean;
```

Then the following are valid element names:

```
<element name="MyRecord.FieldA">
<element name="MyFunction.Arg1">
<element name="MyFunction.Result">
```

It should be obvious what the various elements describe. Note that the descriptions for overloaded functions should be in 1 element.

Inside each of these elements, descriptions for the element can be inserted. The description of the element is divided over various XML tags, each with its own function:

short the **short** tag should contain a one-sentence description of the element. It is used in overviews, and is used as a synopsis in the description. For some identifiers (function arguments, enumeration type elements, record fields) the **short** tag is the only used tag.

description this tag contains the full-length description of a given identifier. It can contain any length of markup text and tags.

errors This tag is used in functions and methods to describe error conditions that can arise when using the function or method.

seealso this tag serves as a cross-reference section: it can contain only links to other identifiers: it is typeset specially depending on the output format.

example is used to include an example in the text. The *file* attribute designates the filename which will be included verbatim in the help text, in a separate section.

The following is therefore a complete description for an element:

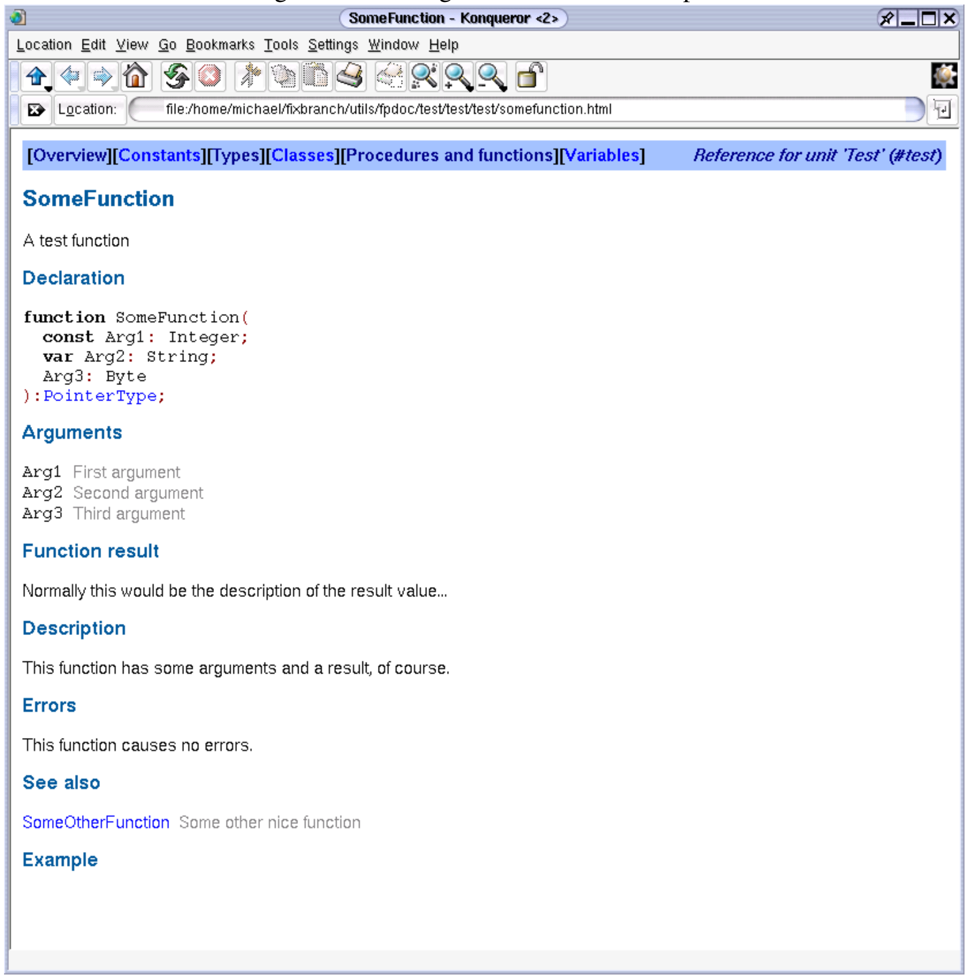
```
element name="SomeFunction">
<short>A test function</short>
<descr>This function has some arguments
and a result, of course.</descr>
<errors>This function causes no errors.</errors>
<seealso>
<link id="SomeOtherFunction"/>
</seealso>
<example file="ex1.pp"/>
</element>
```

This fully describes a fictitious function called `SomeFunction`. The resulting documentation page is shown in figure figure 1 on page 4

4 Description nodes

The **descr** and `short` tags can contain any text, intermixed with markup tags: the markup tags are the same as the HTML markup tags, although there are some extra things (commonly found in XHTML) to take care of:

Figure 1: HTML generated for the example


SomeFunction - Konqueror <2>
Location Edit View Go Bookmarks Tools Settings Window Help
Location:
file:/home/michael/fixbranch/utils/fpdoc/test/test/test/somefunction.html

[Overview][Constants][Types][Classes][Procedures and functions][Variables] *Reference for unit 'Test' (#test)*

SomeFunction

A test function

Declaration

```
function SomeFunction(
  const Arg1: Integer;
  var Arg2: String;
  Arg3: Byte
):PointerType;
```

Arguments

Arg1	First argument
Arg2	Second argument
Arg3	Third argument

Function result

Normally this would be the description of the result value...

Description

This function has some arguments and a result, of course.

Errors

This function causes no errors.

See also

SomeOtherFunction	Some other nice function

Example

- Tags are case sensitive - they must be all-lowercase.
- Each opened tag must be closed again.
- It is not possible to put tables and list environments inside a paragraph tag: the paragraph must be ended before the list or table, and re-opened after the list or table is finished. The following is wrong:

  ```
  <p>
  The following list shows the possibilities:
  <ol>
  </ol>
  from this it is obvious that...
  </p>
  ```

  Correct would be:

  ```
  <p>
  The following list shows the possibilities:
  </p>
  <ol>
  </ol>
  <p>
  from this it is obvious that...
  </p>
  ```

The basic HTML tags are supported:

```
<p>paragraph</p>
</br> linebreak.
<b>bold</b>
<i>italic</i>
<u>underline</u>
<table><tr><td>tables</td></tr></table>
<ul><li>unnumbered lists</li></ul>
<ol><li>numbered lists</li></ol>
<dl><dt>definition</dt><dd>lists</dd></dl>
<pre>Preformatted text</pre>
```

With some extra's:

```
<var>variable</var>
<file>file-references</file>
<remark>remark paragraph</remark>
<code>code fragment</code>
```

The code fragment will be highlighted in HTML output, and the remark is typeset in a special way. The sources of fpdoc contain a small test-file which shows all possibilities.

5 Cross-referencing

A documentation would not be very easy to use if it didn't contain cross-references, so a special tag was made to contain cross-references: the **link** tag. It requires a single attribute (*id*) and can be used in two ways:

```
<link id="SomeOtherFunction"/>
<link id="SomeOtherFunction">link text</link>
```

In the first example, the clickable text will be the value of the *id* attribute: `SomeOtherFunction`.

The value of the *id* attribute is the name of some element: a cross-reference to this element will be inserted in the text (if the output format supports this). References can be made to elements in external packages can be made: in that case, the name of the package/unit should be prepended to the name of the element, like this:

```
<seealso>
<link id="#OtherPackage.NiceUnit.TNiceComponent"/>
</seealso>
```

In order to be able to insert a correct cross-reference to an identifier in another package, documentation for this package must have been generated, and the table-of-contents file - which is generated together with the documentation - must be imported. More about this can be found in the manual.

Instead of a cross-reference, it is also possible to insert the complete synopsis of some element in the text. This can be done with the **printshort** tag. The *id* attribute again identifies the element of which the short description must be printed:

```
<printshort id="TNiceComponent"/>
```

This can be useful when creating a table which shows various elements in an enumerated type.

6 Starting documentation for a new unit

It should be obvious that to start documenting a new unit, a lot of tags must be created. While it is of course impossible to create the documentation out of nothing, it is possible to create an empty skeleton file: based on a unit source file, an initial XML file can be created which contains all the needed elements to document the unit. This has 2 advantages: it avoids a lot of typing, and tags for all identifiers are created, so one does not forget to document an element. Finally, it serves as a reminder for the amount of work which still needs to be done.

The makeskel tool which comes with fpdoc does exactly that: it takes as input one or more unit source files, and as output it creates an XML file with empty tags for all identifiers in the units: for each identifier, an element tag is generated which contains empty **short**,**descr**, **errors** and **seealso** tags. It can be invoked as follows:

```
makeskel --input=class.pp --output=class.xml --package=fcl
```

The generated XML file can be customized to some degree: It is possible to disable generation of certain elements, such as the **errors** tag or **seealso** tags. Depending on what kind of documentation is written, it may also be desirable to skip generation of tags for private or protected members of classes: for internal documentation they may be needed, but not for public documentation. makeskel provides some options to cater for this as well.

7 Customizing the output

The output of the fpdoc tool can be customized, albeit minimally. Out of the box, there are little things that can be customized:

1. Disabling generation of documentation of private and protected methods.
2. Insertion of a 'search' link in the generated HTML pages. This link will be inserted at the top of each page, and can be used to refer to a search page, which could be either a front to a CGI script or a java page containing a search engine.
3. The HTML output depends on a style sheet, so the 'look' of the documentation can be changed by modifying the stylesheet: Almost each element has its own style in the style sheet, and can be modified there.

More elaborate customization must currently be done by creating a descendent of one of the existing back-ends. They are built in a quite modular and straightforward way, which should make it easy to create descendents which create different-looking pages.

8 A GUI editor

Not everybody feels comfortable navigating large XML files, and many people are intimidated by the idea of having to create XML code. For this, a GUI editor for the documentation files was created: fpde, which works on Windows and linux/BSD. This tool takes care of managing the XML structure of the documentation: navigation between elements, creation and deletion of new nodes. The editor can handle multiple files at once, displaying the contents of each file on a separate page.

All that is left to do is edit the documentation itself: For each documentation tag (**short**, **descr**, **errors**,**seealso**), an editing area exists. At this time, it is not yet possible to do WYSIWYG editing of the documentation nodes. This means that the person writing the documentation needs to be familiar with the HTML tags. To make this easier to handle, shortcut keys and toolbuttons exist to insert the most common tags around the current selection. Links to other elements can be inserted by selecting an element, as shown in figure figure 2 on page 8. Complete tables can be inserted as well using a small table wizard. When switching between element nodes, the editor will save the documentation for the current node after it has verified that the node contains valid XHTML. Therefore when the contents of the editor are saved to file, the resulting file is guaranteed to be processable by the documentation editor.

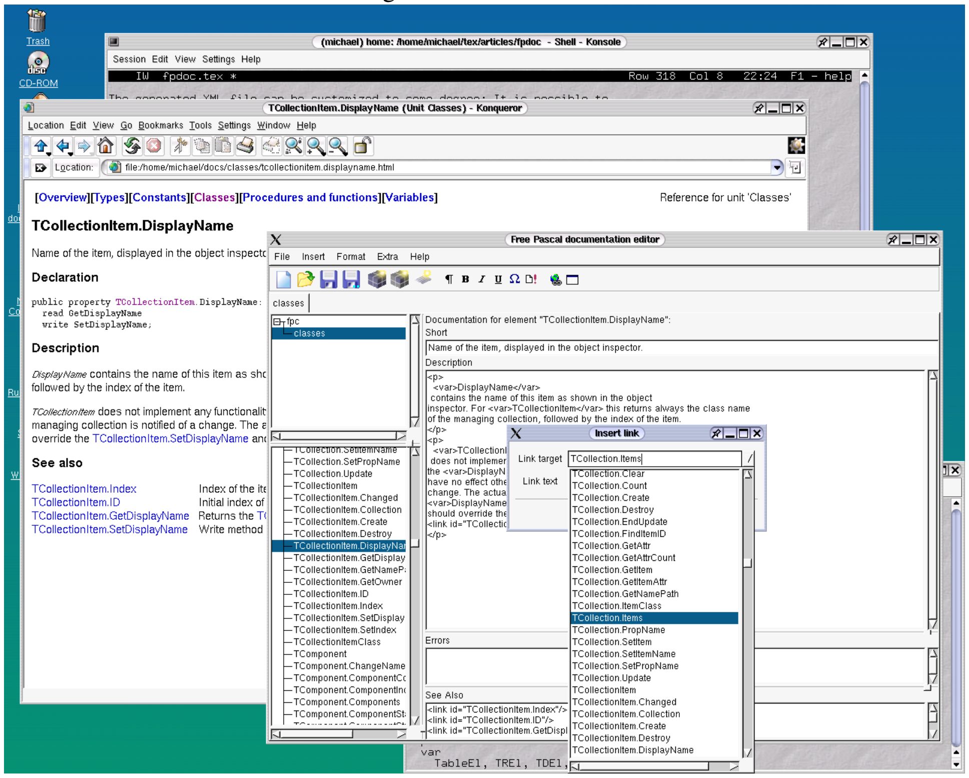
To start a new documentation project, one can start from zero, or open an existing file: the editor does not insert any tags by itself, it understands the documentation format. As an alternative, it can call 'makeskel' to generate an empty skeleton file. A small dialog will appear which allows to set makeskel's various options in a user-friendly way.

The editor is written in fpGTK, an object-oriented wrapper around the GTK units. People familiar with Delphi programming will have no problem understanding how it is built, and extending it should not be very hard.

9 Future plans

The fpdoc tool was born from a need to automate the generation of documentation: in order to have an up-to-date authoritative document, it was necesary to generate this document from the source code. The requirement to have the documentation in a separate file, without hooks in the actual unit code (as found in many other tools), and additionally the desire to have a LaTeX back-end, forced the members of the FPC team to write their own tool: fpdoc. Currently the documentation for the FCL (Free Component Library) is maintained using this new tool. The tool is therefore considered ready for general use, but this doesn't

Figure 2: FPDoc editor


Trash
CD-ROM
(michael) home: /home/michael/tex/articles/fpdoc - Shell - Konsole
Session Edit View Settings Help
TCollectionItem.DisplayName (Unit Classes) - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Free Pascal documentation editor
File Insert Format Extra Help
classes
fpc
classes
Insert link
Link target
Link text
Errors
See Also

```
IW   fpdoc.tex *                                  Row 318   Col 8    22:24  F1 - help
```

Location: file:/home/michael/docs/classes/tcollectionitem.displayname.html

[Overview][Types][Constants][Classes][Procedures and functions][Variables]   Reference for unit 'Classes'

TCollectionItem.DisplayName

Name of the item, displayed in the object inspecto

Declaration

```
public property TCollectionItem.DisplayName:
  read GetDisplayName
  write SetDisplayName;
```

Description

DisplayName contains the name of this item as sho
followed by the index of the item.

TCollectionItem does not implement any functionalit
managing collection is notified of a change. The a
override the TCollectionItem.SetDisplayName an

See also

TCollectionItem.Index	Index of the ite
TCollectionItem.ID	Initial index of
TCollectionItem.GetDisplayName	Returns the T
TCollectionItem.SetDisplayName	Write method

Documentation for element "TCollectionItem.DisplayName":

Short: Name of the item, displayed in the object inspector.

Description:

```
<p>
  <var>DisplayName</var>
 contains the name of this item as shown in the object
inspector. For <var>TCollectionItem</var> this returns always the class name
of the managing collection, followed by the index of the item.
</p>
<p>
  <var>TCollectionI
 does not implemen
the <var>DisplayN
have no effect othe
change. The actua
<var>DisplayName
should override the
<link id="TCollectio
</p>
```

See Also:

```
<link id="TCollectionItem.Index"/>
<link id="TCollectionItem.ID"/>
<link id="TCollectionItem.GetDispl
```

Link target: TCollection.Items

```
TCollection.Clear
TCollection.Count
TCollection.Create
TCollection.Destroy
TCollection.EndUpdate
TCollection.FindItemID
TCollection.GetAttr
TCollection.GetAttrCount
TCollection.GetItem
TCollection.GetItemAttr
TCollection.GetNamePath
TCollection.ItemClass
TCollection.Items
TCollection.PropName
TCollection.SetItem
TCollection.SetItemName
TCollection.SetPropName
TCollection.Update
TCollectionItem
TCollectionItem.Changed
TCollectionItem.Collection
TCollectionItem.Create
TCollectionItem.Destroy
TCollectionItem.DisplayName
```

```
TCollection.SetItemName
TCollection.SetPropName
TCollection.Update
TCollectionItem
TCollectionItem.Changed
TCollectionItem.Collection
TCollectionItem.Create
TCollectionItem.Destroy
TCollectionItem.DisplayNar
TCollectionItem.GetDisplay
TCollectionItem.GetNameP:
TCollectionItem.GetOwner
TCollectionItem.ID
TCollectionItem.Index
TCollectionItem.SetDisplay
TCollectionItem.SetIndex
TCollectionItemClass
TComponent
TComponent.ChangeName
TComponent.ComponentCc
TComponent.ComponentInc
TComponent.Components
TComponent.ComponentSt
```

```
var
  TableEl, TREl, TDEl,
```

mean that development has come to a halt. There are some enhancements planned to the various tools, some of which are being worked on, others are still just ideas:

1. Support for more output formats.
2. Support for **img** tags to include images.
3. Support for a **topic** tag, which would allow to create custom pages, not related to any unit identifier at all. This is useful for context-sensitive help in, for instance, an IDE such as Lazarus.
4. Support for templates for HTML output.
5. Implementation of an 'update' feature in the `makeskel` tool: as a unit is extended with more functions or classes, new element tags should be added to the documentation file.
6. More compact HTML: currently, 1 HTML page per identifier is generated. For units with lots of constants and resourcestrings, this results in a huge amount of pages. It would be better to have all constants concentrated on 1 page.
7. Having the documentation nodes in a database.
8. Generate a documentation page on-the-fly: useful for a webserver, where a small applet can generate the documentation on demand, straight from sources and nodes in a database. This is being worked on.
9. Add real-time editing to the documentation generating applet: this allows to set up a web-server where users can contribute documentation remotely, which can be of great service for a open-source project.

10. Add spport for calling fpdoc directly from the fpde editor.
11. Add support for calling a browser to display a generated page from the fpde editor.

As so often with open source projects, lack of time is the biggest obstacle in the realization of these enhancements. Volunteers or contributors are more than welcome.