Standards

Tijn van Der Zaar & Jean-David Boucher

First release: March 26, 2010
Last modification: February 20, 2013

Abstract

This documentation is devided in two chapter. The first one (chapter 1) is devoted in the manner to create a good documentation. Indeed, the documentation of a project is a touchstone for the developers. Indeed, the latter is necessary to know at which stage the work is, if functionnalities are still implement, how to use and manage their. There are three main tools to make documentations: the *word processors*, LaTeXand *Doxygen* (see respectively the sections 1.1, 1.2, 1.3).

In the second chapter (chapter 2) we detail the way to code in well format. Actually, respecting a strict standard of coding (indentation, syntax of variables, classes etc.) allows to a programmer to read some code that she/he have not developed. This feeling of *déjà-vu* facilitate the modification, the maintenance and the management of it.

Contents

Chapter 1

Documentation

The documentation of a project is a touchstone for the developers. Indeed, the latter is necessary to know at which stage the work is, if functionnalities are still implement, how to use and manage their. There are three main tools to make documentations: the *word processors*, LaTeXand *Doxygen* (see respectively the sections 1.1, 1.2, 1.3).

When someone develops some code in *Python*, *C/C++*, she/he must comment in *Doxygen* manner in order to be understandable and reusable by someone else. Moreover, *Doxygen* program makes *UML* diagrams and *HTML* pages that enable an easy navigation into the project.

In case that (i) the developped code is substantial or (ii) the functionnalities are not a package in programming language, then an explanatory document must be written. For that, two tools are available: the *word processors* and LaTeX. The last one is prefered because a template still exists.

1.1 Word processors

Nowdays, Microsoft Word and Open Writer are the leaders of the *word processors*. Because the robocup project is running on GNU/Linux and Windows plateform all users must respect the interoperability of producted documentations. If you use Microsoft Word you must save your document in *Rich Text Format* (extension file *RTF*). But the best way is to use Open Writer which is a multi-plateform (GNU/Linux and Windows) and save the document in the default format (extension file *ODP*).

On Windows, you can download the executable at
`http://www.openoffice.org/`

On Ubuntu, you can install it by typing the command line:

```
sudo apt-get install openoffice.org
```

1.2 LaTeX

LaTeXallows to produce easily a shared document, in a sens that everybody can use the same template and consequently can create a documentation with a common page setting. Moreover, code and PDF files can be include without changing something of the TEX file but just by compling it. Lot of documentations are available on the web, one of them:
http://en.wikibooks.org/wiki/LaTeX/
Others can be find on the ftp server of the Inserm U846.
The commented template (`template.tex` file) is available in the documentation folder of *mercurial*: It is important to note that three variables must be changed:

```
% Title of the document,
\newcommand{\titleVariable}{Put the title}
% The author(s),
\newcommand{\authorVariable}{Put the authors}
% The date of the first release.
\newcommand{\firstRelease}{Put the date of the first release}
```

1.2.1 Write code

To write many lines of code :

```
\begin{verbatim}
    a multi-line Python or C/C++ code
\end{verbatim}
```

To write one line of code: `\verb|a Python or C/C++ code|`
The symbol `|` can be replaced by anything else like `#` for instance. You can also use a monospace font in order to include a specific code name by using the special command `\texttt{TheClassName}`.

1.2.2 Include code file

You can include an entire code file by using the command `\verbatiminput{path}` from the package `verbatim`.

1.2.3 Include PDF file

You can also include the whole or a part of a PDF file by using the command
`\includepdf[pages = z,x-y]{path}` from the package `pdfpages`.

1.2.4 Include image file

To include an image use the command `\includegraphics` from the package `graphicx`. An example of a command :
`\includegraphics[width=\textwidth or another size]{path}`
You have many options to crop, resize etc. the image. Preferably, use PDF or PNG `\DeclareGraphicsExtensions{.pdf, .png}` file format because. PNG and PDF are the most portable format but note that PDF is a vectorial image.

1.2.5 Figures

To create a figure that floats, use the figure environment.

```
\begin{figure}[placement specifier]
    ... figure contents ...
\end{figure}
```

Specifier	Permission
h	Place the float here, i.e., approximately at the same point it occurs in the source text (however, not exactly at the spot)
t	Position at the top of the page.
b	Position at the bottom of the page.
p	Put on a special page for floats only.
!	Override internal parameters Latex uses for determining "good" float positions.

In the previous section, I was saying how floats are used to allow Latex to handle figures, whilst maintaining the best possible presentation. However, there may be times when you disagree, and a typical example is with its positioning of figures. The placement specifier parameter exists as a compromise, and its purpose is to give the author a greater degree of control over where certain floats are placed.

1.2.6 Tables

Although tables have already been covered, it was only the internal syntax that was discussed. The tabular environment that was used to construct the tables is not a float by default. Therefore, for tables you wish to float, wrap the tabular environment within a table environment, like this:

```
\begin{table}
    \begin{tabular}{...}
```

```
        ... table data ...
    \end{tabular}
\end{table}
```

You may feel that it is a bit long winded, but such distinctions are necessary, because you may not want all tables to be treated as a float.

1.2.7 Caption

It is always good practice to add a caption to any figure or table. Fortunately, this is very simple in Latex. All you need to do is use the `\caption{text}` command within the float environment. Because of how LaTeX deals sensibly with logical structure, it will automatically keep track of the numbering of figures, so you do not need to include this within the caption text.

The location of the caption is traditionally underneath the float. However, it is up to you to therefore insert the caption command after the actual contents of the float (but still within the environment). If you place it before, then the caption will appear above the float. Try out the following example to demonstrate this effect:

```
\begin{figure}[h!]
    \caption{A picture of a gull.}
    \centering
    \includegraphics[width=0.5\textwidth]{gull}
    \caption{A picture of a gull looking the other way!}
\end{figure}

\begin{table}[h!]
    \begin{center}
        \begin{tabular}{| l c r |}
        \hline
        1 & 2 & 3 \\
        4 & 5 & 6 \\
        7 & 8 & 9 \\
        \hline
        \end{tabular}
    \end{center}
    \caption{A simple table}
\end{table}
```

1.2.8 Cross-referencing

Another good point of LaTeX is that you can easily reference almost anything that is numbered (sections, figures, formulas), and LaTeX will take care of numbering, updating it whenever necessary. The commands to be used do not depend on what you are referencing, and they are:

```
\label{marker}
you give the object you want to reference a marker, you can see
it like a name.
\ref{marker}
you can reference the object you have marked before.
This prints the number that was assigned to the object.
```

The main utilizations are:

```
\section{Introduction}
We will see in the section \ref{The section blabla}.
\section{The section blabla \label{The section blabla}}
```

or

```
The figure \ref{The figure blabla} clarify this point of view
\begin{figure}
    \includegraphics[width=\textwidth or another size]{path}|
    \caption[The little caption]{The very long caption}
    \label{The figure blabla}
\end{figure}
```

1.3 Doxygen

Doxygen[1] is a source code documentation generator tool for Python, C/C++, C# and more. It can help you in three ways:

1. It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

[1]The main web page is http://www.stack.nl/~dimitri/doxygen/index.html

2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
3. You can even abuse doxygen for creating normal documentation.

1.3.1 Installation

On Windows, you can download the executable at
http://www.stack.nl/~dimitri/doxygen/download.html#latestsrc
On Ubuntu, you can install it by typing the command line:

```
sudo apt-get install doxygen doxygen-doc
```

In order to apply doxygen, you have to use a configuration file. You find it in the `src` folder on the *mercurial* file tree.

```
doxygen Doxyfile codeFile
```

The best way to do is to link all files inside the root file (localized on `src`) and to execute doxygen on this main file. It will produce global html file summarizing the whole code. In fact, the last command generates two folders where the *Doxygen* is launched. The *html* folder contains the *html* pages, you can access to them by:

```
firefox html/index.html
```

The *latex* folder contains the TEXfile. To create the PDF file:

```
cd latex
pdflatex refman.tex && pdflatex refman.tex
```

1.3.2 Python comments

There are two different ways to comment a *Python* code. For a single line the `#` is necessary.

```
# a simple comment
```

For a multi-line comment you have to use triple quotes `"""`:

```
"""
    a multi-line comment
    with a lot of explainations
    inside
"""
```

This kind of comment is also called *docstring*.

The next code is a template of a *Python* class (available on the *mercurial* server at `usr/documentation/doxygen/Python/PyClassExample.py`). Note that the first line indiquates which version of *Python* will be used and the second one refers to the encoding *utf-8*. The single and the multi-line comments are mixed. A particularity of the three quote comments are we can get them by the command `__doc__`:

```
ipython PyClassExample.py
In [1]: p = PyClassExample()
In [2]: p.__doc__
Out[2]: ' docstring of the class '
In [3]: p.__init__.__doc__
Out[3]: ' docstring of the constructor '
```

1.3.3 C/C++ comments

Chapter 2

Standard coding

Respecting a strict standard of coding (indentation, syntax of variables, classes etc.) allows to a programmer to read some code that she/he have not developed. This feeling of *déjà-vu* facilitate the modification, the maintenance and the management of it. We adopt these conventions:

Module name `publicModuleName` and `__privateModuleName`

`module-rgx=(__)?[A-Z][a-zA-Z0-9]{1,30}$`

Constant name `CONSTANT_NAME`

`const-rgx=(([A-Z_][A-Z0-9_]*)|(__.*__))$`

Class name `PublicClassName` and `__PrivateClassName`

`class-rgx=(__)?[A-Z][a-zA-Z0-9]{1,30}$`

Function name `publicFunctionName` and `__privateFunctionName`

`function-rgx=(__)?[a-z][a-z0-9]{1,30}$`

Method name `publicMethodName`, `__privateMethodName` and `__systemMethodName__`

`method-rgx=(__)?[a-z][A-Za-z]{1,30}$|__[a-z][A-Za-z]{1,30}__$`

Attribute name `publicAttributeName` and `__privateAttributeName`

`attr-rgx=(__)?[a-z][A-Za-z]{1,30}`

Argument name `argumentName`

`argument-rgx=[a-z][a-z0-9]{1,30}$`

2.1 Python

2.1.1 PEP 008 formatting

The *Python* formating follows the *PEP 008* convention[1]. One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of *Python* code. In brief, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

We customize it by adding the above rules. The program `pylint`[2] verifies if the python file is well formated.

2.1.2 Pylint

On Windows, go to the web page and follow the instructions:
http://thinkhole.org/wp/2006/01/16/installing-pylint-on-windows/
On Ubuntu, install pylint by type the command line:

```
sudo apt-get install pylint
```

In order to apply `pylint` on your code write the command line:

```
pylint --rcfile pylintrc PyClassExample.py
```

where `pylintrc` is the configuration file. You find it in the `src` folder on the *mercurial* tree. The statistic report of `pylint` is exhausive. You obtain the detail of one error by typing the following command (in this example, we assume that the error is `C0111`):

```
pylint --help-msg=C0111
```

The rapport is summarized if you use the option `--reports=n`.

```
pylint --reports=n --rcfile pylintrc PyClassExample.py
```

The lines of the errors are mentionned with the corresponding message.

[1] http://www.python.org/dev/peps/pep-0008/
[2] http://www.logilab.org/project/pylint

2.2 C/C++

2.2.1 GNU formating

2.2.2 BCPP