PyUt
Python UML Tool
Quality Management

11th July 2002

Abstract

This little documentation provides the main rules for writing code for PyUt, little UML 1.3 editor.

1 General Information

Quality for the PyUtproject is fundamental for many reasons. Here are some of the most important :

- Each developer has his own coding preferences. Try to read code written by someone else, it's always crazy. From my point of view, it may even be the main cause of the NIH[1] syndrom (just try to imagine how many list handling packages have been written for the last ten years). Having the same writing conventions make code easier to read, even if, when you write it, you have to follow rules too (not so easy in the beginning but quickly adopted).
- Quality makes more robust code. It's easier to find some bugs if the code is clearly commented.
- Try to read some C codes not well indented. For this, Python is great, you will not have the choice.

In conclusion for the introduction, I'll say that quality rules are necessary for a GPL project to allow developers may speak the same language between them.

For external developers that would like to submit code, I have to told you that your code has to respect quality rules or it may be rejected. This document also assume that you know the basics of *pydoc* and *javadoc*, used for technical documentation.

As you can see, we have chosen to write all code and documentation (except project report) in english.

[1]Not Invented Here

2 Quality rules

Ok, let's see these rules. Please contact the team if you think some rules have been forgotten or if you have any comments about them (Quality manager : <*pwaelti@eivd.ch*>).

2.1 Cosmetic

Tabulations

Tabulations are four spaces (real spaces, not four spaces tabs), position specified by the programming language Python. The length of a line must not exceed 78 characters.

Expressions

Spaces should be introduced between operator signs, assignement operations and after each `','` separator.

Names

Names in general should not contain any underline or special characters. Words separations are done with an uppercase letter. Classes always begins with an uppercase letter, variables with a lowercase one. Examples :

```
# A class definition
class PyutApp

# Variables and functions declarations
def myFunction(anInt, ...):
    myIntVar = anInt
```

Headers

There are two main headers : classes and functions headers. Classes headers should explain in detail what the class does. This header is formatted in *reStructuredText*, which may become a standard for documentation strings in the future. Function header is written using the usual *javadoc* style.

Here are the templates :

```
"""
Short description of the class (1 line max) ended by a dot.
Mandatory complete description of the class:
    - what it's for
    - how it works
    - sample use case

reStructuredText samples:
    - *Important* thing, or **vital** thing.
    - `className` or `methodName` or `paramName`

Example of code::
    anObject.doThat()
```

```
        another.sayHello()

    :version: $Revision: 1.4 $
    :author: Laurent Burgbacher
    :contact: lb@alawa.ch
    """
```

This is a class header. The revision id (`$Revision: 1.4 $`) is automatically set by CVS. The complete description of the class is essential.

```
def method(self, param1, param2):
    """
    Short description of what this method does, ended by a dot.
    Long description if needed.

    @param type name : Description
    @param type name : Description
    @return type : What this function returns
    @since 1.0
    @author Name <email>
    """
```

This is a method header. Particular parameters type for Python may be represented with they're usual notations (lists : `[]`, ...). The `@since ...` tag indicates the version from which the method available. For example, if `Revision` is 1.2, method will have the tag `@since 1.3`.

Separators

Each method has to be separated from one another using a line like this one and ends to the 78th column :

```
#>-----------------------------------...
```