Programming handbook

Author: Johan Lissing

Version: 0.3

Date: 2005-11-28

Summary

This document describes coding conventions for all software to be produced in the project "Wazzup DOC?!". It contains information on how to write, comment and manage the source code.

Project identity

Project group

PUM 12 2005
Linköpings tekniska högskola
Institutionen för datavetenskap (IDA)

Project members

Name	Area of responsibility	Telephone	E-mail
Martin Jormedal	Project leader (PL)	073-3121319	ook4mi@gmail.com
Daniel Hilding	Customer relations (CR)	070-7440440	danhi139@student.liu.se
Joakim Svartengren	Documentation manager (DOC)	070-4040005	joasv190@student.liu.se
Jonas Norling	Design manager (DES)	070-3904809	norling@lysator.liu.se
Johan Lissing	Implementation manager (IM)	073-9036256	johli650@student.liu.se
Thobias Bergqvist	Quality manager (QM)	073-6223040	thobe651@student.liu.se
Eric Åberg	Test manager (TM)	070-4058130	eriab522@student.liu.se

Mailinglist for group

pum12@und.ida.liu.se

Web page

http://www-und.ida.liu.se/~pum12/

Customer

Per Karlström, ISY LiU

Customer contact person

Per Karlström, 013-28 29 03, perk@isy.liu.se

Project supervisor

David Broman, 013-28 57 24, davbr@ida.liu.se

Examiner

Robert Kaminski, 013-28 24 57, robka@ida.liu.se

Document History

Date	Version	Changes	Name
2005-10-11	0.1	Document created	Johan Lissing
2005-10-28	0.2	Document updated after informal customer and internal review. Added content to sections 2.1.4, 2.1.7, 2.1.8 and 2.1.12. Changed "clams" to "braces" in section 2.1.3.	Johan Lissing
2005-11-28	0.3	Clarified coding conventions for Flex files in section 2.	Johan Lissing

1 Introduction

This chapter contains information about the disposition of this document. It is intended to work as a guide for the reader.

1.1 Purpose of this document

The programming handbook is a standardization document for the implementation phase of project "Wazzup DOC?!" performed by PUM-group 12. The handbook is specific to this project and therefore its external use is limited.

Use of this handbook will ensure that all source code produced in the project follows the same conventions. This will make the code produced by different programmers easier to integrate. It will also make it easier for future developers to familiarize themselves with the whole code. Since the product will be an extension to Doxygen, many of the standards here are adapted to the coding standard that Doxygen follows.

1.2 Reading instructions

The intended readers of this document are solely members of PUM-group 12. Therefore, unnecessary references and word explanations are left out. If the reader encounters any unclarity, contact the author for an explanation and a document update.

1.3 Distribution

This document will be distributed to the project locker.

2 Coding conventions

Here, the actual conventions for coding are described. The programming language used in the project is C++ along with the tool Flex. A general standard for coding in C++ already exists in RUT 9.3 "Standard for coding in C++", [Kullberg]. Almost all coding conventions in this project will be taken from that document. The code will also be commented for use with Doxygen.

2.1 Code layout and syntax

This chapter will be structured according to chapter 3.3 in [Kullberg]. For each subsection, "(no change)" means that the convention is the same as in [Kullberg]. If a new convention is necessary, or [Kullberg] suggests multiple options, the new convention is given. The "(Doxygen)" comment means that the new convention is necessary to adapt to the Doxygen source code.

When it comes to C++ code embedded in Flex rules, there are restrictions on the code layout, so that it is not misinterpreted by Flex. Any deviations for these code blocks are noted under the respective subsection.

2.1.1 Naming

Header and source files are given the extensions ".h" and ".cpp", respectively (Doxygen).

2.1.2 Comments

See section 2.2 "Commenting for Doxygen" . This applies to functions in Flex files as well, but not C++ code embedded in scanner rules.

2.1.3 Layout

Indentation width is 2 spaces and left braces ({), called "clams" in [Kullberg], are written on a separate line (Doxygen). This layout can be acheived in emacs by adding the following lines to the .emacs configuration file:

```
(require 'cc-mode)
(setq c-default-style "stroustrup")
(setq c-basic-offset 2)
```

In Flex rules, the left brace is placed on the same line as the scanner pattern.

2.1.4 Expressions and Operators

Double operators, such as `+=`, may be used.

2.1.5 Parenthesis

Use a space between the mentioned statements and the following parenthesis (Doxygen).

2.1.6 Conditional Operator

(no change)

2.1.7 Casting of Types

If casting is necessary, use a saturation function as in [Kullberg], but use `static_cast<>` or `dynamic_cast<>` to cast between types.

2.1.8 Functions

Call-by-pointer may be used instead of call-by-reference (Doxygen).

2.1.9 Constants

(no change)

2.1.10 Variables

(no change)

2.1.11 Pointer

(no change)

2.1.12 Program Flow

The `do` statement may be used.

2.1.13 In- and Output

(no change)

2.1.14 Memory Allocation

(no change)

2.1.15 Classes

Public or protected attributes may be used for global structures (Doxygen).

2.1.16 Inheritance and Class Hierarchies

(no change)

2.1.17 Templates

(no change)

2.1.18 Namespace

(no change)

2.1.19 Exception handling

Exceptions may be used for debugging purposes etc, but are not necessary (Doxygen).

2.1.20 The Standard Library

(no change)

2.1.21 Protection Against Double Including

In the FILENAME constant of the include-guard, all letters are uppercase and dots should be replaced by underscores. For example, an include-guard in the file example.h would start `#ifndef EXAMPLE_H` (Doxygen).

2.1.22 File Management

See section 3 "File and version management" .

2.2 Commenting for Doxygen

The source code produced in the project should be commented so that it can be documented using Doxygen. This applies to pure C++ code, as well as C++ functions in Flex files. Embedded C++ code in Flex rules need not be documented with Doxygen comments.

Doxygen supports many comment formats, but for conformity all code in this project should be commented using the so called Qt-style. This section describes how to write Qt-style comments. For more information about Doxygen, see [van Heesch].

2.2.1 Comment blocks

Most important is the multi-line comment block, beginning with `/*!` and ending with `*/`. The lines in between start with *. Such comment blocks should be used for all classes and functions in the header file where they are defined.

Single-line comments begin with `//!`. The general rule is that comments should be written before (above) the code that they describe. However, short single-line comments may be written on the same line as the code, using `//!<` to start the comment. These two single-line comments should be used for class variable declarations in the header files. If a variable needs a thorough explanation, the multi-line comment may be used instead.

See the example in section 2.2.3 for usage of these three types of comments.

2.2.2 Directives

Directives, or tags, are used to indicate what the comment describes. Doxygen supports many different directives, which can be used freely. See [van Heesch] for a complete list.

The following directives should always be present when documenting files (both header and source files) and functions (only header files).

Source and header files:

- `\file` *filename* The name of the source file
- `\author` *name* The name of the author
- `\date` *date* Date of latest change
- `\version` *number change* Version number and list of changes

Functions:

- `\param` *name description* parameter description, one for each parameter (may be left out if no parameters)
- `\return` *description* Return value description (may be left out if void)

See the example in section 2.2.3.

2.2.3 A commented example

This is an example of what a commented file should look like.

```
/*!
 * file Test.h
 * \author Johan Lissing
 * \date 2005-10-09
 * \version 0.2 Added Doxygen-compatible comments
 * \version 0.1 File created
 *
 * This is an example of how to comment source code.
 */

/*!
 * An example class
 */
class Test
{
public:
   /*!
    * A constructor.
    * \param a some value
    */
   Test (int a);

   /*!
    * A destructor.
    */
   ~Test();

   /*!
    * A member function taking two arguments and
    * returning an int.
    * \param d another value
    * \param e yet another parameter
    * \return the results
    */
   int foo(double d, double e);

private:
   //! Short comment about the member b.
   int b;

   int c; //!< Short comment about c on the same line.
};
```

3 File and version management

3.1 Version numbering

Files will be marked with a version number using the `\version` directive, see section 2.2.2 "Directives" . The version number consists of two numbers separated with a dot. The second number is increased after any change in the code. After either an inspection or a test of the code in the file, the first number is increased and the second is set to 0 (zero). The version numbering starts at 0.1.

When a file's version number is to be increased, insert the new `\version` directive first in the list. Leave any older version descriptions intact, so that all changes can be traced. See section 2.2.3 "A commented example" .

3.2 Subversion

The customer has provided a Subversion repository, which will be used for version control of all source code.

3.3 File structure

Doxygen itself uses a completely flat structure, with all header and source files in the same directory. Considering this, and that our own software will use a small number of files, we will also use one directory for all files.

4 References

[Kullberg] Kullberg, Christian, RUT 9.3 "Standard for Coding in C++" version 9.0, 2003, project course RUT-library, WWW:
`www-und.ida.liu.se/~TDDC02/hemligt/RUT/aktuella-pdf/`

[van Heesch] van Heesch, Dimitri, "Doxygen v1.4.4", 2005, WWW:
`www.doxygen.org`