Coding guidelines for SWEET

Martin Schreiber

Last updated: January 20, 2019

1 Purpose

Different developers working together have usually different coding conventions. Therefore an agreement to coding conventions has to be achieved when working together on one project for several reasons. One of the main reasons is to keep the code clean and thus to maintain a better code structure.

2 Coding conventions for Python

The entire workflow to compile and execute code written in SWEET as well as the postprocessing is mainly written in Python. We use Python 3 without any exceptions! Generally, please try to follow PEP8 guidelines which can be found at `https://www.python.org/dev/peps/pep-0008/` with the following exceptions:

- Use tab indentation with tabs configures to take 4 spaces
  In VIM, this can be configured with
  `autocmd Filetype python setlocal tabstop=4 shiftwidth=4`
- Use Unix encoded python source code files

3 Coding conventions for C++

3.1 Eclipse configuration

Eclipse supports features for automatic formattation of code via [Source] → [Format]. The style which is used by SWEET is programmed is closely related to "GNU/Allman" and a profile file "SWEET_eclipse_formattation.xml" is available in the current folder. This can be loaded via [Project] → [Properties] → [C/C++ General] → [Formatter]. Make sure to tick the box "Enable project specific settings" and [Import] the SWEET profile.

3.2 Identation

Blocks and scopes have to be indented by using a single TAB for each identation.

3.3 Naming conventions

3.3.1 Types

Types are distinguished whether they are used as a

- class,
- class with templates
- typedef followed by a specialized class
- classes without templates
- atomic variables (int, char, float, ...)

3.3.2 Variable naming

- For atomic types (int, char, float, etc.), all variables have to be written using underscores and small letters.
- For class types, the class variables have to be written without underscores

3.3.3 Parameters for methods

parameters of methods are prefixed by i_, o_ or io_.

- i_ means that this parameter is accessed read/only (const).
- o_ is used to declare this parameter as being an output reference to write some output values.
- io_ is used to declare an input/output pointer or reference which is read and written.

Output parameters always have to be of type pointer. No references should ever be used for output parameters! Return values handed back to the calling method are still allowed and have so far no convention.

```
/**
 * some comment
 *
 * \return description of return value
 */
char foo(
    const int i_bar_var,
                ///< this is a totally useless variable
    const SomeClass &i_someClass,
                ///< input via referenced parameter
    SomeClass *o_someOutputClass
                ///< output via pointer parameter
) {
    while (true) {
        ...
    }
    return 42;
}
```

3.4 Template parameters

ALL template parameters have to be prefixed with a "t_" to differ between template types and other types.

3.5 Comments

Comments are one of the most important thing in writing code. Therefore as much comments as are necessary or being requested by other developers have to be written.

Equations in comments can be e.g. written with \f$ f(x):=1+2x \f$. See https://www.stack.nl/ dimitri/doxygen/manual/formulas.html for examples.

```
/**
 * comments preceeding functions should
 * follow the doxygen (www.doxygen.org)
 * code-style
 *
 * \return description of return value
 */
void foo(
    const int i_bar_var, ///< this is totally useless
    void *io_foo_var     ///< this variable is not
                         ///< able to drink beer
) {
    while (true) {
        ...
    }
}

constructor(
    int i_val,  ///< index
) :
  some_variable(i_val)
{
        ...
}
```