**SimVascular Coding Standards and Guidelines**

Contact: Adam Updegrove, updega2@berkeley.edu

Doc date: June 6, 2016

**Coding Conventions**

Coding in a consistent style eases shared development. All code that is contributed to SimVascular must conform to the following style guidelines.  Exceptions are permissible, following discussion in code review.

1. **All code that is compiled into SimVascular by default must be compatible with SimVascular’s license.**
2. **Copyright notices should appear at the top of header and implementation files.**

**------------------- OLD, needs to be adjusted for new SV ---------------------------**

1. **All new module files should follow the following format :**

Derived module classes: cvNewnameModelname.h, cvNewnameModelname.cxx

(i.e. for Parasolid, which is derived from the Solid Model module, cvParasolidSolidModel)

Module init files: cv\_newname\_modulename\_init.h, cv\_newname\_modulename\_init.cxx

(i.e. for Parasolid, it is cv\_parasolid\_solidmodel\_init

Module utility files: cv\_newname\_modulename\_utils.h

cv\_newname\_modulename\_init.cxx

(i.e. for Parasolid, it is cv\_parasolid\_solidmodel\_utils

1. **New additional modules and derived modules should be placed according to this:**

New modules: A new directory with the category should be placed under source. Then a sub directory should be created underneath the category for the abstract base class (i.e . Source->Model->SolidModel)

New derived module classes: A new directory underneath the category should be created for the derived class implementation (i.e. Source->Model->PolyDataSolidModel).

-----------------------------------------------------------------------------------------------------

1. **Only one public class per class header file. Internal helper classes may be forward declared in header files, but can then only be defined in implementation files, ie using the PIMPL idiom.**

Rationale: helpful when searching the code and limits header inclusion bloat that slows compilation time.

1. **Class names and file names must match, class names must be unique.**

Rationale: helpful when searching the code, includes are flattened at install.

1. **The indentation style is the "Allman" style. The curly brace (scope delimiter) is placed on the following line and is not indented. The following code is indented two spaces.**

void myfunction(int param)

{

code;

}

Rationale: Readability and historical

1. **Conditional clauses (including loop conditionals such as for and while) must be in braces below the conditional.**

Ie, instead of if (test) clause or if (test) { clause }, use

if (test)

{

code;

}

Rationale: helpful when running code through a debugger

1. **Two space indentation. Tabs are not allowed. Trailing whitespace is not allowed.**

Rationale:  Removing tabs ensures that blocks are indented consistently in all editors.

1. **Only alphanumeric characters in names. Use capitalization to demarcate words within a name (i.e., camel case). Preprocessor variables are the exception, and should be in all caps with a single underscore to demarcate words.**

Rationale: Readability

1. **Try to always spell out a name and not use abbreviations except in cases where the shortened form is obvious and widely understood.**

Rationale: Readability, self-documentation

1. **Following the copyright notice, the name, purpose, and author information should be provided**

/\*\* \class classname – one line description

\* \brief short description

\* \author name

\* \author email

\* \author affiliation

\* \author website

\* \details larger description if necessary

\*/

Rationale: Doxygen generated documentation uses this to describe each class.

1. **Above each function in the header file, provide the description of the function, the input parameters, return, and any other notes.**

/\*\* \brief description of function

\* \param one for each parameter and description

\* \return description of return

\*/

**If additional details need to be provided, provide the details of the function above the function in the implementation file.**

/\*\*

\* \details If any other info, add here

\*/

Rationale: Doxygen generated documentation uses this to generate info.

1. **Use "this->" inside of methods to access class methods and instance variables.**

Rationale: Readability as it helps to distinguish local variables from instance variables.

1. **Include statements in implementation files should generally be in alphabetical order, grouped by type. For example, SV includes first, system includes, STL includes, and VTK includes.**

Rationale: avoid redundant includes, and keep a logical order.

1. **Do not use 'id' as a variable name in public headers, also avoid ‘min’, ‘max’, and other symbols that conflict with the Windows API.**

Rationale: ‘id’ is a reserved word in Objective-C++, and against variable name rules. ‘min’, ‘max’, and less common identifiers listed in Testing/Core/WindowsMangleList.py are declared in the Windows API.

1. **Eighty character line width is preferred.**

Rationale: Readability

1. **Method definitions in implementation files should be preceded by // followed by 78 “-“ characters.**

Rationale: Readability

**Git Commits**

1. **Well formed commit message**

Every commit message should consist of a one line summary optionally followed by a blank line and further details. This is most easily approximated to the subject of an email, and the body in the form of paragraphs.

1. **Start each commit message with an all-caps three letter notifier.**

Here is the list of possible notifiers:

**BLD**: change related to the build system (Make or CMake)

**BUG**: bug fix

**DEP**: deprecate something, or remove a deprecated object

**DEV**: development tool or utility

**DOC**: documentation

**ENH**: enhancement

**MAINT**: maintenance commit (refactoring, typos, etc.)

**REV**: revert an earlier commit

**STY**: style fix (whitespace, PEP8)

**TST**: addition or modification of tests

**REL**: commit specific to a release

Example commit message:

DEV: Very brief header that explains main info of commit

This is a more detailed description if necessary. It should contained changed functions, added features, etc. This should still be less than 50 words.

1. **Valid committer username and email address**

Every developer must have a valid name and email configured in git.

1. **ASCII filename check**

All file names must contain only ASCII characters.

1. **No tabs**
2. **No trailing whitespace**
3. **No empty line at end of file**
4. **Proper file access mode**

Files must be committed with sensible access modes.

1. **One megabyte maximum file size**

**Releases**

Development proceeds along the trunk or “master” branch (taking the form of topic branches that start from and are merged into master), and every so often a release is tagged and branched from it. In general no work goes into the “release” branch, other than the handful of important patches that make up the occasional patch release.

On the master branch, bug fixes and new features are continuously developed. At release time, the focus temporarily shifts to producing a library that is as stable and robust as possible.  The process for cutting releases is as follows:

1. **Inform developers that a release is coming:**

Two weeks before the intended release branch, announce on the mailing list that a new release is nearing. This alerts developers to hold off making drastic changes that might delay the release and gives them a chance to push important and nearly completed features in time for the release.

1. **Polish the dashboards and bug tracker:**

Persistent compilation and regression test problems are fixed. Serious outstanding bugs are fixed.

1. **Forward release branch**

When the dashboards are clean and the outstanding features are finished, we pick a point on the development branch to be the start of the next release branch. Next we move the release branch forward from its current position to the new one.

1. **Gather descriptions of changes**
   1. Compile a list of developers and their changes and send emails to each developer asking them to summarize their work (ideally this should just be a compilation of their git commit messages).
   2. Run the API differencer script in Utilities/Maintainance/semanticDiffVersion.py
2. **Do the release candidate cycle**
   1. Tag the release branch and make and publish release candidate tar balls and change summaries.
   2. Announce the release candidate and request feedback from the community.
   3. Bug reports should be entered into the bug tracker with the **Found in Version** field set to “rc” and **Target Version** field set to the upcoming release.
   4. If no important bugs are reported sevendays after the candidate is published, the source is re-tagged and packaged as the official release.
   5. If the community does report bugs, the manager classifies them in the bug tracker and sees that they are fixed.
   6. Only serious bugs and regressions need to be fixed before the release. New features and minor problems should go into the master branch as usual.
   7. Patches for the release branch must start from the release branch, be submitted through gerrit, and merged into master. Once fully tested there the branch can be merged into the release branch.
   8. When the selected issues are fixed in the release branch, the tip of the release branch is tagged and released as the next candidate and the cycle continues.
3. **Package the official release**

The official SimVascular release consists of tar’s (linus), msi’s (windows), and dmg’s (mac): the Documentation and release notes are also provided.