Developer's guide

Nadir SOUALEM – INRIA

Contents

The developer's guide is a practical introduction to developing applications for H2OLAB.

1 Prerequisite Knowledge

To develop in H2OLAB it is at least required to have a basic understanding of the following points:

- Object Oriented Programming in **C++**
- **MPI** - The Message Passing Interface
- **XML** - File parameters
- Batch Programming (Bash, Windows Command)
- **CMake 2.6** or higher for Linux Users
- **Microsoft Visual Studio 2005** or higher for Windows Users
- Revision control system **Subversion**: update, commit, merge and branching
- Source code documentation generator tool **Doxygen**
- **Latex**

2 Communication

If you have any questions or problems with H2OLAB softwares, you can use the mailing list hydrolab-devel `hydrolab-devel@lists.gforge.inria.fr`. You may subscribe and unsubscribe from this mailing lists using your Gforge-account.

3 Coding Rules

3.1 Documentation

Header files must be completely documented. This means every class, method, and data member must have comments. Header files describe the interfaces of the system, and as such, should contain all the information a developer needs

to use/understand the interface. Code must be documented according Doxygen syntax.

Rule 1 – To document a block of code, the syntax we use is:

```
/**
 * Documentation here.
 */
```

Rule 2 – All functions must be documented

```
/**
* @brief Integration of f in [a,b] using Trapezoid Method
* @param f is a function of one variable
* @param a is the lower bound
* @param b is the upper bound
* @return Integral of f(x) in [a,b]
*/
double integration(double (*f)(double), double a, double b);
```

Rule 3 – All classes must be documented

```
/**
* @brief Short description of test class
*
* Long description of test class
*/

class Test
```

Rule 4 – All Members must be documented

```
int var; //!< Detailed description after the member
```

Rule 5 – All Enum Must be documented

```
/**
* @brief Short description of an enum
*
* More detailed enum description
*/

enum TEnum {
            TVal1, //!< Enum value TVal1 description.
            TVal2, //!< Enum value TVal2 description.
            TVal3  //!< Enum value TVal3 description.
           };
```

Rule 6 – Use Latex to document models or PDE if you can with `\f$` delimiters

```
* @return Integral of f(x) in [a,b] :
* \f$ \displaystyle\int_{a}^b f(x)dx=(b-a)\frac{f(a)+f(b)}{2} \f$
```

gives:

Integral of $f(x)$ in $[a,b]$: $\displaystyle\int_a^b f(x)dx = (b-a)\frac{f(a)+f(b)}{2}$

To display formulas that are centered on a separate line, delimiters are `\f[` and `\f]`. An example:

```
* @return Integral of f(x) in [a,b]
* \f[
* \int_{a}^b f(x)dx=(b-a)\frac{f(a)+f(b)}{2}
* \f]
```

gives:
Integral of $f(x)$ in $[a, b]$

$$\int_a^b f(x)dx = (b-a)\frac{f(a)+f(b)}{2}$$

3.2 Development

Rule 7 – Use the following naming conventions

1. All class names start with an upper case letter.
2. All function names start with a lower case letter.

Rule 8 – Protect header files from multiple inclusion with preprocessing command `#ifndef`. Header files such *MyClass.h* must be defined by:

```
#ifndef MYCLASS_h
#define MYCLASS_h
// Code here
#endif
```

Rule 9 – Do not place *using namespace* directive in header files.
For example, let's use boost's gregorian date library. In my class I want to return dates and use dates in methods. So my header file looks like:

```
#include <boost/date_time/gregorian/gregorian.hpp>

class Calendar
{
public:
   boost::gregorian::date GetEventDate(void) const;
   void  SetEventDate(boost::gregorian::date dateOfTheEvent);
};
```

Clearly these are long names and you are tempted to put at the top of you class file:

```
using namespace boost::gregorian;
```

This would mean you could just use **date** instead of **boost::gregorian::date**. That's nice. But you can't do that. If you do you are making the decision for everyone who uses your class as well. They may have a conflict, "date" is a very common name afterall. So, don't use it in you header file, but you can use it in your source file. Because it's your source file you can make the decision to use short names.
Rule 10 – Do not use `define` to declare constant values, use `const`

```
// Incorrect
#define EPS_POINT_OUT_BORDER 1e-10

// Correct
const double EPS_POINT_OUT_BORDER = 1e-10;
```

Rule 11 – Prefer initialization to assignment in constructors

```
class MyClass
{
private:
string x_;
string y_;

public:
MyClass(){x_ = "Project" ; y_ = "H2OLAB" ;}

};
```

Initialize using initializer list

```
MyClass():x_("Project"), y_("H2OLAB") {}
```

or in the usual way:

```
MyClass(){x_("Project") ; y_("H2OLAB") ;}
```

Rule 12 – Prefer initialization to assignment

```
  // Assignement
  MyClass obj; // call default constructor MyClass()
  obj = value; // call operator =

  // Initialization
  MyClass obj(value); // call copy constructor
```

Rule 13 – Minimize compilation dependencies between files by using forward declaration
Reduce header file dependency by effective use of forward declarations in header files. Sometimes to reduce header file dependency you might have to change member variables from values to pointers. Every time you use a `#include` make sure that you have an extremely good reason to do so.
Example:

```
class Simulation;
class run_global_results;

class Launcher{
protected:
/** Simulation : abstract-based pointer.*/
Simulation *simulation;
/** Run results : abstract-based pointer.*/
Run_Global_Results *run_global_results;
  ...
};
```

By defining pointer and not object themselves, compilers know how much memory they can allocate (size of a pointer !!!).
Rule 14 – Use `const` whenever possible C++ provides powerful support for const methods and fields. `const` should be used in the following cases:

- Methods that do not change the value of any variable in the class should be declared const methods
- If a function is supposed to just read information from a class, pass a const pointer or reference to this function

Rule 15 – Match case *BC_Description.h* is not *BC_description.h*
Rule 16 – Use / separator for include paths

```
#include "Porous_Basis\Grid_Visualisation.h" //KO
#include "Porous_Basis/Grid_Visualisation.h" //OK
```

Rule 17 – Template classes must be in headers
Rule 18 – When you use template argument list prefer > > to >>

```
std::map<T,std::vector<double>> A;  //KO
std::map<T,std::vector<double> > A; //OK
```

Rule 19 – Declare template iterators as typename

```
std::map<T,double>::iterator it=M.begin(); //KO
typename std::map<T,double>::iterator it=M.begin(); //OK
```

3.3 File parameters

Rule 20 – Default parameters modifications must be discussed
Rule 21 – All parameters must be documented
Rule 22 – Possible values have the format **value::value_description**;...

4 Subversion

4.1 Connection

Rule 23 – Prefer **svn+ssh** to **Webdav**
Network protocol **svn+ssh** is stateful and noticeably **faster** than WebDAV. For every day usage, it is highly recommended to use **svn+ssh**.

4.2 Branching

Rule 24 – New branches must be in svn repository branches
Rule 25 – Branching must not exceed **3 months**
Rule 26 – Bring changes from the trunk over to your branch as often as possible
Rule 27 – Remember range revisions when you bring changes from trunk

5 Testing

5.1 Non Regression tests – NRT

Rule 28 – New launchers must be tested. Add a new launcher *MyLauncher* and a new test *MyTest* in
$HYDROLAB_ROOT/svn/non_regression_tests/short/MyLauncher/Mytest
and
$HYDROLAB_ROOT/svn/non_regression_tests/long/MyLauncher/Mytest

1. Add xml parameters files which allow test generation and validation: *generation_of_test_xml_files_.xml*,*validation_parameters.xml*
2. Add *parameters* and *reference_results* directories

Rule 29 – Add or Update tests when you add new features
Rule 30 – Short NRT must not exceed **2 minutes** per Launcher
Rule 31 – Short test must use 32-bits application for Windows Users
Rule 32 – Long NRT must not exceed **1 hour** per Launcher
Rule 33 – Long test must use 64-bits application for Windows Users
Rule 34 – Non-Regression tests must be documented in
$HYDROLAB_ROOT/svn/benchmark_book/NRT/Launcher/description.tex

6 Documentation

Rule 35 – Software must be documented in
$HYDROLAB_ROOT/svn/docbeta/softs/description/Launcher/Launcher.tex