Dernière modification le 09/07/11 09:57:34

C++ Coding Conventions

Naming Conventions

- **Class names** must begin with an upper case character and the case of the following characters must be determined using camel case convention. The 'underscore' (_) character must be avoided in class names but for compelling reasons.

Example:

```
class SatelliteSpecs
{
    /* ... */
};
```

- **Variable and property names** must begin with a lower case character and the case of the following characters must be determined using camel case convention.

Example:

```
SatelliteSpecs satSpecs;
```

- **Variable and property names** must restrict the use of the 'underscore' (_) character to unit postfix. *Example:* `double payloadWeight_kg;`

- **Operation names** must begin with a lower case character and the case of the following characters must be determined using camel case convention. Contrary to usual conventions in C language or in the C++ standard library, the 'underscore' (_) character should be avoided with 2 exceptions: accessor prefix and unit postfix.

Example:

```
double get_payloadWeight_kg();
```

- **Namespace naming** can either follow the same rules as class naming or be restricted to an all-lowercase very short word (no more than 4 characters).

Example 1:

```
namespace DeployModel
{
    /* ... */
}
```

Example 2:

```
namespace dhsm
{
    /* ... */
}
```

Creation and Naming of Files and Directories

- Every **project directory** must contain a **src** subdirectory containing all non-generated source files.

- Every **namespace** defined within the project must correspond to a subdirectory of directory **src**. Namespace subdirectories must follow the ***same nesting order*** as the namespaces and adopt an ***identical capitalization***.

- In principle, ***for every non-template class*** there should be one header file using the ***.h*** suffix and one implementation file using the ***.cpp*** suffix. However, a small set of classes that are deeply linked together can be declared and implemented in a unique header/implementation file pair. ***Template headers*** must bear the ***.hpp*** suffix whereas ordinary class headers must stick to the usual ***.h*** C/C++ suffix.
- The ***capitalization*** of the file names must be ***identical*** to that of the class name.
- The **header and implementation files** of any class must be created within the subdirectory of **src** corresponding to its namespace. For instance, the fully qualified class `DeployModel::SatelliteSpecs` must be declared and implemented in files *src/DeployModel/SatelliteSpecs.h* and *src/DeployModel/SatelliteSpecs.cpp*.
- The header file must be **protected from multiple inclusions** using a leading `#ifdef` that defines a preprocessor macro formed by the fully qualified name of the class, using the 'underscore' character (_) as namespace separator, using an identical capitalization and ending with the header file extension (_h or _hpp) and the label '**_INCLUDED**'. For instance, the content of file *src/DeployModel/SatelliteSpecs.h* declaring the `DeployModel::SatelliteSpecs` class should be enclosed wihtin the following macro definition:

```
#ifndef DeployModel_SatelliteSpecs_h_INCLUDED
#define DeployModel_SatelliteSpecs_h_INCLUDED

namespace DeployModel
{

class SatelliteSpecs
{
    /* ... Class declarations ... */
};

}

#endif
```

Blocks and Indentation

- The curly brackets defining a block must occupy their own line. In particular, they must not be preceded or followed by a C++ expression. Their indentation must be the same as the statement preceding the block:

```
    if (age < 25)
    {
        isFeeReduced = true;
    }
```

- Indentation must be implemented using exclusively the tabulation character. To avoid the visual annoyance of the standard 8-character length of a tabulation, you should change the tab display length preference of your favorite editor, *not* use whitespaces for indentation.
- The indentation of the following elements ***should not be increased:***
  - statements within namespace definition blocks:

```
namespace DeployModel
{

class SatelliteSpecs
{
    /* ... Class declarations ... */
};
```

```
}
```

- case statements in a switch-case expression:

```
    switch (choice)
    {
    case 'a':
        /* ... */
        break;

    case 'b':
        /* ... */
        break;

    default:
        /* ... */
    }
```

- scope statements (`public`, `protected`, `private`) in a class declaration:

```
class SatelliteSpecs
{
public:
    double get_payloadWeight();
    void set_payloadWeight(double value_kg);
    bool isLaunchable();

private:
    double payloadWeight_kg;
};
```

Doxygenation

- All classes, operations and properties must be source-documented using doxygen.
- By default the *javadoc* syntax and conventions are preferred wherever possible. You can find here the typical doxyfile that is to be used.
- Classes and properties must be doxygenated in the header file (*.h or *.hpp for templates) to which they belong.
- Operations should be doxygenated in their corresponding implementation file (*.cpp) rather than in the header in order to minimize comment clutter in the header file.
- Even for template operations (normally implemented in the header itself), an implementation file (*.cpp) should be created containing only the corresponding doxygen comments (using the @fn keyword to refer to the commented operation).