Introduction

The aim of this document is to present a set of coding standards for the EDF's project in the Sysfera company.
These standards are based upon the *Programmer's guide* document used to develop the DIET middleware, and also inspired by the Qt coding style.

1 File structure

1.1 Headers

Each file begins with the DIET header, where are precised:
- a short title (in one line) that explains the role of this file,
- the authors of the file,
- A license field that will be automatically filled in.

As an example, here is the header of the LaTeX source of this document

```
/**
 * \file Name of the file
 * \brief A one line description
 * \author author(s) of the file
 * \section
 *   <licence>
 */
```

The CMakeLists.txt are not submitted to this header marking (since it was considered that there is no technical value attached to them).

These CVS fields are removed by a shell script when a distribution is built. The `$LICENSE$` line will also be processed automatically, and replaced by a short version of the DIET license, so please be very careful to put one and only one space between the `%*` and the `$`, between `$LICENSE$` and the final `*%`, and NO space after the last significant character of this lines.

1.2 C++

Each header must start with a:

```
#ifndef _CLASSNAME_HH_
#define _CLASSNAME_HH_
```

and ends with a *#endif*. A header must contain only one class (unless there are some inner classes/nested classes required) and its name must be *ClassName.hh*. The implementation class must be *ClassName.cc* and must contain the implementation of the functions of the hh file.

In a class declaration, the public part comes first, then the protected one and last the private one. These keywords must always be, event if no protected item. Each inner part declares the fields first, and all methods then.

The include files follow this order:

- The system include
- The DIET include
- The local include

Warning, in C++ include the C++ files (cstdio and not stdio)

See the C++ header example file.

1.3 C

Each header must start with a:

```
#ifndef _FILENAME_H_
#define _FILENAME_H_
```

and ends with a *#endif*. A header must contain the prototypes of the functions linked to the file its name must be *fileName.h*. The implementation file must be *fileName.c* and must contain the implementation of the functions of the header file.

The include files follow this order:

- The system include
- The DIET include
- The local include

Warning, in C include the C files (stdio and not cstdio)

See the C header example file.

1.4 JAVA

A header must contain only one class (unless there are some inner classes/nested classes required) and its name must be *ClassName.java*.

In a class declaration, the public part comes first, then the protected one and last the private one. These keywords must always be, event if no protected item. Each inner part declares the fields first, and all methods then.

The imported files follow this order:

- The system import
- The DIET import
- The local import

See the JAVA header example file.

2 Naming

2.1 JAVA/C++

- All names must be English
- Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML). Acronyms must be uppercase.

```
Wrong:
class lapin
class 3nutella
class Lehomard
class HtmlParser

Right:
class HTMLParser
class AgentImpl
class Toto
class LinusTorvald
```

- Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

```
Wrong:
  addservice();
  chocolat();

Right:
  run();
  addService();
```

  Here is a set of useful verbs *add, run, launch, submit, send, receive, start, stop, get, set, has, is, emit, call*

- Member variable names should be short yet meaningful. The choice of a variable name should be mnemonic - that is, designed to indicate to the casual observer the intent of its use. Moreover, each member variable must be pre-fixed using the 'm' letter. m stands for member. Do not use 'this' to use the variables.

```
Wrong:
  float minimum;
  int size;
```

```
  char *name;

Right:
  float mminimum;
  int msize;
  char *mname;
```

- All instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. They must follow the members naming conventions. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.
  Example: int i, char* name, ...

- The names class constant variables and of ANSI constants, of macros and enum constants should all be uppercase with words separated by underscores. ANSI constants should be avoided, for ease of debugging. They must be in the langage

```
Wrong:
  #define SIZE           10
  static const char *Nom "TOTO"

Right:
  static const char *NOM "TOTO"
  const int TAILLE_CHAT  75
```

- Type names must be suffixed with _t:

```
Wrong:
  diet_type

Right:
  diet_type_t
```

- Getter/setter/boolean functions must start with get/set/is(or has). If they just return/set something, they must be implemented in the header file.

```
Wrong:
  bool local();
  int child();
  void size(int s);
```

```
Right:
  bool isLocal() {return mlocal;}
  int  getNumberChildren() {return mchildren;}
  void setSize(int s) {msize = s;}
```

2.2 C

The C code must follow the C naming rules: with lowercase only and words separated by underscores. The names of variables declared class constants and of ANSI constants, the names of macros and enum constants should be all uppercase with words separated by underscores. ANSI constants should be avoided, for ease of debugging.

```
Wrong:
  #define Size            10
  #define NOMDECONSTANTE "TOTO"

Right:
  #define TRACE_STRUCTURES 10
  #define SIZE             1024
```

Type names must be suffixed with $\_t$

```
Wrong:
  diet_type

Right:
  diet_type_t
```

3 Comments and Documentation

- All comments and the documentation must be in English.
- Both C++ types of comments are allowed.
  Example:
  - // This is a comment
  - /* This is a comment */
- Comments are on the line before, or on the same line as the code they comment.
- /**/ comments must not have code after them on the same line.

```
Wrong:
  int size /* size of object */, length /* length of object */;

Right:
  int size;   /* Size of the object */
  int length; /* Length of the object */
```

- Each method, function, variable, class must be commented with Doxygen comments. See `http://www.stack.nl/~dimitri/doxygen/` Some usefull doxygen keywords: *author, param, return, fn, brief, var, namespace, class, throws*

- Doxygen supports C, C++, JAVA, Python, IDL and PHP. If we use an other language (for WS), we need to specify how to comment automatically.
  Doxygen style to use:

```
C/C++/JAVA:
/**
 * \brief
 * \fn
 * \param
 */
```

4 Manuals

The developers must maintain a user manual and a technical manual up to date to explain to the users how to use the code and the future developers how to maintain the code.

5 Indentation

Do not use tabs inside the code, and wrap lines after the 80th character in a line. We do not like tabs because different editors and editor settings always lead to different displays. A tab is 4 spaces characters.

Lines have to be indented by two spaces for each code block level.

Due to cmake's syntax design choices, `CMakeLists.txt` tend to quickly feed the 80 characters width of a line. For `CMakeLists.txt` files it was thus chosen to use a 2 or 3 characters indentation width. If we strictly conform to the above limited-tabs recommendation, `CMakeLists.txt` should thus remain tab free.

In the CMakeLists.txt files the following standards are to be followed :

- The function/macro commands are to be written in lower case

```
WRONG add_Dependencies, ADD_DEPENDENCIES, ...
RIGHT add_library, add_custom_command, ...
```

- The parameters are to be in upper case with a space between the first parameter and the opening bracket, and no space between the command and the bracket. There is a space between the last parameter nad the closing bracket.

```
WRONG add_library ( TOTO ), add_executable(TITI)
RIGHT target_link_libraries ( myLib,
                              toto )
```

- If a variable is hidden to the user in ccmake, there must be an explanation in the CMakeLists.txt file.

6 Declaration

- Pointers must be initialized at the declaration
- For the asterisk of a pointer declaration and the `&` of a reference the space has to be put in between the `*` or the `&` and the type name.

```
Wrong:
  char* toto;
  int \& titi;

Right:
  char *toto = NULL;
  int \&titi = myObject();
\end{itemize}
\item All keywords (extern, static, export, explicit,
 inline, ...) and the return type must be on the line line the function
 declaration. In the header and the implementation.
\begin{verbatim}
Wrong:
  static int run();
  extern void jump();

Right:
  void
  myFunction();
  static const int
  run();
```

- Exceptions marker must be always present, and the throw statement must be on the line after the function name.

```
Wrong:
void
launch() throws()

Right:
void
launch()
throws()
```

- If the parameters do not fit on the same line, they can be split on various lines
- There must be a space between the opening brace and the closing parenthesis for functions and keywords, on the same line.

```
Wrong:
  void run(){
  for (;;){

Right:
  void run() {
  for (;;) {
```

- The closing brace must always be on a new line. If it ends a huge block, the starting point should be precised in a short comment.

```
Wrong:
  return;}

Right:
  return;
}// End: run()
```

- There is never any space between after an openning parenthesis and before a closing parenthesis.

```
Wrong:
  if ( a < b )

Right:
  int run(int a);
  while (a < 5) {
```

- Fragments of arithmetic expressions must be with spaces.

```
Wrong:
```

```
  a = 3<4;
  b = 5*2;

Right:
  a = 3 * 4;
  b = 5 * 2;
```

- Boolean links must have a space before and after.

```
Wrong:
  (3 * 4)&&(2 + 3)
Right:
 (3 < 4) && (2>3), run(3) && (a>2*4)
```

- There always are spaces around the = symbol.
- There must not be hard coded variables.
- In C, the declaration of the variables must always be at the beginning of a block.
- Use only C++ cast in C++.

```
Wrong:
  size = (int) getSize();

Right: (depending on the situation)
  size = static_cast<int>(getSize())
  size = dynamic_cast<int>(getSize())
  size = reinterpret_cast<int>(getSize())
  size = const_cast<int>(getSize())
```

- Use parentheses to group expressions.

```
Wrong:
  a + b & c
  if (a && b || c)

Right:
  if ((a && b) || c)
  (a + b) & c
```

- Always use curly braces when the body of a conditionnal statement.

```
Wrong:
  if (true)
    return;
```

```
  if (true)
    return a;
  else {
    i++;
    printError(msg);
  }

Right:
  if (true) {
    return;
  }

  if (true) {
    return a;
  }
  else {
    i++;
    printError(msg);
  }
```

- DO NOT USE GOTO OR LABEL