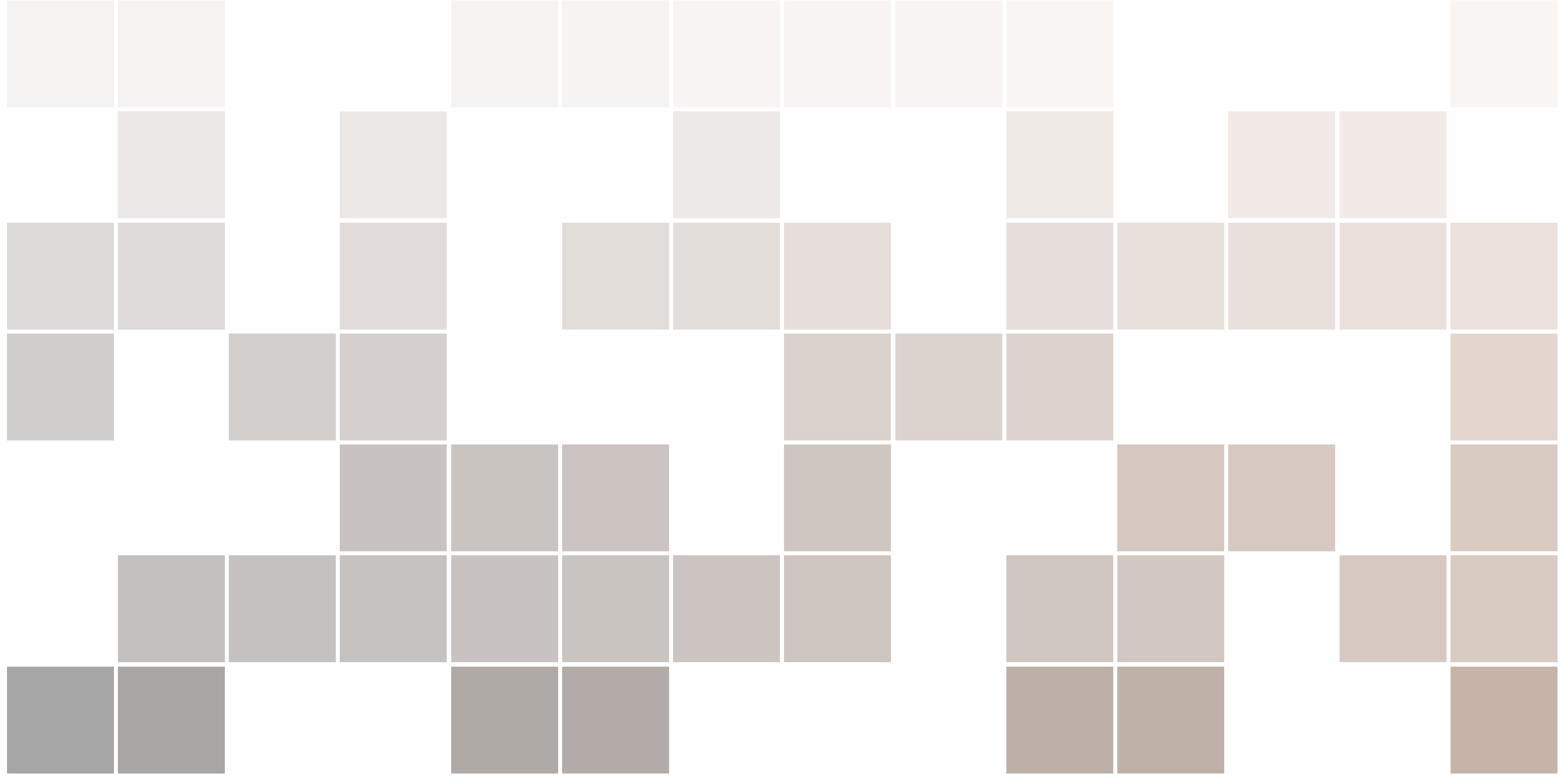
CamiTK Programming Guidelines

Version 1.0 - 21 June 2018

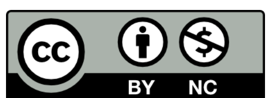
Team playing is fun only if played by the rules...



Releases			
Version	*Date*	*Author*	*Comment*
0.1	2016-02-16	C.Fouard	Creation
0.2	2016-07-25	E. Promayon	Review
0.3	2017-02-04	C. Fouard	XML and Latex Conversion
1.0	2018-06-21	E. Promayon	Review and final touch

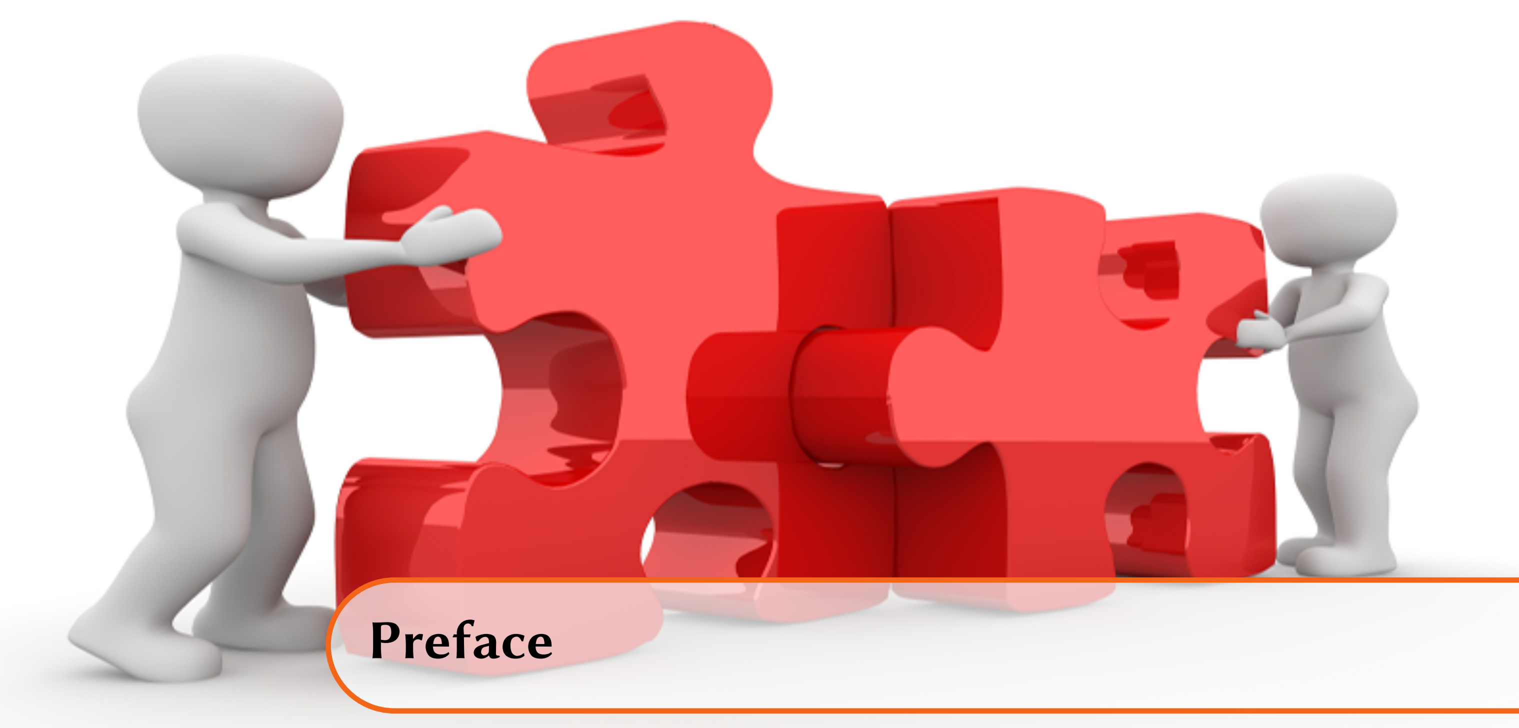
Copyright © 2018 Université Grenoble Alpes

HTTP://CAMITK.IMAG.FR



Licensed under the CC BY-NC 3.0
Creative Commons Attribution - Noncommercial 3.0 Unported License, the "License".
You may not use this file except in compliance with the License. You may obtain a copy of the License at http://creativecommons.org/licenses/by-nc/3.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, June 2018



Preface

CamiTK started as a two-persons project, and evolved to a more international audience. The purpose of this document is to capture what was the initial team coding guidelines in order to ensure a wider and more open participation of other developers.

This document first gives a brief introduction to CamiTK. You should refer to other documentions or training session to understand CamiTK more thoroughly.

The CamiTK process has always evolved since the first commit depending on the number of people in the CamiTK core developer team, their various (mad or practical) ideas, the number and requirement of the old and new users, the interaction with other developers and related software framework and the gradual clarification of the CamiTK goals.

As this is so well written in the VTK software process document: "By setting down the [coding guideline] in writing we seek not to hamper that evolution but rather to provide a defined basis for proposing further changes in order to continue its improvement. We seek a process that does not place needless or difficult restrictions on developers, encourages contributions from new developers, and does not require significant effort to centrally manage."

This Programming Guidelines are written for all the CamiTK Developers. This means, that, whatever CamiTK Extension you would like to develop, you should read and keep in mind this book.

Enjoy!

Céline Fouard, Jean-Loup Haberbush and Emmanuel Promayon

Contents

Preface 3

1 Introduction 7

1.1 Prototyping CAMI Applications 7

1.1.1 The goal of CamiTK 7

1.1.2 Prototyping vs Clinical routine (TRL) 8

1.2 Prerequisite 8

1.2.1 CamiTK General Organization 8

1.2.2 Intended Audience 10

1.2.3 A Free, Open Source, International and Multi-Platform Framework 10

1.2.4 OOP and C++ 10

1.3 Organization of this document 11

2 Generic C++ Conventions 13

2.1 Introduction 13

2.2 Code Genericity and Reutilisability 15

2.3 Code Presentation and Aesthetics 17

2.4 Naming Conventions 26

2.5 Header Files Conventions 33

2.6 Include Conventions 39

2.7 Cpp files conventions 52

2.8 SDK core files 57

3 CamiTK Specific Conventions 59

3.1 Genric Principles 59

3.2 Licensing 63

3.3 CEP organization 65

3.4 Components 67

3.5 Actions 72

Prototyping CAMI Applications
The goal of CamiTK
Prototyping vs Clinical routine (TRL)
Prerequisite
CamiTK General Organization
Intended Audience
A Free, Open Source, International and Multi-Platform Framework
OOP and C++
Organization of this document

1 — Introduction

> Computers are good at following instructions, but not at reading your mind.
>
> *Donald Knuth*

1.1 Prototyping CAMI Applications

CamiTK is a prototyping framework for rapidly building *Computer Assisted Medical Intervention* (CAMI hereafter) applications. It provides a SDK and applications in order to support the maturation cycle of your CAMI Applications or Medical Devices.

1.1.1 The goal of CamiTK

Computer Assited Medical Intervention (CAMI) is a complex multi-disciplinary field. CAMI research requires the collaboration of experts in several fields as diverse as medicine, computer science, mathematics, instrumentation, signal processing, mechanics, modeling, automatics, optics, etc. CamiTK is a modular framework that helps researchers and clinicians to collaborate together in order to prototype CAMI applications by gathering the knowledge and expertise from each discipline. It is an open-source, cross-platform, generic and modular tool written in C++ which can handle medical images, biomedical simulations, robot control and surgical navigation.

The goal of the CamiTK project is not only to gather the knowledge of the specialists from each area of CAMI, but also to gather their know-how by facilitating data exchange, software prototyping of applications and the production of reproducible results. The underlying objective is to bring clinicians, scientists and industry faster and more efficiently toward clinical validation and benefits to the patient. CamiTK's general design is inspired by the Component-Based Software Engineering (CBSE) concept and therefore encourages reusing rather than reinventing the wheel. Rapid prototyping of CAMI applications is made easy by assembling pre-built

CamiTK components (called CamiTK extensions) instead of continuously patching onto existing code.

1.1.2 Prototyping vs Clinical routine (TRL)

Using CamiTK core C++ SDK, the `camitk-wizard` that generate new CamiTK Extensions and the CamiTK workbench application (called `camitk-imp`), you can quickly design new algorithms or new process in order to shape CAMI algorithm or product. Once your proof of concept is defined, CamiTK provides another application (called `camitk-actionstatemachine`) to test your prototype over a larger number of realistic cases.

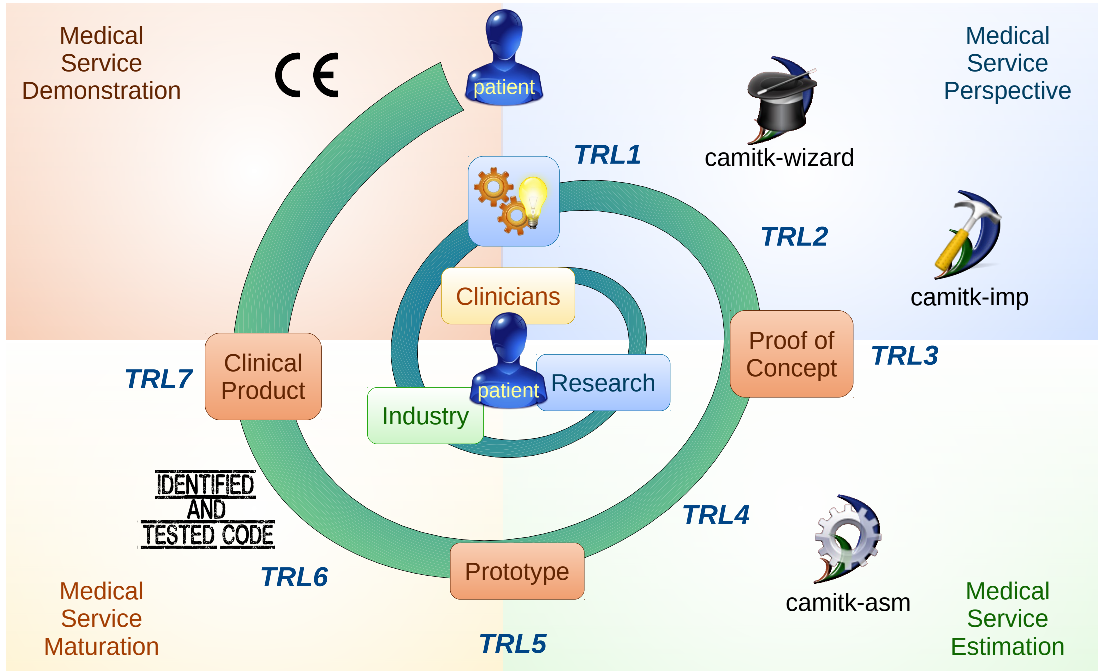


Figure 1.1: Innovation Spiral, Technology Readiness Levels and CamiTK-Applications.

Figure 1.1 (greatly inspired and reproduced by courtesy of MAXITHEC Center of Clinical Investigation - Innovative Technology) shows the maturation cycle of a new medical device and the associative innovation spiral in regards with three applications provided by CamiTK. CamiTK is dedicated to help you during the two first phases of the development of a new CAMI application or medical device: the development of the proof of concept and the prototype. It can not only help you to define the product perspectives but it can also support you during the estimation of the medical service of your prototype.

1.2 Prerequisite

1.2.1 CamiTK General Organization

CamiTK can be seen three different ways (hence the logo):

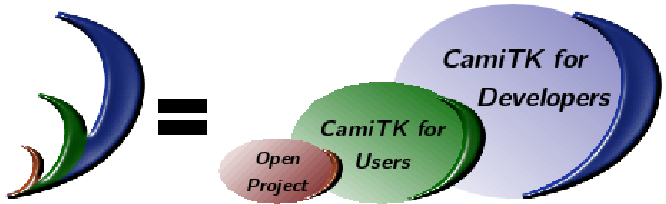


Figure 1.2: CamiTK logo seen as a 3 usability layers

- Open Project: CamiTK is an Open Source Project. The essential source code (CamiTK SDK) is not only available under the LGPL-v3 licence, but there are also a lot of collaborative tools, the software forge being the first of them, where anyone can help (you don't need to be a programmer for that, there are a lots of areas where improvements are needed, not only in the source code!).
- CamiTK for Users: CamiTK directly provides CAMI applications that can be useful to do something immediately. The installation is easy (there are packages for Linux and Windows for example).
- CamiTK for Developers: CamiTK is a framework, and developers can create their own pieces of puzzle thanks to the Open Source *Software Development Kit (SDK)*.

CamiTK extensively uses the Component-Based Software Engineering (CBSE) principle.It organizes the development of new features in three different levels (hence the logo):

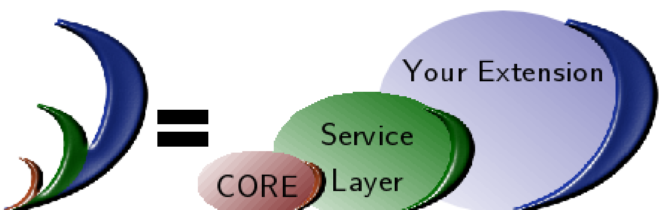


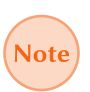
Figure 1.3: CamiTK logo seen as a 3 development layers

- CamiTK Core: this is where all the concepts are defined and implemented. The CamiTK experts are the only one who have to understand and master it well (the in-depth explanation of CamiTK Core architecture is beyond the scope of this document).
- CamiTK Service Layer: this is where the glue between the source code developed in/for a specific field and CamiTK is defined. All developers need a basic knowledge of the service layer. Note that the `camitk-wizard` application facilitates the work of the developer and that CamiTK-SDK itself is developed with simplicity in mind. You will acquire this knowledge little by little. Most of the usage are covered by default behaviors already implemented in CamiTK-SDK helper classes.
- CamiTK Extensions: this is where the domain-specific code is written. This is where most of the work has to be done in order to provide a new feature in CamiTK for a specific field. This is where you should focus your attention as a developer. And this is where your own expertise is mostly needed.

1.2.2 Intended Audience

This Programming Guidelines are written for all the types of CamiTK Developers. This means that you should read and keep in mind this book if your are developing a new CamiTK Extension. CamiTK applications users, like `camitk-imp` and `camitk-actionstatemachine` users do not need to write a line of code, so do not need to read this book. However, `camitk-wizard` users, as they will have to modify code provided by the wizard to create new extensions will need to follow these guidelines.

If you intend to develop within CamiTK-SDK Core, not only will you need to follow these guidelines, but you will also have to follow other, more restrictive guidelines and understand CamiTK SDK Core architecture in-depth. A dedicated documentation is in preparation.



These guidelines are not extensive (nor probably be ever entirely and satisfactorily completed!). Indeed, there is no mention of code versioning for example, which mainly depends on your institution/company/laboratory. Please refer to your institution guidelines for this subject. We advocate using gitlab or at least a bare git repository.

1.2.3 A Free, Open Source, International and Multi-Platform Framework

CamiTK aims at being able to reuse and share code between programmers (whether they are from your team or not). To ensure this modularity and know-how technological transfer, some elementary principle should be followed.

These guidelines are written with the *big picture* in mind. To this end, CamiTK-SDK and CamiTK-Extensions **must** be developed for an international audience and be compatible with Linux, Windows and MacOS. CamiTK contributors come from several country, and use several technologies, work environment and several mindsets. To be as generic as possible, CamiTK **must** therefore be written in English (Code, Comments and Documentation) and **must** be compatible/compilable on the most common Operating Systems (Linux, Windows, MacOS).

The *Free* and *Open Source* character of the CamiTK-SDK framework (labelled by the LGPL v3 licence) ensures you that any CamiTK-Extension you develop will be your property (whether or not you decide to distribute it under LGPL v3 licence or any other FOSS licence) and that you will always be able to compile and execute it in the future. Indeed, this specific licence ensures your independence regarding the code you link to CamiTK.

1.2.4 OOP and C++

The choice has been made to write the CamiTK Framework using the C++ language. It strictly follows the *Object Oriented Programming* (OOP hereafter) paradigm. In this case, C++ is used as a pure OOP language (more like the Java language does) and static and non OOP characteristics of the C++ language are most of the time avoided.

We recommend you to find an open class or a book if you wish to learn OOP and C++. Their

basic principles will not be described in this book. *Thinking in C++* from Bruce Eckel is a good book to start with (and freely available).

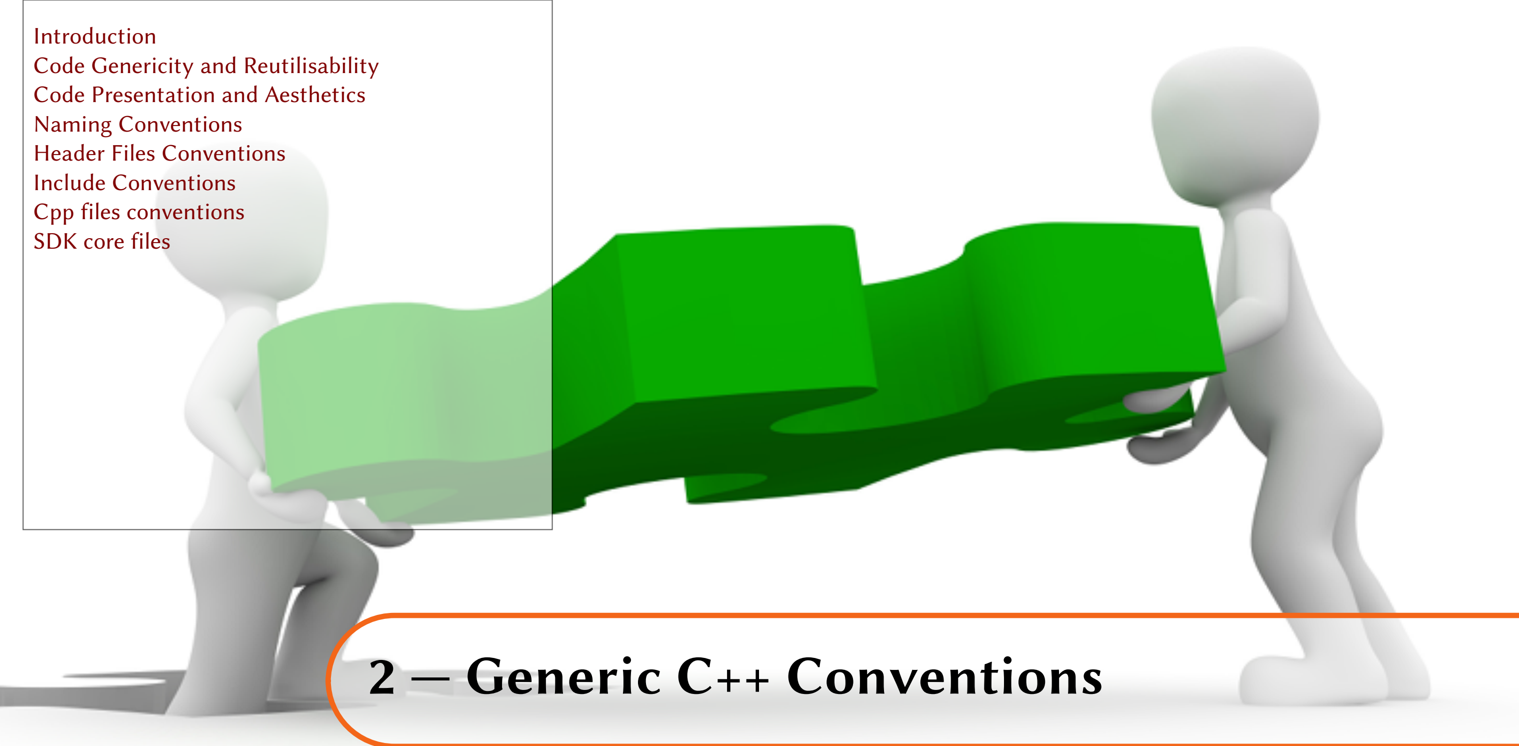
1.3 Organization of this document

Although basic principles of OOP and C++ are beyond the scope of this book (see previous paragraph), the first part is dedicated to general C++ conventions adopted for CamiTK programming. Indeed, for the code to be easily reusable and maintainable by a large community of programmers, conventions must be followed to ensure that reading anyone's code will lead to the correct interpretation of the initial programmers intentions. As the Qt documentation has expressed it very well: *"Keep in mind that code is read much more often than written."*

A second part explains what is expected as a correct usage of the CamiTK-SDK service layer.

In this document, each guideline is numbered with **CPG** (for CamiTK Programming Guideline) and a 3 digits number, so you can refer to it in your forum messages or bug reports.

All along this guide, a **must** requirement must be followed, a *should* is a recommendation and a can is a general guideline.



Introduction
Code Genericity and Reutilisability
Code Presentation and Aesthetics
Naming Conventions
Header Files Conventions
Include Conventions
Cpp files conventions
SDK core files

2 — Generic C++ Conventions

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> *Martin Fowler*

2.1 Introduction

C++ is a very permissive language. It is thus very powerful and allows you to code in any way you like to. However, as one of the main goals of CamiTK is to share know-how between researchers and developers, the basic rules are to make generic and reusable code. This idea should always guide you as you write your code. this chapter gives you some (non exhaustive) mandatory guidelines to make your code more generic and easier to use by any other CamiTK developer.

As you probably know by now, CamiTK is mainly based on Qt and VTK. After reading the current document, we advise you to read the two following coding conventions, on which some parts of this document is based:

- the Qt coding convention: it contains useful information that will improve your understanding.
- the VTK Software Process: it also contains extra bits about how the VTK software life-cycle is managed, including information about continuous integration.
- the KDE coding style: KDE is a big project with a lot of contributors, and like CamiTK it is based on Qt and uses CMake. The KDE community process is a great example of how collaboration are organized between a lot of developers.
- The Google C++ Style Guide: it contains a ton of well documented information with detailed rationale. It is worth reading it for improving your general knowledge about C++.

If you want to go further and learn more best practices, C++ coding guidelines, rules to develop

more secure programs, or just pieces of wisdom from experience C++ authorities, here is a list of helpful references:

- Overview of the New C++ (C++11/14). Scott Meyers. 2015.
- Effective C++ book series (Effective C++, Effective Modern C++, More Effective C++, and Effective STL). Scott Meyers.
- C++ Coding Standards by Herb Sutter and Andrei Alexandrescu.
- Joint Strike Fighter Air Vehicle C++ Coding Standards, December 2005, Lockheed Martin Corporation
- MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, June 2008, MIRA Limited
- SEI CERT C++ Coding Standard, 2016. The current updated online version is also available here.
- High Integrity C++ Coding Standard Version 4.0. 2013

This chapter will not be easily read by non C++ developers as the CPGG described here are all about C++. So if you do not understand the following CPGG, you may want to find a book or Open Classroom on C++ coding.

2.2 Code Genericity and Reutilisability

CPG-001: *Default language is English*

Summary
All code, comments and documentation **must** be in English

Best Practice

```
// Whether they are on one line
/* Or on several lines...
   All comments should explain the programmer's intentions in English
*/
```

Rationale

As CamiTK is an Open Source *International* project, all code comments and documentation **must** be in English. Translation can be provided in specific files, but the default language is English. Simple English is easier to understand and correct if needed (commenting code is not a work of literature): use simple sentence constructions, be direct, stay simple and factual.

CPG-002: *No absolute file name*

Summary

Absolute file names (like e.g. `/some/path/to/filename.vtk`) **must** be avoided.

Rationale

The goal of CamiTK is to share know-how between several users through several computer and operating systems. An absolute file name prevents this kind of use as the absolute file path will not work on another computer and/or another account. You can use Qt Resource System [a] to embed specific required files within the code.

[a] (http://doc.qt.io/qt-5/resources.html)

2.3 Code Presentation and Aesthetics

Even though code presentation may not influence how code compile and execute, it is very important that anyone reading your code understands it without too much effort. Other guidelines are also important for this matter, but code presentation is the easiest and most straightforward way to ensure it. All the following code presentation conventions can be arranged and checked thanks to the utility Artistic Style automatic code formatter.

The easiest way to apply the coding conventions described in this section is to run *Artistic Style* (`astyle`) on your code using the following option:

```
astyle --style=java \
       --break-closing-brackets \
       --add-brackets \
       --unpad-paren \
       --pad-oper \
       --pad-header \
       --align-pointer=type \
       --indent-switches \
       -R *.cpp *.h
```

See below for explanations about these options or check the `astyle` documentation

CPG-003: *4 spaces instead of one tab*

Summary

Code indentation **must** use 4 spaces instead of one tab everywhere. You *should* not use tab character for indentation. There *should* be no trailing whitespace and no empty line at end of files.

Rationale

As CamiTK uses CMake as project generator, you are free to use your favorite IDE and/or your favorite code editor to write your code. However, indentations **must** use 4 spaces everywhere. There **must** be no indentation for namespaces. Special characters like TAB and page break **must** be avoided. Please refer to the corresponding wiki page to correctly setup your editor.

This is the default for `astyle` (same as using the `-indent=spaces=4`) and will be automatically applied when using `astyle`.

CPG-004: *Bracket Style: Java*

Summary

The bracket style *should* be Java: open bracket at the end of the line and closing bracket at the beginning of the last line

Best Practice

```
1 int methodName(bool parameterName) {
2     if (isBar) {
3         doSomething();
4         return 1;
5     } else
6         return 0;
7 }
```

Rationale

You should use java style for bracket positions. You can make your editor format automatically or use the `astyle` option `-style=java` to automatically apply this CPGG.

CPG-005: *Space Between Operators*

Summary

To ease reading, ther *should* be space between operators and operands, and parenthesis for each condition.

Best Practice

```
1 if (( a == b ) || ( a == c ))
```

Rationale

Spaces ease the reading of the code. In addition, event though boolean operators have priority rules, it is a good practice to write unambiguous code for the reader. You can use the `astyle` option `-pad-oper` to automatically apply this CPGG.

CPG-006: *Breaks closing headers from their immediately preceding closing brackets*

Summary
To ease reading, all the closing headers ('else', 'catch', etc.) *should* start on a new line.

Best Practice

```
void Foo(bool isFoo) {
    if (isFoo) {
        bar();
    }
    else {          // Note the else starts on a new line
        anotherBar();
    }
}
```

Rationale

Readability: this should be easier to read and not miss the closing header blocks. You can use the `astyle` option `-break-closing-brackets` to automatically apply this CPGG.

CPG-007: *Add brackets to unbracketed one line conditional statements*

Summary
To ease reading, all the one line statements (e.g. 'if', 'for', 'while', etc.) *should* use a bracket.

Best Practice

```
if (isFoo) {        // There are brackets even if the statement is only on
    isFoo = false;  // one line
}
```

Rationale

Readability: this should be easier to see that there is one statement following the conditional statement. You can use the `astyle` option `-ad-brackets` to automatically apply this CPGG.

CPG-008: *Insert Space Padding Between a Header and the Following Paren*

Summary

There should be a space between header (i.e., a C++ test statement for example) and the following parenthesis.

Best Practice

```
1 if (isFoo((a+2), b)) {  // note the space between the "if" and the parenthesis
2     bar(a, b);           // note the absence of space between method name and parenthesis
3 }
```

Rationale

Readability: this makes it easier to see the statement and conditional. You can use the option `-pad-header` in `astyle` to automatically apply this CPGG.

CPG-009: *Attach pointer and reference operator (*, &)*

Summary

The pointer or reference operator should be attached directly to the variable type.

Best Practice

```
Component* foo1;
Viewer& foo2;
```

Rationale

Readability: this should be easier to see if all the pointer and reference operators are attached to the left (i.e. the type). This can also avoid confusion between the reference operator and the binary operator &. You can use the `astyle` option `-align-pointer=type` to automatically apply this CPGG.

CPG-010: *Indent Case Statement in Switch Blocks*

Summary

Indent `switch` blocks so that the `case X:` statement are indented in the `switch` block. The entire case block is indented.

Best Practice

```
switch (foo) {
    case 1:
        a += 1;
        break;

    case 2: {
        a += 2;
        break;
    }
}
```

Rationale

Readability: this should be easier to read and distinguish between the switch blocks. You can use the option `-indent-switches` in `astyle` to automatically apply this CPGG.

2.4 Naming Conventions

Most of the CamiTK naming conventions follow the Java naming convention. The following text is highly inpired by (https://en.wikipedia.org/wiki/Naming_convention_%28programming%29).

CPG-011: *Directory Names*

Summary
Directory, package names *should* be in lower case (with no underscore or space or any type of separators between words) and use only 26 unaccentuated lowercase characters of the roman alphabet (no accents, no language specific, only the lowercase letters of the ASCII table).

Rationale

There are many things to consider (portability, ascendant compatibility, multi-OS support, etc.).

CamiTK directory names follow an old Unix idea, that proved later to be save when using the same tree on Microsoft Windows. It also follows the more modern Object Oriented Programming principle, where a directory is used/can be directly used as a namespace: the directory name is thne used as the namespace's name.

It is also on the same wavelength as the Java convention *"Package names are written in all lower case to avoid conflict with the names of classes or interfaces"*, from the Java documentation (Oracle).

Therefore, CamiTK conventions do not allow dash (-), uppercase, space or any separator in the directory names.

CPG-012: *Class Names*

Summary

Class names *should* be `UpperCamelCase`, with the first letter of every word capitalized. Use whole words (avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as URL or HTML). Use only ASCII characters (no accents).

Best Practice

```
class ImageComponent;
class MeshComponent;
```

Rationale

CamelCase (also called camel caps or medial capitals) is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter. To know more on CamelCase, see (http://en.wikipedia.org/wiki/CamelCase) As the filenames are automatically deduced from the class names, all file names **must** contain only ASCII characters.

This deffers from the VTK naming convention.

CPG-013: *Method Names*

Summary

Methods *should* be verbs in `lowerCamelCase` or a multi-world name that begins with a verb in lowercase ; that is, with the first letter lowercase and the first letters of any subsequent words in uppercase.

Best Practice

```
void run();
void runFast();
Image * getImage();
bool isOpened();
```

Rationale

When reading the name of the function, the reader should understand what the function does. As in Object Oriented Programming, functions are most of the time *actions* on objects, their names sould express verbs. With editors auto-completion, one does not have to restrain oneself on the length of a method name, so it should be as explicit as possible. Method returning a boolean, which indicates the state of an instance, can begin by `is`.

CPG-014: *Accessor / Mutator*

Summary

Class data should be encapsulated, i.e., each attribute (instance variable) should be declared as private. You should use an accessor and mutator to enforce encapsulatoin and control the access to the data (unless very very very rare specific cases). The accessor *should* be named `setAttributeName()` and mutator *should* be named `getAttributeName()` (where `AttributeName` is... the name of the attribute).

Best Practice

```
public:
    void setAttributeName(type newValue);
    type getAttributeName() const;
private:
    type attributeName;
```

Rationale

Public attribute are allowed only in exceptional situations (that you probably won't meet in CamiTK code, that can be different in a library). For better clarity, an accessor for a boolean attribute can also be called `isSomething()` or `hasSomething()`

This differs from the Qt naming convention (where the `get` prefix is omitted).

CPG-015: *Variables*

Summary

Local variables, instance variables and class variables are also written in `lowerCamelCase`. Variable names *should* be meaningful. The choice of a variable name should be mnemonic, useful and meaningful (that is, designed to indicate to the casual observer the intent of its use and clearly and quickly self explain what it is there for).

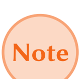
Each variable declaration *should* start on a new line. Please avoid abbreviations and cantraction of names (e.g., `strPtr` should be renamed `structurePointer`). Do not use short names, except for counters and temporary variable whose purpose is very obvious. Variable and method names **must** always start with a lowercase letter.

Best Practice

```
1 float width;
2 QString arbitraryLength;
```

Rationale

Variable names *should* not start with underscore (_) or dollar sign ($) characters. This is in contrast to other coding conventions that state that underscores should be used to prefix all instance variables. One-character variable name 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.



(from (https://msdn.microsoft.com/en-us/library/hh289390.aspx)) "The C++ language standard reserves the use of identifiers that begin with an underscore character for implementations such as libraries. Do not use names beginning with an underscore in your code."

Calling a variable with metasyntactic words (`foo`, `bar`, `fobar`, `toto`, `titi`, `tata`, `tutu`, `truc`, `bidule`, `stuff`, `azerty`, etc.) or words like `tmp`, `temp`, etc. is to be avoided at all cost !

CPG-016: *Constants and Enums*

Summary

There *should* be **no constant** declared in public class fields. Public constant values expose the API. If a modification is required it will be very hard to propagate it to developer codes. If you really need to define a constant, use a static method (e.g. `C::getConstantValue()`). This way if the value itself must be changed, your API won't disturbed anything. But, again, do you really need this constant value?

Enumerated values *should* be written inside an enum C++ structure. All enumerated values must be in uppercase characters separated by underscores if needed. Constant names may also contain digits if appropriate, but not as the first character. As for all variables, constants must have meaningful names.

Enums are good. Enum values will be replaced by the compiler at compile time, resulting in faster code and can be easily commented. Using `#define` *should* be prohibited. They are difficult to manage regarding to `namespace` safe (and look ugly) [a] .

[a] (https://wiki.qt.io/Coding_Conventions#Aesthetics)

Best Practice

```
// in the C.h header
class C {
...
public:
    /** \enum MyEnumType gives an example of how to use enum (and easily document it!).
     */
    enum MyEnumType {
        VALUE_1,        ///< this represent the VALUE_1 enum
        ANOTHER_VALUE,  ///< each enumerated value can be documented separately
        ...,            ...
        LAST_VALUE      ///< This is C++: las value does not have any trailing comma
    };

    ...
    /// if you really (really) need to declare a constant, use a static method
    static const int getMyConstant();
}

...
```

Rationale

Constant declared in the global scope may cause strange (and difficult to debug) behaviors when used in dynamic libraries (e.g. in a CamiTK extension). Again: do not use constant!

2.5 Header Files Conventions

A particular attention must be paid for header files as they may be included in other files and deeply impact other projects.

CPG-017: *A Header File is a Class Declaration*

Summary

A header file *should* declare only one class and **must** have the same name that this class.

Rationale

A header file *should* be extended by `.h`. All definitions *should* reside in source files. The header file *should* only declare an interface (i.e. a class with its attributes and method signatures).

Header files may contain more than one class when inner classes or private classes are used (but it should remain exceptional and follow the PIMPL design pattern).
There may be inline methods in `.h` header files.

CPG-018: *Header Guards*

Summary

There **must** be header guards in all header files. Header guards should not start with an underscore [a] .

[a]See DCL51-CPP.

Best Practice

```
#ifndef NAMEOFMYFILE_H
#define NAMEOFMYFILE_H
// Here should be the content of the header file
...
#endif // NAMEOFMYFILE_H
```

Rationale

Multiple inclusions of a header file can have unexpected and unwanted effects. Indeed, if you include `A.h` in `B.h` and in `C.h`, and also include `B.h` in `C.h`, then `A.h` will be included twice in `C.h`. To avoid this, guarding pre-compiler code **must** be added to each header file.

CPG-019: *Header File Anatomy*

Summary

In order to enhance readability, please use the following order for your headers:

- `// copyright and licence`
- header guards (see ())
- includes (see ())
- class name (see ())
- enum and typedefs
- public methods (constructors firsts)
- public signals/slots
- protected methods
- protected signals/slots
- private members
- inline method (a minimal number, inline should be well justified)

Best Practice

```
/*****************************************************************************
 * $CAMITK_LICENCE_BEGIN$
 *
 * CamiTK - Computer Assisted Medical Intervention ToolKit
 * (c) 2001-2018 Univ. Grenoble Alpes, CNRS, TIMC-IMAG UMR 5525 (GMCAO)
 *
 * Visit http://camitk.imag.fr for more information
 *
 * This file is part of CamiTK.
 *
 * CamiTK is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License version 3
 * only, as published by the Free Software Foundation.
 *
 * CamiTK is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License version 3 for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License version 3 along with CamiTK.
 * If not, see <http://www.gnu.org/licenses/>.
 *
 * $CAMITK_LICENCE_END$
 ****************************************************************************/

#ifndef CAMITK_COMPONENT_H
#define CAMITK_COMPONENT_H

// -- Core stuff
#include "InterfaceNode.h"
...
// -- QT stuff
#include <QPixmap>
...

// -- vtk stuff
```

```
38 #include <vtkWindowLevelLookupTable.h>
39 ...
40 // -- vtk stuff Classes
41 class vtkActor;
42 ...
43
44 namespace camitk {
45 // -- Core stuff classes
46 class Geometry;
47 class Slice;
48 class Viewer;
49 class Frame;
50
51 /**
52  *
53  * @ingroup group_sdk_libraries_core_component
54  *
55  * @brief
56  * A Component represents something that
57  * could be included in the explorer view, the interactive 3D viewer,
58  * and that could have or not a contextual
59  * popup menu (open by a right click in the explorer),
60  * a property dialog (to change some properties)
61  * Thus, a Component inherits from many abstract classes.
62  * A Component can only have one implemented representation.
63  *
64 ...
65  */
66 class CAMITK_API Component : public InterfaceProperty, ... {
67     Q_OBJECT
68
69 public:
70     /** \enum Representation The different representation that can be implemented to represent this
        Component in the InteractiveViewer.
71       * use getRepresentation() to get the information about a specific Component.
72       * \note the representation cannot be NULL; if a Component
73       * does not have any representation, then getRepresentation() should return NO_REPRESENTATION (
        default).
74       */
75     enum Representation {
76         GEOMETRY, ///< this Component can be displayed as a GEOMETRY
77         SLICE, ///< this Component can be displayed as a SLICE
78         NO_REPRESENTATION      ///< this Component has no representation implemented
79     };
80
81     /** Component constructor for top-level component (please use the other constructor for sub-level
         components).
82      *  parentComponent is set to NULL (=> isTopLevel() will return true).
83      *  @param file the file to get the data from
84      *  @param name the Component name
85      *  @param rep the representation concretely implemented by this Component (default=
        NO_REPRESENTATION)
86      */
87     Component(const QString & file, const QString & name, Representation rep = NO_REPRESENTATION);
88
89     /** Component constructor for a Component that is a child of another Component
90      *  You should not use this constructor for a top-level component.
91      *  @param parentComponent the parent Component
92      *  @param name the Component name
93      *  @param rep the representation implemented by this Component (default=NO_REPRESENTATION)
94      *  @throws AbortException if parentComponent is NULL.
95      */
96     Component(Component *parentComponent, const QString & name, Representation rep =
        NO_REPRESENTATION) throw(AbortException);
97
98     /** default destructor.
99      *  The Component class destructor is automatically called after the inherited destructor has
        finished (C++ standard).
00      *  This destructor delete all the children, clear all this component's viewer list, delete all
        helper class instance
01      *  (Geometry, Slice or Frame), which in turns will delete the VTK pipeline and any additional
```

```
         prop,
     *  and finally delete and all additional CamiTK Properties.
     *  @see deleteChildren()
     */
    virtual ~Component();

    /** return the type of representation concretely implemented by this Component in the
     InteractiveViewer.
      * \note if a Component does not have any representation, then getRepresentation() returns
     NO_REPRESENTATION (default).
      */
    Representation getRepresentation() const;

    /// return true if this component is a top-level component
    bool isTopLevel() const;

    /// get the parent component
    virtual Component * getParentComponent();

    ...

protected:
    /// myGeometry is the 3d representation of this Component, the Component delegates all
     InterfaceGeometry activity to myGeometry (delegation pattern)
    InterfaceGeometry * myGeometry;

    /// mySlice is the slice representation of this data component, the Component delegates all
     InterfaceBitMap activity to mySlice (delegation pattern)
    InterfaceBitMap * mySlice;

    ...

private:
    /** @name Instance members
      */
    /// method called in constructors for general initialization
    void init();

    /// the service implemented to be represented in the InteractiveViewer
    Representation myService;

    /// my name
    QString myName;

    ...

    /** @name Static members
      * All the static member (manage the application-wide list of Component + the application-wide
     current selection + viewers)
      */
    ///@{
    /// set of used viewers
    static QSet<Viewer*> allViewers;
    ///@}

};


// -------------------- isSelected --------------------
inline bool Component::isSelected() const {
    return isSelectedFlag;
}
...
}

#endif
```

Rationale

Respecting this guideline fasten header file reading.

2.6 Include Conventions

CPG-020: *Angle-bracket includes convention*

Summary

You *should* use quoted form includes for project local file includes and angle-bracket includes for other libraries includes.

Best Practice

To include a file from another library (or from CamiTK in a CEP), use

```
#include <Application.h>
```

To include a file local to your CEP, use

```
#include "MyOwnClass.h"
```

Rationale

Quoted form include instructs the preprocessor to look for include files in the same directory of the file that contains the `#include` statement, and then in the directories of any file that includes that file. The preprocessor then searches along the path specified by the `/I` compiler options. Angle-bracket form include instructs the preprocessor to search for include files first along the path specified by the `/I` compiler option.

Use angle-brackets (e.g. `#include <XXX>`) to include headers from external libraries (e.g. Qt, VTK). This is also true about CamiTK API when you are developing outside the CamiTK core library (i.e. most of the time !). In headers, it is strongly advised to use angle-brackets and this becomes mandatory if your headers are exported to other projects.

Use double quotes (e.g. `#include "XXX"`) in implementation files that are in the same project/directory.

CPG-021: *Include Statement Order*

Summary
Include statements in implementation files should generally appear in logical order from the most generic / most important class to the less important class (i.e. the class that are used the most are on top), and grouped by type.

Best Practice

We recommand the following order:

- CamiTK includes
- Qt includes
- VTK includes
- STL includes
- other includes (specific API directly related/required by your project)

CPG-022: *Include Path*

Summary

Include statements **must** never use an absolute path. For example, the following statement:

```
#include <C:/Programms/Dev/CamiTK/include/Application.h>
```

is *wrong* while

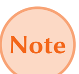
Best Practice

```
#include <Application.h>
```

is right

Rationale

Your code might be compiled on other computers, even on other Operating Systems. The tree organization of header and library files will probably be completely different on another computer. Let CMake handles path finding.



Although it is possible to have relative path for inclusion (`#include "../include/MyOwnFile.h"`), it is deprecated as this tree organization *should* be specified in the `CMakeLists.txt`.

CPG-023: *Minimize the number of includes in header files*

Summary

Do not #include a definition when a forward declaration is possible. Use forwarded declarations in header files judiciously. Only include the header files that are really necessary (i.e. for classes of which the structures are used in the header, for super class and dependencies). Try to include all other headers in the implementation file.

However, for a class that constitutes an API and will be exported and used by other projects (e.g. CamiTK API will be exported in CEP), try to avoid forward declaration of entities defined in another API (e.g. Qt)

Best Practice

```
class C; // forward declaration of class C, it just says to the compiler "C is a class"

class A {
    C* myOwnC;  // for the compiler, a pointer to a class instance is ok
}
// You may then include C.h in your file A.cpp
```

Rationale

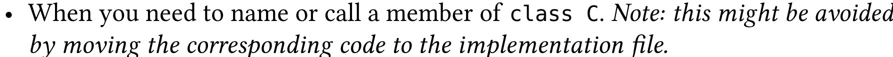
You *should* prefer forward declarations whenever possible for the 2 following reasons:

- Cyclic dependency: you cannot include `C.h` in `A.h` and include `A.h` in `C.h`
- Transitive dependency: more importantly, if you include in your header `A.h` a header `CFromAnotherLib.h` from an external library, then people using your code and including your `A.h` header will also have to include `CFromAnotherLib.h` and thus add the other library include path in their project where, if you had use foward declaration, a link path whould have been sufficient. This is more tricky to configure and slows down the compilation

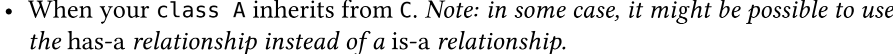
The logic here is that forward decaration is good if you don't export your own header file to be used outside your project and if its usage will not automatically imply to include the class you forward declared.

You may need a full definition of a `class C` in `class A` in only 3 main cases:

- When you need to know the size of a `C` object (for example when `A` has a member of `class C`). *Note: this might be avoided by using smart pointers*



- When you need to name or call a member of `class C`. *Note: this might be avoided by moving the corresponding code to the implementation file.*
- When your `class A` inherits from `C`. *Note: in some case, it might be possible to use the* has-a *relationship instead of a* is-a *relationship.*



For more details, see item 59 of Sutter and Alexandrescu's "C++ Coding Standards: 101 Rules, Guidelines and Pest Practices"

CPG-024: *No 'Using namespace' in header files*

Summary

There should be no

```
using namespace camitk;
```

nor

```
using namespace anything;
```

in header files

Best Practice

Within header files always use `namespace::type` whatever the namespace.

Rationale

Namespaces deliver the powerful advantage of unambiguous name management. Most of the time, you will use no namespace or the `std` or `camitk` namespace. This is why namespace using declarations and directives are really useful.

However, if a using namespace statement is present in a header file `C.h`, anyone including `C.h` will get the corresponding namespace. This defeats the logic and purpose of namespace and can lead to problems that will be difficult to tackle.

Written in your header file, any user who wants to include your file will have a using namespace something, declared before is own code (as it will be declared in the include.h).

For more detailed explanation, see item 59 of Sutter and Alexandrescu's "C++ Coding Standards: 101 Rules, Guidelines and Pest Practices".

CPG-025: *About windows.h*

Summary

As you can guess, the header file `windows.h` contains declarations that are specific to Microsoft Windows platform. Thus `windows.h` **must** never be included anywhere.

Best Practice

Extract the needed lines from `windows.h` (and only them) and include them referably in your `.cpp` files.

Find another solution that is not based on `windows.h`. Most of the time the OS independant Qt library provides system level features that will ensure an equivalent, if not better, to the required Microsoft Windows specific function/class. For instance, Qt is able to handle USB port communication, read/write access in the windows registry, manage files and directories, network communication, thread, etc. Have a look at the Qt documentation, it is a well-documented secure and stable framework.

Rationale

When using third party library, we may be tempted to include the file `windows.h` (or include header files that include `windows.h`).

Here is why we must resist this temptation:

1. Platform-specific file
   First, as `windows.h` is a file specific to Windows platform and CamiTK is by essence targeting multi-platform. You should make sure that if including `windows.h` is truly what you need, several precautions are taken (e.g., using `#if define(_WIN32)` blocks). In addition, you should be aware that a library including the file `windows.h` may not work on other platforms (Linux, Unix, MacOS, etc.) and produce errors in the continuous integration process. Such a library should only be used if absolutely no multi-platform equivalent exist. Your CEP will consequently be stuck on the Windows platform. This defeats one of the main CamiTK objective.
2. Redundant definition of Macros (`ERROR`, `min`, `max`, etc.)
   The main reason why this particular file `windows.h` must not be included into CamiTK header files is that it defines many redundant macros like for example `ERROR` which replaces every occurrence of the word `ERROR` in all the following files by a windows predefined code during compilation. As the word `ERROR` may be used for example in error messages, this will lead to many compilation errors on Windows. This is an example, other macros, such as `MIN` and `MAX`, may also generate other conflicts.

CPG-026: *declspec(dllexport) declarations*

Summary

Add a `_declspec(dllexport)` declaration in your extension (wheather it is a Component or an Action Extension) when you want to allow other Extensions (wheather they are Component or Action Extensions) to use one of your class.

Best Practice

1- Create a file `MyExtensionNameAPI.h` file whith the following code:

```
#ifndef MYEXTENSIONNAME_API_H
#define MYEXTENSIONNAME_API_H

#if defined(_WIN32) // MSVC and mingw
#ifdef COMPILE_MYEXTENSIONNAME_API
#define MYEXTENSIONNAME_API __declspec(dllexport)
#else
#define MYEXTENSIONNAME_API __declspec(dllimport)
#endif
#else
// for all other platforms MYEXTENSIONNAME_API is defined to be "nothing"
#ifndef MYEXTENSIONNAME_API
#define MYEXTENSIONNAME_API
#endif
#endif // MSVC and mingw

#if defined(_WIN32) && !defined(__MINGW32__) // MSVC only
#pragma warning( disable : 4290 )
#endif // MSVC only

#endif // MYEXTENSIONNAME_API_H
```

2- Include this `MyExtensionNameAPI.h` file in the header files containing classes you want to export and declare you class in the following way:
`class MYEXTENSIONNAME_API MyClassName ...`

3- Add the following line in the camitk_extension macro parameters of the `CMakeLists.txt` of your extension:
`DEFINES COMPILE_MYEXTENSIONNAME_API`

Rationale

On Windows Platform, during link-edition, the linker needs to know which functions are exported from one dynamic library to another. In CamiTK, all Component or Action Extensions are dynamic libraries, so they must declare code that will be used by other Extensions.

CamiTK is multi-platform. So, even if you develop on a Unix-like platform and do not have

> this export declaration problem to handle on your system, you code will very probably be used by CEPs on a Windows Platform. The proposed code will keep it fully functional on your platform, but make it usable for everyone.

CPG-027: *Copyright and Licence*

Summary

Each header and implementation files should start with the copyright and licence statement, even if they differ from the default LGPLv3 licence.

Best Practice

Here is the default LGPLv3 licence:

```
/************************************************************************
 * $CAMITK_LICENCE_BEGIN$
 *
 * CamiTK - Computer Assisted Medical Intervention ToolKit
 * (c) 2001-2018 Univ. Grenoble Alpes, CNRS, TIMC-IMAG UMR 5525 (GMCAO)
 *
 * Visit http://camitk.imag.fr for more information
 *
 * This file is part of CamiTK.
 *
 * CamiTK is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License version 3
 * only, as published by the Free Software Foundation.
 *
 * CamiTK is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License version 3 for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License version 3 along with CamiTK.
 * If not, see <http://www.gnu.org/licenses/>.
 *
 * $CAMITK_LICENCE_END$
 ***********************************************************************/
```

Note that

- The year interval should be up to date
- the *CAMITK_LICENCE_BEGIN* and *CAMITK_LICENCE_END* lines are mandatory and are useful to updte the year for instance using regexp recursive replace.

Here is a copyright and licence example for non-open source code (it might not be the best one or it might not suit your need):

```
/************************************************************************
 * $YOUR_PRIVATE_CEPNAME_LICENCE_BEGIN$
 *
 * Copyright (c) 2018-2xxx xxxxxxxxxxxxxxxxxxxxxxxxxx
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of xxxxxxxxxxxxxxxxxxxxx
 * You shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with xxxxxxxxxxxxxxxxx
```

```
12  *
13  * This code can not be copied and/or distributed without the express
14  * permission of the copyright holder.
15  *
16  * $YOUR_PRIVATE_CEPNAME_LICENCE_END$
17  ***************************************************************************/
```

Rationale

Licence type and copyright notices should appear at the top of each C++ file or alongside XML or data files. If some of your files are distributed outside the protected environment of you university or company, people can only know the licence by looking at these lines. Our advice and recommendation is to use LGPLv3 whenever possible. When in doubt, you should speak about this with your project/team and see if you have to use a non-open source copyright and licence notice or not.

The default licence for CamiTK and the code generated by the CamiTK-Wizard is LGPLv3.
It might not be what you want.

CPG-028: *Doxygen Documentation*

Summary

All class **must** be commented for Doxygen.

Best Practice

```
/** Class:
*  Purpose of this class: a useful comment helping the next developer
*  to fully understand the intent of the original developer explaining
*  why and how this class was developped.
*/
class MeaningfulName {
public:
    /// One sentence to explain what this method does, returns
    ///   and what these paramters are
    void doSomethingMeaningful ();
protected:
    ///one sentence to explain what this field is, does, what is its unit...
    int keyToEverything;
}
```

Rationale

The purpose of CamiTK Extension project is to be re-used by other users, but also other developpers. Thus all the class must be commented. CamiTK uses Doxygen to produce API Documentation, so this format must be adopted by any CamiTK developer.

The @authors and @date fields are not needed. If the class implements published research work, please add the reference to the paper in the Doxygen comments as well as in the description fields (when available). With the future automatic cataloging this can be important for the dissemination of knowledge, which is an objective of the CamiTK project.

2.7 Cpp files conventions

CPG-029: *Core file extension must use .cpp file name extension*

Summary

The C++ core files **must** have a `.cpp` file name extension.

Best Practice

```
// Header file:
NameOfTheClass.h
// Core file:
NameOfTheClass.cpp
```

Rationale

CamiTK uses CMake to generate projects that can be compiled on any platform.

Instead of entering manually cpp file names in `CMakeLists.txt` files, a CamiTK macro searches for all files with the file extension `.cpp`. If your C++ core files have the file extension `.cxx` for example, this CMake macro will not find and include them automatically into your project.

CPG-030: *Tricky code*

Summary

Tricky code *should* not be commented but rewritten !

Rationale

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and explicit logical structures.

CPG-031: *Magic Numbers*

Summary

The use of magic numbers in the code **must** be avoided.

Best Practice

Number other than 0 and 1 should be considered at best as parameters or at least declared as apporpiately named constants (see ()).

CPG-032: *Comments to indicate a method implementation*

Summary

There *should* be comments like `//--- methodName----` in all C++ files just above each method implementation

Best Practice

```
//---------- methodName ------------
void methodName() {

}
```

Rationale

This guideline is for better human readability as it helps clearly separating methods implementation.

If there is another method implementation with the same name (but different signature), you should not repeat this comment.

CPG-033: *Use of this->*

Summary

The use of this-> to specify instance variables or instance methods should be limited to cases where there is an ambiguity.

Best Practice

```cpp
int A::A(int a) {
    this->a = a;
}

void A::changeA(int b) {
    a = b;
}
```

Rationale

Do not use "this->" inside of methods to access class methods and instance variables unless needed as a parameter is shadowing the member.

This should make you think about the names you used for the variables. If too many local variables have the same name as an instance attribute, there might be something wrong: either too many variables, too badly-named local variables or too many badly-named members!

The use of your editor color coding should be enough to avoid useless and redundant auto-reference.

2.8 SDK core files

These conventions are required for all the C++ files inside the open-source CamiTK SDK library. They should be applied when possible as a good practice everywhere else.

CPG-034: *CamiTK namespace*

Summary

Write everything in the camitk namespace only for code and classes inside the CamiTK core library (classes in the sdk/library/core of the CamiTK opensource project).

Outside this library, use no namespace or your create your own one if you wish (e.g., for a library).

Rationale

camitk namespace should be restricted to a small number of class that constitutes the basic framework.

Genric Principles
Licensing
CEP organization
Components
Actions

3 — CamiTK Specific Conventions

> Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.
>
> *Rick Osborne*

3.1 Genric Principles

All the extensions developed in CamiTK **must** be grouped in *CamiTK Extension Projects* (CEP hereafter). Building an extension is easy and is simplified by using the CamiTK-Wizard to generate the code skeleton and buid it on your development platform. Once an extension is build and ready, it can be automatically integrated with all the existing extensions, i.e., once ready you can start using your extension immediately.

CamiTK does not want you to modify any of your own code. Your code is your specialty, and you are the only one to know what is best to do there. Instead CamiTK asks you to provide a glue to the service layer: you need to encapsulate your work in a pre-defined extension type. This generally means a minimal or no modification of yoru own code. There is no modification needed if you already have a well defined API where the GUI is completely separated from the rest. This is generally the case if you have followed well known/recommended software design. If not, your code can only gain from the new design!

An extension in CamiTK is distributed as a plugin. There are three types of possible plugins in the CamiTK architecture. The learning curve for CamiTK should therefore be minimal.

To simplify the development and distribution of extensions, CamiTK extensions are grouped in CEP (CamiTK Extension Projects). A CEP basically groups all the extensions developed by a specific person or group for a specific field or CAMI project. CEP facilitates task oriented project and helps you to focus on the most important/valuable code.

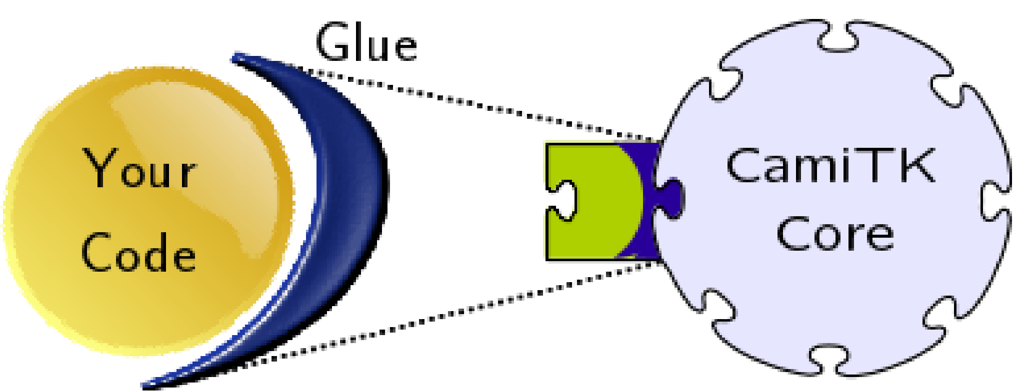


Figure 3.1: Principle of the CamiTK plugin interface.

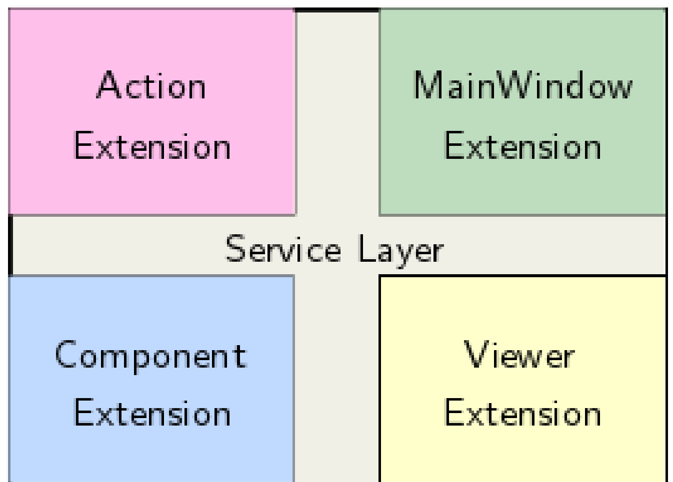


Figure 3.2: CamiTK Service layer 01

There are three types of extensions:

- Component extensions, they provide data I/O features
- Action extensions, they provide data processing features,
- MainWindow extensions, they provide interaction and visualization features, and
- Viewers which come with Action extensions and/or Component extensions ; they provide a way to design a GUI main window.

There are the four types of glue available in CamiTK. In a project you need to specify a given number of extensions of each types. Most of the cases are covered by action and component extensions. The Service Layer provides the definition of all these types of extantion and the SDK provides a lot of default behaviors for the extensions. The default behaviors are usually the one you need 90
behavior (in Object Oriented Programming, this will lead to redefinition of an inherited method).

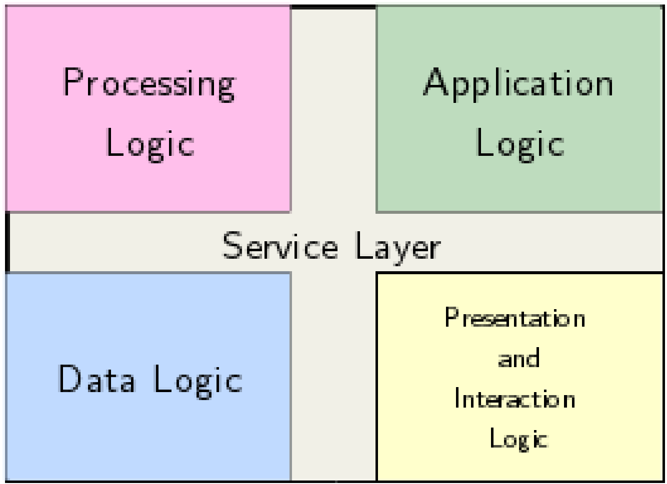


Figure 3.3: CamiTK Service layer 02

These four extensions correspond to four well-known principles in software achitecture design, and lead to the separation between, respectively:

- Processing logic
- Data logic
- Application logic
- Presentation and interaction logic

The table below gives you a list of use cases for the four types of extensions.

Type	Description/Need	Examples	Solutions
Component	View a new mesh, graph scene, image format, etc. Connect a new device to view its data, control its parameters/actions	• import a .obj format • view 3D ultraound images • connect a medical robot or sensor • connect a motion tracker device to interact with 3D viewer	• create a new component that inherits from basic ones (`ImageComponent`, `MeshComponent`) • or create a new generic Component • or combine existing components in a new one (with sub-Components).
Action	Transform some data. Generate new information from some data. New way of interacting with available view.	• Image filter, volume reconstruction • Biomechanical simulation using a specific method • Mesh/Image registration • Fly camera in 3D using a path of key positions and timer	1. Choose which component is concerned (what is the input data of the new algorithm) 2. Create a new action extension 3. Fill in its parameters as camitk::Property
Application	You need a specific GUI for your specific CAMI application	• No 2D/3D interaction, just data explorer and property viewer • A specific application to test on a group of user (including specific log or trace)	1. Choose which software component you need in the main window and 2. Create a new MainWindow extension
Viewer	New type of interaction with the data. A new way of visualizing the data	• Graph of values computed by an action • Histogram of data from a component	1. Create a new Viewer inherited from existing one (Viewer or InteractiveViewer) and add/refine a behavior. 2. Your viewer may be inserted into a Component Extension.

3.2 Licensing

CamiTK-SDK, CamiTK-Imaging and CamiTK-Modeling are under the Free and OpenSource license LGPLv3.

CPG-101: *LGPLv3 license for original open source code*

Summary

If you add new original code, data or anything to the open-source project (CamiTK SDK, imaging or modeling CEP), you **must** use the LGPLv3 license.

Rationale

The mail default license of the CamiTK open-source project is LGPLv3. We would like to keep it that way and make as little exception as possible.

CPG-102: *License compatibility for imported source code*

Summary

If you import some code, data or anything from another project to the open source SDK, please ensure that its license is compatible with the CamiTK LGPLv3 OpenSource license. Otherwise it could not be included in the CamiTK Open-Source project.

Rationale

If you include any files (C++ code, xml, data examples, icons, or any other resources) in the Open-Source project (e.g, SDK, imaging and modeling CEP), you need to make sure they are compatible with the Debian Free Software Guideline1 and even more importantly that, if it is source code (C++, XML or otherwise), it is compatible and can be linked with code under the LGPLv3 license (see for instance https://www.gnu.org/licenses/gpl-faq.html#AllCompatibility).

3.3 CEP organization

CPG-103: *CEP Directory Tree*

Summary

The source code of a CEP should have the following directory tree structure:

```
cepname
  -> src
    -> actions
    -> components
    -> applications
    -> viewers
    -> libraries
    - COPYRIGHT
    - CEPContent.xml
    - FindCamiTK.cmake
    - README
    - CMakeLists.txt
  -> doc
  -> build
```


Best Practice

This example shows a CEP called `cepname`. Everything useful for CEP development is inside a directory called `cepname`.

There *should* be 3 sub-directories at the top level of `cepname`:

- subdirectory `src` contains all the CEP source code. You can also name this directory `cepname`, which facilitates the use of a VCS.
- subdirectory `doc` contains all the documentation about this CEP.
- subdirectory `build` is the build directory needed by CMake ; it contains all generated files and can be deleted/regenerated at any time (it **must** not be stored in a VCS). Note: it can be anywhere on your computer, not necessarily near the `src` directory.

The source subdirectory itself contains:

- `README` is optional, it contains some general information about your CEP (this is not the same as the description found in `CEPDescription.cmake`). You can explain some specific things about your CEP in the README (this is good practice). It will be the first thing other CamiTK developers will read before bombarding you with questions (got the tip?)
- `COPYRIGHT` contains the copyright information about the source code of the entire CEP. By default it follows the same COPYRIGHT as CamiTK (LGPLv3), so if you have a different copyright **do not forget** to edit this file for reflecting yours.

- `CEPContent.xml` contains an xml description of what is inside your CEP. It is generally generated by the CamiTK-Wizard and helps making a catalog of existing extensions
- the top-level `CMakeLists.txt` and the `FindCamiTK.cmake` files are needed to build the CEP (Theses files are generated by CamiTK-Wizard. Default version of this file should work out-of-the-box, once you have set up `CEPDescription.cmake`)
- subdirectories `actions`, `components`, `applications`, `viewers` contain all the extensions provided by the CEP. Depending on your CEP you can have any of those directories.
  subdirectory `libraries` includes all the specific libraries that are needed for this CEP.

Your CEP may contain only actions and/or components subdirectories, or more, but they **must** be organized on the same level. This standardize the data/view/algorithms notions separation in a CamiTK reliable way.

All the directories of the CEP tree organization **must** follow [CPG-011]

3.4 Components

CPG-104: *Component Representation*

Summary

A component **must** have one of the following representation: `BitMap`, `Mesh` or `None`.

Rationale

A Component implements by default 5 interfaces:

- `InterfaceNode` to be visualized as a tree structure in the Explorer viewer,
- `InterfaceProperty` to be able to visualize/use its properties in the Property Explorer
- `InterfaceFrame`, to be localize in the 3D viewer
- `InterfaceGeometry` **OR** (exclusive) `InterfaceBitMap` to be visualize either in a 3D or a 2D viewer

All Components are viewed by default in the Explorer Viewer (you do not have to change/reimplement InterfaceNode, excepti if you really want to), same thing for InterfaceProperty and InterfaceFrame. For the last one, however, you should precise weather your Component should implement InterfaceGeometry (i.e. be a Mesh) or InterfaceBitMap (i.e. be an or a series of 2D image(s)), or none of them (in this case it won't be seen in default 2D and 3D viewers.

CPG-105: *Component Attributes should be camitk::Property*

Summary

`camitk::Property` is an overlay of `Object::Property`, itself an overlay of what an Object Oriented Programming attribute could be. Whenever it is possible, Components attributes *should* be managed thanks to `camitk::Propertys`

Best Practice

Suppose you want to model a patient's data component with an identifier and a birth date year for the patient.

```
// file InterventionData.h:
#ifndef INTERVENTIONDATA_H
#define INTERVENTIONDATA_H

#include <QObject>

#include <Component.h>

class InterventionData : public camitk::Component {

    Q_OBJECT

public:
    /// Default Constructor
    InterventionData(const QString & file) throw(camitk::AbortException);  virtual ~InterventionData
     ();

    /** Inherited from Component.
     * InterventionData has no representation, so initRepresentation should not do anything
     */
    virtual void initRepresentation() {};

    /** Inherited from Component.
     *  This method encapsulates all the setters for Properties.
     */
    virtual void updateProperty(QString name, QVariant value);

    /**
     * Sets the subject ID.
     * The subject ID must be the first letter of the first name,
     * followed by the first letter of the last name,
     * followed by the number of trial inclusion.
     * This methods encapsulates the data of subject ID
     */
    virtual void setSubjectID(QString value);

    /**
      * Returns      * should be given in the format YYYY.
    the subject ID (2 capital letters followed by a number
      */
    virtual QString getSubjectID();

    /**
     * The birth year of the subject
     * should be given in the format YYYY.
     */
    virtual void setSubjectBirthYear(int value);
```

```
    /**
     * Returns the birth year of the subject in the format YYYY.
     * This method encapsulates the data of subject birth date year
     */
    virtual int getSubjectBirthYear();

    /**
    * Return the current age of the patient (in years)
    */
    virtual int getSubjectAge();

};
#endif // INTERVENTIONDATA_H

// File InterventionData.cpp
// CamiTK includes
#include "InterventionData.h"
#include <Property.h>

// Qt Includes
#include <QFileInfo>
#include <QDateTime>

using namespace camitk;

// The name of the properties come back several time. They should be declared as constants
static const char * SUBJECT_ID_PROP_NAME = "Subject’s ID";
static const char * SUBJECT_BIRTH_DATE_YEAR = "Subject’s birth year";
static const char * SUBJECT_AGE_PROP_NAME = "Subject’s age";

// All hard coded numbers should be declared as constants (ssee CPGC#015 Constants and enums)
static const char * DEFAULT_SUBJECT_ID = "XX0";
static const int MIN_BIRTH_YEAR  = 1916; // 100 years old in 2016
static const int MAX_BIRTH_YEAR  = 1998; //  18 years old in 2016
static const int DEFAULT_BIRTH_YEAR = 1986; //  30 years old in 2016


// -------------- Constructor ------------------
InterventionData::InterventionData(const QString & file) throw(AbortException)
    : Component(file, QFileInfo(file).baseName())
{
    //-- For this example, we discard the input file and do not read it.

    //-- We declare all the object variables as properties.
    // Subject’s Name
    Property * subjectID = new Property(SUBJECT_ID_PROP_NAME, DEFAULT_SUBJECT_ID, "Subject Unique
     identifyer. Must be the first letter of the fist name, followed by the first letter of the last
      name, followed by the number of the trial inclusion.", "No Unit");

    //-- Let us use a regular expression to check that it follows the rules for subject’s ID
    // Regular expression that checks that there are 2 letters followed by un number between 0 and
     1000
    QRegExp rx("[A-Z][A-Z][0-9]{1,3}");
    subjectID->setAttribute("regExp", rx);
    addProperty(subjectID);

    //-- Subject’s birth date
    Property * subjectBirthYear = new Property(SUBJECT_BIRTH_DATE_YEAR, QVariant(DEFAULT_BIRTH_YEAR),
      "Birth date of the subject in the format YYYY", "Integer");
    subjectBirthYear->setAttribute("minimum", MIN_BIRTH_YEAR);
    subjectBirthYear->setAttribute("maximum", MAX_BIRTH_YEAR);
    addProperty(subjectBirthYear);

    int defaultAge = QDate::currentDate().year() - DEFAULT_BIRTH_YEAR;
    Property * subjectAge = new Property(SUBJECT_AGE_PROP_NAME, QVariant(defaultAge), "Subject’ Age
     at the lauch of the application", "year");
    //-- This property is automatically calculated and should not be set by external user on the GUI.

    //!! This property will be read only, only on graphically on the Property Editors (i.e. GUI).
     Other class may modify it
```

```
11     //   from the outside with setProperty. So it is safer to user getters ans setters in this case.
12     subjectAge->setReadOnly(true);
13     addProperty(subjectAge);
14 
15 }
16 
17 // --------------- Destructor -------------------
18 InterventionData::~InterventionData() {
19 
20 }
21 
22 // --------------- updateProperties -------------
23 void InterventionData::updateProperty(QString name, QVariant value) {
24     //-- If the birsth date has changed, we should change the age also...
25     if (name == SUBJECT_BIRTH_DATE_YEAR) {
26         int newAge = QDate::currentDate().year() - value.toInt();
27         setProperty(SUBJECT_AGE_PROP_NAME, QVariant(newAge));
28     }
29     else if ((name != SUBJECT_ID_PROP_NAME) && // Do nothing for the property subject's ID
30              (name != SUBJECT_AGE_PROP_NAME)) {  // Do nothing for the subject's age
31         // if the property is not handled locally, let Component handle it !
32         Component::updateProperty(name, value);
33     }
34 
35     //-- Do not forget to refresh your component after having changed its properties.
36     refresh();
37 }
38 
39 
40 // --------------- setSubjectID ----------------
41 void InterventionData::setSubjectID(QString value) {
42     //-- Check that the format is ok
43     // Regular expression that checks that there are 2 letters followed by un number between 0 and
       1000
44     QRegExp rx("[A-Z][A-Z][0-9]{1,3}");
45     if (rx.exactMatch(value)) {
46         setProperty(SUBJECT_ID_PROP_NAME, QVariant(value));
47     }
48 }
49 
50 
51 // --------------- getSubjectID ----------------
52 QString InterventionData::getSubjectID() {
53   return property(SUBJECT_ID_PROP_NAME).toString();
54 }
55 
56 
57 //--------------- setSubjectBirthYear-------------
58 void InterventionData::setSubjectBirthYear(int value) {
59     // Check if the value is between min and max values
60     if ((value >= MIN_BIRTH_YEAR) && (value <= MAX_BIRTH_YEAR)) {
61         setProperty(SUBJECT_BIRTH_DATE_YEAR, QVariant(value));
62     }
63 }
64 
65 //--------------- setSubjectBirthYear-------------
66 int InterventionData::getSubjectBirthYear() {
67   return property(SUBJECT_BIRTH_DATE_YEAR).toInt();
68 }
69 
70 //--------------- setSubjectBirthYear-------------
71 int InterventionData::getSubjectAge() {
72   return property(SUBJECT_AGE_PROP_NAME).toInt();
73 }
```

Rationale

> Contrary to private attributes with public accessors and modifiers, `camitk::Property`(ies) are accessible at runtime without knowing their names at compile time. Which allows for example the Property Explorer to display and edit them at runtime for any Component.

3.5 Actions

CPG-106: *All Action Parameters should be camitk::Property*

Summary

The parameters of your Action *should* not be attributes or GUI specific methods, but `camitk::Property`.

Rationale

Actions can be called on the graphical interface of CamiTK-Imp. So you may want to redesign your Action Widget. However, other CamiTK applications, like the CamiTK-StateMachine for example may pre-set your Action parameters without using a GUI (but a configuration file). These applications should thus be able to access all of your Action's parameters without the widget. You may still have a specific widget to modify these `camitk::Property`(ies), but you should use properties instead of Action's attributes in the same way as for Components.

Index

4 spaces instead of one tab (CPG-003), 18

A Header File is a Class Declaration (CPG-017), 33
About windows.h (CPG-025), 45
Accessor / Mutator (CPG-014), 29
Add brackets to unbracketed one line conditional statements (CPG-007), 22
All Action Parameters should be camitk::Property (CPG-106), 72
Angle-bracket includes convention (CPG-020), 39
Attach pointer and reference operator (*, &) (CPG-009), 24

Bracket Style: Java (CPG-004), 19
Breaks closing headers from their immediately preceding closing brackets (CPG-006), 21

CamiTK namespace (CPG-034), 57
CEP Directory Tree (CPG-103), 65
Class Names (CPG-012), 27
Comments to indicate a method implementation (CPG-032), 55
Component Attributes should be camitk::Property (CPG-105), 68
Component Representation (CPG-104), 67
Constants and Enums (CPG-016), 31
Copyright and Licence (CPG-027), 49
Core file extension must use .cpp file name extension (CPG-029), 52

declspec(dllexport) declarations (CPG-026), 47
Default language is English (CPG-001), 15
Directory Names (CPG-011), 26
Doxygen Documentation (CPG-028), 51

Header File Anatomy (CPG-019), 35
Header Guards (CPG-018), 34

Include Path (CPG-022), 41
Include Statement Order (CPG-021), 40
Indent Case Statement in Switch Blocks (CPG-010), 25
Insert Space Padding Between a Header and the Following Paren (CPG-008), 23

LGPLv3 license for original open source code (CPG-101), 63
License compatibility for imported source code (CPG-102), 64

Magic Numbers (CPG-031), 54
Method Names (CPG-013), 28
Minimize the number of includes in header files (CPG-023), 42

No ’Using namespace’ in header files (CPG-024), 44
No absolute file name (CPG-002), 16

Space Between Operators (CPG-005), 20

Tricky code (CPG-030), 53

Use of this-> (CPG-033), 56

Variables (CPG-015), 30

CamiTK Programming Guidelines
Version 1.0 - 21 June 2018