Style Manual

In order to have easily readable and professional-looking code in Kactus2, the coding style should be the same regardless of the programmer. This manual defines the coding style and practices used in developing Kactus2 to ensure unified outfit and style for all source code files.

1. General

1.1. Language

The natural language used for comments, naming variables is English.

1.2. Indentation

For indentation, four (4) spaces are always used. Using tabs for indentation is prohibited.

1.3. Line Width

The maximum width for a source code line is 115 characters.

2. Comments

Kactus2 source code is commented in a way that allows auto-generation of Doxygen documentation. Doxygen tags are used extensively to make the documentation as comprehensible as possible. JavaDoc notation (@param) is preferred over the classic Doxygen notation (\param).

2.1. General

All comments must be full sentences, i.e., they begin with a capitalized letter and end to a period, exclamation mark or question mark.

2.2. File Comment

All C++ source files begin with the following file comment:

```
//-----------------------------------------------------------------------------
// File: File.h
//-----------------------------------------------------------------------------
// Project: Kactus 2
// Author: <Name>
// Date: <Date in the format dd.mm.yyyy>
//
// Description:
// <Short description of the class/file contents>
//-----------------------------------------------------------------------------
```

2.3. Function Comments

Functions are commented extensively in header files. Following example illustrates a typical function comment:

```
/*!
 *  Stores a new command to the edit stack.
 *
 *      @param [in] command   The command to add.
 *      @param [in] autoExec  If true, the command's redo() is automatically
 *                            executed inside this function.
 */
void addCommand(QSharedPointer<QUndoCommand> command, bool autoExec = true);
```

The function comment can contain the following tags with the following syntax:

- **@param** [in]/[out]/[in/out] <paramName> <Description>
- **@return** <Description of the return value> *(Only if the function returns something)*
- **@remarks** <Important notes about the function behavior and special pre/post-conditions> *(Only if necessary)*
- **@throw** <exceptionClass> <Description of when the exception is thrown> *(Only if the function throws exceptions)*

The parameter or return value descriptions must not state explicitly whether the parameter is a pointer, a reference or a value. This information is already visible in the parameter declaration and should not be repeated in the comment, because it adds unnecessary burden for e.g. cases when the parameter is changed from a reference to a pointer.

In source code files, the functions have a minimal start comment which follows the style:

```
//-----------------------------------------------------------------------------
// Function: GenericEditProvider::addCommand()
//-----------------------------------------------------------------------------
void GenericEditProvider::addCommand(QSharedPointer<QUndoCommand> command,
                                     bool autoExec)
{
    ...
}
```

2.4. Comments for Enhancing Appearance

The appearance of the source code files can be enhanced using separator comments:

```
//-----------------------------------------------------------------------------
```

Different sections in the source code can also be separated using a section comment:

```
//-----------------------------------------------------------------------------
// <Section description>.
//-----------------------------------------------------------------------------
```

2.5. Data Type Comments

When a class, a structure or an enumeration is defined in a header file, it is commented using a section comment (notice the exclamation mark before the description):

```
//-----------------------------------------------------------------------------
//! <Class description>.
//-----------------------------------------------------------------------------
class <ClassName> : public <BaseClassName>
```

2.6. Commenting Variables and Types

Member variables and type definitions are commented using the following format:

```
//! The history size for undo/redo stack.
unsigned int historySize_;

//! Point list type.
typedef QList<QPointF> PointList;
```

Variables in a structure, on the other hand, can be commented on the same line as the actual variable definition with the following format:

```
bool redoing_;      //!< Boolean flag for redoing.
```

3. Naming Conventions

3.1. Files

Following extensions are used for each file type:

File Type	Extension
Implementation file	.cpp
Header file	.h
Inline file (if inline functions are needed for performance reasons)	.inl

The names of the C++ source code files are derived from the file contents. All words are written with a starting upper-case letter. For example, if the source file contains the implementation of class GenericEditProvider, the source files are named GenericEditProvider.cpp and GenericEditProvider.h.

3.2. Variables

The name of the variables is written so that the first word is in lower-case and the next ones start with an upper-case letter. Member variables are always post-fixed with an underscore. Parameters and local variables are written without the underscore.

Examples: pointList_ *(member)*, currentState *(non-member)*

3.3. Pointers and References

The * and & characters are part of the pointer and reference types.

Examples: PointList* pointList; GenericEditProvider& editProvider; **NOT** PointList *pointList;

3.4. Functions

Functions start with a lower-case word and the next words start with an upper-case letter.

Example: void doSomething();

3.5. Namespaces, Classes, Structures and Enumerations

All words in namespace, class, structure or enumeration names start with an upper-case letter.

Examples: namespace General, class GenericEditProvider, enum State, struct Point

3.6. Enumeration Values and Constants

Values of an enumeration and constants are written using capitalized words. Words are separated from each other using underscores.

Examples: enum HighlightMode {HIGHLIGHT_OFF, HIGHLIGHT_HOVER};
int const MAX_HISTORY_SIZE = 100;

4. Code Appearance

4.1. Control Statements

Control statements (if, while, for) must always be encapsulated using brackets even if there is only one statement inside the control statement.

4.2. Code Block Brackets

Code block brackets are always placed on their own lines.

Example:

```
if (qFuzzyCompare(dir, QVector2D(0.0f, -1.0f)))
{
    setRotation(0.0);
}
```

4.3. Whitespace in Control Statements

The following example illustrates how whitespaces are placed in a control statement:

```
if (statement1 && statement2)
  ^           ^  ^
```

5. Practices

5.1. Global Variables

Global variables must not be used. Only global constants are allowed.

5.2. Const-correctness

Const-correctness must be used at all times. Const is used in member functions that do not modify the object's data. For variables, the place of const is after the type and before the &-sign.

Examples: PointList const& pointList;

5.3. Friends

Friend classes are not allowed since it breaks the modularity of the code.

5.4. Contents of One Compilation Unit

Usually, only one class should be defined in a compilation unit (meaning one header and one source file). However, if the class requires some enumerations, structures or constants, they can be defined in the same compilation unit.

5.5. Function Implementations in Header Files

Function implementations are never written into the header files. Normal functions are implemented in the .cpp files and inline functions in the .inl files.

5.6. Forward Declarations

Forward declarations are used in header files always when it is possible. This is the situation when only pointers or references to the class instances are used in the header file.

5.7. Disallowing Copying

All classes that should not be copied must have private a copy constructor and an assignment operator declaration with the following syntax:

```
// Disable copying.
DiagramConnectionEndPoint(DiagramConnectionEndPoint const& rhs);
DiagramConnectionEndPoint& operator=(DiagramConnectionEndPoint const& rhs);
```

Copying instances should be avoided, so this is the default policy for any classes.

5.8. Header File Guards

Header files must be guarded using pre-processor statements with the following format:

```
#ifndef CLASSNAME_H
#define CLASSNAME_H
...
#endif // CLASSNAME_H
```

5.9. Organization of Implementation Files

The functions must be implemented in the implementation files in the same order as they have been declared in the header files.

5.10. *Using* Statements

Using statements must not be used in header files. *Using namespace* should only be used with extra caution in the implementation files. It is preferred to use *using* statements only to retrieve a specific class, structure or other type from a namespace.

5.11. Initializing Variables

All variables are initialized when they are declared. All member variables must be initialized in the class constructor. Member initialization list must be used if it is possible.

5.12. #include Statements

A forward slash (/) must be used instead of a backslash as the #include directory separator to allow cross-compatible code.

5.13. C-style Coding

C-style coding is not allowed. Using, for example, functions abort(), exit(), malloc() and dealloc() or goto statements is prohibited.

5.14. Type Casts

Implicit or C-style style type casts must not be used. Instead, static_cast<>, dynamic_cast<> and reinterpret_cast<> should be used. const_cast<> should be avoided.

5.15. Function Parameters

Large function parameters are always passed as references or pointers.

5.16. Class Organization

All member variables are private. If access to private member variables is required in a derived class, protected access functions are used. A class contains first the public part, then the protected part and last, the private part.

5.17. Constructors and Destructors

Every class must have a constructor and a destructor.