CLAM

USER AND DEVELOPMENT DOCUMENTATION

RELEASE 0.7.0

REVISION 3

Table of Contents

CLAM RELEASE 0.7.0

USER AND DEVELOPMENT DOCUMENTATION

April 2004

IUA-UPF

Music Technology Group

Source:	**MTG**
Title:	**CLAM RELEASE 0.7.0 USER AND DEVELOPMENT DOCUMENTATION**
Revision:	**3**
Current Developers:	**Xavier Amatriain** **Pau Arumí** **Maarten de Boer** **David García** **Miquel Ramírez**
Past Developers:	**Xavier Rubio** **Enrique Robledo**
Contact:	**clam@iua.upf.es**

I Introduction

1 Disclaimer

This document is offered 'as is'. There may still be formatting mistakes or non-coherent sections or comments. Please report any such elements to the editor of the document.

2 License of this document

3 What is CLAM?

CLAM is a Free Software framework licensed under GNU-GPL that allows to fully develop multiplatform audio applications in C++ using advanced processing algorithms. It is not limited to the processing part of your application; it can help you providing multiplatform solutions for most problems that an audio application should face:

- accessing audio and MIDI devices,
- managing threads,
- serializing objects in formats such XML and SDIF,
- displaying and controlling your application data,
- integrating visualization using several multiplatform graphical toolkits,
- interconecting your application modules in a decoupled way
- ...

CLAM is able to do complex audio processing involving:

- Management of heterogeneous signal data: not only samples but also spectral data, symbols, structured data...
- Complex data flows: with asyncronous events (controls), diferent rates of data feeding...
- Scaling up by composition of smaller processings.
- Dynamic creation and interconnection of processing networks.

And last but not least, it comprises a big repository of already done algorithms concerning areas such as:

- Spectral modeling and transformations
- Feature extraction
- Classification
- ...

4 What does ’CLAM’ mean?

CLAM stands for **C**++ **L**ibrary for **A**udio and **M**usic and in Catalan means something like ’a continuous sound produced by a large number of people as to show approval or disapproval of a given event’. It is the best name we could find after long discussions and it is certainly much better than its original name (MTG-Classes).

5 Historical background

CLAM was formerly developed as an internal project on the Music Technology Group at UPF named *MTG-Classes*. The aim of the project was to create a foundation of C++ classes to be used by all the research projects at the MTG. Literally (from the first written draft) the goal was:
’To offer a complete, flexible and platform independent Sound Analysis/Synthesis C++ platform to meet current and future needs of all MTG projects.’

The three main axes of these goals were defined as:

- **Complete:** should include all utilities needed in a Sound Processing Project (input/output, processing, storage, display...)
- **Flexible:** Easy to use and adapt to any kind of need.
- **Platform Independent:** Compile under UNIX, Windows and Mac platforms.

These initial objectives have slightly changed since then mainly to accommodate to the fact that the library is no longer seen as an internal tool for the MTG but as a library that is made public under the GNU-GPL in the course of the Agnula IST European Project.

6 Supported platforms

CLAM code is standard ISO/ANSI C++. It uses multiplatform libraries to abstract platform dependant issues. Non portable code and even library dependant code is very localized and isolated. This makes it easier to port CLAM to whatever platform.

Currently, the following platforms are supported:

Linux	gcc 2.95	Fully supported
	gcc 3.X	Fully supported
Windows	VisualC++ 6 **(SP5)**	Supported until release 0.6.1, not currently supported any longer
	VisualC++ 7.1	Fully Suported
MacOSX	gcc 3.X	Fully Supported

Reports on CLAM ports to any other platform will be appreciated.

7 Recommended previous skills

Althought CLAM goes toward a visual environment, currently, this "visual builder" (quoting R.E. Johnson) is in beta stage and CLAM functionalities are only accessible by doing your own programs. Thus, a good level in object oriented C++ programming experience is needed, although we have tried our best to keep interfaces as simple as possible.

Of course, CLAM is a framework for audio and music processing so some knowledge in those areas (as well as some DSP basis) is also recommended.

8 Basic principles

The CLAM framework is built on top of some architectural basic elements that are used as building blocks and should therefore be mastered. Most of this document is about them but, just to have a first impression, these are the basic principles used in CLAM:

8.1 Processing architecture

A CLAM based system can be viewed as a set of **Processing** objects deployed as an interconnected network. Each Processing can retrieve **ProcessingData** tokens and modify them acording to some algorithm.

Programmers can keep control over the ProcessingData flow between Processing or they can delegate this task to one of the many automated **FlowControl** schedulers.

A set of Processings can be arranged to form a new processing. Thus you can use that new processing to abstract what the full bunch of Processings does and then scale up to a more complex system. Processings can be arranged on compile-time (**ProcessingComposites**), or dinamically on run-time (**Networks**).

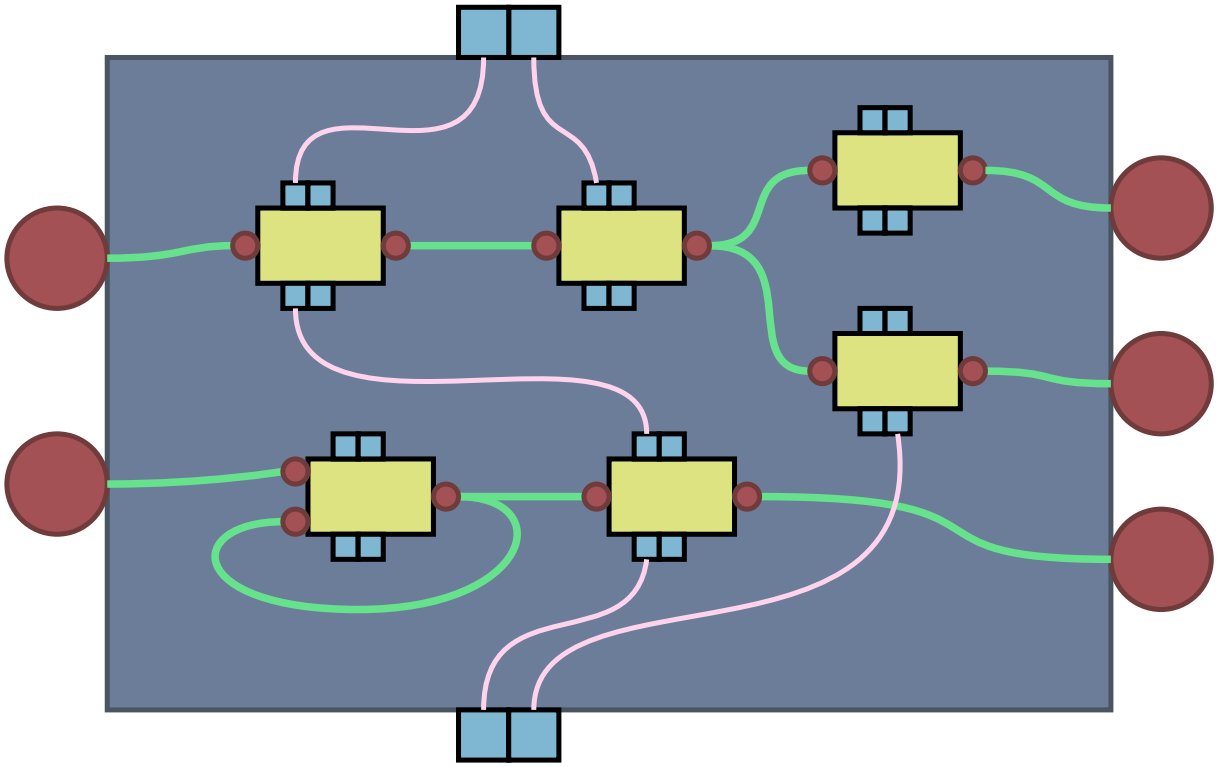


Figure 1: CLAM Network

Networks allow to build a CLAM system without no C++ knowledge, using only the graphical interface. They can be also used to build up rapid prototypes of a later optimized system. The prototyping environment is still some how limited.

8.2 Processing classes

The Processing classes are the main building blocks of the CLAM framework. All processing in the CLAM must be performed inside a Processing class.

Interaction between Processings follows a very bounded but flexible interface. This way, CLAM can manage Processing in a very general way and it boosts the reusability of Processings between different CLAM systems.

All the Processing configuration process is done by giving it a configuration object before the Processing object is on the running state. While the Processing is running, the processing algoritm will be executed every time the `Do()` method is called at the processing execution rate.

While the Processings is running receive and emits two kinds of output:

- **Continuous data:** That will be fed from and by the Ports every time a Do method is called. Basically they consist on ProcessingData.
- **Asyncronous data:** Fed from and by the Controls whenever an event happens. They usually changes the internal state/mode of the algorithm

When no automated FlowControl is involved, you can also overload the Do() method to pass as parameters the ProcessingData that would be otherwise accessible from the ports. This way of implementing Processing is now deprecated but still used.

The following figure illustrates all the different components of a Processing class.

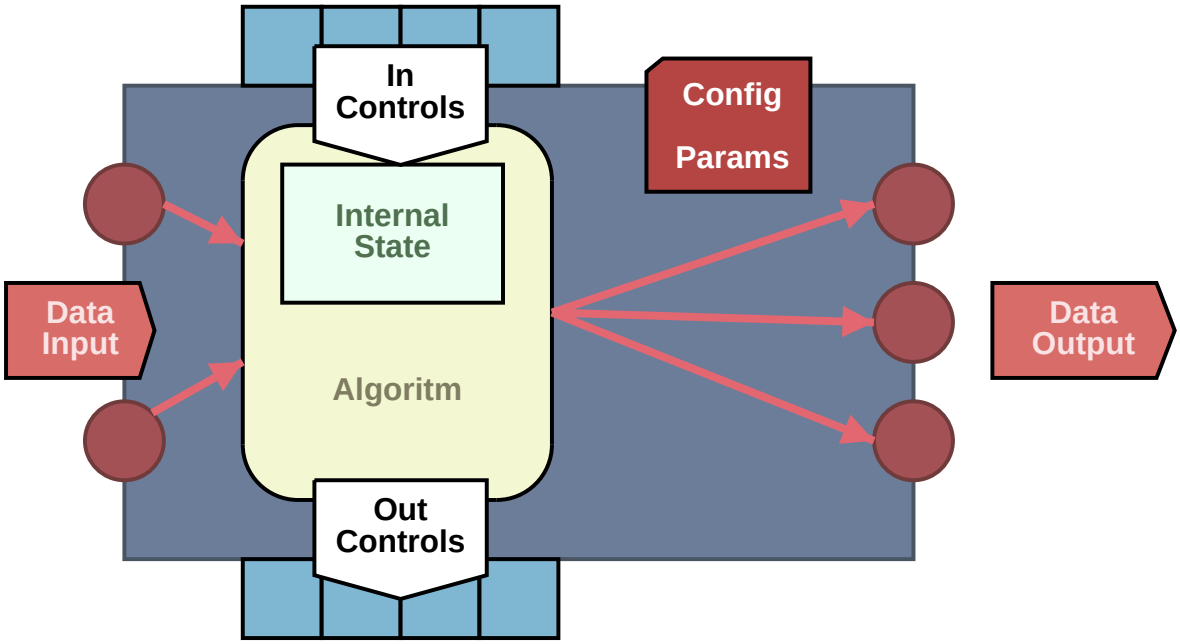


Figure 2: Processing Object Representation

- See chapter VII and others.
- See also section chapter VIII for information on ProcessingData classes.
- Controls are explained in section 40

8.3 Dynamic Types

Both, Configurations and ProcessingData, are implemented as DynamicTypes (often DT for short). DynamicTypes are an abstraction that allows to have objects with not all attributes instantiated. DynamicType are implemented using preprocessor macros that will expand each attribute declaration in accessors and instanciation interface among other useful methods.

One of the most used features in dynamic types is that they provide some kind of introspection, and thus CLAM can provide some useful functionalities in a general way on every new DynamicType you will define. Examples of those free functionalities that you get by defining some class as DT are XML serialization, generic atribute visit, automatic interface generation...

You can see section chapter VI for more information on DynamicTypes use.

(See section 35, chapter XX, chapter IX for XML support)

8.4 Visualization Module

The CLAM Visualization Module fullfills two different developers needs. The first one is to inspect graphically CLAM objects as a debugging aid. The other one is to build a complete GUI based application that can be used in interaction with the Processing part of a CLAM system.

CLAM-VM has been designed in a very decoupled way so that it can be fully removed harmlessly from a CLAM system. It provides some general services that are toolkit independent such as thread safe data caching, model-view comunication and sincronization...

That CLAM-VM infrastructure can be used with any toolkit such as Qt, Fltk... In fact, it provides already done widgets, the toolkit dependant part. They are mostly implemented using FLTK and OpenGL but next releases will provide more support on Qt.

(See chapter XIV)

8.5 System utilities

CLAM provides integrated and platform independent support for system dependant tasks. For example:

- Threading
- Midi devices access
- Audio devices access
- Audio files I/O
- SDIF I/O

9 Structure of this document

Although the structure of this document is a bit complex, hopefully by the time you get here you will be able to understand why the index has been set that way.

Basically, there are three parts to this document:

Part 1. User Documentation: Is the information any user of the CLAM framework should be aware of.

Part 2. Developer Documentation: Includes all the necessary information for more advanced users/developers.

Part 3. CLAM Sample Applications: In this part, we explain the most important sample applications included with CLAM.

Part 4. Migration: Is just a brief summary of the differences between the different releases (for already introduced and advanced users).

So, this document is intended to work as a more or less progressive introduction to CLAM. Start reading the first page and stop whenever you think details are becoming overwhelming. Of course for some particular users some sections may be 'skipable' and some may even prefer to start by reading about the sample applications just to get a grasp of what the framework is able to offer.

Going into a bit more details of the different parts, each of them includes the following topics.

Part 1 starts off with some information any CLAM user will need the first time he/she tries to use the framework: in the 'Deploying CLAM in your System' chapter you will be guided on how to configure CLAM in your system; and in the 'CLAM Build System Documentation'chapter you will be introduced to the features and functionalities of the rather particular build system that is used in CLAM. In the next chapter you will find a 'Usage Tutorial' (chapter IV) that introduces the basic functionality of the CLAM framework, furthermore in this same section we include a listing of other relevant examples that are kept in the repository. All following chapters focus on the different functionalities of the framework from a user's point of view (you will find chapters on Dynamic Types, Processing classes, Processing Data classes, XML Support, Audio I/O, MIDI I/O, The Application classes, and the Visualization Module (GUI)).

Part 2's first chapter, XIII. CLAM Coding conventions, gives some hints on coding recommendations and conventions for developers. The next chapters (Dynamic Types, Processing Data classes and XML) give an inside view on how to develop using the internal features of these CLAM's building blocks.

Part 3 includes an explanation of the following applications: SMSTools, SALTO, Spectral Delay and Rappid. The last chapter in this part explains how to use CLAM to develop LADSPA plug-ins. After you have read the introduction and have managed to compile CLAM, you may wish to go directly to these examples, before getting deeper into the library.

Part 4 gives a summary of the differences between this release and previous ones. You can skip this if you are not already familiar with the CLAM framework. Note: this part explains main differences between three latest releases, all the previous ones, though, have only been internal to MTG.

10 Where to find more information on CLAM

There are different sources of CLAM related information. All of them, though, are linked and updated at CLAM's website at www.iua.upf.es/mtg/clam .

USER DOCUMENTATION

II Deploying CLAM in your system

11 Roadmap

This section explains how to:

- Obtain CLAM and the libraries it depends on
- Install these packages in your computer for working with CLAM

12 Obtaining the CLAM sources

CLAM source code and third party libraries binaries can be found on our web page --download area-- http://www.iua.upf.es/mtg/clam in both tarball and zip formats.

At this moment CLAM is provided as a source package which is not ment to be compiled as a binary library. Although in the future this is likely to change we provide an automatic build system in order to ease the job to the user (you will find information on this build system in the next chapter).

12.1 GNU Tools distributed with MacOS X 10.2 Specific Issues

- **autoconf upgrade**
  Please try to upgrade the autoconf program that Apple bundles with MacOS X 10.2 to latest version (2.59). You can download its sources from http://www.gnu.org/software/autoconf/. If you cannot install new applications on your system, then please use the *configure.macosx* script included in *build* folder, instead of anything generated by the autoconf tool provided by Apple.

13 Dependencies on third-party libraries

CLAM depends on other external libraries in order to implement low level task such as XML parsing or cross-platform abstractions (devices, threads, GUI, etc.). This implies that they must be present on your development environment in order to build applications with CLAM.

The full list of libraries CLAM depends on is:

- Any platforms:
  - **FLTK 1.1.4**
    the Fast Light ToolKit, a GUI library. **Mandatory**
    Website: http://www.fltk.org
  - **Qt 3**
    a wideused GUI toolkit **Optional**: a few example uses it.
    Website: http://www.trolltech.com
  - **Xerces-C++ 2.3.0**
    XML tool: implements a DOM API implementation. **Mandatory**
    Website: http://xml.apache.org/xerces-c/index.html.
    You can find the sources of this version in Apache Archive
  - **FFTW 2.1.3**
    a library that implements very efficiently the FFT algorithm. **Mandatory**
    Website: http://www.fftw.org
  - **CppUnit 1.8.0**
    a C++ testing framework. Is the C++ version of the xUnit family. **Optional**: for running library tests.

Website: http://cppunit.sourceforge.net

- **OpenGL 1.x**
  a computer graphics API. Major graphic hardware vendors provide with its own implementation (usually bundled with their driver packages). However, the SDK is usually provided either by IDEs like Microsoft Visual Studio or are directly provided by the vendor: check your hardware manufacturer. **Mandatory**
  Website. There is also a free non-hardware accelerated implementation: MesaLib at http://www.mesa.org.
- **libsndfile 1.0.6**
  a library for reading and writing several audio file formats. **Mandatory**
  Website: http://www.mega-nerd.com/libsndfile
- **Underbit's libmad 0.15**
  Underbit's Mpeg Audio Decoding library. **Mandatory**
  Website: http://www.underbit.com/products/mad
- **Xiph.org Ogg/Vorbis SDK 1.0.1**
  Xiph.org free implementation of Vorbis I encoder and decoder. **Mandatory**
  Website: http://www.xiph.org
- **id3lib 3.8.3**
  a library for parsing ID3 tags found on Mpeg audio bitstreams. **Mandatory**
  Website: http://id3lib.sourceforge.net/

- Only on GNU/Linux:
  - **ALSA Library**
    the library to interface the Advanced Linux Sound Architecture. That is: the sound devices. **Mandatory**
    Website: ALSA project http://www.alsa-project.org.
- Only on Windows:
  - **PortMIDI**
    a library for accessing in a cross-platform way to MIDI devices. **Optional**: a few examples use it.
    Website: http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/music/web/portmusic
  - **Win32 Pthreads**
    a POSIX Threads standard implementation on top of Win32 API. **Mandatory**
    Website: http://sources.redhat.com/pthreads-win32
  - **Microsoft DirectX SDK 8.1**
    own Microsoft SDK for using their Multimedia API. **Mandatory**
    Website: http://www.microsoft.com/windows/directx/default.aspx

The following sections will give you guidelines on how to deploy them in either GNU/Linux and Windows platforms. If, after having read these instructions, you still have problems please browse through the CLAM mailing list archives or post a new mail to the list if you find that your question has no answer there either.

13.1 External libraries on GNU/Linux

Your GNU/Linux distribution should have the suited versions for the required libraries. However, it is possible that the normal package has some problems: it is inexistent, it is a wrong version, it is compiled with the non-default compiler, or it is compiled with wrong options.

In case you run on some of the above problems, check if there is a suitable tarball in our web: We provide some compiled libraries for Debian-Woody and Red Hat 9.

If this is not your case, or you still have problems we recommend to download and compile the sources directly.

If you do that take into account these hints:

- **fltk 1.1.4**
  Enable shared libraries when configuring fltk, they are disabled by default. Once downloaded and extracted the source package, configure it with the command:

```
$./configure --enable-shared --enable-threads --enable-xdbe --enable-xft
```

- If you don't have administrator rights to install libraries in the standard location, our build system expect them the same directory as CLAM root. Actually this is the way they are deployed on Windows.

13.2 External libraries on Microsoft Windows

To make it easy for Windows users we have precompiled binaries of all the needed libraries. They are available for download in the CLAM web, download section

All of them are generated using Microsoft VisualC++ 7.1.(.NET) although we still keep the ones compiled with VisualC++ 6.0 for backwards compatibility.

Windows has no convention on how to install libraries --both precompiled binaries and development headers. We have configured our build system in order to find libraries at the same directory level than the CLAM root directory.

For example:

```
devel\CLAM\
devel\fltk\
devel\xercesc\
devel\fftw\
devel\...
```

Note on fftw: fftw can work on two modes with different precission and speed: 'float' and 'double'. Both versions are available on the web and you must make sure that the fftw directory you place in the same level as CLAM dir must fit with the value of `CLAM_DOUBLE` build system variable. (See chapter III for more details on this).

CLAM uses 'float' as default so to start using CLAM extract the float-version zip.

Note on DLLs: Once you have downloaded all the development kits, you should scan the `lib\` subdirectories inside each library folder, and copy all the Dynamic Linking Libraries (.dll) to some place pointed by your system `PATH` environment variable.

We recommend to create a directory named `dll` at the same level as the libraries directories. Copy there the dlls and add it to the `PATH` environment variable.

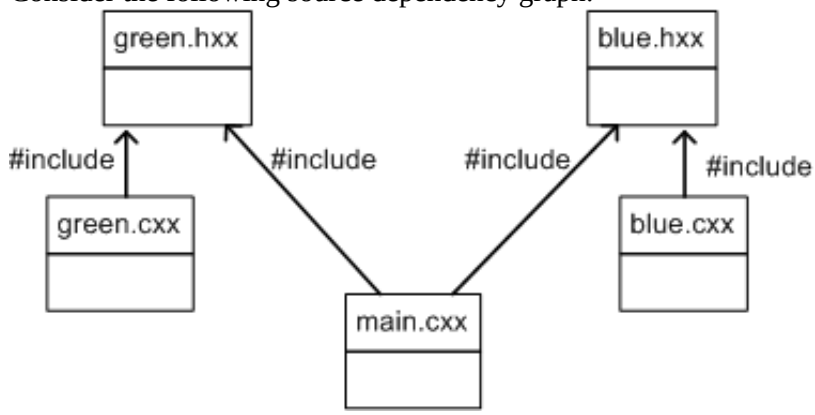
III CLAM Build System Documentation

14 Overview

CLAM provides an automatic build system which allows you to generate and maintain GNU Makefiles and VisualC++ 6/7.1 project files (.dsp/.prj).

This automatic build system is most useful taking in account that at the moment CLAM is not distributed in a compiled (binary) library. Thus it manages to hide to the user all details regarding: necessary implementation files, include dirs, external libraries and compilation flags.

Consider the following source dependency graph:



Without a build system helper you would have to add by hand each `.cxx` file into your Makefile or Visual project in order to compile it. The CLAM automatic build system uses an small and fast application called `srcdeps` that finds the source files dependencies.

In the previous example, supposing we say to the `srcdeps` that we want to compile `main.cxx`. It would make the following deduction:

```
'As main.cxx must be compiled and includes both blue.hxx and green.hxx
then blue.cxx and green.cxx must be also compiled'
```

So, when `srcdeps` finds a header dependency to `Foo.hxx`, it looks for the `Foo.cxx` implementation file. And in the case that it is found, it applies the same search recursively.

Of course additional implementation files --not reachable from dependent headers-- can be specified as starting points. The common case is that only the .cxx with the main function has to be specified.

For each executable --or binary in general-- you want to generate, you must use a file named `settings.cfg` in order to provide those starting points and other build specifications. See section 19.1 for more details on how to do this.

Because the automatic build system relies on a compiled application `srcdeps` to do its job, after downloading CLAM we'll need to compile this application. Next section describes how to do it.

15 Setting up CLAM Build System

16 USE / HAS variables

In order to keep track of what external libraries to use for each project, and whether they are available, the srcdeps configuration files used by CLAM contain several `USE_[feature]` and `HAS_[feature]` variables.

On Windows, the user has to make sure that the `HAS_[feature]` variables reflect it's system installation, on Linux and MacOSX they are set to 1 or 0 when running the `configure` script in `build/`

When `srcdeps` parses the configuration files, it makes sure that all `HAS_[feature]` and `USE_[feature]` variables are consistent. In other words, it will give an error if it noticed that a specific `USE_[feature]` is set to 1 while the corresponding `HAS_[feature]` is set to 0

Sometimes, the settings may specify only the use a certain feature when it is indeed possible. This can be done by setting

```
USE_[feature] = $(HAS_[feature])
```

See section 19.3 for a full list of these variables.

16.1 Included configuration files

Looking at a typical `srcdeps settings.cfg` file, you can see the following files are included (directly or indirectly):

- `$(TOP)/build/defaults.cfg`
  Default settings for all `USE_[feature]` variables. Typically these are overwritten in the `settings.cfg` file.
  - `packages.cfg`
    - `packages-win.cfg` *or*
      `packages-posix.cfg`
    All `HAS_[feature]` variables, to indicate which external libraries are present on the system. Human-edited on Windows, generated by `configure` on Linux/MacOSX (from `packages-posix.cfg.in`)
- `$(TOP)/build/system.cfg`
  - `system-win-common.cfg`
    Common (operating-system/platform indepenend) configuration
  - `system-win.cfg` *or*
    `system-posix.cfg`
    Operating-system/platform dependend configuration, typically library files, include paths, etc. Human-edited on Windows, generated by `configure` on Linux/MacOSX (from `system-posix.cfg.in`)

16.2 Editing the packages-win.cfg file

Before continuing, make sure that `build/packages-win.cfg` file reflects your installation. This file will be read (through includes) when you run `srcdeps`. If you make any change to this file, you will need to rerun `srcdeps`.

16.3 Setup on MS Visual Studio

Open the Visual workspace file `srcdeps.dsw` located on `CLAM_DIR\build\srcdeps\`, and compile the `srcdeps` project. Make sure you are building the binary using the "release" configuration. That’s just for efficency reasons.

Visual will leave the output executable `srcdeps.exe` on the `CLAM_DIR\build\srcdeps\` directory. Now we have to configure Visual in order it can automatically execute `srcdeps` when some CLAM project (.dsp) needs to be redone.

16.3.1 Configuring Visual 6 to use srcdeps

1. Select Tools->Options ...
2. Click on the ’Directories’ tab
3. From the combo box labeled ’Show directories’ select ’Executable files’
4. Scroll the list below labeled ’Directories’ up to the bottom and double-click on the blank line.
5. Enter or browse the path to srcdeps.exe

16.3.2 Compiling our first CLAM example

Now we are going to compile the example application "SMSTools2" that comes with CLAM. Doing so you both will be able to play with an application that gives a taste of what CLAM can do, and will confirm --hopefully-- that the CLAM environment is correctly set up.

Open the Visual project `SMSTools.dsp` which is on `CLAM_DIR\build\Examples\SMS\Tools\` and compile it (it takes a while...)

If something went wrong check you followed the above steps and see the Windows Troubleshooting section See section 22.

You can see that the file `settings.cfg` appears in the project files view. This file defines settings of the project and after any change you are able to apply them to the dsp project by just right clicking over that file and choose 'compile' from the contextual menu. This execute `srcdeps` taking `settings.cfg` as a parameter

Important note:

Don't change configuration values or add files directly to the project, instead you should modifiy `settings.cfg` and re-run `srcdeps` as explained before. Otherwise you'll lose the manual changes the next time you run `srcdeps`

16.4 Setup on GNU/Linux

Change dir to `CLAM_DIR/build/srcdeps` and invoke `'make'` to build the `srcdeps` binary.

Once `srcdeps` has been created, we will check if all the required external libraries. To do so, change dir to `CLAM_DIR/build` and type:

```
$ autoconf -f
$ ./configure
```

16.4.1 Compiling our first CLAM example

Now we are going to compile the example application "SMSTools2" that comes with CLAM. Doing so you both will be able to play with an application that gives a taste of what CLAM can do, and will confirm --hopefully-- that the CLAM environment is correctly set up.

Change dir to `CLAM_DIR/build/Examples/SMS/Tools/`. Execute `make CONFIG=release` to generate dependencies and compile the sources (it can take a while...).

If something went wrong check you followed the above steps and see the GNU/Linux Troubleshooting section See section 22.

17 How to set up your own programs using CLAM

This chapter is a little tutorial for setting up an example application using CLAM and its build system. In first place we're going to compile a given simple application --very straight. And in second place we're going to go through all possible customizations in the build settings that the user might need to set up more complicated projects.

17.1 An out-of-the-box example

Following this steps you'll get an application which performs an FFT of a random generated audio and stores its spectral equivalent in a file in XML

1. Go into your new unpacked CLAM dir. Inside /build you'll find this dir: `compiling_against_CLAM_example`. Copy this dir at the same directory that contains the CLAM sources.

Note that this copied directory, apart from the .cxx file with a main function, contains a subdir named `build/` with the configuration files necessary to use the automatic CLAM build
Hint: in linux, cp -ar <src> <dst> copies a directory recursively

2. Edit the build/clam-location.cfg and change CLAM_PATH so it points to the CLAM dir --maybe you'll find that this value is already properly set.
3. At this point we need to have CLAM correctly deployed and srcdeps already compiled. See section 15 for how to do it.
4. Now our way forks depending on the choosed compiler
   - **gcc**
     Just call make with 'debug' or 'release' as argument:

     ```
     $ make CONFIG=debug
     ```

     This will generate the needed dependencies and compile the program
     Note: before running make you must have made the configure dance. See section 16.4.
   - **MS Visual 6**
     The first time we'll need to generate the .dsp file. For that we'll need to use the command line.
     - change your current dir to the `buid` subdir of our example directory
     - from this dir execute the srcdeps --which is in the `CLAM/build/srcdeps` dir-- passing the `settings.cfg` as argument. For example:

       ```
       $ ../../CLAM/build/srcdeps/srcdeps settings.cfg
       ```

     - srdeps generate a .dsp with the smallest subset of the CLAM implementation files needed to compile the example.
     - further runs of srcdeps can be achieved directly from the MS Visual IDE (compile settings.cfg). See section 16.3 for more details.

Hopefully we've been able to run the example and play with spectral data in XML. From here we'll see how to customize the build options and to create more complex projects

17.2 Customizing your project

Here we're going to go through the things you should know when setting your own project that compiles against CLAM and using its automatic build system.

First of all, we assume that you allready have created a root directory for your project

- we recomend to create a `build/` directory inside your project dir. There you're going to place all the configuration files that the build-system needs. There you must copy --from the `compiling_against_clam_example` dir-- the following files:
  - clam-location.cfg
  - defaults.cfg
  - Makefile
  - Makefiles.rules
  - settings.cfg
  - system.cfg
- maybe you'll like to organize your code starting from a `src/` directory. This is up to you but has to be taken in account when configuring the build files
- modify **clam_location.cfg** making the CLAM_PATH variable to point to the CLAM root directory. You should use the absolute path.
- modify the TOP variable of **Makefile**. It must point to your project root dir. Its value should

probably be just `../` but is also feasible that you want to spread severals Makefiles in different subdirectories of `build/` thus the TOP variable must be defined accordingly.

- **settings.cfg** is the most important file you'll need to tweek. Notice that if you want too build several executables you'll need to create a directory for each one containing both `settings.cfg` and `Makefile`.
  What follows is a list of the settings.cfg variables you might need to modify. See section 19.1 for more details.
  - TOP: must point to your project root.
  - PROGRAM: the name of your binary --the executable.
  - various library flags: changing its value {0,1} you can choose to link or not against several libraries (i.e. fltk, fftw, xerces-c, etc)
  - PRJ_SEARCH_INCLUDES: the directories that contains your headers. Notice that you don't have to add any CLAM directory here.
  - PRJ_SEARCH_RECURSE_INCLUDES: the same as before but now including all the recursive directory tree
  - SOURCES: all the entry points: the implementation files (i.e. .cxx files) that you'll want to compile and can not be automatically deduced. Write the source files relative to the TOP. I.e.:

    ```
    SOURCES = $(TOP)/program_using_clam.cxx
    ```

    Normally you'll only need to include the implementation file that contains the `main` function.
  - Note: More info about how the CLAM automatic build system works can be found here: See chapter III-Overview

  Now we should be ready for a nice compilation !

  If we are using **gcc** just call `make CONFIG=[debug|release]` standing in the proper directory --where the settings.cfg and Makefile are. This will both generate the dependencies and will compile the project.

  Otherwise, if you are stucked with **MS Visual** run the CLAM srcdeps.exe application from your directory --where the settings.cfg stands-- and you'll get a ready-to-use dsp file.

That's it, hopefully you are compiling your project agains CLAM, without dealing in manualy finding the CLAM sources it needs.

18 CLAM and QT toolkit library

CLAM gives to the user complete support if he wants to use QT to develop his graphical user interfaces. This library uses a special kind of macros in its code that needs a preprocessing step in order to work correctly. In order to activate this support USE_QT=1 must be declared in the settings.cfg file of the project.. There are two kinds of actions:

- Create classes with Q_OBJECT macro declared - The project needs to create a 'moc' file for each header which declares the macro
- Srcdeps searches for Q_OBJECT in the source files, and if the macro is found a moc file will be generated inside $(build_dir)/moc/.
- Use .uic files generated from qt designer - The project needs to generate moc files using this .ui, and compile them with the rest of the source code. In this case the user needs to specify inside settings.cfg the .ui files to be preprocessed, using the variable UI_FILES. (for example, UI_FILES = ControlPanel.ui). The header generated will have a .h extension (ControlPanel.h), and will be stored in $(build_dir)/uic/ directory. This header must be included from the source file that needs it.

19 CLAM build system configuration variables reference

There are two main kinds of config variables depending on the values they may take:

- Boolean variables - these can only have values of 0 or 1. Usually 0 means that the variable effect is disabled, and 1 that it is enabled.
- Textual variables - they are a string, for instance, a relative path to some file

Note for Windows users: you should use frontslashes '/', as directory separator, instead of backslashes '\', which is the usual way in Windows.

Depending on the effect, there are three kinds of variables:

- Build System variables - variables whose value just affects the CLAM build system behaviour while naming binaries or searching for certain files.
- External Libraries variables - variables whose value determines whether the build system will make your application link or not to some (or any) of CLAM external libraries.
- CLAM internal variables - these variables are mainly compile-time flags that activate/deactivate certain framework features or change some framework behaviour.

Now we will see a detailed explanation for each of the variables meaning and possible values.

19.1 Build system variables reference

- `TOP`
  (Textual) - Should contain the relative path from the settings.cfg file to the 'top' of the project source tree
- `PROGRAM`
  (Textual) - Should contain the name for the program binary
- `PRJ_SEARCH_INCLUDES`
  (Textual) - Should contain the lists relative paths, from settings.cfg location, to folders where you want srcdeps to look for binary dependencies, usually the folders where you have your sources. Note that srcdeps *will not* perform a recursively descent search on these folders.
- `PRJ_SEARCH_RECURSE_INCLUDES`
  (Textual) - Should contain the list relative paths, from settings.cfg location, to folders where you want srcdeps to look for binary dependencies, usually the folders where you have your sources. Note that srcdeps *will* perform a recursively descent search on these folders.
- `SOURCES`
  (Textual) - Should contain the source file that contains the application entry point. While building library binaries or not following for some reason the rule 'for each header file there exists a source file with the same name' then you should add the source relative paths, from current settings.cfg location.

Note that textual variable values should follow the GNU Makefile variable syntax. See section 21 if you are unfamiliar with GNU Makefile. If you are not working with GNU development tools and you are not interested in learning how to write Makefile scripts then take into account the following.

Note that you can assign several tokens (variable values separated by spaces) to the same variable as in:

```
PRJ_SEARCH_INCLUDES= ../../../foo/bar/whatever ../../../foo/bar/stuff ../../../foo/bar/thingamajig
```

sometimes you could have so many tokens that reading the variable assigned value can be hard. Then you may break the tokens into several lines as in:

```
PRJ_SEARCH_INCLUDES= \
        ../../../foo/bar/whatever \
        ../../../foo/bar/stuff \
        ../../../foo/bar/thingamajig \
```

note that the backslash, '\', tells srcdeps that the values assigned to PRJ_SEARCH_INCLUDES variable span several lines. Note that whenever you begin a new line you must type in a 'tab' character (spaces are not enough) and when you finish the last line you must ensure it finishes in a newline character (pressing Return should suffice).

19.2 CLAM configuration variables

- `CLAM_DOUBLE`
  (Boolean) - This variable controls whether CLAM::TData datatype is either a single precision floating-point number (ANSI C++ float type) or a double precision floating-point number (ANSI C++ double type).
- `CLAM_USE_XML`
  (Boolean) - This variable controls whether CLAM code is built with XML-based Object External Storage support. Disabling it could improve compiling speed as well as reduce code size.
- `CLAM_DISABLE_CHECKS`
  (Boolean) - This variable controls whether CLAM internal precondition, postcondition and invariant verification checks are performed or not. Deactivating it could improve code speed in spite of robustness.
- `CLAM_USE_RELEASE_ASSERTS`
  (Boolean) - This variable controls whether CLAM Asserts (see [documentación de los asserts]) behave equally in "debug" and "not debug" mode.

19.3 External libraries variables

- `USE_ALSA`
  (Boolean) - Tells the build system to make applications to link against ALSA. Note that this variable can only have effect on GNU/Linux systems.
- `USE_FFTW`
  (Boolean) - Tells the build system to make applications to link agains the FFTW library.
- `USE_FLTK`
  (Boolean) - Tells the build system to make applications to link against FLTK.
- `USE_DIRECTX`
  (Boolean) - Tells the build system to make applications to link against DirectX SDK. Note that this variable only has effect on Microsoft Windows(c) systems.
- `USE_PORTMIDI`
  (Boolean) - Tells the build system to make applications to link against Portmidi. Note that this variable only has effect on Microsoft Windows(c) systems.
- `USE_RTAUDIO`
  (Boolean) - Tells the build system to make applications to link against RtAudio. Note that this variable only has effect on Microsoft Windows(c) systems.
- `USE_PTHREADS`
  (Boolean) - Tells the build system to make applications to link against pthreads (POSIX threads library). Note that this variable only has effect on Microsoft Windows(c) systems.

- `USE_QT`
  (Boolean) - Tells the build system to make applications to link against Qt Toolkit.
- `USE_CPPUNIT`
  (Boolean) - Tells the build system to make applications to link against cppUnit library.

20 Generating CLAM binaries

TODO: No information about generating a CLAM library binary is available yet

21 Some useful links

To get some insight on the 'source dependencies issue' and GNU Makefile syntax you may visit the following links:
http://www.eng.hawaii.edu/Tutor/Make/

22 Build system troubleshooting

- **Problem:**I get some external library related compile or undefined symbols errors.
- **Cause 1:** Check your `USE_XXXX` directives in `settings.cfg` for the library to be linked.
- **Cause 2:** Check you have already downloaded CLAM external libraries and that the libraries are placed following the directions specified for your system.
- **Problem:** When "compiling" settings.cfg from within Visual Studio IDE a message appears telling me: 'Cannot open myproject.dsp for writing'
- **Solution:** Use the command-line method for generating the .dsp.
- **Problem:** Srcdeps crashes in Windows while generating dependencies.
- **Solution:** VisualC 6 have some bugs on code generation, please upgrade to the Service Pack 5.

23 Some common problems while using Microsoft Visual C++

23.1 Getting lots of LNK2001 errors: redefinition of C/C++ Standard Library symbols

Microsoft C/C++ ANSI Standard Library run-time binaries come in six different versions: Single Threaded Release, Single Threaded Debug, Multi Threaded Release, Multi Threaded Debug and Multi-threaded with Dynamic Linking Release and Multi-threaded with Dynamic Linking Debug. This multiplicity makes possible the following pitfall: suppose you have some external library, such as FLTK which links statically against, say, the Single Threaded Release version, and then you try to link against FLTK binary (and, of course the standard library binary) you get a number of linking errors about duplicate symbols such as malloc, realloc, etc. The way of getting rid of this is to make your project link dynamically against the Multithreaded Dynamic Linking Library. This can be achieved by doing the following: go to Project->Settings (or ALT+F7), select your project from the project browser on the left and then click on the C/C++ tab on the right, select Code Generation option on the Category combo-box and then you will see another combo-box labelled "Use run-time library:". Select there the "Debug Multithreaded DLL" option if your project is in Debug or the "Multithreaded DLL" option if otherwise. Note that this phenomenon not only arises with FLTK: it would be wise configure always the projects in this manner, since it is unharmful and saves many headaches about obscure linking errors.

23.2 Getting lots of compiling errors not related to your Project (What's config.h about?)

It is common practice amongst library developers to allow library users to alter some inner mechanisms or configure the library for a concrete platform (either OS, hardware or compiler) by "prefixing" all library source files with a C header file containing some macros that define a concrete behaviour or policy for the library binary. Then the users can undefine or define the necessary macros for obtaining its desired configuration. However, especially under Microsoft Visual Studio it is easy to forget about including this prefixing file. Whenever you get some unexpected or unrelated to your work compiler errors it is wise to check that Visual Studio is prefixing Library sources by doing this: go to Project->Settings (or ALT+F7), select your project from the project browser on the left and then click on the C/C++ tab on the right, and check that inside the Project Options textbox on the right appears the following /FI"config.h". If it doesn't appear add it. If it is there check that your "Additional Include Directories" on the "Pre-processor" category are pointing to.../../src/Defines/Windows. If the compiler still complains then the problem could be elsewhere and that is beyond the scope of this document :).

23.3 Not finding a user defined header

Usually this pitfall is caused by having not specified to the compiler the needed additional include directories. This can be done by: go to Project->Settings (or ALT+F7), select your project from the project browser on the left and then click on the C/C++ tab on the right, select on the "Category:" combo-box on the right the "Pre-processor" option. Then appears on the right a textbox labelled "Additional include directories:", and then just add the path **RELATIVE TO THE DIRECTORY WHERE THE .DSP FILE IS IN YOUR HARD DISK**. This could have the form ".../../draft/mything" or something similar.

23.4 My dynamic_cast's are failing for no apparent reason

Just check that your C/C++ Project Settings, on the category *Language* enable RTTI (Run Time Type Information). If this setting is already set then you should revise your code (and remember that temporaries have not a reliable virtual function table).

23.5 I am getting an Internal Compiler Error message!!!

The dreaded error messages '*par excellence*'. These 'errors' are signalled by Microsoft IDE whenever the compiler dies ungracefully due to a **Segmentation Fault**. These compiler crashes happen whenever the parser found something extremely standard or some of extremely convoluted C++ code. Here you have a list of possible causes for these errors:

- Using precompiled headers

  Sometimes, under Windows 2000, and especially whenever you have the option set but you are not using explicitly this Visual C++ feature you may get one these messages. If you do not intend to use them in your project, and you should not, deactivate this option from the project settings.
- Hardcore templates

  There are a number of C++ features related to templates that sometimes, even using them 'correctly' will make the parser to die hard. For instance things like:

```
template < typename A, typename B >
void foo_function( int t )
{
}
```

```
template <>
void foo_function< someA, someB >( int t )
{
}
int main( void )
{
   foo_function< someA, someB >( 3 );
   return 0;
}
```

The parser will die at the line containing the call to foo_function.

- Not closing properly strategically placed brackets

  Not closing namespace declarations usually make the parser to die, since for some reason it confounds and match the bracket with the first available. Even if this 'first available bracket' is on another file. This may also happen with scope declarations such as class scopes, loop scopes, etc.
- Using nested namespaces

  Visual C++ 6.0 behaviour is erratic at best. Some features do not work as expected (i.e. as stated on the ANSI) and others, such as references to identifiers declared inside nested namespace from outside those namespace (i.e. MTG::MTGTest::foo or nasty errors as FFT::FFT::Do) may kill the parser outright under certain conditions.
- Convoluted C

  Difficult expressions such as
  `MemFunc &memFunc(*(MemFunc*)(void*)(ftor.memFunc))`
  are also Microsoft parser's silver bullets.

23.6 My Visual C++ is behaving weirdly and signalling non-sense error messages

Visual C++ hosts a quite impressive list of bugs, so this is clearly a sign that you have uncovered some particularly nasty or weird. We cannot give you any outright solutions, since many times we are hassled by this kind of issues. But sometimes rebuilding your project from scratch or disable function-level linking (Incremental linking feature) helps to make thins saner. Good Luck!

23.7 The compiler does not find FL/Flxxxx.H or DOM/xxxx.hpp

Note that when in your code you use #include <something.h> you are telling the compiler that that header can be found in the System Headers path. This feature has a lot of sense under UNIX where you have a /usr/include or similar dir where all headers are neatly deployed. But under Microsoft Visual Studio this can mean two things: that either you have to copy manually the folders with the includes into Microsoft Visual Studio directory/VC98/Include dir or follow our guidelines for creating a *sandbox*.

This FAQ-like section does not covers all possible issues you might find while using Visual C++ but we hope they solve the most typical of them. Feedback about new issues or alternative solutions to situations described above is welcomed.

24 Some common problems while using GNU/Linux and GNU C++ Compiler

The motivation of executing autoconf & configure in CLAM_PATH/build is to check if your system configuration will be able to compile and execute CLAM applications. You can get a lot of different problems related to external libraries, so we have created a compilation of the most common,

classified by the step where this problem arises.

24.1 FFTW

24.1.1 Getting error when trying to locate fftw header/libs

You can search in config.log where is the error if the configure stops at the lib checking. Surely it will be related to the installation of fftw, because CLAM needs both versions (float and double) of libs and headers. Please check that.

24.2 FLTK

24.2.1 Checking fltk libs fails and config.log contains compiler errors

Check that you use fltk-1.1.4, because some interface has been changed (and some has been created), so the errors can be related to the fltk version you are trying to compile against.

24.2.2 Checking fltk libs fails and config.log contains linking errors, or the program test couldn't be executed.

Make sure you have `LD_LIBRARY_PATH` pointing to all the locations needed

24.2.3 fltk-config not found

You have a fltk version without this utility (older than 1.1), or the `PATH` variable doesn't point against its location. Set `PATH` correctly or download a new version from fltk official web (or the binaries you can get in CLAM web). Anyway, it should work without this program if you have fltk-1.1.4 or newer.

24.3 QT

24.3.1 No qt headers found! having qt installed correctly in the system

Use `QTDIR` variable to tell the configure where you have qt (like `export QTDIR=/usr/qt/3`).

24.3.2 Found qt headers but crashed testing lib because library (qt or qt-mt) not found.

Make sure `QTDIR` is correctly pointed, and you have a qt version newer to 3.0.

24.3.3 Compiler errors related to exit and throw functions

Remove `config.cache` and rerun `autoconf` with 2.5X version. (like `autoconf-2.57`).

24.4 XERCES

24.4.1 Checking xerces libs fails and config.log contains compiler errors

You'll need exactly xerces 2.3, because in newer versions they have broken the interface, and CLAM won't be capable of link against it.

24.4.2 Checking xerces libs fails and config.log contains linking errors, or the program test couldn't be executed

Make sure you have `LD_LIBRARY_PATH` pointing to all the locations needed

24.5 STL

24.5.1 Getting these errors:

```
checking for std::vector::at() method in libstdc++... no
checking for standard sstream header in libstdc++... no
checking if stringstream::str() returns std::string in libstdc++... no
```

Remove `config.cache` and rerun `autoconf` with 2.5X version. This error is related to a bug inside autoconf-2.1X combined with gcc-3.X, so you'll need to run an autoconf version newer in order to create a correct configure for this compiler version.

24.6 Common problems trying to compile and execute CLAM applications

24.6.1 Compiling is ok but getting errors trying to link/execute the program

Make sure you have `LD_LIBRARY_PATH` pointing to all the locations needed. It's needed when you have installed external libraries in a local location. If you have, for example, fltk and xerces installed in the same directory where you have CLAM, you should set `LD_LIBRARY_PATH` in this way:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/xerces/lib:/path/to/fltk/lib/
```

IV Usage tutorial

25 Introduction

This document provides a first glance to the usage of CLAM as a signal processing and sound synthesis framework for C++ applications, following several simple code example. The source code this the examples is included with the source code of the library (in the directory `examples/Tutorial`.

CLAM consists of a set of C++ classes. There are two main kinds of classes in the library:

1. *Processing Classes*, which perform the signal processing operations.
2. *Processing Data Classes*, which are placeholders for the data that flows between Processing objects.

A Processing class has a set of data **inputs** and a set of data **outputs**. Each of them will receive *Processing Data* of an specific type. Several such data types exist (also as C++ classes): Audio, Spectrum, statistical descriptors, etc.

A processing class can also have a set of input and output **controls**. These are attributes that can be used to change the Processing object behavior once it is ‘‘running’’ setting internal run-time execution variables.

Finally, a Processing Object is created and configured using a **configuration** object, which may define its architecture and its behavior. This configuration may be set at object construction time, or later.

26 Instanciating Processing objects

Each processing class (and its configuration class) is defined in a different library header file. To instanciate an FFT processing object, for example, you should do the operations shown below.

```
// We include the FFT class and associated Processing Data objects
#include <FFT.hxx>
#include <Audio.hxx>
#include <Spectrum.hxx>

// And C++ input-output library to write the output message.
#include <iostream>

int main(void)

{

   // This creates an FFT configuration object
   CLAM::FFTConfig my_fft_config;

   // This sets some configuration parameters.
   my_fft_config.SetAudioSize(1024);
   // And this finally creates the FFT object.
   CLAM::FFT my_fft(my_fft_config);
   std::cout << my_fft.GetClassName() << std::endl;
}
```

If you have a look at the declaration of the FFTConfig class in the `FFTConfig.hxx` file (you will find this header files in the `src/Processing/Analysis` directory of CLAM sources), you will notice that it is not declared as a standard C++ class. This is because FFTConfig is a so called

DynamicType. More information about CLAM dynamic types is provided in the specific chapter in this document, although some useful features will be described below.

This example also shows that all the CLAM related classes are declared in the `CLAM` namespace. Thus you must either include the `CLAM::` prefix in each of the names, or include the following line of code at the beginning of your program, to avoid prefixing:

```
using namespace CLAM;
```

Note: Importing a namespace globally as it is done in the previous line though is not usually a good idea in a less than trivial project.

27 Processing Data

Instanciation of processing data objects is usually simpler that the creation of processing objects, as shown below.

When working with processing data, you must keep in mind that the data classes are *Dynamic Types*. This allows you to add or remove data attributes at run time (which means that memory for those attributes is allocated/freed at run time).

Both `SpectralRange` and `Scale` are dynamic attributes in the spectrum class, and as such they can be accessed using `Set` and `Get` methods like in the sample code above.

Some complex processing data classes, like the Spectrum, provide dynamic attributes for alternative representations of the same data. You can have the spectral samples stored as `CLAM::Complex` objects (Cartesian coordinates), as `CLAM::Polar` objects, or as separate magnitude an phase floating point arrays. You can add or remove these dynamic attributes to fit your application.

The `Add`*AttributeName*`()`, `Remove`*AttributeName*`()` and `UpdateData()` methods are the general mechanism for adding and removing dynamic attributes to/from dynamic types.

Some complex data classes, such as the Spectrum, also provide a configuration class, which can be used to ease the construction of multiple objects with common settings.

For example, if you know what dynamic attributes you need when you construct the object, you can specify them in the configuration object. The `SpectrumConfig` has a `Type` attribute, which stores a set of Boolean `Flags` describing which dynamic attributes in the `Spectrum` object are to be "added" at construction time. This is shown in below.

You have two different ways to set or change the value of a dynamic attribute. The easiest one is completely overwriting the previous value. This is illustrated in below. It also introduces a new processing data class (`Audio`).

This example has also introduced the class `Array<T>`. This is one of the many utility template classes defined in the library. Some of them are similar to some STL classes. `Array<T>`, for example, is similar in functionality to the `std::vector<T>` class (the reasons for not actually using the std vector have been thoroughly discussed and are still a matter of controversy between CLAM developers). The attributes which contain sample chunks in both the `Spectrum` and the `Audio` classes are Array attributes. Note that the `Array<T>` class is not a ProcessingData class (i.e., it cannot be fed directly to a processing object).

In the example shown below, we can see the other way to access dynamic attributes: getting a reference to them and modifying it. This is a bit more obscure, but more efficient. Back to the Spectrum class, the example shows how to convert the complex samples among the different representations (which are stored as different dynamic attributes in the Spectrum object).

28 Usage examples

Now that we are familiar with ProcessingData classes, we can start to use Processing Objects to perform computations on data objects. This is best shown through some examples.

The first one, in shows how to perform the FFT of an Audio object and store it in a Spectrum object. The key parts are the calls to the `Start` and `Do` methods of the FFT.

If you are wondering about the strange data sizes used in this examples, you should take in account that the FFT processing class actually performs a real DFT. Because of spectrum symmetry, only the first half of the spectrum is needed, so the second half is not calculated.

The next example illustrates how to use controls in a processing object. A Frequency Domain Filter is created, and the filter characteristics are set using its input controls.

V Usage examples

Appart from the previous tutorial the CLAM repository also includes some examples in order to show how can you do the basic operations with CLAM: loading audio files, playing sounds, using Processing class... This is the list of related examples, each one of them fully detailed with comments.

- **Audio file Reading example**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/AudioFileReading/`
  - Sources location:
    `examples/AudioFileReading_example.cxx`
  - Complexity:
    `Low/Medium`
  - Keywords:
    `Audio file I/O, Processing usage`
  - Pre-requisites:
    `Knowledge about basic Processing usage.`
  - Description:
    `Shows how to load an arbitrarily formatted audio file using CLAM utilities.`
- **Audio file Writing example**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/AudioFileWriting/`
  - Sources location:
    `examples/AudioFileWriting_example.cxx`
  - Complexity:
    `Low/Medium`
  - Keywords:
    `Audio file I/O, Processing usage`
  - Pre-requisites:
    `Knowledge about basic Processing usage.`
  - Description:
    `Shows how to save an arbitrarily formatted audio file using CLAM utilities.`
-
- **Frequency domain filter usage example**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/FDFilter`
  - Sources location:
    `examples/FDFilterExample.cxx`
  - Complexity:
    `Low`
  - Keywords:
    `Processing usage, Digital Signal Processing, Frequency Domain.`
  - Pre-requisites:
    `Knowledge about basic Processing usage.`
  - Description:
    `Shows how to configure and use the FDFilterGen Processing`

object.

- **FFT (fftw implementation) usage example**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/FFT
  - Sources location:
    examples/FFT_example.cxx
  - Complexity:
    Low
  - Keywords:
    FFT, Digital Signal Processing, Spectrum usage, XML, Persistent objects.
  - Pre-requisites:
    Basic knowledge of Processing interface.
  - Description:
    Shows how to obtain the real Fourier transform of a given audio signal. Also shows how to *store* a CLAM object i.e. obtain a persistent copy.
- **Descriptor Computation Example**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/DescriptorsComputation
  - Sources location:
    examples/DescriptorComputation_example.cxx
  - Complexity:
    Medium
  - Keywords:
    Descriptors, Feature Extraction, Statistics
  - Pre-requisites:
    Previous knowledge on CLAM Processing classes and the Processing Data classes included in the repository.
  - Description:
    A Descriptor is a special data Container that holds the result of applying statistical computations to an existing Processing Data. In this example the basic functionality of descriptors and statistics is shown. Descriptors are finally dumped into XML.
- **Processing Life Cycle example**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/ProcessingLifeCycle
  - Sources location:
    examples/ProcessingLifeCycle_example.cxx
  - Complexity:
    Medium
  - Keywords:
    Extending Processing abstract class, Processing life cycle.
  - Pre-requisites:
    Knows about CLAM Dynamic Types.

- Description:
  Shows how write a new CLAM::Processing. Tries to give some insight into Processing life cycle - start, Configure(), Start(), Stop(), -etc.

- **Object persistence through Dynamic Types**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/PersistenceThroughDTs
  - Sources location:
    examples/PersistenceThroughDTs_example.cxx
  - Complexity:
    Medium
  - Keywords:
    Object persistence, XML processing, XML Schema, Dynamic Types.
  - Pre-requisites:
    Knowledge of DynamicType's API, Knowledge of XML and XML Schema standards.
  - Description:
    This example shows how to obtain a persistent copy of an object, encoded as a well-formed, valid, XML document.

- **Visualization Module Plots: single function plot**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/SinglePlot_1
  - Sources location:
    examples/SinglePlot_example.cxx
  - Complexity:
    Low
  - Keywords:
    CLAM GUI services, simple data visualization.
  - Pre-requisites:
    Familiarity with CLAM::Array and CLAM::BPF.
  - Description:
    This example shows how to plot on the screen some data object part of a simple DSP application.

- **Visualization Module Plots: multiple function plot**
  - Makefile / Visual C++ project location:
    build/Examples/Simples/MultiPlot
  - Sources location:
    examples/MultiPlot_example.cxx
  - Complexity:
    Low
  - Keywords:
    CLAM GUI services, simple data visualization.
  - Pre-requisites:
    Familiarity with CLAM::Array and CLAM::BPF. It is recommended to take a look first on the single function plotting example.

- Description:
  This example shows how to plot on the screen some data object part of a CLAM-based DSP application, as well as combining several functions in the same plot window.

- **LPC usage example**
  - Makefile / Visual C++ project location:
    `build/Examples/LPC`
  - Sources location:
    `examples/LPCAnalysis_example.cxx`
  - Complexity:
    `Medium`
  - Keywords:
    `Linear Prediction Coding, Fourier Transform, multiple function plotting.`
  - Pre-requisites:
    `Notions of DSP analysis techniques.`
  - Description:
    `This example shows how to analyze a given audio signal using the LPC and associated ProcessingData's. Also shows to compare the approximation achieved by the LPC algorithm, and the one achieved by Fourier Transform.`

- **A simple threaded speech analysis application**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/ThreadedProcessing`
  - Sources location:
    `example/ThreadedProcessing_example.cxx`
  - Complexity:
    `Medium`
  - Keywords:
    `Concurrent programming, threads, GUI`
  - Pre-requisites:
    `Notions about concurrent programming issues, some background in signal processing, some familiarity with most usual CLAM objects ( Processings, etc.)`
  - Description:
    `Unfinished example - growing overly complex.`

- **Playing a WAVE file**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/FilePlayback`
  - Sources location:
    `examples/FilePlayback_example.cxx`
  - Complexity:
    `Low`
  - Keywords:
    `Audio device I/O`
  - Pre-requisites:

Minimum familiarity with CLAM objects such as Processing and ProcessingData.

- Description:
  This examples shows how to load a WAVE file and play it with your soundcard.

- **SDIF I/O, Segments and plots**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/SDIF_And_Segment
  - Sources location:
    examples/SDIF_And_Segment_example.cxx
  - Complexity:
    Medium
  - Keywords:
    DIF, CLAM Segment, SMS Synthesis process, Audio Device I/O.
  - Pre-requisites:
    Basic knowledge of Processing objects interface, basic knowledge of SMS Analysis algorithm byproducts.
  - Description:
    Shows how to restore a CLAM::Segment object stored into a SDIF file, and inspect visually its contents.

- **Using CLAM Networks**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/NetworkUsage
  - Sources location:
    examples/NetworkUsage_example.cxx
  - Complexity:
    Low
  - Keywords:
    CLAM Network, Flow Control, Supervised Mode.
  - Pre-requisites:
    Knowledge of Processing objects interface.
  - Description:
    Shows how to use the CLAM Network Processing interface to create and connect easily CLAM Processings.

- **Storing and loading CLAM Networks**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/NetworkPersistence
  - Sources location:
    examples/NetworkPersistence_example.cxx
  - Complexity:
    Medium
  - Keywords:
    CLAM Network, Flow Control, Supervised Mode, Serialization, Audio File I/O, Audio Device I/O
  - Pre-requisites:
    Basic knowledge of Processing objects interface, basic

knowledge of serialization module, knowledge about using CLAM Processing Networks.

- Description:
  Shows how to store a CLAM::Network to a xml file, restoring its definition to another network. It show how to load and store audio files and play them, too.

- **How to create and use a Processing with Controls**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/Controls
  - Sources location:
    examples/ProcessingObject_controls_example.cxx
  - Complexity:
    Medium
  - Keywords:
    Processing, Controls
  - Pre-requisites:
    Knowledge of Processing objects interface and some previous reading on what CLAM controls are and how they are supposed to behave.
  - Description:
    A Processing class with different kinds of input and output controls is declared. Then it is used by illustrating how controls are connected, modified and read.
- **How to create a Processing with Ports and Controls and use it**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/PortsAndControlUsage
  - Sources location:
    examples/PortsAndControlUsageExample/
  - Complexity:
    High
  - Keywords:
    Flow Control, Audio Device I/O, Extending Processing abstract class, Processing life cycle, Ports, Controls, Nodes
  - Pre-requisites:
    Advanced knowledge of Processing objects interface, knowledge of flow control and nodes system.
  - Description:
    Shows how to create a custom processing class with ports and controls interface, in order to use it with another processings creating an small processing chain.
- **Creating and using a CLAM Composite Processing Object**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/POComposite
  - Sources location:
    examples/POCompositeExample.cxx
  - Complexity:
    Medium
  - Keywords:

Composite, Children

- Pre-requisites:
  Good knowledge of Processing classes and their whole interface.
- Description:
  This example illustrates how to create a static composition of Processing objects using the Processing Composite construction available in CLAM. A "big" Processing class is declared by composing with basic Processing classes. Then this class is used to show how it interfaces and behaves.

- **Creating a Basic Audio Application using the CLAM Application classes**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/AudioApplication
  - Sources location:
    examples/AudioApplicationExample.cxx
  - Complexity:
    Low
  - Keywords:
    Audio I/O, Application, Oscillators
  - Pre-requisites:
    Basic knowledge of Processing objects interface and audio IO in CLAM
  - Description:
    Shows how to develop a basic audio application using the CLAM application classes. For doing so a basic Oscillator is used and its output is sent to the soundcard.
- **Converting a MIDI file into an XML Melody**
  - Makefile / Visual C++ project location:
    build/Examples/Simple/MIDI2XML
  - Sources location:
    examples/MIDI2XMLExample.cxx
  - Complexity:
    High
  - Keywords:
    MIDI, XML, Data, Controls
  - Pre-requisites:
    This example may be used as is without not much previous knowledge. But in order to understand its internals a good knowledge on CLAM Processing classes, including controls and ports, is necessary. Prior basic knowledge on the MIDI protocol is also necessary.
  - Description:
    This examples batch processes all the .mid files contained in a given folder (and subfolders recursively) and converts them into xml files. In order to do the input MIDI controls are converted to data and then dumped into XML using CLAM infrastructure. This example was implemented after a question posted to the MIR (Music Information Retrieval) mailing list.
- **Creating a MIDI Synthesizer**

- Makefile / Visual C++ project location:
  `build/Examples/MIDISynthesizer`
- Sources location:
  `examples/MIDI_Synthesizer_Example.cxx`
- Complexity:
  `High`
- Keywords:
  `MIDI, Synthesizer, Instrument, Controls`
- Pre-requisites:
  `Previous knowledge on MIDI and additive synthesis may help. Previous knowledge on CLAM Processing classes and Controls is also required.`
- Description:
  `This example implements a basic additive synthesizer with ADSR control. The synthesizer is controlled from incoming MIDI messages.`

- **Inputing and outputing MIDI messages**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/MIDIIO`
  - Sources location:
    `examples/MIDIIOExample.cxx`
  - Complexity:
    `Medium`
  - Keywords:
    `MIDI, Controls`
  - Pre-requisites:
    `Previous knowledge on the MIDI protocol may help but is not strictly necessary. Some prior knowledge on CLAM Processing classes and controls is required.`
  - Description:
    `A basic input MIDI stream is opened and the input events are sent on the output.`
- **Implementing a spectral analysis**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/SpectralAnalysis`
  - Sources location:
    `examples/SpectralAnalysis_example.cxx`
  - Complexity:
    `High`
  - Keywords:
    `Spectral Analysis, FFT, spectrum, Composite`
  - Pre-requisites:
    `Previous knowledge on the basic concepts in the framework (Processing objects, Processing Data objects...) from a user's perspective. Some previous knowledge on signal processing might also help.`
  - Description:
    `A spectral analysis scheme is implemented. This schem is based on the STFT and includes windowing, overlapping and`

zero padding.As a matter of fact this example includes much of the functionality in the SMSTools, but it is more isolated and therefore easy to read/understand

- **Implementing a spectral peak detection algorithm**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/SpectralPeakDetect`
  - Sources location:
    `examples/SpectralPeakDetect_example.cxx`
  - Complexity:
    `Medium`
  - Keywords:
    `Spectral Analysis, spectrum, Spectral Peaks, Composite`
  - Pre-requisites:
    `Previous knowledge on the basic concepts in the framework (Processing objects, Processing Data objects...) from a user's perspective. Some previous knowledge on signal processing might also help. As a matter of fact, before reading this example it is recommended that you first understand the previous Spectral Analysis example.`
  - Description:
    `Using the output from a previously performed spectral analysis, the peak detection algorithm in the CLAM repository is used in order to find the most prominent peaks in the spectrum.`
- **Using the Spectrum class**
  - Makefile / Visual C++ project location:
    `build/Examples/Simple/Spectrum`
  - Sources location:
    `examples/Spectrum_example.cxx`
  - Complexity:
    `Medium`
  - Keywords:
    `Spectrum, ProcessingData, Dynamic Types`
  - Pre-requisites:
    `Previous knowledge and experience with Dynamic Types and basic Processing Data classes is recommended.`
  - Description:
    `The Spectrum class is the most complex Processing class in CLAM and this example gives a thorough insight on its interface and usage.`

VI Dynamic Types

29 Scope

This section is addressed to the users that want to learn the basics aspects and usage of Dynamic Types (DTs for short). You may also find a section devoted to Dynamic Types in the "Developers" part of this document. This latter is aimed to explain how to create new classes that derive from the base DT, and goes into some details regarding their functionality.

30 Why Dynamic Types ?

Though it might be a quite controversial issue, there are three main reasons for the decision of implementing and using DTs in CLAM.

1. There is a need in some core classes of the library, of working with types with a large number of attributes, i.e.: the descriptors of audio segments, that in some cases only a small subset is needed, and so could represent a waste of space if its memory is always allocated. DT can instantiate and de-instantiate attributes at run-time, and do it in such a way that its interface is the same as if they where C++ normal attributes.
2. We want support for working with hierarchic or tree structures. That means not only composition of DTs but also aggregates of them (lists, vectors, etc. of DTs). With such compositions of DTs, we can use assignation, and two clone member functions: ShallowCopy () and DeepCopy(), the good thing is that they come free; we don't need to write these members in none of the DT concrete classes.
3. We obtain introspection of each DT object. That is the ability to know the name and type of each dynamic attribute, to iterate through theses attributes, of having some type specific handlers for each. One clear application of introspection is storage support for loading from, and storing to a file, of a tree of DTs. Of course all this is implemented generically, so appears transparent to the user. At this point we have XML support implemented. Other profits we take from introspection in DT are debugging aids.

31 Where can DT be found within the CLAM library?

All classes deriving from ProcessingData base class are DT. The concrete ProcessingConfig classes as well as the ProcessingDataConfig classes are also DT.

32 Declaring a DT

When we say that dynamic attributes can be instantiated at run time, we mean that we can do so with the previously declared dynamic attributes. Let's see an example. Imagine we want to model a musical note with a DT. We declare it like this:

This is a macro-based declaration, right now there are three different macros: `DYNAMIC_TYPE` for expanding the concrete DT constructors, `DYN_ATTRIBUTE` for declaring each dynamic attribute and `DYN_CONTAINER_ATTRIBUTE` for declaring any STL interface compliant container.

We will now explain these three more in depth:

1. DYNAMIC_TYPE this macro expands the default constructor of the concrete DT being declared. The first parameter is the total number of dynamic attributes, and the second one the class name.

If the writer of a DT derived class sees the need of writing a customized default constructor or other constructors it can be done using the initializers. See section 82.

2. DYN_ATTRIBUTE it is used to declare a dynamic attribute. It has four parameters, the first one is the attribute order (needed for technical reasons of the DT implementation), the second one is the accessibility (public, protected or private) the third one is the type: it can be any C++ valid type including typedef definitions **but not references** (& finished) **or pointers** (* finished). If you still think you need pointers as dynamic attributes then read the pointers section (section 85).

The forth and last parameter is the attribute name, it is important to begin in upper-case because this name (let's call it XXX) will be used to form the attribute accessors GetXXX() and SetXXX(.), thus the XXX must start in upper-case, following the coding style of the library (See chapter XVI)

Returning to the example above, each DYN_ATTRIBUTE macro will expand a set of usable methods:

```
float& GetPitch(), void SetPitch(const float&),void AddPitch(), void RemovePitch(), bool HasPitch()void StorePitch(Storage&) const, bool LoadPitch(Storage&)
```

Of course GetPitch and SetPitch are the usual accessors to the data. AddPitch and RemovePitch, combined with UpdateData that will be explained latter on, will instantiate and de-instantiate the attribute. HasPitch returns whether Pitch is instantiated at this moment. Finally StorePitch and LoadPitch are for storage purposes, and will be explained in section 35

3. DYN_CONTAINER_ATTR: The purpose of this macro is to give storage (only XML by now) support to attributes declared as containers of objects. For providing this service, we need that container to fulfill the STL container interface, so all the STL collection of containers is usable. This macro has five parameters, one more that DYN_ATTRIBUTE: the attribute numeration, accessibility, the type, the name of the attribute and finally the new one: the label of each contained element that will be stored.

33 Basic usage

Once, the concrete DT Note has been declared, we can use it like this:

```
Note myNote; // create an instance of the DT Note
```

Now myNote, have no attribute instanciate. We can activate attributes this way:

```
myNote.AddPitch(); myNote.AddNSines(); myNote.AddSines();
```

Or in the case that we want all of them, it is better to use AddAll. (This method is not macro generated as AddPitch, but is a DT member available in any concrete DT.
)

```
myNote.AddAll();
```

As this kind of operations require memory management, we want to update the data, with its possible reallocations only once for every modification of the DT shape or structure (which can imply many individual adds and removes). We'll use the DynamicType UpdateData operation for that purpose:

```
std::cout << myNote.HasPitch() // writes out: 'false'myNote.UpdateData();std::cout << myNote.HasPitch() // writes out: 'true'
```

And now all the instantiated attributes can be accessed as usual, using the accessors GetXXX and SetXXX. For example:

```
MyNote.SetPitch(440);cout<<myNote.GetPitch()<<endl;myNote.SetSines().push_back(440).push_back(440*2);myNote.GetSines().push_back(440*3).push_back(440*4); // etc. int i=my
```

More about adding, removing and updating attributes:

It is important to learn the exact behaviour of adding and removing DT attributes. It can be summed up to three ideas:

- `GetXXX` and `SetXXX` operations over non-instantiated attributes will rise an exception `ErrDynamicType`. So these operations should be protected with an if (`HasXXX()`) clause in places of the code where there is no safety about the presence of `XXX`.
- Besides, `AddXXX` and `RemoveXXX` can be used safely (don't rise any exception). Basically what they do is set and unset internal flags. For example, if the attribute Pitch exists and we do RemovePitch() it will be marked as "removed" (waiting for the `UpdateData()` to perform the actual removal). Now, if before updating data we call `AddPitch()`, then it detects the mark and the effect is just to unset this flag. If we insist with another `AddPitch()` it will have no effect (and no flag will be set), because it is allready instantiated. The case of adding a non-instantiated attribute is symmetric as the case of removing an instantiated one.
- `UpdateData()` is a safe and efficient operation: with safe what we mean is that it has no effect (and doesn't rise exceptions) in case that the DT needs no memory update, and it is efficient in the sense that the checking for changes in the dynamic shape doesn't involve any traversal of the attributes but only cheking a global flag. Moreover, `UpdataData()` returns a boolean that says if it has been necessary to update the data. Sometimes it is very useful do things such as :

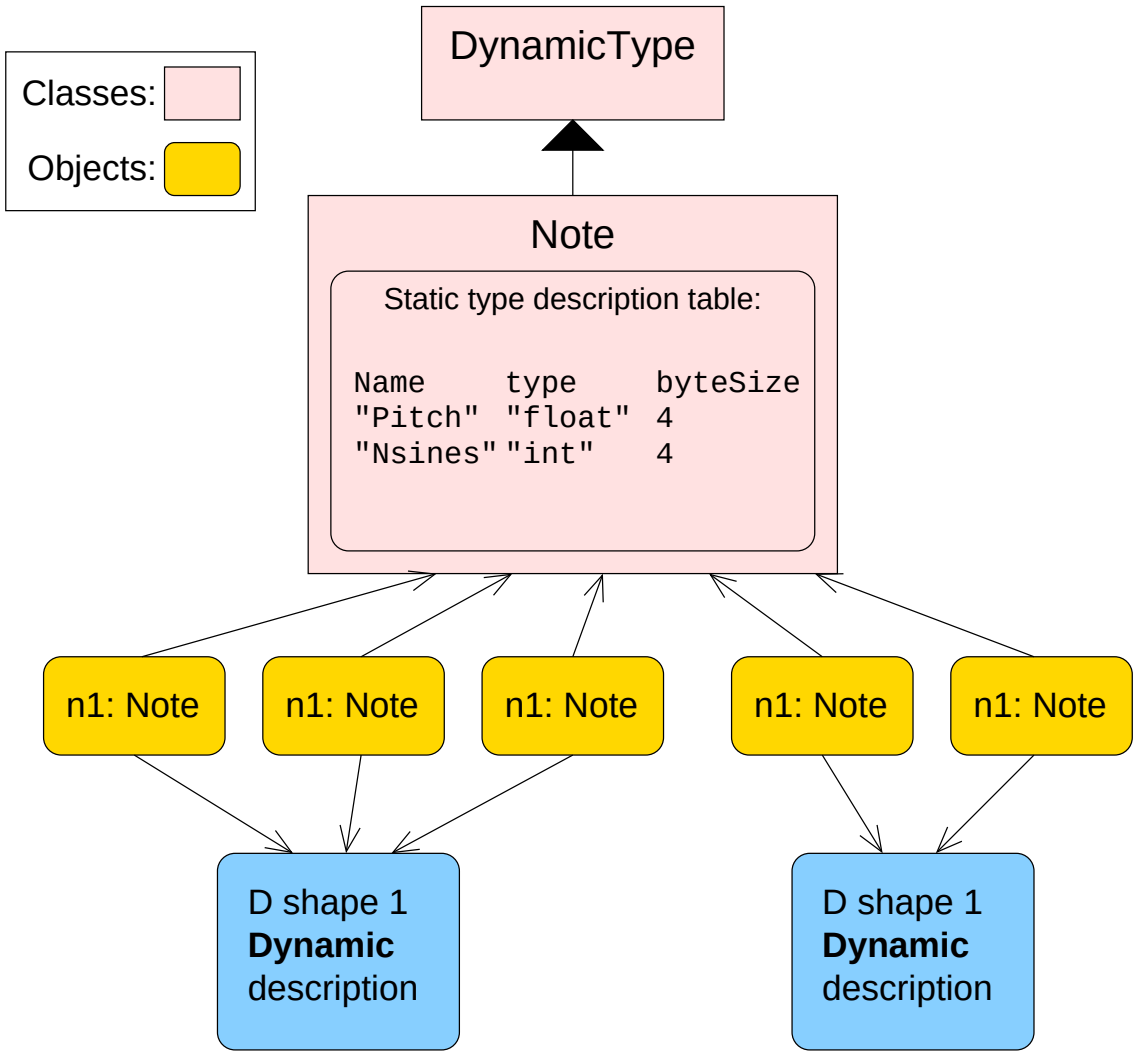
```
// here we don't know if Pitch exist.myNote.AddPitch();if (myNote.UpdateData()) {   // yes, it existed   // ...} else {   // no, it didn't was there till now.   // ...}
```

34 Prototypes and copy constructors

It's been said that a the dynamic shape of a DT is the set of instantiated attributes. The description of this shape is stored in a table and can be shared between various DTs. Thus this brings us to the idea of prototypes and creating objects by cloning the dynamic shape of others, this is exactly how the default copy constructor works.

```
Note n1(myNote), n2(myNote), n3(myNote), n4(myNote);
```

Now n1, n2, n3 and n4 all share both static information of their type and the dynamic shape information, and we say that myNote is their prototype, or that they all share the same prototype. It is important to notice that this happens even if the prototype (myNote) doesn't have its data updated. If we decide to change the shape of one of these objects, i.e. we want to add new attributes to n4, this operation will automatically create a new prototype and so a new dynamic shape as the next figure shows:



Of course, the copy constructor also copies the data of each instantiated attribute from the source object into the new object. This copy is made using the corresponding copy constructor. This means that the copy constructor makes a so-called "deep copy" (that means recursives copies of its sub-elements) of any composition of DT and aggregates (i.e. using STL containers) of DTs.

More on copy constructors (i.e. with the `shareMemory` flag) can be learnt directly from the javadoc source documentation.

35 Storing and Loading DTs

This section only explores a feature that is particular to the Dynamic Types: debug-time information. If you should need more details about how to store and load DT from and to XML please refer to chapter IX.

35.1 How to explore a DT at debug time

Since the dynamic data of a DT is not stored as regular C++ attributes, it might be difficult to explore in the usual way from the debugger. Thus we have provided the DT base class with a `Debug()` method that can be called from the debugger environment and basically does two things:

- writes information regarding internal parameters and the dynamic shape of the object to the standard output, and
- writes out a ’Debug.xml’ file (placed at the working directory) with the XML content of the object.

Note that the XML load and store, including storing the debug file, will only work if the CLAM macro `CLAM_USE_XML` was set when compiling.

VII Processing classes

36 Introduction

This chapter gives a short introduction to CLAM library development issues to those who intend to write a new processing class. The chapter assumes certain familiarity with the use of the library, so if you are completely new to it, you might want to read the usage tutorial first.

The rest of this section will give an overview to the library class hierarchies. Section 37 gives a brief overview of the main tasks you will need to perform while implementing a processing class. In section 38 the construction and configuration interface of processing classes will be described. Section 39 will describe their execution interface. Finally, the following sections will summarize some implementation details to keep in mind while writing a processing class, especially exception handling and code testing.

36.1 Class hierarchies

When making a new processing class, it is important to be familiar from the beginning with the `Processing` and the `ProcessingConfig` classes hierarchies shown in figure f 5. This figure shows a small collection of the available processing classes, and their related configuration classes.

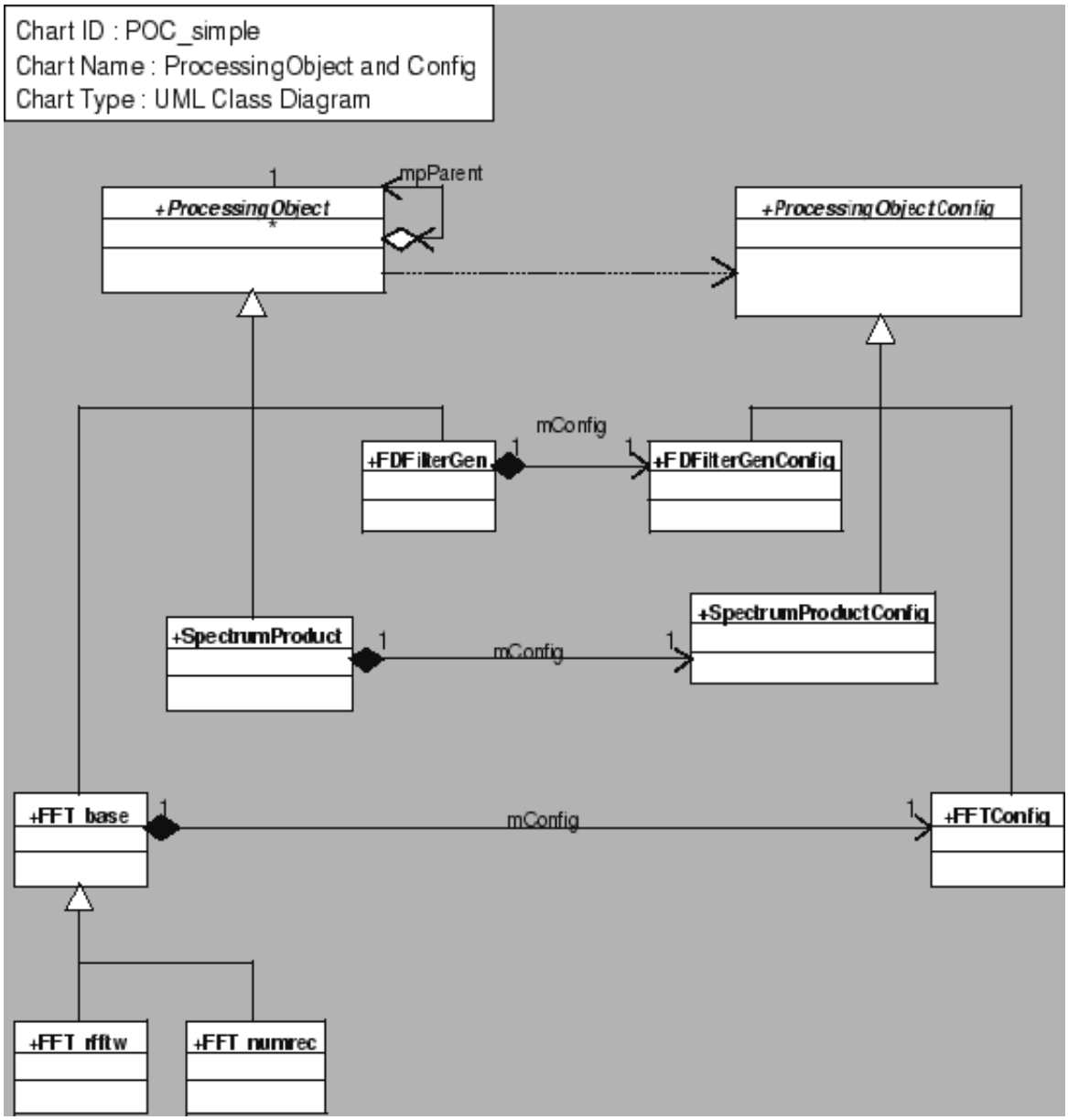


Figure 5: Processing and Configuration classes hierarchy

As you can see in the figure, there is a base `Processing` class from which all other processing classes must derive. This class provides an abstract interface for processing classes. The class is defined in the following source file:

`$(TOP)/src/Processing/Processing.hxx`

You should also be familiar with `ProcessingData` classes, at least with the fundamental ones. They are described in a different chapter of this document.

You may never need to create a new `ProcessingData` class, but you will have to know how to use them in order to write a new `Processing` class.

36.2 Coding style and philosophy

If your class will be part of the CLAM library, you should write it keeping in mind these objectives, in suggested order of priority.

Safety.

Your class should work as expected, contain as few bugs as possible, and should detect as often as possible when it is not being used properly. More about this in section 43 and section 44 .

Clarity.

Your code should be easy to understand. Reading it should give a clear idea of the algorithms that are being used.

Efficiency.

Your class should work fast. This is, after all, why are we using C++ instead of a more sane language.

Once you have chosen an efficient algorithm, it is usually not a good idea to think too much about code efficiency while you are writing your class. It is usually hard to figure out what will be the real bottleneck in your code until you run a profile on it, and you can not do that until it is finished and working correctly.

Also, clearly written code is easier to modify later, and thus to optimize. If you obfuscate your code trying to avoid some imaginary efficiency problem, and you later find that efficiency problems are caused by a different reason, it will be harder to fix the problem. If your code is clear, optimizing it later will usually be an easy task.

37 Overview of the processing class implementation tasks

The mayor tasks you need to undertake when writing a new processing class are briefly described below. All of them will be further described in the next sections.

37.1 Declaring the processing interface attributes

Ports

: Your processing class will have a set of inputs and outputs. You should declare the related `Port` attributes in it.

Warning: Ports, although recommended, are still not mandatory so you may find classes in the repository that still make no use of Ports. These classes though will necessarily be re-factored soon when the Ports interface becomes mandatory

Controls

: If your class is to have input or output *Controls*, you also have to declare the related `Control` attributes in it (section 40). Note that you must declare as a control any attribute that is supposed to be modified while the Processing object is running.

37.2 Implementing the construction mechanism

This requires writing a helper configuration class and writing your processing class constructors, in which all the non-configuration related initialization can be performed.

37.3 Implementing the configuration mechanism

Processing objects reconfiguration is performed using the `Configure()` method provided in the `Processing` base class. You don’t have to write this method. But you do need to write a `ConcreteConfigure()` method that will be called from the `Configure()` method in the base class (more in section 38.3). This method must do all the initialization stuff dependent on

configuration parameters.

37.4 Implementing the execution methods

You have to provide a `Do()` method that reads the data in the ports and runs a processing ‘‘cycle’’ on it (section 39.2).

You may also write a `Do(...)` method with the data arguments specific to your processing class. This is usually the method where the actual processing algorithm is implemented.

37.5 Implementing other optional standard methods

Processing object execution state (section 39.1) is controlled using the `Start()` and `Stop()` methods implemented in the `Processing` base class. You can not overload these methods.

If objects of your class need to perform any special operations at start or stop time, you can overload the `ConcreteStart()` and `ConcreteStop()` methods, which will be called from the base class `Start()` and `Stop()` methods when the user calls them. See section 40.3 to see what kind of operations need to be done in your ConcreteStart and ConcreteStop methods.

37.6 Writing the tests

You should always write a ‘‘class test’’ program for your processing class, the why and how are explained in section 44.

38 Object construction and configuration interface

The most important item related to object construction and configuration are the processing *configuration* classes, which will be described in section 38.1. Section 38.2 will describe how to write the Processing constructors, and section 38.3 will describe how Processing objects can be reconfigured, and what needs to be implemented in the concrete classes to support this mechanism.

38.1 Processing configuration classes

As figure f 5 suggested, each processing class requires a related configuration class.

38.1.1 The role of processing configuration classes

The role of the configuration class is to store all the necessary information to configure an object of the related Processing class. In fact, configuration classes may be described as a place-holder for parameters which would otherwise be specified as individual arguments in the processing class constructors.

Processing classes must provide both a constructor and a `ConcreteConfigure` method taking an object of this configuration class as argument (more about this latter, in section 38.2 and section 38.3.

Configuration classes are implemented as dynamic types that store collections of configuration attributes (implemented as dynamic attributes).

When writing a configuration class, you will usually include configuration attributes such as:

- Number of inputs or outputs.
- Values for parameters of the processing algorithm that the user may want to specify, and which will be fixed during the execution.
- Initial values for input control parameters (whenever they are needed, see section 40).

It is important to keep in mind that the configuration mechanism can not be used to change parameters of a processing object during processing execution, only during an initial configuration stage (see section 39.1 for more details on this). Your processing class should provide *input controls* (section 40) for run-time parameter changes (a configuration attribute can be used to set an initial value for those controls, though).

38.1.2 Configuration class implementation

Processing configuration classes must derive from the base `ProcessingConfig` class, defined in the file

```
src/Processing/Processing.hxx
```

The name of the configuration class should be the same as the name of the processing object, adding the `Config` suffix.

In some cases several processing classes may share the same configuration class. The FFT is a clear example of this. For example, figure f 5 shows that several FFT classes exist for different algorithm implementations, but they all derive from a common FFT_base class, and they share a common configuration class: FFTConfig.

In other words: you will have to write a Configuration class for your processing class, unless the latter is a different implementation of an existent processing class, and may share its existent configuration class.

You will typically place the declaration of both classes (configuration and processing) in the same header file. This file should have the same name as the processing class.

Once you have written your configuration class, you can write the declaration of your processing class constructors and configuration methods.

38.2 Processing constructors

Processing classes must provide two constructors: a default constructor and a constructor with the configuration class as its argument type. In most classes they will look just like this (explicit member constructor calls are omitted):

```
SpectrumProduct::SpectrumProduct(){   Configure(SpecProductConfig());}SpectrumProduct::SpectrumProduct(const SpecProductConfig &c){   Configure(c);}
```

The `Configure` method called inside the constructors is implemented in the base class, and is described in section 38.3.

You will usually need to include some member constructor calls from your constructors. If you are using controls, for example, you should call the constructors of the control attributes, as described in section 40

You may need to perform more tasks in the constructor, but most of them will probably be better implemented in your `ConcreteConfigure` method (see below), which will be automatically called from `Configure`.

You should not try to provide a copy constructor for a processing object. It will not work well[3].

38.3 Configuration methods

Once a processing object is instantiated, it can be (re)configured using the `Configure` method, implemented in the Processing base class[4]. This method will check that the object can be actually reconfigured (i.e., that the object is not ‘‘running’’) and will call the concrete configuration method (which you have to write), `ConcreteConfigure()`, in the concrete processing class.

These two methods (The Configure method and the concrete configuration method) are declared in the base class as follows:

```
protected:   virtual bool ConcreteConfigure(const ProcessingConfig&) = 0;public:   bool Configure(const ProcessingConfig&) throw(ErrProcessingObj);
```

So you only have to implement a protected or private `ConcreteConfigure` method in your new processing class, and you may forget about the `Configure` method (although it is useful to call it from the constructors, as described above).

Additionally, you need to provide a configuration accessor:

```
const ProcessingConfig &GetConfig()const;
```

This method must return a reference to a configuration object holding the current object configuration. The easiest way to do this is to store a copy of the object passed to the `ConcreteConfigure` method, so `GetConfig` only has to return it.

You should keep in mind while writing other processing class parts that the configuration attribute (if you keep it) must store only *initial state* of the object. The object should not change configuration parameters itself. This means that the object returned by the `GetConfig` method must show the latest values passed either to the constructor or to the `Configure` method.

39 Object execution interface

39.1 Execution states

Processing objects are in a certain *execution state* at any moment. This is best shown in a state diagram, figure f 6. The object is initially in the `Unconfigured` state.

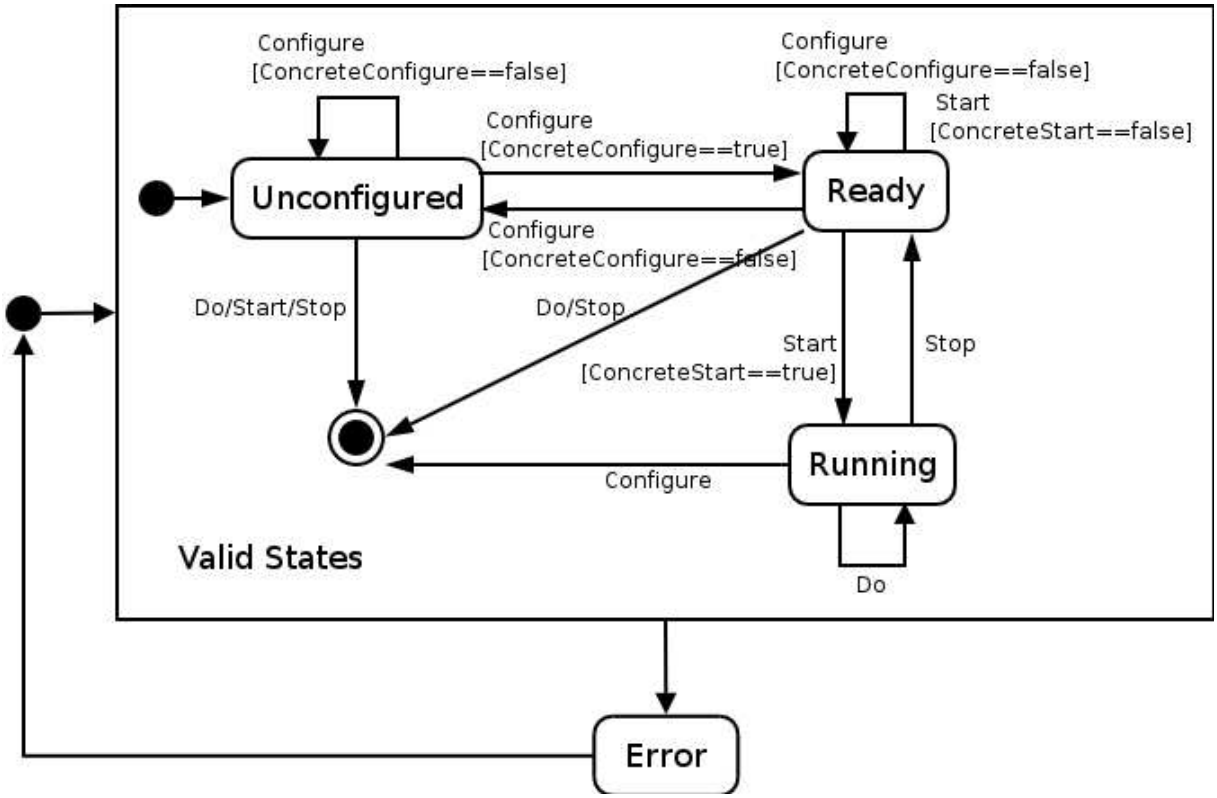


Figure 6: Processing Objects execution states.

All the methods shown in the state diagram, except the `Do() method` are implemented in the `Processing` base class. These are briefly described below. Note that all the state transitions are done via these methods; the state variable is also kept in the base class, so you do not need to worry about execution state management when implementing a new processing class.

- `bool Configure(ProcessingCofig&)` This puts the object in ready mode, if configuration is correct. See section 38.3 for more details.
- `bool Start(void)` This puts the object in execution mode. This method must be called before the first call to any `Do()` method. Calling `Do()` before calling `Start()` may throw an exception.
  Once the object is running (i.e. in execution mode), it can not be reconfigured. Calling `Configure()` in execution mode will throw an exception.
- `bool Stop(void)` This puts the object out of execution mode.
- `ExecState GetExecState()`. This method returns the current execution state of the object. This may be used for debugging or to keep tack of the object state in the application.

You may need to perform some specific operations in your class at certain state transitions. There are several virtual methods that you can override to do so, and which are described in the following table:

concrete method	**called from**	**mandatory**
`ConcreteConfigure(ProcessingConfig&)`	`Configure(ProcessingConfig&)`	yes
`ConcreteStart()`	`Start()`	no
`ConcreteStop()`	`Stop()`	no

Note that the last two methods are very tied to supervised mode operation.

39.2 Execution methods

The main execution methods are the `Do` methods. They are the ones which actually perform the processing action.

There are two different kinds of `Do` methods:

- A `Do(void)` method, with no arguments. This operation is declared in the Processing base class and is therefore called "generic" Do. In future releases of CLAM this will become the standard way of using processing objects. This method is related to the input and output ports that have been previously declared. As already commented although you may still find some Processing classes in the repository that do not declare Ports and therefore provide an empty implementation of this method, you are encouraged to provide one so your Processing class can be used from within a Network.
- `Do(...)` methods taking data objects as arguments and called "concrete" Do's. They will have some input data arguments first, and then some output data arguments. A typical processing class will need a single `Do` method of this kind. These methods are discussed in section 39.3

Both kinds of `Do()` operations work in the same way: they read a certain number of data objects from each of the inputs, and write a certain number of data objects to each of the outputs. The difference is that the concrete `Do()` method takes this data objects as arguments (and therefore does not use ports), while the generic `Do()` operation has no arguments, and accesses the Data through the Ports objects.

figure f 7 shows a sequence diagram for the execution of a processing object with a single input and a single output.

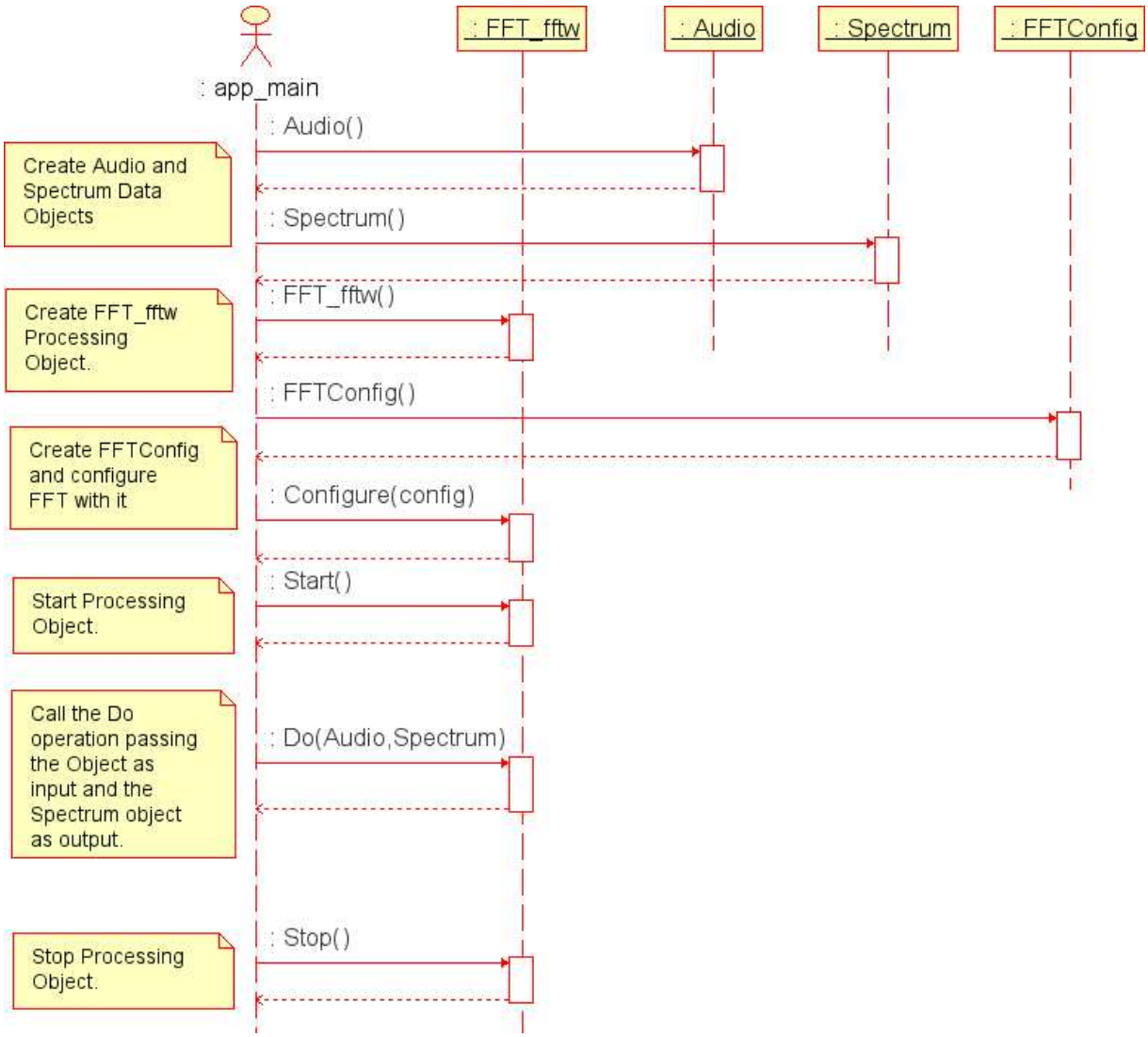


Figure 7: Execution sequence

39.3 Object execution not using ports

Processing objects must provide a `Do` method which takes the input and the output objects as its arguments. You may provide several `Do` methods for different object configurations (such as different sets of active inputs and outputs).

You should carefully check that the dynamic attributes you need are instantiated in the input and output objects passed to a `Do`.

A good approach to this is having some kind of *prototype state* variable which is updated when `SetPrototypes` is called. Your `Do` methods can then use a `switch` statement to choose the processing code to execute.

You can, for example, do a fast execution when the inputs and the outputs have the attributes most convenient for your computations, and a slow execution for other cases, in which you perform previous and subsequent data conversions.

In this slow cases, if the data classes you are handling provide attribute conversion routines, you can usually just add the attribute you need to the data object, and call a proper conversion routine to update it. See the FFT_rfftw code for an example of this.

39.3.1 `Do` method argument conventions

1. You should clearly document the `Do` methods arguments. You should use Java-Doc comments in your code for this.
2. Inputs go first and should be "const", outputs go last.
3. By default, it is assumed that a Processing can process inplace, thus it is possible to pass the same reference to input and output (of course, if the data type is the same). If the Processing you implement cannot process inplace you should override the `CanProcessInplace` method, return false and document this fact.
4. You should take both input and outputs as references whenever it is possible.
5. Inside a Processing, you should never assume ownership of objects passed as inputs. You should not delete them.
6. You should avoid passing as outputs data objects which you have created. If you do so, document it clearly, explaining why you do so, and how to handle them, specially their lifetime, or who should destroy them.
7. Variable number of arguments must be handled passing a reference to an array of pointers (typically as a reference to a pointer). You can implement an extra `Do` method which takes a reference to an array of objects instead, to avoid dereferences (you must anyway provide the pointers version, because the user may not always have an array of objects handy).
8. If the number of arguments is variable, you must clearly document this fact, and provide methods to get this number.

40 Controls

Some processing classes need to allow external entities to change the behavior of the objects asynchronously during their execution. Input controls are the mechanism to perform this kind of run time changes.

Also, a processing class may be used to detect some kind of event. Output controls are the way to make notifications on asynchronous events.

An application can connect output controls from some processing objects to the input control of others.

When we use the ‘‘asynchronous’’ word here, we mean that control values do not flow with a given rate, as data does.

In CLAM, control values are floating point numbers.

40.1 Input Controls

There are two different mechanism to implement input controls. Controls using the first mechanism simply store a value, and allow an externally connected output control to change this value. These controls are described in section 40.1.1.

For the second mechanism, you have to write a special method in your class, a *call-back method* which will be called whenever a new value is sent to the input control. This mechanism is described in section 40.1.2

40.1.1 Regular input controls

To use a regular input control, you need to

1. Declare an `InControl` attribute in your class with a descriptive name. For example, if you have a couple of input controls with pitch and amplitude values, you should declare them like:

```
InControl mInPitch;   InControl mInAmplitude;
```

2. Call the `InControl` constructors from your processing class constructors. They take two arguments: the control textual name, and a pointer to the processing object containing the control. For example, the constructor of a processing class called `MyClass` containing two input controls would look something like:

```
MyClass(const MyClassConfig &c)     : mInPitch("Pitch", this),       mInAmplitude("Amplitude", this)   {     Configure(c);   }
```

3. Give an initial value to the control. You should do this in the `ConcreteStart()` method of your class. Input controls provide a `DoControl(`*value*`)` method to change the value.

40.1.2 Input controls with call-back method

In some cases you may want to have a call-back method executed each time an input control changes its value. Some reasons for this might be:

- You may want to send some output controls in some situations when an input controls arrives.
- You may want to keep track of multiple changes in a control value between two consecutive calls to your `Do()` method.
- If you have many controls, you may want to avoid checking all of them for changes each time the `Do()` method is called.

In order to use this call-back mechanism, you have to:

1. Declare your control attributes to be of type `InControlTmpl<MyClass>` (where `MyClass` is the name of your processing class). Following the example in previous section, you would have:

```
InControlTmpl<MyProcObj> mInPitch;   InControlTmpl<MyProcObj> mInAmplitude;
```

2. Define the call-back method(s) in your class, which must take a single argument of type `TControlData`. For example

```
int InPitchControlCB(TControlData val);   int InAmplitudeControlCB(TControlData val);
```

3. Call the `InControlTmpl` constructors from your processing class constructors. They take three arguments: the control textual name, a pointer to the processing object containing the control, and the address of the call-back method. Following the example:

```
MyClass(const MyClassConfig &c): mInPitch("Pitch", this, &MyClass::InPitchControlCB),  mInAmplitude("Amplitude",                &MyClass::InAmplitudeControlCB)){  Configure(c);}
```

40.2 Output Controls

You add output controls to your class in the same way you add regular input controls, but taking into account that the name of the control class is now `OutControl`. So, you need to:

1. Declare the `OutControl` attributes in your class.
2. Call their constructor in the initializer lists of yours.

Now you can send values through the output control. You will usually do it from time to time in your `Do()` method, using the `SendControl(TControlData val)` method of the `OutControl` class.

40.3 Controls initialization

Input controls must be initialized in the `ConcreteStart()` method. If the initial value of a control should be chosen by the user, a configuration attribute can be provided in the configuration class for this task.

41 Internal object state

‘‘Object state’’ usually refers to the specific values that the attributes of this object have in a given moment of time.

For processing objects, it is useful to consider two different kinds of attributes, and thus two different kinds of ‘‘state’’:

- Configuration related attributes.
- Execution related attributes.

Initialization, usage and destruction of these attributes should be done in different ways, as described below.

41.1 Configuration related attributes

The first ones would be attributes which may only change when the processing object configuration is changed (i.e., when the `Configure()` method is called).

For example, if your configuration class includes an attribute which defines the size of some internal buffers in your processing objects, you should create or resize these internal buffers when the `ConcreteConfigure()` method is called, instead of doing so directly in the class constructors.

41.2 Execution related attributes

Some processing classes may need to keep internal computed values between different calls to Do methods. This is usually done using normal private class members.

Some examples of this are:

- Keeping track of time,
- Performing some kind of integration/accumulation of the input.
- Performing some kind of differential operation for which values from previous execution calls are needed.

Sometimes it may be a good idea to provide a public accessor (getter) method to some of these internal values, so that applications using the class can easily implement some run time debugging. But it is usually a better idea to provide this access in the form of input or output Controls.

41.2.1 Initialization

The initialization of internal state attributes related to object execution (such as execution counts, accumulated values, time references, etc) must be performed in the `ConcreteStart()` method.

Also, if you want to liberate some resources when the object stops being run, you can implement this in the `ConcreteStop()` method.

42 Processing Composite

Sometimes you need to write a large processing object which uses other processing objects internally to perform some parts of the algorithm.

There is a standard way to do this:

1. You should derive your class from `ProcessingComposite` instead of deriving it directly from `Processing`.
2. You should configure your children processing objects, calling their Configure method in the ConcreteConfigure method.

See the file `examples/POCompositeExample.cxx` for more details.

43 Exception Handling

Your classes may some times throw exceptions. This will usually happen in two circumstances:

1. When some consistency checks inside your code fails (section 77.3).
2. When some external run time problem happen (a file which does not exist, a device which refuses to be opened, etc). This is discussed in section 79.2.

43.1 Assertions

As you probably remember from section 36.2, one of the main self imposed requirements in the CLAM library is code safety.

One of the best tools to achieve this is using ‘‘assertions’’ (condition checks) in your code to check that things are like they are supposed to be.

Assertions are a general mechanism inside the CLAM library, and are thus also discussed elsewhere. But it is worth making some comments about how they should be used in processing classes.

43.1.1 Where to use assertions

There are mainly two kinds of assertions:

1. Internal consistency checks about the state of your class. These are usually conditions which should NEVER fail, and which would mean that something is really wrong in your class. They are useful for you as a developer, to find bugs in your own code.
2. External precondition checks. Class methods usually expect some restrictions in they arguments, or in the order in which they will be called. These restrictions should be documented, but someone using your class may make a mistake and misuse your class. ‘‘External’’ assertions make checks on the values of arguments provided by the user, or check if the internal state of the class is adequate to perform an operation requested by the user.

A clear example of this kind of check is testing the value of an index provided by the user to see if fits the size of a data array.

43.1.2 How to make assertions

Easy. Just use the `CLAM_ASSERT` macro. This will usually be fine for you. If not, the assertions mechanism is fully described in a different section of this document.

In some circumstances, you may want to make a check while debugging, and remove it later for efficiency reasons. An example of this is checking the value of an index in the `Array<T>` class indexing operators. For this kind of check, you can use the `CLAM_DEBUG_ASSERT` macro. The checks made using this macro will be disabled when compiling the library in ‘‘release’’ mode.

43.2 Run time problems

These may rise when you are performing some operation which may fail, such as trying to open a file, trying to allocate some memory, trying to start an input/output system device, etc.

These are not assertions, in the sense that even if the program is absolutely correct, these conditions may fail.

The behavior when you find one of these problems depends on the context where you find it:

1. You should not throw an exception from your `ConcreteConfigure()` method. You should return a *false* value instead, and maybe add a textual explanation to the `mStatus` member. This will leave the object in an unconfigured state, and allow the user to fix the problem. If he does not, the Start() method will complain for you.
2. You should throw an exception elsewhere, and you should clearly explain the fact in the documentation of the methods in your class from where this exception may come out.

If you have this kind of situation in your code, you should use one of the standard exception classes defined in CLAM library. See the `src/Errors/` directory for the collection of available exception classes. If no one fits your needs, it may be a good time to write a new exception class, but you can always throw the base `CLAM::Err` class meanwhile.

44 Writing tests for your classes

44.1 Why?

Mainly for two reasons:

1. This way you will be able to find trivial run time errors easily and as soon as possible. Otherwise, if you find them when you are using your class in a bigger system with many other classes, it will sure be much more difficult to trace the problem.
2. This way, once the class is finished, it will be really easy to introduce further changes, and check that everything keeps working as expected.

A non-trivial bug may take just a minute to detect, understand and fix if you run a test program often while you are coding, because you will always be almost sure that the problem belongs to the latest changes you have made.

It will usually take you more than ten times longer (say, 10 minutes) if you first encounter this bug when you are using your class in a larger system, after making many modifications without testing hard.

If it is someone else who runs into this bug while using your class in a bigger program, it can take him a hundred more times (say, a couple of hours) to trace the problem and fix it (or ask you for a fix).

Some argue that you should write the class test even before you start programming your class, so that you already have a program against which you can try it all along the process of coding.

44.2 How?

You should definitely have a look around the `tests/` directory in CLAM sources to take a feeling of how the test thing goes. There you can find a test file ‘‘skeleton’’ (`TestSkeleton.cxx`) to take as starting point, but you usually have complete freedom in the way to implement a test. Anyway, it should follow a standard convention to communicate its results:

- Your test program should know when things are working fine, and when they are not.
- The `main()` function in your program should return 0 when everything has worked as expected.
- It should return a non-zero value when some error is found, such as an exception being thrown somewhere, or output values from your class containing incorrect data.
- Of course, it is usually nice to have some output written when something fails, explaining the details of the failure.

45 Helper classes

45.1 Enumeration classes

In the situations where you would normally use a standard C++ `enum` type, you should consider using a CLAM `Enum` class instead.

CLAM Enums are much like C++ enums, with the advantage that they have storage capabilities built in. In run-time, C++ enums only provide the integral value, CLAM::Enum’s also use the symbolic value (the string). See the `src/Standard/Enum.hxx` class Doxygen documentation for more information.

45.2 Flags classes

`CLAM::Flags<N>` also provides symbolic usage and storage capabilities to the `std::bitset<N>` class. You should use the `Flags` CLAM class, instead of a `std::bitset`, or instead of an integer plus bit-mask mechanism. That way you gain storage capabilities.

See the `src/Standard/Flags.hxx` Doxygen documentation for more information.

46 Prototypes

NOTE1: In previous CLAM releases the Prototype interface was included in the Processing base class and was indeed a recommended way of working. Experience has demonstrated that this feature is only necessay in very specific cases and has therefore been removed from the base class interface. Nevertheless, some particular classes such as the FFT still keep this functionality, which can indeed improve efficiency. Only read the following paragraphs if you are really worried about efficiency issues and have the suspicion that the use of prototypes at the input/output of your processing may help

NOTE2: This section discusses some concepts related to dynamic types. If you are not familiar with them, you should read some relevant documentation on the issue.

The processing data objects used as inputs and outputs for a processing object can sometimes have multiple alternative dynamic attributes, which are often different representations of the same data. In other words, they can have different *dynamic prototypes*. An example of such multiplicity is the `Spectrum` class.

Trying to read a dynamic attribute that is not instantiated is a run-time error, so it should never happen. One should use the `Has...()` dynamic method to check if a certain attribute is instantiated before using it.

Some structural parameters of the data objects must also be checked before using them. Buffers size, for example, is often stored as a dynamic attribute in the data object. One can not assume these values, they must be checked *for each data object* before relying on them.

If the processing data classes you are handling do not have such kind of prototype alternatives or structural parameters, you don't need to provide the `SetPrototypes()` method.

Otherwise, processing classes must be ready to handle this attribute diversity. Instances of a well written processing class should be able to cope with every valid prototype at its inputs, probably with different performance in different cases.

In normal cases all of the above requires quite a few checks, which may degrade the performance of the execution method.

The `SetPrototypes` functions can be used to avoid this performance problem. When the application (or the flow control) calls a `SetPrototypes` method, the object can assume that subsequent calls to the execution method will pass data objects with the same prototypes and structural parameters as the ones specified with `SetPrototypes`, until a new call to `SetPrototypes` is made, or until the `UnSetPrototypes` method is called.

These methods are just informative; the object is not required to perform any specific action. It can safely ignore these calls. If you want to take advantage of them, you will typically do an exhaustive prototype checking, and set some internal state attribute in your processing object according to what you find in the data. This state variable can be checked with a single `switch` statement in your `Do` method.

46.0.1 Footnotes

3 This is due to some missing functionality in the processing object composite system

4 You must not override the base class `Configure` method.

VIII Processing Data classes

47 Scope

This section is written having in mind average CLAM users. In the "Developer" part of this document you will also find a chapter on Processing Data that is mainly addressed to developers who want to implement a new Processing Data class.

48 Introduction

In CLAM terminology, a Processing Data (PD for short) class is a class designed for storing all sort of data that will be used in the processing process. All Processing Objects have Processing Data objects as inputs/outputs (therefore all arguments of the concrete `Do()` method must be Processing Data). Examples of Processing Data classes include Spectrum, Audio, SpectralPeakArray, Fundamental, Segment, Frame.

49 Basic structural aspects

Any PD class is in fact a concrete Dynamic Type Class (it derives from abstract ProcessingData class, which itself derives from DynamicType). Therefore most of the PD attributes are macro-derived dynamic attributes. For example, in the declaration of the class you will see something like `DYN_ATTRIBUTE(1,public, Spectrum, ResidualSpectrum)`, which means that the given class has a public dynamic attribute called ResidualSpectrum that is an object of the Spectrum class.

All dynamic attributes have associated automatically derived Setters and Getters that may be used from outside the class. Furthermore, attributes can be Added and Removed at run-time (please refer to Dynamic Type's chapter if you need further explanation on these issues).

Some classes have private dynamic attributes that cannot be accessed directly but offer an alternative public interface. If you encounter a private or protected attribute with a name starting with the 'pr' prefix (i.e. prSize) you should look for its associated public interface (i.e. GetSize() and SetSize()).

Very rarely, some PD class have an attribute that is not dynamic. In that case, you should be granted the corresponding Set/Get interface so its usage will not be different.

50 Efficiency Issues

Most PD classes offer convenient shortcuts for accessing and setting elements in their buffers that should be very useful in a developing stage but should be avoided if seeking efficiency in a given algorithm.

Ex:

The Spectrum class has a MagBuffer and a PhaseBuffer for storing spectrums in a magnitude-phase manner. It also offers shortcut methods for accessing and setting both phase and magnitude of a given bin. Let's take a look at the code of the `GetMag()` method.

The main advantage of this shortcut is its flexibility and the fact that the memory layout of the data is transparent to the user. But it is not a very 'efficient' method!

Let’s now suppose that we have an algorithm that computes the average of the magnitude in the spectrum. We could think of doing something like:

```
TData ComputeAverage(const Spectrum& spec){   int i;   TData sum=0;   for (i=0;i<spec.GetSize();i++)      sum+=spec.GetMag(i);   return sum/spec.GetSize();}
```

This method could be made much more efficient if the usage of the shortcut was avoided. In that case, the code would become something like:

```
TData ComputeAverage(const Spectrum& spec){   int i;   TData sum=0;   DataArray mag = spec.GetMagBuffer();   for (i=0;i<spec.GetSize();i++)      sum+=mag[i];   return sum/spec.GetSize();}
```

Another consideration about the algorithm that needs clarification is the fact that the GetSize() method is being called in every step of the loop! That is not a very efficient way to go either. The finally optimized ComputeAverage() method should look somewhat like:

```
TData ComputeAverage(const Spectrum& spec){   int i;   TData sum=0;   DataArray mag = spec.GetMagBuffer();   int size= spec.GetSize();   for (i=0;i<size;i++)      sum+=mag[i];   return sum/size;}
```

Finally, let’s think of a method that instead of computing the average out of a given spectrum, sets the magnitude to twice the original one. If we tried to follow the previous example we would come up with the following code:

```
void DoubleMagnitude(const Spectrum& spec){   int i;   TData sum=0;   DataArray mag = spec.GetMagBuffer();   int size= spec.GetSize();   for (i=0;i<size;i++)      mag[i]*=2;}
```

The previous code would not work. The reason is that when we are calling the GetMagBuffer() method and assigning the result to our mag variable, we are actually doing a copy of the Array. All modifications that are done afterwards to the mag array are only done on the local copy. Then, what is the solution? The following code gives you the right way to deal with this case (assigning the result to a DataArray reference):

```
void DoubleMagnitude(const Spectrum& spec){   int i;   TData sum=0;   DataArray& mag = spec.GetMagBuffer();   int size= spec.GetSize();   for (i=0;i<size;i++)      mag[i]*=2;}
```

Note: Due to the nature of Dynamic Types, it is very error-prone to Add/Remove dynamic attributes from a Dynamic object after a reference has been assigned. The following are examples of code likely to fail:

```
Spectrum& spec=myFrame.GetSpectrum();myFrame.GetSpectrum().AddComplexArray():myFrame.GetSpectum().UpdateData();//Do something with spec: error, reference may be lost!!

Spectrum& spec=myFrame.GetSpectrum();spec.AddAll();spec.UpdateData();//Error, reference may be lost!!

Spectum& spec=myFrame.GetSpectrum();spec.SetType(myFlags);/*Error, reference may be lost!! SetType operation may add/remove attributes to Dynamic Object*/
```

51 Introduction to CLAM‘s Core PD classes

In this section, a brief overview of the Processing Data included in CLAM’s core is given. If more details are needed, it is better to refer yourself to the code or the DOXYGEN generated documentation.

51.1 Audio

Attributes

In short, the Audio class has three basic attributes (SampleRate, BeginTime, and Size), one buffer (Buffer) and associated descriptors (Descriptors). All of them have associated Getters and Setters. The Size attribute is, in fact, a structural attribute. Thus, a change in its value implies a change in the existing buffer. On the other hand, the BeginTime and SampleRate attributes are purely informative and thus, a change in their value only implies a change in the attributes but not on the buffer.

Additional interface

The Audio class has some additional interface for working with time tags instead of indices or sizes. The Getters/Setters for EndTime and Duration do not belong to an associated attribute but are rather different ways of changing the size of the Audio.

There is also an additional interface for working with audio chunks and slices. An audio chunk is defined as another Audio object that has a copy of a subset of the data in a given Audio. On the other hand, an sudio slice is defined as an Audio object that has a reference to a subset of the samples in a larger Audio object. Therefore the difference is that while in asking for an audio chunk you are getting an actual copy of the data in the audio, an audio slice will have the same effect but without actually copying the data but rather referencing the original one. In an audio slice if the original audio is deleted the audio slice will be left with no valid audio data.

51.2 Spectrum

The Spectrum is possibly the most complex PD class in the CLAM repository. It is a fundamental class for the library's purposes and some 'extra' care and effort have been put into it.

A Spectrum can be represented in one of the following formats: array of complex numbers, array of polar numbers, a pair of magnitude/phase arrays, and a pair of magnitude/phase BPF's (Break Point Function). The Spectrum class is designed in such a way so as to be able to keep consistency of the data in its different representations. This is accomplished through the `SetTypeSynchronize` and the `SynchronizeTo` methods and some conversion routines (which are private and cannot be accessed directly). Note though, that the `SetType` method does not perform this sort of data consistency check and only instantiates the necessary attributes with the existing Size.

There is an accessory interface for accessing/setting Magnitude and Phase regardless its internal representation. These methods are not efficient but help in keeping consistency between different representations.

The spectrum also has two different sizes: Size and BPFSize. We won't go into many details about the reasons for being so but if you feel you need to understand this difference maybe it is time you just take a look at the code and Doxygen documentation and look at the BPF class.

The Spectrum class is right now the only PD class that has an associated configuration. This configuration is used for initialization purposes and a local copy is not kept in the object. Whenever the `GetConfig(SpectrumConfig& c)` method is called, the argument passed and used as output of the method is synchronized with the internal structure of the spectrum.

If the previous explanation did not seem enough or you are still left with many doubts about this class we strongly recommend that you take a look at the SpectrumExample in the CLAM repository.

51.3 SpectralPeak and SpectralPeakArray

A SpectralPeak is a simple storage PD class that has the following dynamic attributes: Scale, Freq, Mag, Phase, BinPos, and BinWidth. By default, only frequency, magnitude and scale are instantiated, all others, if needed, must be added by hand. It has also a couple of operators like product, distance and log/linear scale converting routines. By itself it is seldom used and should be preferably used through the SpectralPeakArray class.

The SpectralPeakArray is not, as its name may imply, just a simple array of SpectralPeaks. As a matter of fact, the SpectralPeakArray class does not hold SpectralPeak's inside but rather a set of buffers containing magnitude, frequency, phase, bin position, bin width and index. By default only the MagBuffer and FreqBuffer are instantiated, all others must be added by hand. Apart from these, it includes other non-array attributes such as Scale, nPeaks (number of peaks currently available) and nMaxPeaks (maximum number of peaks allowed). By default all these dynamic attributes are instantiated.

Even though the most efficient way to deal with a PeakArray is to work directly on the buffers (see section 50), two accessory interfaces are offered: first, you can access/modify any of the attributes of a given peak by using the interface offered by methods like `GetMag()`or `SetPhase()`; but also, you can use an interface using SpectralPeak objects through the `GetSpectralPeak()` and `SetSpectralPeak()` methods. Note that these methods do not return a pre-existing peak but

rather construct the peak object on the fly. Therefore, they are far from efficient.

Another particularity that needs mention is the IndexArray. It is a multi-purpose array of indices. Currently it is used for fundamental detection, peak continuation and track assigning. It is sometimes indeed a very convenient way of dealing with many insertions/deletions of peaks into the array as they can be substituted by a simple change in index (having the negative values mean that the particular peak is not valid, for instance). An accessory interface (consisting of several methods) is also offered for working through indices.

51.4 Fundamental

The Fundamental class is a basic storage PD class used for storing the result of a fundamental (pitch) detection algorithm: a set of candidate frequencies and the computed estimation error if present. It has two integer dynamic attributes that hold the current number of candidates and the maximum allowed and two arrays: one of frequencies and the other one containing the errors. All the dynamic attributes are instantiated by default.

51.5 Frame

A Frame class has two time related attributes that are instantiated by default: CenterTime and Duration. Apart from that, it has 'a bunch' of other attributes that belong to one of the PD classes explained in the previous sections. Namely, we have: two Spectrum (one for the general spectrum and the other for the residual component), a SpectralPeakArray, a Fundamental and an Audio attribute that is usually used for storing the windowed audio chunk that has been used for generating the other data.

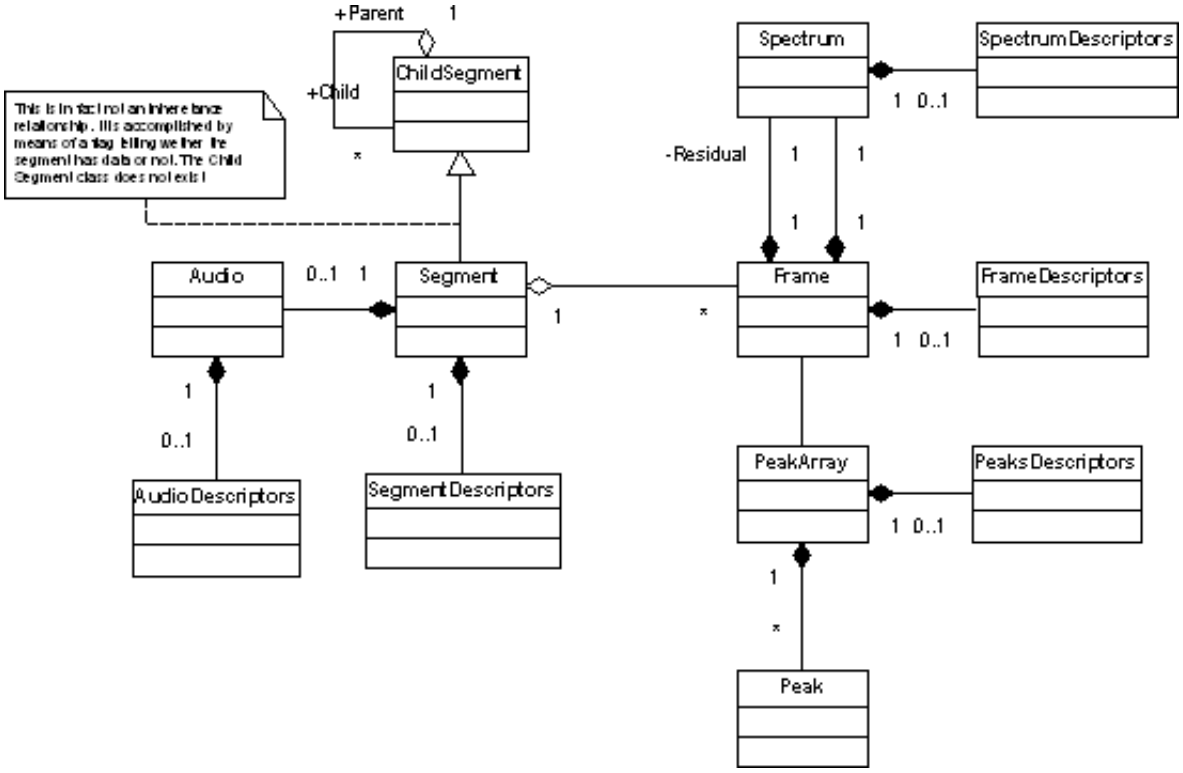
All other methods are just shortcuts for the getters and setters of the previous attributes and may come in handy for some applications that do not bear efficiency requirements.

51.6 Segment

A Segment consists basically of an audio frame (Audio dynamic attribute) and an aggregate of Frames. This aggregate is implemented as an `CLAM::List` so as to favor fast insertions and deletions and supposing that access is usually going to be sequential. This list of frames can be searched upon, using its begin time as the sorting criteria. Apart from this, the Segment follows a composite pattern so a segment can in turn hold an aggregate of other Segments (which are known as children and thus stored in the Children dynamic attribute). In the composite structure, only the root segment may hold data (frames and audio) but this data may be accessed from a child located at any level. For doing so, all children have a pointer to their parent. (This member is not a dynamic attribute and will not therefore be stored if doing an XML dump). In order to know if the Segment holds data or not, a structural attribute is included: `prHoldsData`, which may be accessed through the `GetHoldsData/SetHoldsData` interface. The `SetHoldsData` method is not just an accessor, if set to true, the child will actually detach itself from its parent and copy the data that corresponds to its time interval. If set to false the child will remove the data attributes (frames and audio). Note that you should do a SetpParent afterwards in order to keep that segment consistent.

A Segment also has a couple of informative attributes: BeginTime and End Time and a set of associated descriptors (SegmentDescriptors).

The following UML class diagram illustrates the inner structure of the Segment class and its associates:



51.7 Descriptors

Descriptors are a special kind of ProcessingData that are always bound to another ProcessingData class. They describe numerical attributes that are usually computed from the data in the PD object using 'basic' statistical computations. At the time being the Descriptor functionality is being completely refactored. If you feel that you cannot wait you can take a look at the DescriptorComputationExample in the repository or at the development documents at the CLAM webpage to get a grasp of what will be very soon offered.

52 Basic XML support

As all PD classes are concrete Dynamic Type classes have automatically built-in XML support. At this moment XML input/output is fully supported. Please refer to chapter IX for more details.

IX XML Support

53 Scope

This section is addresed to anyone who wants to get an already implemented CLAM object and *passivate* it as XML or to *activate* such a CLAM object from a previous passivated XML document.

If you want to provide XML support to your own classes or you want to customize the default XML output that any DynamicType has, you may also refer to the other XML related sections on the developer's part of this document.

54 Brief introduction to XML

XML is a text based format to represent hierarchical data. XML uses named tags enclosed between angle brackets to mark the begin and the end of the hierarchical organizators, the XML elements. Elements contains other elements, attributes and plain content. Let's see a sample XML document:

```
<?xml version='1.0' encoding='ISO-8859-15' ?>
<mainElement>
        <subelement1 attribute1="atribute Content">
                plain content here
                <subsubelement>plain content</subsubelement>
                plain content here
        </subelement1>
        <subelement2 attribute2="atribute Content">
                <subsubelement>plain content</subsubelement>
                <subsubelement>plain content</subsubelement>
                <subsubelement>plain content</subsubelement>
        </subelement2>
        <emptyelement attribute="foo" />
</mainElement>
```

Both attributes and plain content are simple text data. The main different between them is that an attribute is named and plain content is not. Elements also have a name. Names for attributes must be unique inside its hierarchic context, though this restriction doesn't apply to elements' name.

The power of XML is that you can adapt your own tags (elements) and tag attributes (attributes) in order to describe your own data.

For more information on XML, visit W3 Consortium XML page.
There is also a MTG report about XML related standards and API's.

55 Storing components

A very fast example on how to store a Component (that means DynamicTypes, ProcessingData, ProcessingConfig... and any other instance of any Component subclass) would be:

```
MyComponent comp;

// Here you modify your component
// ...

XMLStorage::Dump(comp, "MyComponent", std::cout);
```

The first parameter of the Dump method is the object to be stored, the second one the name of the XML document root, and the last one is the std::ostream that you want the XML be written to. You can use `std::cout` or any other `std::ostream` like `std::ofstream`,

`std::stringstream` or any other you specialize.

```
std::ofstream out("mycomponent.xml");
XMLStorage::Dump(comp, "MyComponent", out);
```

For short, instead of an open stream you may specify the file name and will also work:

```
XMLStorage::Dump(comp, "MyComponent", "mycomponent.xml");
```

(TODO: Update this piece documentation to the last changes) By default, XMLStorages uses no indentation. That means that all any output is done in one single line without tabulators. If you want to pretty format it you should use the `UseIndentation` method like this:

```
storage.UseIndentation(true);
```

56 Loading components

The inverse way is nearly the same.

```
// An unmodified default constructed object!!!
MyComponent comp;

XMLStorage::Restore(comp, "mycomponent.xml");
// or alternatively
std::ofstream in("mycomponent.xml");
XMLStorage::Restore(comp, in);
```

Attention: Loading methods suppose that the objects are default constructed and unmodified. If not, weird things can happen.

57 Detailed step interface

The `XMLStorage` static methods used above provide shortcuts for the widely used funcionalities. But you may want to do something special like:

- storing optimally the same object it onto two different streams,
- updating an existing xml by adding some objects to it,
- extracting an object from a part of a document,
- writing a document fragment
- ...

Static methods are not enough, but you still can instantiate an XMLStorage object and use the non-static methods with it. Non-static methods implements smaller steps than static methods do. and you can combine them in order to obtain some concrete behaviour.

The non-static methods that XMLStorage provides are:

- **Create:** Creates an Empty DOM document.
- **Read:** Creates a DOM document from the XML that comps from an istream.
- **WriteDocument:** Writes on a stream the whole document
- **Select:** Changes the selected node (by default the root is selected)
- **WriteSelection:** Writes on a stream the selected target
- **DumpObject:** Dumps the CLAM object on the selected DOM node
- **RestoreObject:** Restores the CLAM object from the selected DOM node

For example, if you want to update an xml document by adding an object on XPath /Doc/element/subElement, you can use the sequence Read-Select-DumpObject-WriteDocument.

Check the Doxygen documentation for more information on the usage of those methods.

X Audio File I/O in CLAM

58 What is able to do?

Currently, CLAM is able to decode the following audio file formats:

- Ogg/Vorbis
- Mpeg Audio Layer I, II, III
- Microsoft's WAVE/RIFF
- SGI/Apple's AIFF
- Sun's AU, SND
- Paris Audio File (PAF)
- Creative's VOC
- and several more

obviously we haven't implemented ourselves the routines for achieving this from scratch. We use several free libraries like:

- libsndfile (http://www.mega-nerd.com/libsndfile) an excellent library by Erik de Castro.
- Ogg/Vorbis SDK by http://www.xiph.org
- Underbit's libmad http://www.underbit.com/products/mad

With exception of Mpeg Audio, due to IP fuss around Fraunhöfer, all decodable file formats can be encoded with the help of CLAM.

Besides being able to read and write audio data, there is also the possibility of extracting meta-data embedded in Vorbis or Mpeg bitstreams. In the case of Mpeg bitstreams, this implies to deal with ID3, the format for tagging Mpeg audio streams. We handle ID3 through id3lib http://id3lib.sourceforge.net

59 Usage examples

There are available several usage examples of new CLAM Audio file I/O tools:

59.1 Audio File I/O: File information extraction example	
Makefile / Visual C++ project location:	build/Examples/Simple/FileInfo
Sources location:	examples/FileInfo_example.cxx
Complexity:	Medium
Keywords:	Audio file I/O tools, Textual meta-data extraction
Pre-requisites:	Familiarity with CLAM::Processing usage
Description:	This example shows how to extract useful information about audio files using the CLAM::AudioFile class.

59.2 AudioFile I/O: Audio file reading example	
Makefile / Visual C++ project location:	build/Examples/Simple/AudioFileReading
Sources location:	examples/AudioFileReading_example.cxx
Complexity:	Medium
Keywords:	Audio file I/O tools, Read operations
Pre-requisites:	Familiarity with CLAM::Processing
Description:	This example shows how to access an arbitrary audio file and perform some simple analysis on the data it contains.

59.3 Audio file I/O: Audio file writing example	
Makefile / Visual C++ project location:	build/Examples/Simple/AudioFileWriting
Sources location:	examples/AudioFileWriting_example.cxx
Complexity:	Medium
Keywords:	Audio file I/O tools, Write operations
Pre-requisites:	Familiarity with CLAM::Processing
Description:	This example shows how to create an stereo file using CLAM file I/O tools.

59.4 Playing an arbitrary audio file	
Makefile / Visual C++ project location:	build/Examples/Simple/FilePlayback
Sources location:	examples/FilePlayback_example.cxx
Complexity:	Low
Keywords:	Audio device I/O
Pre-requisites:	Minimum familiarity with CLAM objects such as Processing and ProcessingData
Description:	This examples shows how to play an arbitrary sound file with your soundcard

XI Audio I/O

60 The AudioManager

The core of CLAM audio input/output is the *AudioManager* class. The *AudioManager* takes care of all administrative tasks concerning the creation and initialization of audio input and output streams, using the internal, system dependent *AudioDevice* class.

The first thing you need to do in order to use audio is create an *AudioManager* object. While this object is present all subsequent audio I/O objects created will use it. You should specify samplerate and latency. The latency is used to control the internal buffersize, and depends on your hardware. Typically, you will be safe with a value of 1024, but especially with a low-latency patched linux you could safely use 256.

```
AudioManager audioManager(44100,1024);
```

61 The AudioIn and AudioOut classes

The actual audio I/O classes, called *AudioIn* and *AudioOut*, can than be uses to create processing endpoints to retrieve audio from, or write audio to. Note that each of these objects is mono (the argument to the `Do(..)` function is a single Audio object). The *AudioIn* and *AudioOut* objects have to be created with an *AudioIOConfig* object that can be used to specify the device, the channel and the sample rate to use.

61.1 Specifying the device

The device is referred to with a string that has the following form:

```
"ARCHITECTURE:DEVICE"
```

Currently, implemented architectures are `alsa`, `rtaudio`, `portaudio` and `directx`. The available devices depend on your hardware and system configuration, and which objects you link your application with (in `src/Tools/AudioIO/...`. You can use the class AudioDeviceList, to obtain a list of available devices for the platform you use:

```
audioManager.FindList(arch)->AvailableDevices()
```

This returns a `const std::vector<std::string>&`

However, if you don’t specify the device, or use the string `"default:default"`, the *AudioManager* will choose the device that seems most adequate for your architecture. Similar, it is possible to obtain a list for the default architecture, passing `"default"` to the `AudioDeviceList::FindList` method.

61.2 Specifying the channel

In order to have a flexible multi channel system, you can specify the channel you want to use for each *AudioIn* and *AudioOut*. The *AudioManager* will use this information to initialize the internal audio handling. Typically, you may want to use `0` for left and `1` for right.

Example:

```
AudioManager audioManager;

inCfgL.SetChannelID(0);
inCfgR.SetChannelID(1);

AudioIn inL(inCfgL);
AudioIn inR(inCfgR);
```

If you want mono input/output, you can simply leave this out, and just create a single default *AudioIn*/*AudioOut* object.

XII MIDI I/O

The MIDIIO approach has several similarities with the AudioIO, and it is recommended to read the AudioIO documentation first. Basic knowledge of the MIDI protocol is required.

The following documentation reflects the MIDIIO implementation since CLAM 0.6.1. The main difference with earlier releases is the addition of MIDI output, a simplification of the *MIDIInControl* (no more ChannelMasks and MessageMasks - you'll have to create a *MIDIInControl* for each channel (or one for all) / message type)

62 The MIDIManager

The core of CLAM midi input is the *MIDIManager* class. The*MIDIManager* takes care of all administrative tasks concerning the creation and initialization of midi input streams, using the internal, system dependent *MIDIDevice* class.

The first thing you need to do in order to use MIDI, is to create a *MIDIManager* object. This object will be a singleton, and all subsequent MIDI I/O objects created will use it.

```
MIDIManager midiManager;
```

63 MIDI I/O Processings and their configuration

63.1 The MIDIIn and MIDIInControl class

The actual MIDI input class, called *MIDIIn*, can be used to parse incoming MIDI data, and handle it in any way. But more useful, in the CLAM context, is the derived class, *MIDIInControl*. A *MIDIInControl* has one or more *OutControl*s, and can be used to convert the incoming MIDI data to *ControlData*. *MIDIIn* and *MIDIInControl* objects have to be configured with a *MIDIIOConfig* object.

63.2 The MIDIOut and MIDIOutControl class

TODO: Example MIDIOut

63.3 The MIDIIOConfig class

Both *MIDIIn*, and derived *MIDIInControl*, and *MIDIOut*, and derived *MIDIOutControl*, have te be configured with *MIDIIOConfig*. This *ProcessingConfig* contains the following fields:

Name	Set with	Meaning for MIDIIn	Meaning for MIDIOut	Observations
Device	SetDevice	Specifies the midi device to use.	Specifies the midi device to use.	See section 64.1.

Channel	SetChannel	Specify channel (1-16) to listen to for incoming MIDI messages. If set to 0, the *MIDIInControl* will listen to all channels, and create an *OutControl* (the first) to output for each message what channel it appeared on	Specify channel (1-16) for outgoing MIDI messages. If set to 0, the *MIDIOutControl* will create an *InControl* (the first), that can be used for specify the channel for each message.	
Message	SetMessage	Specify message type to listen to for incoming MIDI messages.	Specify message type for outgoing MIDI messages.	MIDI message types are specified in `src/Tools/MIDI/MIDIEnums.hxx`, see section 65
FirstData	SetFirstData	Specify first data byte to listen to for incoming MIDI messages of the specified message type. If left unconfigured, or set to 128 (default), the *MIDIInControl* will have an *OutControl* that outputs the first data value of each incoming message.	Specify first data byte for outgoing MIDI messages. If left unconfigured, or set to 128 (default), the *MIDIOutControl* will have an *InControl* the control it.	This is especially usefull for control change messages, where the first data byte specifies the type of control change. (For example, 1 is modulation, 11 is breath/expression).

On the input side, the *MIDIManager* and *MIDIDevice*s use this information to create a very efficient MIDI parsing table.

63.4 Dynamically created InControls and OutControls

The number of *InControls* / *OutControls* on a *MIDIOutControl* / *MIDIInControl* resp., depends on the configuration used. This is reflected in the table above. Resuming, in case of the *MIDIInControl*, you will only get *OutControl*s for those fields that you do not specify a specific filter value for. In the case of the *MIDIOutControl*, you will only get *InControls* for those fields that you do not specify a specific default value for. In both cases, this is done through the *MIDIIOConfig*

The name of each *InControl* / *OutControl* is set to `"MESSAGE:FIELD"` automatically, for example `"NoteOn:Key"`, `"NoteOn:Vel"`, etc. You can obtain this with the method

```
const std::string& OutControl::GetName()
const std::string& InControl::GetName()
```

64 The MIDIDevice class

64.1 Specifying the MIDI device

The device is referred to with a string that has the following form:

```
"ARCHITECTURE:DEVICE"
```

Currently, the implemented architectures are `alsa` and `portmidi`, and the "virtual" MIDI file devices `file` (input only) and `textfile` (output only). The available devices depend on your hardware and system configuration. You can use the class MIDIDeviceList, to obtain a list of available devices for the platform you use:

```
MIDIManager::FindList(arch)->AvailableDevices()
```

This returns a `const std::vector<std::string>&`

However, if you don't specify the device, or use the string `"default:default"`, the *MIDIManager* will choose the device that seems most adequate for your architecture. Similar, it is possible to obtain a list for the default architecture, passing `"default"` to the `MIDIDeviceList::FindList` method.

64.2 Clocking the MIDI device

MIDIEnums.hxx defines the following:
When the MIDI input comes from a device, typically live input, MIDI messages gets delivered through the controls as soon as they come in. In the case of the special "virtual" *FileMIDIDevice*, this situation is slightly different, and you will have to add a *MIDIClocker* to control the sequencing of the data in the MIDI file. Please look at the MIDI_Synthesizer_example for info.

65 MIDI Enums

MIDIEnums.hxx defines the following:

```
eNoteOff = 0,
eNoteOn = 1,
ePolyAftertouch = 2,
eControlChange = 3,
eProgramChange = 4,
eAftertouch = 5,
ePitchbend = 6,
eSystem = 7
```

XIII The Application Classes

CLAM provides several *Application Classes* that provide a basic framework for typical application situations, such as audio, or audio + graphical user interface. When necessary, threads or setup, and several virtual functions are provided, which can be implemented by deriving from the relevant application subclass (*AudioApplication* or *GUIAudioApplication*).

66 BaseAudioApplication

This class is the base of all (derived) *AudioApplication* classes. It sets up a (high priority) audio thread, and specifies several virtual functions:

- `void AudioMain(void)`, which will be executed inside the audio thread. This is where the derived classes implement the actual audio processing.
- `void UserMain(void)`, which will be executed by the main thread. This is where the derived classes implement the actual (graphical) user interface.
- `void AppCleanup(void)`, which will be executed when the applications ends. This is where the derived classes implement any extra resource cleanup.
- `bool Canceled(void)`, which can be used in the `AudioMain`, to check if the audio thread has been canceled.

This class should never be used directly. If you want a standard audio application, use *AudioApplication* instead.

67 GUIAudioApplication

This class is derived from *BaseAudioApplication*, and additionally provides a standard user-interface, with start/stop functionality. This user interface uses the **FLTK** library.

The virtual function `void GUIAudioApplication::UserMain(void)` by default just calls `Fl::run()`, but a derived class could add a more complex user interface here, before calling `Fl::run()`.

The function `void GUIAudioApplication::Run(int argc,char** argv)` has to be called to execute the application.

68 AudioApplication

This class behaves different on Linux and Windows.

- In Windows, DirectX audio can only be used in combination with a window open. Therefore, *AudioApplication* is derived from *GUIAudioApplication*, where the GUI is just a basic window with start/stop buttons.
- In Linux, it is perfectly possible to have an audio application without a user interface. Therefore, *AudioApplication* is derived from *BaseAudioApplication* directly.

Obviously, if you are developing an *AudioApplication* under Windows, but with a custom user interface, you should use *GUIAudioApplication* instead, so your application will be cross platform.

The function `void AudioApplication::Run(int argc,char** argv)` has to be called to execute the application.

69 Creating and running an Application

If you want to write an (GUI)`AudioApplication`, you have to implement the `void AudioMain(void)`, and optionally the `void UserMain(void)` and `void AppCleanUp(void)` methods.

- The `void AudioMain(void)` typically creates an *AudioManager*, instantiates some audio processing objects, and enters a loop, until the audio thread is canceled:

```
do{   /* processing code here, calling Do(...)'s */} while (!Canceled());
```

- The `void UserMain(void)` typically creates a (graphical) user interface. In a Linux command line *AudioApplication* this might just be waiting for a keypress. In a *GUIAudioApplication* (and therefore in a Windows *AudioApplication*, it typically sets up a **FLTK** user interface, and finally calls `Fl::run()`.

To execute the derived AudioApplication:

XIV Visualization Module

CLAM Visualization Module can be described as a set of tools that try to help application developers with the task of providing a Graphical User Interface to their applications. This help comes in the following forms:

- As easy to deploy and simple objects called *Plots*, that are able to render on the screen common data structures, such as CLAM::Array, or CLAM ProcessingData objects, such as the Spectrum.
- A set of toolkit-independent and efficient algorithms for generic data display implemented with OpenGL.
- A set of custom Widgets which are commonly needed when adding GUI features to a CLAM's domain application.
- A set of abstract classes that model the problem of projecting CLAM objects onto a GUI, allowing users to both visually inspect these objects as well as to interact with them.

Note that these different 'tools' might be used alone or combined. It's up to you to decide what do you need of the utilities the Visualization Module offers.

70 Plots

New!: We have just added new and fancier Plots based on the QT framework. The documentation is currently on a different document .

The VM *Plots* facilities are very similar to the ones offered by **MathWorks MATLAB (c)** or **GNUplot**, where the user is able to render easily a piece of data allowing some level of customization as curve color, axis scales, etc. while details as concrete rendering algorithms or interaction with widget toolkits is transparent to her. As in both **MATLAB** or **GNUplot**, VM *Plots* main purpose is to become a tool for debugging algorithms (or obtaining nice graphics for an article or lab report), not to be efficient but functional, not flexible but simple. So they are **not meant nor recommended** to be used in *production-level* code.

There are two kinds of VM *Plots*: the *generic plots* and the *specific plots*:

- **Generic plots** allow to render data contained in generic data objects such as arrays or CLAM::BPFs. Range and meaning of the axis is left to users to define. *Generic plots* are subdivided into two kinds: the *single function plots* that are able to render just one function, and the *multi function plots* that are able to render multiple functions simultaneously.
- **Specific plots** allow to render 'special' kinds of data such as audio signals, spectrums and the like. They are 'smarter' than *generic plots* since some parameters as axis ranges or meaning are implicit on the kind of object, and they might be using rendering algorithms quite specific for them. Also some features that are specific to them (such as playing back an audio signal) are accessible from these *plots*.

We provide you with some examples that should help you to become familiar with *Plots* facilities. Don't be afraid to experiment with these examples - you might discover different ways of using them that we did not foresee!

70.1 Plots examples

Currently, in CLAM 0.5.3 there are available the following examples on VM *Plots*:

70.2 Visualization Module Plots: single function plot	
Makefile / Visual C++ project location:	build/Examples/Simple/SinglePlot_1
Sources location:	examples/SinglePlot_example.cxx
Complexity:	Low
Keywords:	CLAM GUI services, simple data visualization
Pre-requisites:	Familiarity with CLAM::Array and CLAM::BPF
Description:	This example shows how to plot on the screen some data object part of a simple DSP application

70.3 Visualization Module Plots: multiple function plot	
Makefile / Visual C++ project location:	build/Examples/Simples/MultiPlot
Sources location:	examples/MultiPlot_example.cxx
Complexity:	Low
Keywords:	CLAM GUI services, simple data visualization
Pre-requisites:	Familiarity with CLAM::Array and CLAM::BPF. It is recommended to take a look first on the single function plotting example.
Description:	This example shows how to plot on the screen some data object part of a CLAM-based DSP application, as well as combining several functions in the same plot window.

70.4 SDIF I/O, Segments and plots	
Makefile / Visual C++ project location:	build/Examples/Simple/SDIF_And_Segment
Sources location:	examples/SDIF_And_Segment_example.cxx
Complexity:	Medium
Keywords:	SDIF I/O, CLAM Segment, VM Plots
Pre-requisites:	Basic knowledge of Processing objects interface, basic knowledge of SMS Analysis algorithm byproducts.
Description:	Shows how to restore a CLAM::Segment object stored into a SDIF file, and inspect visually its contents.

71 Model Adapters and Presentations

In order to decouple the model elements (mainly CLAM Processing Data such as Audio or Spectrum) a variant of the Model-View-Controller architectural pattern was implemented. In this new version the main actors are the Presentation, the Model Adapter, and the Model Controller.

A Presentation is a graphical metaphor through which some information contained in the model object is shown to the user. A Presentation can be anything from a simple widget to a full application graphical interface, depending on the complexity of the model object to be presented. A Presentation can be activated and deactivated, therefore its existence does not imply its visibility.

The ModelAdapter class defines the interface that is common to all model object adapters in CLAM Visualization Module. It offers the interface required by the Observable actor in the GOF Observer pattern [GOF]. The Adapter concept was chosen in order not to taint the model object interface and to separate effectively the model objects from its representation. The main operation in the ModelAdapter class is the abstract Publish operation that must be implemented in all subclasses in order to publish the updated model object internal state.

The ModelController class is similar to the ModelAdapter except in that, besides from publishing the model object state, it also allows to modify it. For that reason it adds the Update operation to the previously mentioned Publish.

The CLAM Visualization Module also implements a Signal&Slots mechanism similar to that offered by frameworks such as QT [QTProgramming]. The basic rationale behind the Signal&Slot mechanism is the following: Sometimes it is required that an object notifies a change in internal state or the reception of a message to any number of listeners. This situation can be modeled in different ways but most of them suffer from a major drawback: coupling. In this sense, the caller must know to some extent the callee interface. Because of this, reuse capabilities are reduced. The Signal&Slot idiom gives solution to this problem. The Signal models the concept of "event notifying", and signals are connected to Slots that represent "event handlers".

In CLAM the Signal&Slot idiom is implemented through three main classes: the Signal class, the Slot class and the Connection class. The Signal and Slot classes model the obvious concepts previously explained. On the other hand, the Connection class models the knowledge a signal has about who has to be notified whenever a client invokes the Emit() operation on it. Each time a Signal and Slot objects are bound together a Connection object is created, tagged by a Global Unique IDentifierIdentifier (GUID). This particular implementation was loosely derived from R. Hickey’s article "Callbacks in C++ using Template Functors" published in C++ Report ’95.

Apart from the previous tools, the non-dependency from graphical toolkit implementation is also accomplished through the use of a Widget Toolkit Wrapper. This Creator/Singleton class produces objects that are abstract wrappers for accessing a GUI Toolkit low-level functionality such as triggering the event loop, triggering the execution of a single iteration of the event loop or setting the refresh rate for graphic displays.

XV SDIF SUPPORT

SDIF or Sound Description Interchange Format is a binary format defined and supported by various research teams. It was created with the goal of having a common format for exchanging synthesis samples, usually spectral domain data coming from a previous analysis.

The mapping of CLAM data to a SDIF File is fairly simple; it is always done from a CLAM::Segment (see section 51.6). The Segment internal structure can very easily be mapped to SDIF as it basically holds inside an array of time-ordered frames. Out of the different data inside a frame, only the necessary for the synthesis process is stored into SDIF. That is, residual spectrum, sinusoidal peaks with track number and fundamental frequency. Due to the SDIF specification, all magnitude data needs to be stored in linear (as opposed to what is usual in CLAM, where data is in dB).

All this is done using two CLAM Processing: SDIFIn and SDIFOut. SDIFIn takes a Segment in its output port because it needs a single reference where to store the created frames. SDIFOut takes frames in its output port and enables storing frames even from different segments.

As an example, this is the way the Analysis/Synthesis Example application handles the loading and storing of SDIF data.

Loading:

```
SDIFInConfig cfg;
cfg.SetMaxNumPeaks(100);
cfg.SetFileName(inputFileName);
cfg.SetEnableResidual(true);
SDIFIn SDIFReader(cfg);
SDIFReader.Output.Attach(mSegment);

while(SDIFReader.Do()) {}
```

and Storing:

```
SDIFOutConfig cfg;
cfg.SetSamplingRate(mGlobalConfig.GetSamplingRate());
cfg.SetFileName(mGlobalConfig.GetOutputAnalysisFile());
cfg.SetEnableResidual(true);
SDIFOut SDIFWriter(cfg);
int nFrames=mSegment.GetnFrames();

for(i=0;i<nFrames;i++)
{
   SDIFWriter.Do(frames[i]);
}
```

In order to add SDIF support to an application you might be developing, you also have to add to your project/meakefile the files included in the /src/Tools/SDIF. These files implement the necessary classes for mapping the sdif file content, namely the definition that the specification gives of the following concepts: Frame, Matrix, Stream and File.

DEVELOPER DOCUMENTATION

XVI CLAM Coding Conventions

72 Indenting code

These guidelines aims that the code will be always well indented with the indentation size that each programmer prefers and keeps some parts of the code vertically aligned.

- Use **only tabs to indent** the beginning of each line.
- Use **only spaces to align vertically** (For example when writing variables declarations in two aligned columns or splitting a long line)
- If the aligned word is the first one in the line, you must keep the same number of tabulators that has the line you want to align with.

On the following example `[tab]` represents tabulators and `.` represents spaces used for alignment.

```
[tab] [tab] double.a; [tab] [tab] int....b; [tab] [tab] if (a<b && b<c) { [tab] [tab] [tab] std::cout << "a is less than c"; [tab] [tab] [tab] .........<< std::endl;
```

Ensure that your editor does not change tabulator by spaces or spaces by tabulators.
Some hints about some used editors:

- MS VisualC++ : `Tools->Options->Tabs-> FileType:C/C++, Keep Tabs, AutoIndent: Smart`
- Emacs: Add something like this to your .emacs file: `(setq c-default-style '((other . "bsd"))) (custom-set-variables '(tab-width 4))`

73 Naming conventions

- When an identifier is composed of several words, they must be appended together (without "_') and distinguishing each word with the initial in upper-case.
- **Functions** and methods starts with upper-case.
- **Variables** and members start with lower-case.
- Normal **members** should start with a lower-case 'm' like in `mMemberName`
- **Static members** should start with a lower-case 's' like in `sMemberName`
- Identifiers inside an **enum** declaration must start with a lowercase 'e' like in `enum MusicalStyles { eRock, ePop, eClassic };`
- **Dynamic Types** members: Mustn't start with 'm' or 's' and in upper-case. Dynamic types are implemented using macros that will create code for the member's accessors. So, when registering a dynamic type member like `DYN_ATTRIBUTE(0,public, ComplexArray, Array<Complex< TData> >)` keep in mind that the identifier (3rd parameter) will be used for creating the methods: `GetComplexArray()` and so on.

§ It's possible that in the future we'll change to a template-based implementation, so that it will turn to a normal member declaration. Then the naming convention is likely to be like `dComplexArray`.

74 Programming style

Use **const** whenever is possible:

- Declare **`const`** a member function (method) when it doesn't modify the object
- Declare **`const`** the parameters that are not going to be modified inside the function

- Declare **`const`** the return if it is something of the object we don't want to be modified outside
- Use **`const &`** parameter passing instead of by value, for efficiency

In general is better to pass parameters by reference than by pointers, because we implicitly are not allowing delete the object.

75 Error Conditions

- See Error notification and managing for more details.
- If an error condition is to be handled by the function caller throw an exception and **publish it on the function prototype with the `throw` keyword**.
- If an error condition is not defined as interface use `CLAM_ASSERT`.
- Every published exception must be caught. Not catching a published exception is a bug.
- Catch the exceptions locally. Don't group several functions that may throw exceptions on the same `try` clause.
- Catch the exceptions concretely. Catch for one concrete type of exception so you can recover for it.
- When you can't handle the exception locally, translate the exception to the caller context and throw.

76 Debugging aids

There are some useful coding techniques for finding "well hidden" errors very quickly, that otherwise might escape tests but can appear at most undesirable circumstances: sometimes this is called defensive programming. Here we give some guidelines:

- Systematically check that the state is consistent with asserts. Use the macro `CLAM_ASSERT(`*`bool_expression, info message`*`)` for that. Its code will be removed when compiling with optimization options, see the Error notification and managing for more details.
- The **precondition** and **postcondition** of a method are the restrictions --about the parameters and object state-- that a method must satisfy at the entry and return points respectively. Is very recommendable to use asserts for checking these conditions.
- An **invariant** of an object is the set of restrictions that the state of the object will always satisfy during its life. Write a method: `bool FulfillsInvariant(void)` that checks the most relevant restrictions of the invariant and uses it like this : `CLAM_ASSERT(FulfillsInvariant(), "anything...").` Moreover, is very useful that FulfillsInvariant() call the same method of its members if it exist.
- Another possible signature for this method is : `void Fulfillsinvariant(void) throw ErrAssertionFailed`. In that case the invariant conditions will be checked using `CLAM_ASSERT` and no boolean return is necessary, this can be an advantage if we want each condition to have a different informative message.
- Companion removable code of a `CLAM_ASSERT` must be enclosed by `CLAM_BEGIN_CHECK` and `CLAM_END_CHECK` macros that remove de code when some optimization options are enabled.

```
CLAM_BEGIN_CHECKfor (int i=0; i<N; i++) {   CLAM_ASSERT(array[i].IsValid(),"Invalid Element                                    Found");}CLAM_END_CHECK
```

See examples of `FulfillsInvariant()` in the code of the library classes `DynamicType` whose file is placed in src/Base/ or any `XMLAdapter class` that can be found at src/Storage/XML.

XVII Error Handling

77 Use case analysis

77.1 Actors

There is two kinds of actors, humans actors and automated actors.

- **Automated actors:** They detect the exceptional conditions, but also can receive exceptional conditions notifications.
  - **The CLAM library:** The library we are developing. This also may include users add-ons that match the interface we provide.
  - **Third party libraries:** They are libraries that our library uses. For example the Standard C++ Template Library, Fltk, OpenGL, XercesC++... They report the library error conditions to our library.
  - **User application:** Source code that uses the library. Parts of CLAM may adopt this role when they use some interface also offered to the outside of the world.
- **Human actors:** They are only notifications receivers. Humans needs a text based interface.
  - **Library programmer:** That is a programmer that is developing CLAM library. He receives notifications that are useful for debugging the library.
  - **Library user (application programmer):** The programmer of the final application. He receives notifications that are useful for debugging its application.
  - **The final application user:** It is the last notification receiver when the application is released.

77.2 Stages

Human actors implied are different on the different stages of the library code:

- **Library development**
- **Application development**
- **Application usage**

Each development stage has a different main human error notification receiver and different behaviours are expected.

77.3 Mechanisms

There are three main error notification mechanisms:

- A **public exception** is a documented error condition for a function that callers can manage in some way. An exception object is thrown to let the caller know the error nature and its details in order to recover properly. So, public exceptions are error notifications from one piece of code to other that uses the former one. **Receiver:** Any automated actor.
- An **assertion** is a check for some precondition for a piece of code. This precondition is suposed not to occur, so failing the check means that a bug is happening and the following code will have unexpected results. **Receiver:** Programmers, but via the application on release mode.
- A **user message** is a direct text based notification to the user. Because this kind of notification is application dependent, **it will not be issued by the library**. **Receiver:** The application user.

78 Sanity checks and assertions

78.1 Expression assertions

You can create an assertion of an expression by using the `Assert` macro like that:

```
CLAM_ASSERT(i<mSize, "Accessing past the end of an array");
CLAM_ASSERT(mBuffer!=NULL, "Array buffer points to NULL");
return mBuffer[i];
```

On development stage, that is, with the `DEBUG` macro defined, what the programmer wants is to interrupt the program where the assert failed in order to start debugging from this point.

In release mode, that is, when the `DEBUG` macro is not defined, we have choosed not to ignore asserts, as does the default standard `assert` macro behaviour, because not fullfiling an assert can lead to unpredictable state. But instead of aborting the program, it will throw an `ErrAssertFailed` exception that can be catched. So an application catch the exception and, for example, do a backup of the data or open an error reporter window before the application does finally crash.

78.2 Statement based 'assertions' (checks)

`CLAM_ASSERT` macro is only useful for expressions. When you have a check whose code is not only based on a simple expression, but in a complex statement, then you must use the following construct:

```
CLAM_BEGIN_CHECK
   for (int i=0; i<N; i++) {
      CLAM_ASSERT(array[i].IsValid(),
         "Invalid Element Found");
   }
CLAM_END_CHECK
```

78.3 Documenting assertions

When you add assertions to a function body about the status derived from caller context, and they are not so obious conditions, you should **documentate them as a function preconditions**. You can use the `@pre` doxygen directive for this.

This way you are telling the caller that it may assure to fullfil those preconditions.

78.4 Optimization and assertions

WARNING: Please, use this section only when you are very sure about, the needing of disabling one assertions and checks on release mode. Mantaining the assertions on release code will give the final application a chance for doing a 'gracefull crash' by catching at top level the `CLAM::ErrAssertionFailed` exception.

Because assertions do take time, on critical parts, you may decide that one assertion will not be part of the release code by using the `CLAM_DEBUG_ASSERT` macro instead of the `CLAM_ASSERT` macro.

There is also a debug only version for statement based asserts, for example:

```
CLAM_BEGIN_DEBUG_CHECK
   for (int i=0; i<N; i++) {
      CLAM_DEBUG_ASSERT(array[i].IsValid(),
         "Invalid Element Found");
   }
CLAM_END_DEBUG_CHECK
```

Please, use `CLAM_ASSERT`, `CLAM_BEGIN_CHECK` and `CLAM_END_CHECK` instead of `CLAM_DEBUG_ASSERT`, `CLAM_BEGIN_DEBUG_CHECK` and `CLAM_END_DEBUG_CHECK` every where you can.

Another extreme option in order to speed up the library by disabling completely all the checks is to compile defining the `CLAM_DISABLE_CHECKS` global macro which disables all the asserts and checks.

So, debug-only checks and assertions are used to remove some selected checks on debug mode only, and `CLAM_DISABLE_CHECKS` is used to remove all of them whatever the mode you are compiling in.

78.5 Managing assertions from the application

- Callback: You can use the `SetAssertFailedHandler` function in order to change the function that is called when an assert is failed in debug mode.
- Catching: In release mode, you can catch the `ErrAssertionFailed` exception at the top level application in order to do some crash management and backups.

78.6 Debugging the release mode

Because there is some parts of the code that changes between the debug mode and the release mode, some bug can happen on the release mode and not on the debug mode. We have left the possibility of simulating asserts as they work on the release mode but being on the debug mode.

If you globally define the macro `CLAM_USE_RELEASE_ASSERTS`, the asserts will be defined as it will be on the release mode althought the `DEBUG` macro is also defined. Then you may check whether the bug is on the release assert code or not.

79 Exceptions

79.1 Previous note

Most of the guidelines provided here for exception use, are not still applied to the inner CLAM code. By now only the Assert/Exception use cases have been revised.

Future CLAM releases will match more faithfully this and users must use the library as it would.

79.2 When to use Exceptions

Exception handling is a mechanism provided by C++ for recovering from an error situation which solution is unknown in the context where the error condition is given but the solution is known somewhere in a higher level in the call stack.

So, when a function meet an error condition, it must throw an exception only when:

- The error is recoverable (if not, you might use an assertion instead)
- Recovery strategy is not fully known on the error detection context.

In C programmers terms, exceptions usage in CLAM, as suggested by the standard, is like function returned error codes but without interfering the return process and handling the error several function calls thru the call stack.

In short, Exceptions and Assertions differ on the following:

- Exceptions are error conditions that may be anticipated, and recovered by the caller code.
- But failed assertions are error conditions that you can recover from in runtime, they are detected runtime bugs.

79.3 Contract between throwers and catchers

Because callers must know which exceptions are thrown by the method in order to catch and handle them, exceptions are part of the called method prototype, say the contract with its client (the caller).

Forming part of this contract, exceptions must be present both on the prototype and the Doxygen documentation.

```
/**
 * Gets a property
 * @param file The property file
 * @param value The property name
 * @returns The value of the property
 * @throws NotFound when the property is not present
 * @throws IOError when an error ocurs accesing the file
 */
std::string getProperty(const std::stream & file,
                        const std::string & name)
   throw (NotFound, IOError)
{
   ...
   if (somecondition) throw NotFound(name);
}
```

Exceptions must be documented by using the Doxygen tag `@throws`. You also may explicitly define the `throws` clause in the method prototype. Just like the previous listing does. You may want to be less conservative and make the client assure that the key is always present before calling your method. Then you will not throw any exception and you may use an assertion instead. A `@pre` entry in the Doxygen documentation tells that the function needs to assure that the key exists. Just like the following code.

```
/**
 * Gets a property
 * @param file The property file
 * @param value The property name
 * @returns The value of the property
 * @pre The property must exists on the file
 * @throws IOError when an error ocurs accesing the file
 */
std::string getProperty(const std::stream & file,
                        const std::string & name)
   throw (IOError)
{
   ...
   CLAM_ASSERT(!somecondition, "Property not found");
}
```

Notice that the client will not be able to handle this error condition.

Responsability facts:

- Not assuring the precondition means a bug in the client code.
- Throwing an undocumented exception is a bug on the called function.
- Not catching a documented exception is a bug on the client code unless published as thrown by the client code.

Exceptions must be documented as part of the interface because the client must handle this error condition. Also, often assertions should be documentated because they require a precondition, that the client must filfull before calling the method.

```
std::string getProperty(const std::stream & file,
                        const std::string & name)
{
   try {
      ...
      if (somecondition) throw NotFound(name);
   }
   catch (IOError) {throw;} // rethrow it
   catch (NotFound) {throw;} // rethrow it
   catch (...) { // Any other, call unexpected
      std::unexpected();
   }
}
```

So any unespecified exception will be translated to an `unexpected` exception and it could not be managed by any client.

79.4 Exception data and exception hierarchy

Throwers must provide enough information to the catchers for them to recover the best way. This information must be meaningfull to C++ code because human readable messages are complicated to parse and understand by handling code.

Human readable messages are useful for debugging and bugreporting tasks and asserts covers this functionallity.

The catcher can get this information from several sources:

- The exception type allows to do a first filter on what has happened and which error handling block you must use.
- Error codes can complement this first classification
- Realtime information of the error can be attached as attribute members of the exception object. This can help the catcher to improve the recovery.

79.5 Exception handling

All CLAM exceptions subclasses from `CLAM::Err`. `CLAM::Err` itself, subclasses from `std::exception`.

As a general rule, you ought not to catch exceptions on a general way. That is caching elipsis (`...`), `std::exception`, `CLAM::Err`... Catching an exception means giving a solution for it. Probably, if you are catching exceptions in a general way you aren not giving a suited solution.

So take a look to the function interface and catch only the documented exceptions providing a solution for them.

Developers tend to group in a try statement a very large piece of code when they refuse to do any handling. This is not what is intended with exceptions in CLAM. This kind of ’handling’ is for assertions.

79.6 Contextualization

What an exception means on a function context may not be the same on a higher context. For example, an AudioIO processing configuration method may receive an exception about an already used selected audio device. If you decide not to handle this error here, you may catch the device exception and throw an exception about the configuration fail that has more sense on the context.

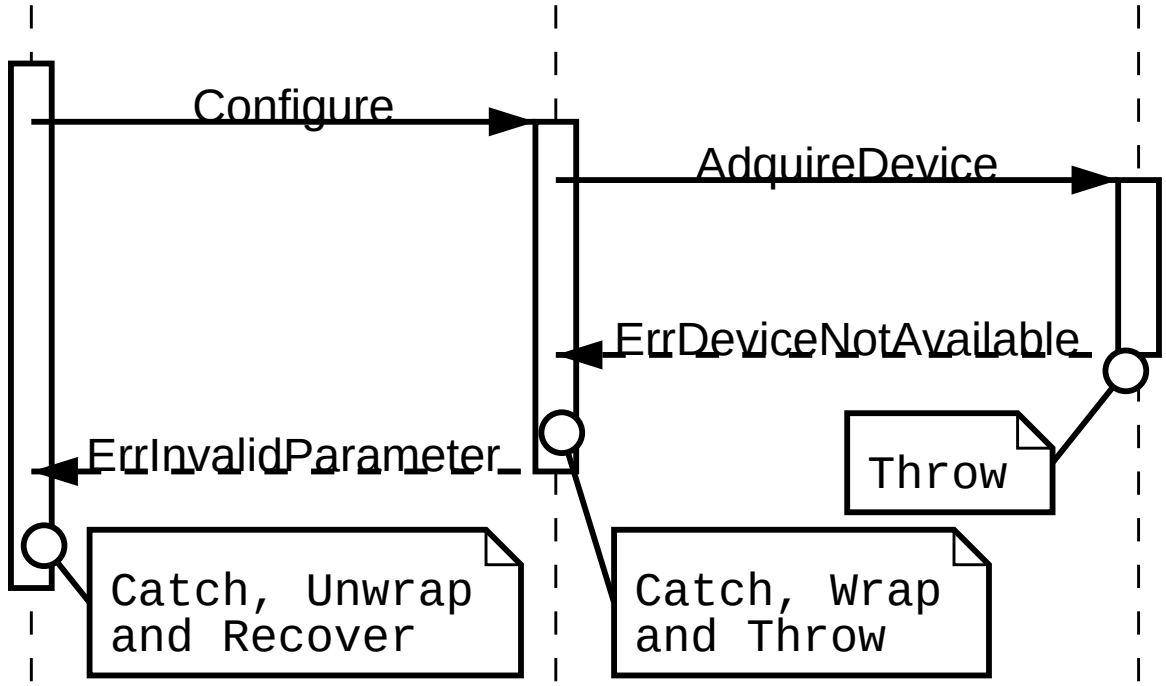


Figure 9: Contextualization example

Although throwing a different exception you may embed the old exception onto the newer one. If the error handling code needs more information it can extract it from the embeded exception.

`CLAM::Err` defines the interface that allows to embed and disembed exceptions.

XVIII Dynamic Types

80 DTs that derive from an interface class

Implementing such dynamic behavior and load/stores methods by default, just by deriving from a base class, and declaring attributes using macros, requires a null efford to the user but leads, of course, to certain limitations. The most important is the inheritance constraint, that can be expressed this way: *dynamic attributes can only be declared in a class that derives directelly from the DT base class*. That means: if we want to create a MyDynamicType class this must derive directly from `DynamicType` and if we want to extend it, actually we can, but we cannot declare new dynamic attributes in the lower class.

Anyway we found very interesting to give the ability of placing an interface class between the concrete DT and the DT base class. This can be done using a new (and long) macro, for example:

```
class Audio: public ProcessingData {
public:
   DYNAMIC_TYPE_USING_INTERFACE (Audio, 4, ProcessingData);
   DYN_ATTRIBUTE (0, public, TData, SampleRate);
   // . . .
}
```

In this example, `ProcessingData` is an interface or pure abstract class. It is basically usefull for: **i)** organizing concretes DTs, **ii)** forcing the existence of certain dynamic attribute XXX, by means of declaring a virtual method `GetXXX`, and **iii)** forcing all derived classes to be DTs.

In the unlikely case of having to declare new interface classes, it is just matter of following the pattern given by this example:

```
class ProcessingData : public DynamicType
{
public:
   /** Constructor of an object that will contain the number of
   *  attributes passed by parameter  */
   ProcessingData(const int n) : DynamicType(n) {};

   /** Copy constructor of a ProcessingData object  */
   ProcessingData(const ProcessingData& prototype,
      bool shareData=false, bool    deep=true) :
      DynamicType(prototype, shareData, deep){};

   virtual ~ProcessingData(){};
};
```

81 Typical Errors

81.1 Detected errors at compile time:

Lukily enough, we can detect a number of basic errors at compile time. This is possible by means of an implementation with extensive use of template and function overloading techniques. Lets see these detected errors with an example:

81.1.1 Constructor errors

Here we will show two examples of typical errors related to the concrete DT constructor:

```
class Note : public DynamicType {
   public:
      DYNAMIC_TYPE     (CNote, 5)
// ...
```

The compiler will report that the constructor name CNote doesn't match with the class name Note.

```
class Note : public DynamicType {
   DYNAMIC_TYPE     (Note, 5)
// ...
```

Here we have left the public: keyword before the DYNAMIC_TYPE macro, this will cause a compilation error when it tries to instantiate an object of the type Note.

81.1.2 Attribute position out of bounds

```
class Note : public DynamicType
{
public:
   DYNAMIC_TYPE     (Note, 4)
   DYN_ATTRIBUTE    (0, public,  float,   Pitch)
   DYN_ATTRIBUTE    (1, public,  unsigned,    NSines)
   DYN_ATTRIBUTE    (2, public,  ADSR,        Envolvent)
   DYN_CONTAINER_ATTRIBUTE (3, public, std::list<Sine>, Sines,                                    harmonic)
   DYN_ATTRIBUTE    (4, private,     Audio,   Wave)
};
```

Here we have a too big identifier the compiler will complain saying that the following symbol is undefined:
`CLAM::MyDT::AttributePosition<6>::CompilationError_AttributePositionOutOfBounds`

81.1.3 Attribute not defined

```
class Note : public DynamicType {
public:
   DYNAMIC_TYPE     (Note, 5)
   DYN_ATTRIBUTE    (0, public,  float,       Pitch)
   DYN_ATTRIBUTE    (2,  public,  unsigned,    NSines)
   // ...
```

Now we've left an empty position and it will complain about this symbol:
`CLAM::MyDT::AttributePosition<1>::CompilationError_AttributeNotDefined`

81.1.4 Duplicated attributes

```
class Note : public DynamicType {
public:
   DYNAMIC_TYPE     (Note, 5)
   DYN_ATTRIBUTE    (0, public,  float,       Pitch)
   DYN_ATTRIBUTE    (0,  public,  unsigned,    NSines)
   // ...
```

In this case we will get an error of duplicated methods that recive parameters of type: `AttributePosition<0>`.

Morover, we'll get this message if we `define DYN_ATTRIBUTE(5,.)`. Yes, it shoud be an Attribute position out of bounds error, but we couldn't manage to do so.

81.2 Detected errors at run time

Within the CLAM library we have established two compilation macros that gives us a way of choosing the degree of verifications to be done at run time. Of course going deeper into this issue is not done in this section. You can find it at chapter XVII.

The important thing to know is which kind errors will be catch in run-time depending on the compilation option choosen:

81.2.1 Compiling in debug mode (the macro `_DEBUG` defined)

In this mode, when adding an attribute for the first time, an assert will stop the execution if a dynamic attribute name has been repeated, as in this example:

```
class Note : public DynamicType {
public:
   DYNAMIC_TYPE     (Note, 5)
   DYN_ATTRIBUTE    (0, public,  float,       Pitch)
   DYN_ATTRIBUTE    (1,  public,  unsigned,    Pitch)
   // ...
```

81.2.2 Compiling in a non debug (release) mode

we stay safe in this kind of things: the use of `GetXXX` or `SetXXX` of not instantiated attributes will throw an assert exception.

81.2.3 Compiling for the best run-time efficency

For reaching the best run-time efficiency we compile in release mode and we must set the compilation flag: `CLAM_DISABLE_CHECKS`.

In this case if the code calls a `GetXXX` or `SetXXX` where `XXX` is not instantiated we'll get a segmentation fault (if we are lucky!)

81.3 Non detected errors

Furthermore, there is a kind of errors that can not be detected even in debug mode: they are caused by inconsistency of a reference to a dynamic attribute. It is very important, when writing a DT, to keep in mind that any `UpdateData()` can cause a movement of every dynamic attribute, and so references (pointers) to these attributes can turn out inconsistent. The first rule of thumb would be: never keep references to dynamic attributes. But if this is really necessary, then we can take profit of the copy constructor of the dynamic attribute, because all these movements are done calling this constructor. So is a duty for the dynamic attribute copy constructor to keep the references to itself consistent.

There is another version of this error that is much more subtle: Imagine that we have a DT class MyDynamicType with an attribute A. Now, myDT is an instance of this class, and the attribute A have a method `DoMess()` : what it does is an `UpdateData()` of its parent (that is: the object myDT). This can be catastrophic because this UpdateData can move the attribute A, invalidating the this pointer of the method `DoMess()` in the middle of its execution. Ugly, indeed! If we need such object coupling wetween a dynamic types an its attributes then the flag `SetPreAllocateAllAttributes()` should be used. see section 83

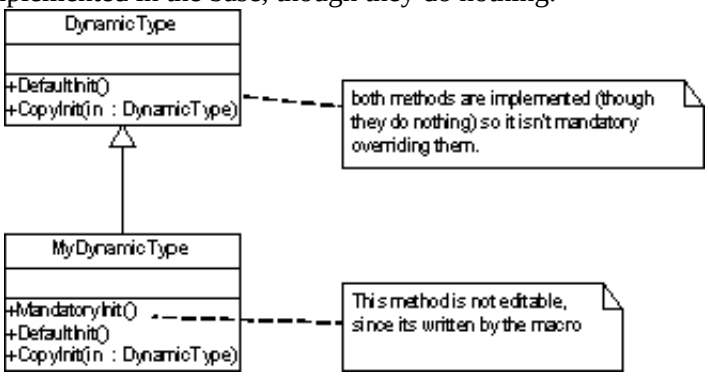
82 Constructors and initializers

This section addresses the issue of where to place initialization code for DTs and how to write new constructors.

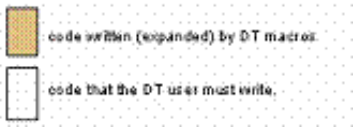
Basically what makes a DT different from a normal C++ class is that the first needs, at construction time, initializing some static table (for type description information) in the case of being the first instance of its class. And, of course, this job cannot be done at the DT base class instead.

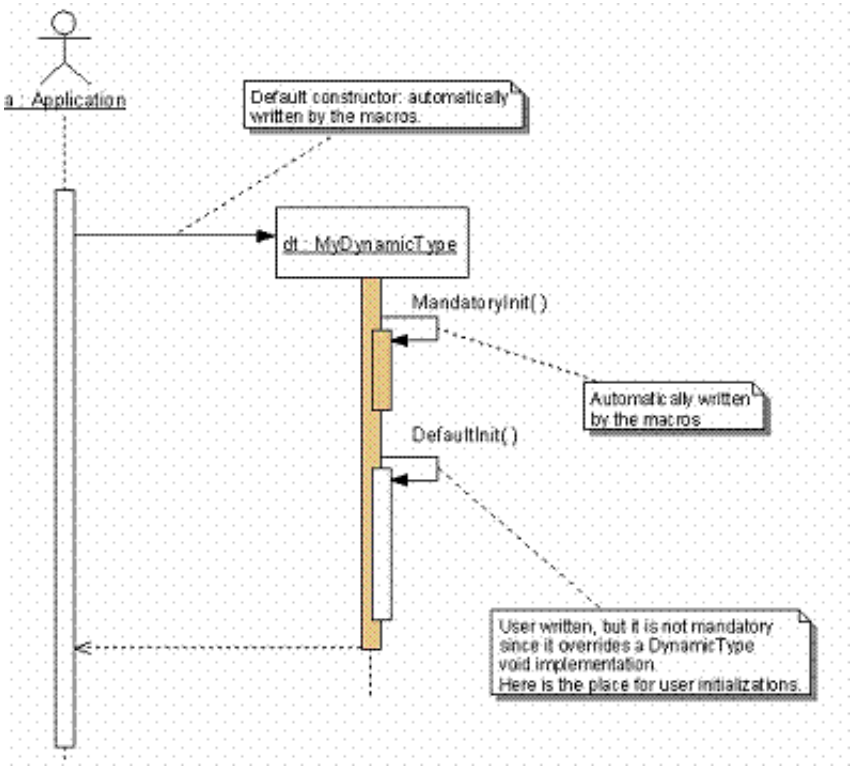
The `DYNAMIC_TYPE` macro expands two constructors : the default one (without arguments) and the default copy constructor (with argument of the same concrete type).

So what we have done is, in these automatically written constructors, is let them call an initialization function, that will be written by the user. They are `DefaultInit()` and `CopyInit()`, and are represented in the next class diagram. Notice that they are virtual and implemented in the base, though they do nothing.

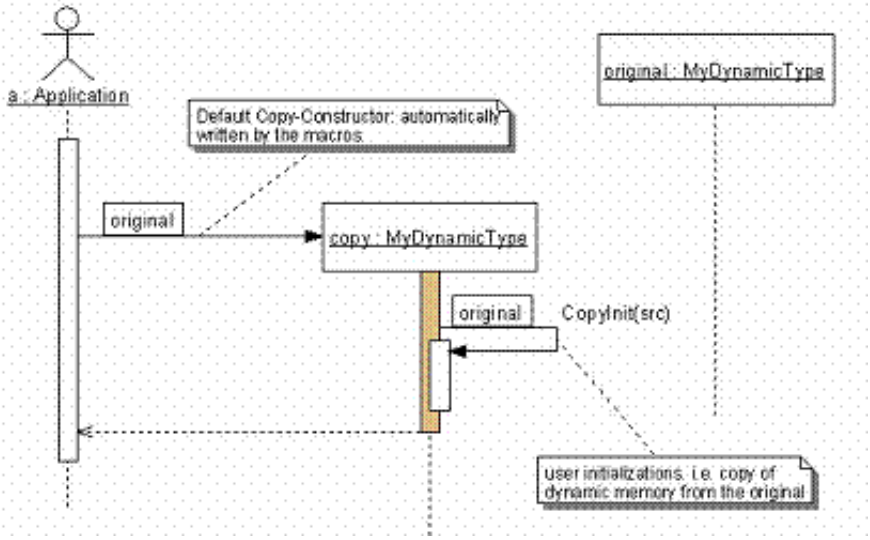


On the other hand, the macro writes a concrete (not virtual) method `MandatoryInit()` that contains all the static table initializations. This is the sequence diagram for a normal DT constructor:





In the case of the copy constructor, the `DefaultInit()` won't be called, but the `CopyInit()` will instead. In some cases we could want the same behavior for both initializators, then we could implement the first and let the second call the first. In the next sequence diagram you can notice also that the `MandatoryInit()` is not called, that's only an implementation detail: when a copy constructor occurs we are certain that at least one instance exist of the current concrete DT.



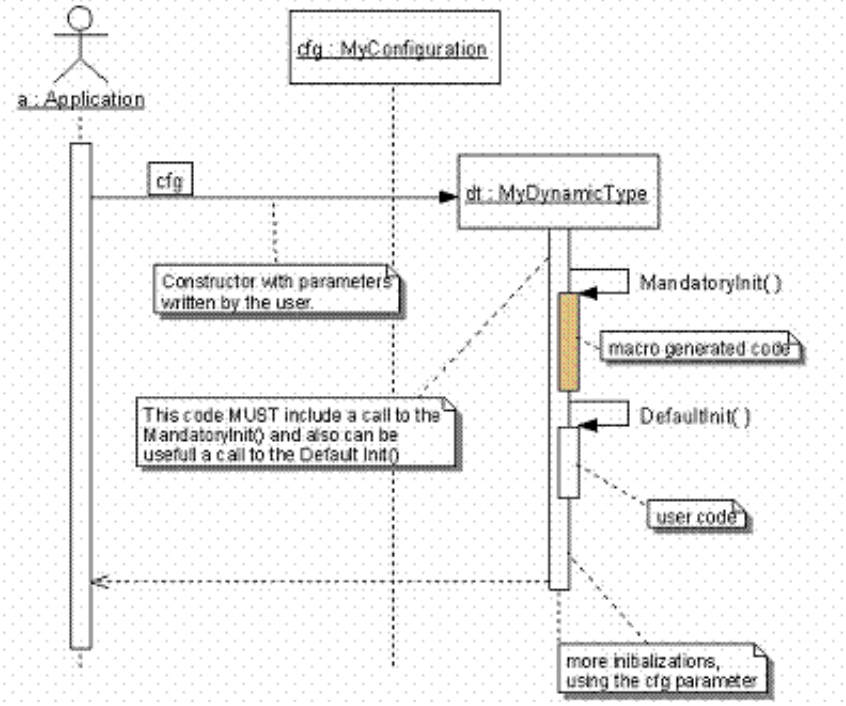
We can go further with the customization of our DTs: writing new constructors, (i.e. passing a configuration object as parameter). Here we have to be very carefull following these steps:

Call the super constructor with a parameter N, where N is the number of dynamic attributes. (this way: `MyDynamicType() : DynamicType(N) { . }` or otherwise using the interface class, if this exist. An example of this can be found at this source file: src/Data/Spectrum.hxx)

Call the macro-expanded `MandatoryInit()`

You may also want to call the user writed `DefaultInit()`.

This next figure illustrates the calling sequence of such a constructor. Notice that unlike the previous ones, here almost all code is unpainted, and so meaning: written by the user.



83 Tuning a DT

In some ocasions the force of not moving dynamic attributes is stronger than the force of optimizing the dynamic memory. This is specially true in cases of attributes that are large buffers (i.e. using `Array`, `std::vector`.) or large composition structures. In other words: the innocent action of adding a new attribute and updating data in some classes might carry a huge copy of buffers and tree structures, thus making dynamic operations (adds and removes) extremelly inneficient.

That is the reason for introducing a global flag in the DT class that can be set with this method:

`SetPreAllocateAllAttributes()`. When this flag is set the next `UpdateData()` will perform the last reallocation (if necessary) of the DT object life. From this point, the maximum data memory is allocated and all dynamic attributes offsets are fixed and even further dynamic shape changes (adds and removes) will not produce dynamic attributes movements.

Then, this kind of classes with 'heavy attributes' should call `SetPreAllocateAllAttributes()` in its `DefaultInit()`.

Notice that, by now, this flag cannot be unset: that is just because we couldn't find a single use to allow it.

84 Debugging aids and compilation flags

As it has been explained in section 35, the DT base class implements a `Debug()` method which will display some internal information as well as will write a 'Debug.xml' file with its xml content (only when compiling with the flag `CLAM_USE_XML`).

The next example of console dump shows for each attribute features like its size in bytes, name, and its current pointer. At the left side of each attribute information we can read '--' or with some dash substituted with an 'A' or 'R'. This are the added and removed flags, meaning that its attribute will need memory update in the next `UpdateData()`. Is important to note that when an attribute XXX is

marked in one or other way, `HasXXX()` will return false.

```
Class Name: Dyn at: 0012FEB4
[#attr.], dyn offs, name, type, {comp,dynType,ptr,strble}, exist, size, memPos
---------------------------------------------------------------------
{ size, allocatedSize } = { 28 , 0 }

 -R [0] 0 , Int , int , {0,0,0,0} , 1 , 4 , 00892F38
 A- [1] 4 , MyA , CompWithBasics8 , {0,0,0,0} , 1 , 24 , 00892F3C
. . .
```

Apart from `CLAM_DIABLE_CHECKS` that will increase a lot the run-time performance of DTs, exists another one: `CLAM_EXTRA_CHECKS_ON_DT` that will do exactly the opposite, but at least will automatically check the DT invariant in every DT operation (adds, update data, etc). This can be very usefull if we are getting paranoids about some possible bug in the DTs and for finding it, if this is the case (hopefully not).

85 Pointers as dynamic attributes

This is an easy point, right now: pointers as dynamic attributes are not supported. Anyway we have foreseen the use of pointers in two scenarios: as references to the same hierarchic structure and for holding polimorphic types. This will involve the introduction of a couple of new macros, and these changes are scheduled for a near future.

86 Copies of DTs

There are several ways of getting DTs copies, all of them implemented at the DT base class:

`DeepCopy()` : this method is declared in the Component abstract class, so it returns a Component type and a cast can be necessary. As its name says, it behaves deeply or in a recursive way. It's done following these rules:

Copying each attribute that is Components (i.e. DTs) calling its `DeepCopy()`.

- Copying the rest of non-Component attributes using its copy-constructor.

The `CopyInit(const DynamicType&)` method is called for non-dynamic attributes copies. This method can be overrided in the concrete class, see section 89 for more details about this.

Copy constructor : it just calls the DT `DeepCopy()` method.

`ShallowCopy()`: it calls the `ShallowCopy()` for each Component attribute and the copy-constructor for the rest. The `CopyInit()` is also called.

87 DTs and XML

87.1 The default XML Implementation for DynamicTypes

Any Dynamic Type has a default XML implementation. By default, all the dynamic attributes are stored as XML elements with the attribute name as tag name and following the order specified in their declaration. If a dynamic attribute is not instantiated, it is not stored.

On loading, first all dynamic attributes are instantiated, and, then, each one are tried to be load. Those attributes that are not in the XML source are marked as removed.

87.2 XML aware dynamic attributes

Every dynamic attribute is stored as element but the way the attribute content is stored depends on the kind of object.

Dynamic attributes that are **components** have direct XML support and are stored recursively.

Basic objects like C primitive types and some others (`std::string`, `CLAM::Complex<TData>`, `CLAM::Polar<TData>`, `CLAM::Point<TData>`...) use their extraction and insertion operator to generate plain content.

You can define any class, for example `MyBasicType`, to be used in XML as a basic type doing the following:

- defining their extraction (>>) and insertion (<<) operators over std::streams
- and using the following macro call at namespace level:

```
CLAM_TYPEINFOGROUP(CLAM::BasicCTypeInfo, MyBascType);
```

About the insertion and extractor operators, you must be careful to choose a parseable format that will allow not to waste with the extractor more input than the necessary from the stream.

Although **`char*`** has been defined as a basic type to easily inserting string literals, do not use it to load because it may lead to buffers overflows. Use the `std::string` class and then extract a char* from it if you need it.

Neither char* nor std::string works loads correctly with strings containing spaces, because their extraction operators only loads the first word. Use CLAM::Text which supports multiword strings.

STL compliant containers have XML support if they are declared as `DYN_CONTAINER_ATTRIBUTE`.

When the contained class is a component, then each of the contained objects are stored as elements inside the container element. The fourth macro parameter is for the subitems tag name. So:

```
DYN_CONTAINER_ATTRIBUTE(1, public, std::list<MyComponent>, ComponentList, AComponent);
```

will look like

```
<ComponentList size='10'>
   <AComponent> ...  </AComponent>
   <AComponent> ...  </AComponent>
      ...
   <AComponent> ...  </AComponent>
</ComponentList>
```

And when the contained class is a basic type, all the container items will be stored in a single XML element separated by spaces.

```
DYN_CONTAINER_ATTRIBUTE(1, public, std::vector<double>, LeafList, Ignored);
```

will look like

```
<LeafList size='256'>342.243 2342.252 .... 0.234 0 0</LeafList>
```

Note that in this case the last macro parameter is ignored.

87.3 Customization basics

Let see a sample Dynamic Type class:

```
class ConcreteDT : public CLAM::DynamicType
{
public:
   DYNAMIC_TYPE(ConcreteDT, 5);
   DYN_ATTRIBUTE      (0, public, DummyComponent, MyComponent);
   DYN_ATTRIBUTE      (1, public, Array<Complex>, MyArray);
   DYN_ATTRIBUTE      (2, public, FooDTClass, MyDynType);
   DYN_CONTAINER_ATTRIBUTE(3, public, std::list<int>, MyList);
   DYN_ATTRIBUTE      (4, public, int, MyInt);
   public:
      virtual ~ConcreteDT() {}
   protected:
      void DefaultInit()
      {
         AddMyDyn();
         AddMyA();
         UpdateData();
      }
   // Some non dynamic attributes
   private:
      FooComponent mExtraNonDynamicAttribute;
};
```

This Dynamic Type, as is, will generate default XML. In order to customize it we have to redefine two storage related methods:

```
void MyDyn::StoreOn(Storage & s);
void MyDyn::LoadFrom(Storage & s);
```

When a MyDyn is stored/loaded on/from a Storage, and the Storage detects that it is a component, it calls those functions in order to store/load all meaningful MyDyn subparts if it has any.

So by redefining those functions we will change its XML representation.

87.4 Reordering and skipping

Dynamic Types macros expand some useful methods that allow simplifying the customization.

For each dynamic attribute named XXX, dynamic type macros expand the methods:

```
void ConcreteDT::StoreXXX(Storage & s);
void ConcreteDT::LoadXXX(Storage & s);
```

Using such methods you can easily store/load a concrete dynamic attribute separately. Be careful, **`LoadXXX` requires the attribute XXX to be instantiated** before calling it and it will mark it automatically as removed if the attribute is not present in the XML file. It is important to store the attributes in the same order you load them.

The following example will store and load its attributes in the inverse order to the default one, and skips the third attribute (MyDynType).

```
void ConcreteDT::StoreOn(CLAM::Storage & storage) {
   StoreMyInt(storage);
   StoreMyList(storage);
   // MyDynType is not stored
   StoreMyArray(storage);
   StoreMyDummyComponent(storage);
}

void ConcreteDT::LoadOn(CLAM::Storage & storage) {
   // First of all asure that all attributes are instantiated
   AddAll()
```

```
   UpdateData();
   // Then load them
   LoadMyInt(storage);
   LoadMyList(storage);
   // MyDynType is not loaded
   LoadMyArray(storage);
   LoadMyDummyComponent(storage);
}
```

87.5 Recalling the default implementation

`StoreAllDynAttributes()` and `LoadAllDynamicAttributes()` are another macro expanded methods. They are called from the default StoreOn and LoadFrom implementation. So, by calling them we can reproduce them and it is easy to add non dynamic subparts before or after them or forcing some attributes to be or not present before them. The first step of `LoadAllDynamicAttributes()` is to instantiate all the dynamic attributes that will be marked as erased if they are not in the XML document.

87.6 Adding content not from dynamic attributes

If you simply want to add a non dynamic attribute to the XML representation, you may call those expanded functions and then using a suited XML adapter for the attribute and store it. Refer on how to define the XML format for a normal (non DynamicType) Component to know about those adaptators and how they are used.

The following example stores two extra items on the XML. An existing member of the class (`mExtraNonDynamicAttribute`) and a literal string as an XML attribute (the false value).

```
void ConcreteDT::StoreOn(CLAM::Storage & storage) {
   // Store a temporary object in the first place
   CLAM::XMLAdapter<char*> adapter1("Addedcontent", "Added", false);
   storage.Store(&adapter1);

   // Call the default implementation
   StoreAllDynAttributes();

   // Store a non dynamic attribute member
   CLAM::XMLComponentAdapter adapter2(mExtraNonDynamicAttribute,
       "ExtraNonDynamic", true);
   storage.Store(&adapter2);
}

void ConcreteDT::LoadOn(CLAM::Storage & storage) {
   // std::string is not vulnerable to buffer overflows on loading
   std::string foo; // A temp
   CLAM::XMLAdapter<std::string> adapter1(foo, "Added", false);
   storage.Load(&adapter1);

   LoadAllDynAttributes();

   CLAM::XMLComponentAdapter adapter2(mExtraNonDynamicAttribute,
      , "ExtraNonDynamic", true);
   storage.Load(&adapter2);
}
```

87.7 Storing not as XML elements or changing the tag name

Of course, we can also use Adapters with the dynamic attributes instead of using `StoreXXX` and `LoadXXX`. This is useful to store a dynamic attribute as XML attribute or XML plain content or to change the name from the one the attribute has. Again, refer to the XML developer guide.

When using adapters with dynamic attributes you must take care of some dynamic attributes tasks:

- When storing a dynamic attribute XXX you must check that it is instantiated using the function `HasXXX`.
- When loading you must check that the Storage::Load returns true. When it returns false it is advisable to mark it as removed.

```
void ConcreteDT::StoreOn(CLAM::Storage & storage) {
   StoreMyDummyComponent(storage);
   StoreMyArray(storage);
   StoreMyDynType(storage);
   StoreMyList(storage);

   // MyInt is stored as an attribute (the default is element
   // and with a different name ('Size').

   if (HasMyInt()) {
      CLAM::XMLAdapter<int> adapter(GetMyInt(), "Size", false);
      storage.Store(&adapter);
   }
}
void ConcreteDT::LoadOn(CLAM::Storage & storage) {
   // First of all asure that all attributes are instantiated
   AddAll()
   UpdateData();
   // Then load them
   LoadMyDummyComponent(storage);
   LoadMyArray(storage);
   LoadMyDynType(storage);
   LoadMyList(storage);

   // MyInt is loaded as an attribute (the default is element
   // and with a different name ('Size').

   CLAM::XMLAdapter<int> adapter(GetMyInt(), "Size", false);
   if (!storage.Load(&adapter)) {
      RemoveMyInt();
   }
}
```

87.8 Keeping several alternative XML formats

Normally you will define the storage customization on the same concrete dynamic type class. But sometimes, you want to keep the default implementation o several customized implementations.

A good way of doing this is by subclassing the concrete Dynamic Type and redefining the storage related methods as above but in the subclasses.

XIX Processing Data

88 Basic structural aspects II

A data storage class derives publicly from *ProcessingData*. Thus, it is a concrete Dynamic Type class and must use the DYNAMIC_TYPE_USING_INTERFACE macro.

Ex:

```
class SpectralPeak: public ProcessingData
{
public:
   DYNAMIC_TYPE_USING_INTERFACE (SpectralPeak, 6, ProcessingData);
   DYN_ATTRIBUTE (0, public, EScale, Scale);
   DYN_ATTRIBUTE (1, public, TData, Freq);
   DYN_ATTRIBUTE (2, public, TData, Mag);
   DYN_ATTRIBUTE (3, public, TData, BinPos);
   DYN_ATTRIBUTE (4, public, TData, Phase);
   DYN_ATTRIBUTE (5, public, int, BinWidth);
  (.)
```

Remember that all attributes registered using the DYN_ATTRIBUTE macro are granted associated Getters and Setters.

89 Constructors and initializers

Apart from the default constructor (already available as a result of the Dynamic Types macros), other constructors may be implemented. All these constructors must call the constructor of the *Processing Data* base class using the member initialisation syntax and passing the number of Dynamic Attributes as parameter.

Ex:

```
Segment::Segment(const SegmentConfig &newConfig):ProcessingData(6)
{
(.)
```

Apart from that, these constructors must call a macro-derived method called `MandatoryInit(),` which is in charge of initialising concrete Dynamic Type's internal structure.

Another initializer that is often useful is the `DefaultInit()` method. This method has to be implemented by the developer and is in charge of initializing default attributes and values. This method is automatically called from the Processing Data's default constructor and may also be called from all other constructors.

The most usual non-default constructors that a Processing Data class is bound to have are the Copy constructor and the Configuration constructor. The former is already implemented in the Processing Data base class and this implementation is sufficient as long as all attributes of the concrete class are Dynamic and require no initialisation. If not (for example if the class has a non Dynamic member), the developer may make use of the `CopyInit()` method. This method has to be implemented by hand, but is automatically called from the macro derived Copy constructor.

Ex:

The Segment PD class has a non-dynamic member called pParent. Thus, the copy initializer is implemented as:

```
void Segment::CopyInit(const Segment& prototype)
{
   pParent=prototype.pParent;
}
```

The configuration constructor is sometimes desirable for constructing a Processing Data out of its associated configuration object or out of some sort of initial value (flags, size.). In this case the constructor must explicitly call the `MandatoryInit()` method and then call any other necessary configuration methods.

Ex:

```
Spectrum(const SpectrumConfig &newConfig) : ProcessingData(12)
{
   MandatoryInit();
   Configure(newConfig);
}
```

90 Private members with public interface

In many cases, it is desirable to keep some members private and offer an accessory public interface for modifying their value. This is especially so if the member is related to some structural aspect of the processing data and a modification in its value implies a re-structuring of the object itself. Imagine, for example a `Size` attribute that is related to the size of the buffers in a PD class. If the user modifies this attribute, we want to keep the size of the buffers consistent.

In that case, it is recommended you keep the member as private and declare it adding the prefix ‘pr’ before its common name.

Ex:

```
class Spectrum : public ProcessingData
{
public:
   DYNAMIC_TYPE_USING_INTERFACE (Spectrum, 12, ProcessingData);
   ...
private:
   DYNAMIC_ATTRIBUTE (2, private, int , prSize);
   ...
```

Then you need to offer the public interface implementing by hand the appropriate Setter and Getter.

Ex:

In the previous example, that of the spectrum, the public interface is offered through the GetSize() and SetSize() methods, implemented as follows (see how consistency is always kept between prSize member and size of all existing buffers:

```
int Spectrum::GetSize() const
{
   int size= GetprSize();
   CLAM_BEGIN_CHECK
   if(HasMagBuffer() && GetMagBuffer().Size())
      CLAM_ASSERT(GetMagBuffer().Size() == size,
         "Spectrum::GetSize(): Mag size and Size mismatch.");
   if(HasPhaseBuffer() && GetPhaseBuffer().Size())
      CLAM_ASSERT(GetPhaseBuffer().Size() == size,
         "Spectrum::GetSize(): Phase size and Size mismatch.");
   if(HasComplexArray() && GetComplexArray().Size())
      CLAM_ASSERT(GetComplexArray().Size() == size,
         "Spectrum::GetSize(): Complex size and Size mismatch.");
   if(HasPolarArray() && GetPolarArray().Size())
      CLAM_ASSERT(GetPolarArray().Size() == size,
```

```
            "Spectrum::GetSize(): Polar size and Size mismatch.");
      if (HasprBPFSize()) {
         if(HasMagBPF() && GetMagBPF().Size())
            CLAM_ASSERT(GetMagBPF().Size() == size,
               "Spectrum::GetSize(): MagBPF size and Size mismatch.");
         if(HasPhaseBPF() && GetPhaseBPF().Size())
            CLAM_ASSERT(GetPhaseBPF().Size() == size,
               "Spectrum::GetSize():PhaseBPF size and Size mismatch.");
      }
      CLAM_END_CHECK
      return size;
   }

   void Spectrum::SetSize(int newSize)
   {
      SetprSize(newSize);
      if(HasMagBuffer()) {
         GetMagBuffer().Resize(newSize);
         GetMagBuffer().SetSize(newSize); }
      if(HasPhaseBuffer()) {
         GetPhaseBuffer().Resize(newSize);
         GetPhaseBuffer().SetSize(newSize); }
      if (HasPolarArray()) {
         GetPolarArray().Resize(newSize);
         GetPolarArray().SetSize(newSize); }
      if (HasComplexArray()) {
         GetComplexArray().Resize(newSize);
         GetComplexArray().SetSize(newSize); }
      if (!HasprBPFSize()) {
         if (HasMagBPF()) {
            GetMagBPF().Resize(newSize);
            GetMagBPF().SetSize(newSize); }
         if (HasPhaseBPF()) {
            GetPhaseBPF().Resize(newSize);
            GetPhaseBPF().SetSize(newSize); }
      }
   }
```

91 Configurations

PD classes may also use associated configuration classes in a similar way to the Processing Objects. As it has been shown up until now, both informative and structural attributes of a PD class can be added as regular attributes. The only need for offering an associated configuration class is if a PD class has too many of these attribute to handle them one by one and it becomes more ’friendly’ to use a configuration wrapper. As a rule of thumb, you can say that if more than one non-default constructor needs to be implemented for a PD class you should start thinking of implementing an associated Configuration. Thus, configurations are only seen as initialization shorthands and should therefore never be stored inside a PD class.

Configuration classes derive from the *ProcessingDataClass*.

Ex:

```
class SpectrumConfig : public ProcessingDataConfig
{
public:
   DYN_CLASS_TABLE_USING_INTERFACE(SpectrumConfig, 5, ProcessingDataConfig);
   DYN_ATTRIBUTE (0, public, EScale, Scale);
   DYN_ATTRIBUTE (1, public, TData, SpectralRange);
   DYN_ATTRIBUTE (2, public, int, Size);
   DYN_ATTRIBUTE (3, public, SpecTypeFlags, Type);
```

```
    DYN_ATTRIBUTE (4, public, int, BPFSize);
    protected:
    void DefaultInit();
    void DefaultValues();
};
```

92 Customizing XML output

If you want to have a specific XML output that does not exactly match the one automatically derived and mentioned in 43 you have two options:

1) If the class you are dumping is not part of the CLAM core you may override the StoreOn() method and implement it by hand.

2) Otherwise you are suggested to write another PD class that fits your needs and acts as an adapter of the real data you have on the existing PD class and the desired output.

93 Specific attributes: flags and enums

When implementing a PD class it is common that you need to implement an associated enumeration or flag. If you use a standard C++ enum or std::bitset you will get no more than a meaningless integer representation. CLAM::Enum and CLAM::Flags<N> will give you a proper string representation. See, Doxygen documentation for these base classes.

It is far beyond the scope of this document to go into the implementation details of both classes and the developer is suggested to implement his own Enum or Flag class departing from an existing class. For the case of Enums, let’s see the EInterpolation class:

Ex:

The following code is included in the .hxx file:

```
class EInterpolation: public Enum
{
public:
    static tEnumValue sEnumValues[];
    static tValue sDefault;
    EInterpolation() : Enum(sEnumValues, sDefault) {}
    EInterpolation(tValue v) : Enum(sEnumValues, v) {};
    EInterpolation(std::string s) : Enum(sEnumValues, s) {};

    typedef enum {
        eStep,
        eRound,
        eLinear,
        eSpline,
        ePolynomial2,
        ePolynomial3,
        ePolynomial4,
        ePolynomial5,
        ePolynomialn
    } tEnum;

    virtual Component* Species() const
    {
        return (Component*) new EInterpolation;
    };
};
```

and the following in the .cxx:

```
Enum::tEnumValue EInterpolation::sEnumValues[] = {
   {EInterpolation::eStep,"Step"},
   {EInterpolation::eRound,"Round"},
   {EInterpolation::eLinear,"Linear"},
   {EInterpolation::eSpline,"Spline"},
   {EInterpolation::ePolynomial2,"2ond_order_Polynomial"},
   {EInterpolation::ePolynomial3,"3rd_order_Polynomial"},
   {EInterpolation::ePolynomial4,"4th_order_Polynomial"},
   {EInterpolation::ePolynomial5,"5th_order_Polynomial"},
   {EInterpolation::ePolynomialn,"nth_order_Polynomial"},
   {0,NULL}
};

Enum::tValue EInterpolation::sDefault = EInterpolation::eLinear;
```

Deprecated: since the `flags` application is not available. Meanwhile, see the doxygen documentation for an example.

For the case of flags, the easiest way of writing your own class is making use of the flag generation script available at mtg150.upf.es/flags.

XX XML

This section is intended to explain underlaying mechanisms that are useful to know when serializing XML of non-DynamicType Components. DynamicTypes are components that have already a default XML implementation. If you don't like that default DynamicType implementation them you should try the extension mechanisms explained also on the DynamicType chapter. When those extensions are not enough, then you should try this.

94 Components and XML

Any Component must implement two virtual methods related to storage stuff:

```
void MyComponent::StoreOn(Storage & s) const;void MyComponent::LoadFrom(Storage & s);
```

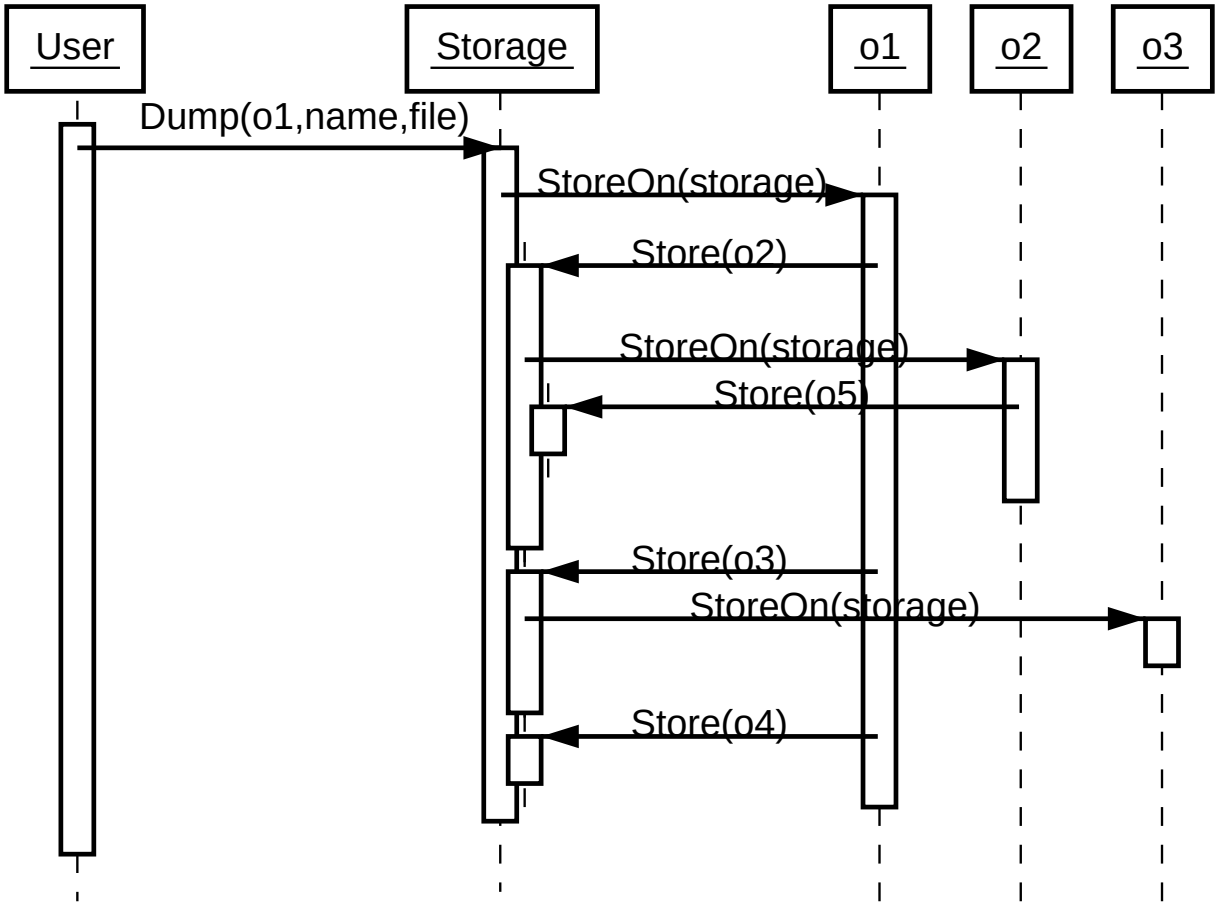
Those methods are suposed to store and load onto/from the storage all the inner members by the component.

When is this StoreOn/LoadFrom method used? When a component is stored/loaded onto/from an storage, the storage detects that this object is also a Component and then calls StoreOn/LoadFrom method in order to allow the component to also store its childs.

Imagine the following scenario:

- o1, Component containing o2, o3 and o4
- o2, Component containing o5
- o3, Component containing no subcomponent
- o4, Non Component
- o5, Non Component

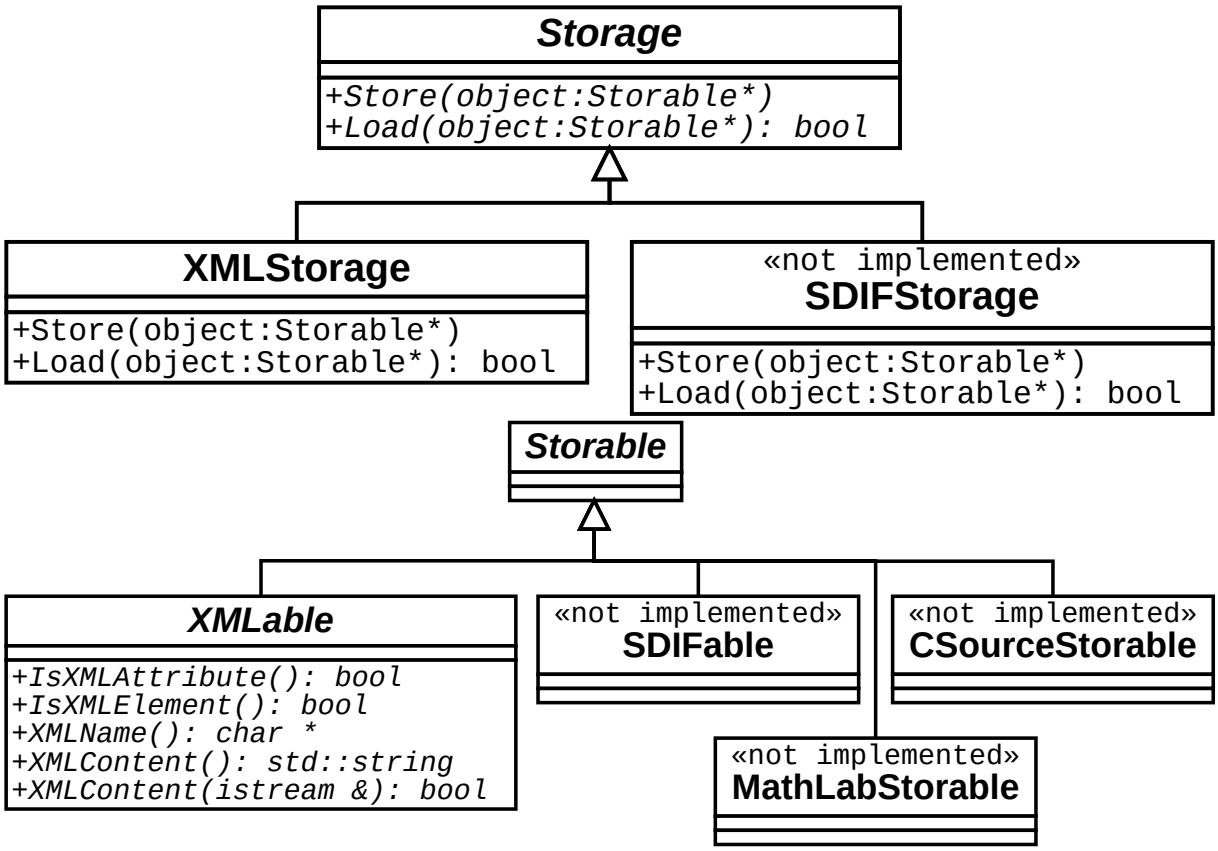
The process of storing o1 on a Storage could be the one depicted here:



The load process is the same. You can check the return value for the Storage::Load function in order to know if the object was present or not in the Storage.

95 XML Adapters

Both methods, `Storage::Load` and `Storage::Store` receive an `MTG::Storable` as parameter. Moreover, this parameter will be ignored if it doesn't fulfills the concrete Storable interface for the format of the concrete Storage interface. For the MTG::XMLStorage, the Storable interface is MTG::XMLable.



Just in order you don't have to derive every object from that interface, a pool of adapters are available in order to make your objects fullfill the MTG::XMLable interface.

In short, your implementation of StoreOn and LoadFrom methods will consist on selecting the suited XML adapter to wrap each subpart of your Component and call Storage::Store or Storage::Load having it as parameter.

As a general rule, adapters take several parameters. Former parameters describe the adaptee (for example a reference to it), and later parameters adds the format dependent information.

For XML, the format dependent information is the name and a boolean.

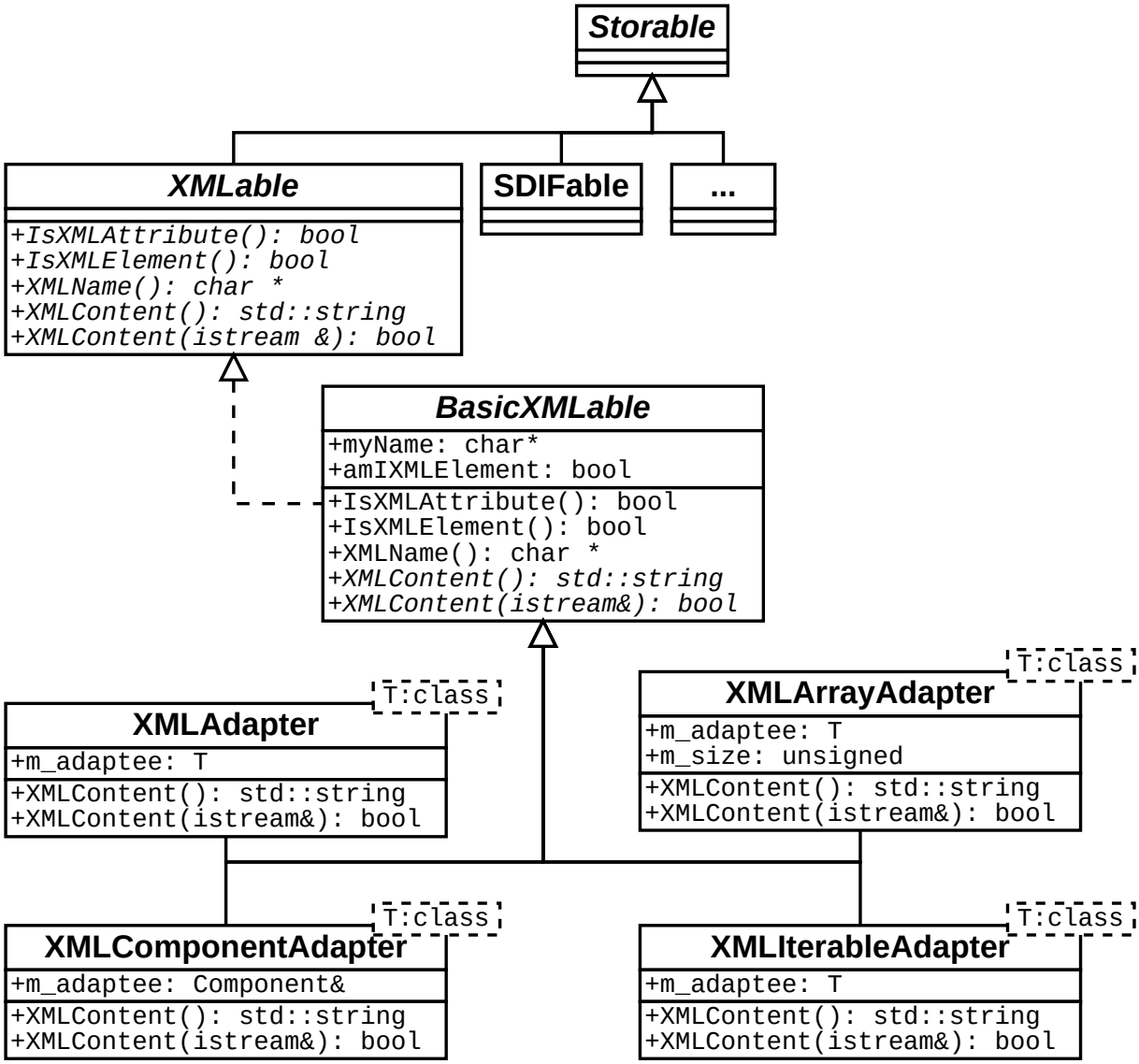
- If the name is a NULL pointer, the adapters is considered plain text content. (Both, attributes and elements, must have names)
- If not, the boolean determines whether is an element or not (thus, it is an attribute)

Those parameters have default values to NULL and false respectively so a tipical Adapter constructor definition would be:

```
MyAdapterClass( /* Here goes the adaptee info */ , char * name = NULL, bool beElement = false );
```

Important: The adapter only copies the pointer to the the null-terminated string, not a copy of it. So it is dangerous to delete or modify this string until the adapter has been stored on the Storage.

The whole set of adapter classes is shown in the following UML class diagram:



95.1 Simple types adapters

We consider a simple type that one that implements the insertion operator (<<) and the extraction operator (>>) to a C++ ostream. Most basic C types define it (int, float, char, char* ...).

Also user classes and structures can define its own insertion operator. Because simple type adapters are based on such operator to extract the XML content, they are useable with a large amount of objects not being MTG specific.

You also need to instantiate a CLAM::TypeInfo<T> for this type, having `StoreAsLeaf` typedef defined as `StaticTrue`. You can simply instantiate this using the macro statement:

```
CLAM_TYPEINFOGROUP(CLAM::BasicCTypeInfo, YourClass)
```

For any 'simple type' in this sense, you may adapt it with a `XMLAdapter<T>` where T is the type.

```
class MyComponent : public CLAM::Component {
   int i;
   double d;
   char c;
   std::string s;
public:
```

```
    MyComponent()
    {
       i = 3;
       d = 3.5;
       c = 'a';
       s = "Hello";
    }

    void StoreOn(CLAM::Storage & storer) const
    {
       // Storing an integer as plain content
       XMLAdapter<int> intAdapter(i);
       storer.Store(intAdapter);
       // Storing a double as an attribute
       XMLAdapter<double> doubleAdapter(d, "myDouble");
       storer.Store(doubleAdapter);

       // Storing a char as a element
       XMLAdapter<char> charAdapter(c, "myChar", true);
       storer.Store(charAdapter);

       // Storing an standard library string as an attribute
       XMLAdapter<std::string>(s, "myString");
       storer.Store(strAdapter);
    }
    void LoadFrom(CLAM::Storage & storer)
    {
       // Storing an integer as plain content
       XMLAdapter<int> intAdapter(i);
       storer.Load(intAdapter);

       // Storing a double as an attribute
       XMLAdapter<double> doubleAdapter(d, "myDouble");
       storer.Load(doubleAdapter);

       // Storing a char as a element
       XMLAdapter<char> charAdapter(c, "myChar", true);
       storer.Load(charAdapter);

       // Storing an standard library string as an attribute
       XMLAdapter<std::string>(s, "myString");
       storer.Load(strAdapter);
    }
};
```

The result of storing this component would be:

```
<Document myChar='a' myString='Hello'>
   3
<myDouble>3.5</myDouble>
</Document>
```

Notes:

- We are using std::string and not a char pointer. Char pointer is limited to having a buffer limit that can be surpassed when loading an arbitrary content. std::string is safe.
- std::string and char* extraction operator reads only a word while insertion operator writes every word it finds in the string. In order to be loaded properly, string cannot contain any space in C sense (spaces, tabs, returns...)
- If your string must contain such space caracters, you can use the CLAM::Text class instead

std::string which feeds all the available content incluiding spaces.

95.2 Simple type C array adapters

This adapter (XMLArrayAdapter) adapts dinamically an array of simple type objects. The content is generated concatenating of the individual objects separated by an space.

The XML array adapter constructor is declared this way:

```
XMLArrayAdapter(T* adaptee, unsigned size, char * name = NULL, bool beElement = false );
```

See the following example of the XMLArrayAdapter usage:

```
// Supose that MyComponent has as fields mSize and mBuffer
void MyComponent::StoreOn(CLAM::Storage & storer) const
{
   XMLAdapter<int> sizeAdapter(mSize, "Size", false );
   storer.Store(intAdapter);

   XMLArrayAdapter<double> doubleAdapter(mBuffer, mSize, "Buffer", true);
   storer.Store(doubleAdapter);
}

void MyComponent::LoadFrom(CLAM::Storage & storer) {

   XMLAdapter<int> sizeAdapter(mSize, "Size", false );
   storer.Load(intAdapter);

   if (mBuffer) delete [] mBuffer;
   mBuffer = new double[mSize];

   XMLArrayAdapter<double> doubleAdapter(mBuffer, mSize, "Buffer", true);
   storer.Load(doubleAdapter);
}
```

95.3 Component adapters

Previous adapters only adapts XML data with trivial structure. In order to have a more structured data, Components are needed. Although you can adapt a Component with a XML adapter, the XMLStorage only sees that the adapter is not a Component and then it doesn’t look for subitems.

A XMLComponentAdapter is an adapter that it is itself a Component and adapts a Component. So, when the storage confirms that the Adapter is a Component and executes the StoreOn method, the adapter forward the calling to its adaptee.

```
if object is not an XMLable
  return;

if object->IsXMLElement() {
  store it as element
  if object is a Component
    object.StoreOn(*this)
  }
else if object->IsXMLAttribute()
  store it as attribute
else {
  store it as content
  if object is a Component
    object.StoreOn(*this)
  }
```

`XMLComponentAdapter` usage is very similar to other adapters usage. You may obtain very variated behaviours:

When you store a component **as element**, element tags with the name are written and all its **subitems will be placed inside**.

```
<theParent aSibblingAsAtrib="aSibbling">   <aSibbling>sibbling</aSibbling>   <theComponent aSubitemAsAtrib="subitem">      theContent      subitemAsPlainContent      <aSubitemAsElem>subitem</aSubitemA
```

When you store a component **as plain content**, its subitems will be placed **on the same level** than the component would be.

```
<theParent aSibblingAsAtrib="aSibbling" aSubitemAsAtrib="subitem">   <aSibbling>sibbling</aSibbling>   theContent   subitemAsPlainContent   <aSubitemAsElem>subitem</aSubitemAsElem></theParent>
```

If you store a component **as an XML attribute** the XMLStorage **will not recurse** to subitems as it does with elements and plain content.

```
<theParent aSibblingAsAtrib="aSibbling" theElement="theContent">   <aSibbling>sibbling</aSibbling></theParent>
```

95.4 Loading Considerations

Keep in mind those considerations related on how to make things deserialized back to the system:

- If you use the insertion operator provide a complementary extraction operator that would not consume extra input and test that it works with generated output.
- Extraction operator must manage stream error flags.
- Strings with spaces: Don't store strings containing spaces. When parsing them, the standard C++ library >> operator gets only the first word: Use the CLAM::Text class instead!
- CLAM::Text loading will consume all the contiguous content left, so, if you store it as a XML content, don't put XML content sibblings afterwards.
- Contiguous plain content: Several contiguous plain content nodes are separated with spaces.
- If you need them, you can give hints from the StoreOn to the LoadFrom, for example, inserting in the XML the number of elements for a variable length collection of subitems, telling which subitems are instanciated or no or not.

XXI C pre-processor macros defined and used by CLAM sources

We define a set of C preprocessor macros that pollute the global namespace, so we document them here explicitly, for minimizing the possibility of macro names clashes. Some of them are global flags that affect the entire library behaviour, some other are part of the DynamicType API, a few needed for ensuring cross-platformness (especially math constants) and some macros that give support to *Defensive programming* techniques. Besides, we also rely on some macros to be defined by the compiler or development environment, to select a particular behaviour for a given platform.

96 Global flags

TODO Should be moved

You can define these macros in the project (VisualC++) or makefile (GNU) in order to change CLAM behaviour.

- `CLAM_EXTRA_CHECKS_ON_DYNAMIC_TYPES`
  Define it to perform paranoid very very slow checks on dynamic types.
- `CLAM_USE_STL_ARRAY`
  Define it to use the CLAM::Array implementation based on the STL std::vector

97 Cross-platformness macros

These macros can be used inside user code.

- `PI`
  PI number in double precission
- `TWO_PI`
  Two times PI number in double precission

98 Dynamic Types Macros

All these macros are more accurately documented in Dynamic Types reference.

- `DYNAMIC_TYPE(newClass, nAttributes)`
  Insert it inside a DynamicType subclass definition in order to generate some macro expanded functions.
- `DYNAMIC_TYPE_USING_INTERFACE(newClass, nAttributes, subClass)`
  Insert it inside a DynamicType descendant class definition in order to generate some macro expanded functions.
- `DYN_ATTRIBUTE(order, access, type, name)`
  Inside a dynamic type class definition, it defines a dynamic attribute. See Dynamic Types.
- `DYN_CONTAINER_ATTRIBUTE(order, access, type, name,elemName)`
  Inside a dynamic type class definition, it defines a dynamic attribute with STL container like interface. See Dynamic Types.
- `CLAM_TYPE_INFO_GROUP(group, class)`
  Defines the class CLAM type information like the one defined in the group. (ie. BasicCTypeInfo, ContainerTypeInfo...) See DynamicType-XML

99 Defensive programming macros

Although some may consider an oxymoron to use C preprocessor macros for defensive programming they are very useful for giving users enough flexibility to enable or disable code auto-checking:

- `CLAM_ASSERT(expressionToBeTrue, messageWhenFalse)`
  A normal CLAM assert. See Error handling.
- `CLAM_BEGIN_CHECK`
  Marks the start of check related code. See Error handling.
- `CLAM_END_CHECK`
  Marks the end of check related code. See Error handling.
- `CLAM_DEBUG_ASSERT(expressionToBeTrue,messageWhenFalse)`
  A debug only CLAM assert. See Error handling.
- `CLAM_DEBUG_BEGIN_CHECK`
  Marks the start of debug only check related code. See Error handling.
- `CLAM_DEBUG_END_CHECK`
  Marks the end of debug only check related code. See Error handling.
- `CLAM_BREAKPOINT`
  Sets a compiler independent execution breakpoint.

100 preinclude.hxx Macros

These macros deal with system dependant issues and are defined (or not) in the `src/Defines/{platform name}/preinclude.hxx` file. This file is generated by `GNU autoconf` on UNIX platforms, and is hardcoded for specific platforms such as Microsoft Windows.

- `HAVE_NON_COMPLIANT_STANDARD_LIBRARY`
- `HAVE_STANDARD_VECTOR_AT`
- `HAVE_STANDARD_SSTREAM`
- `HAVE_STANDARD_SSTREAM_STR`

- CLAM_BIG_ENDIAN
- CLAM_LITTLE_ENDIAN

101 Platform dependant macros

CLAM code expects these macros to be defined by the compiler in order to identify the platform and the compiler:

- _MSC_VER
  This macro is defined by Microsoft Compilers, and identifies the major and minor version of the compiler being used.
- __MWERKS__
  This macro is defined by Metrowerks Codewarrior family of compilers
- __GNUC__
  This macro is defined by all recent versions of the GNU C/C++ Compiler
- WIN32
  This macro is defined by default by Microsoft Visual Studio IDE. If your IDE is not defining it by default, and you are developing on Microsoft Windows you should hack your makefiles (or similar) since it is vital for some CLAM cross-platform functionalities to work properly.
- macintosh
  When this macro is defined CLAM code assumes that the current platform is some MacOS version.
- POSIX
  The generated binary is for a POSIX compatible operating system (i.e. headers such as unistd.h are expected to be found).

102 Private Macros

Those macros are not intended for user use. They are documented here only for completeness.

- CLAM_NUMERIC_ARRAY_INITIALIZATIONS(type)
- CLAM_FAST_ARRAY_SPECIALIZATIONS(type)
- CLAM_DB_SCALING
- CLAM_ABORT(message)
- __COMMON_DYNAMIC_TYPE(CLASS_NAME,N)
- __COMMON_DYN_ATTRIBUTE(N,ACCESS,TYPE,NAME)

CLAM SAMPLE APPLICATIONS

XXII Introduction

CLAM Sample applications can be found in the /examples folder. This directory includes large applications and smaller usage examples. In this section we will focus in the former.

The next sections explain the main functionality and structure of these applications. But as a first overview here is a brief description of each one:

103 SMS Example

Functionality: This application performs a spectral analysis of an input sound following the SMS model (http://www.iua.upf.es/~sms). The analysis can then be transformed and synthesized back.

Focus on this example if: (1) Your main interest in CLAM is related to Spectral Analysis or Transformations; (2) You have previously used SMSTools or SMSCommandLine and you are looking for a similar tool; (3) You would like to see an off-realtime signal processing application; (4) You are wandering what XML is used for in CLAM; (5) You are interested in content-analysis or MPEG7.

104 SALTO

Functionality: This is a real-time sax and trumpet synthesizer based on a spectral model. It can take in MIDI data (both from a keyboard and a wind controller) and XML Melody Data.

Focus on this example if: (1) You are interested in real-time synthesis applications; (2) Your main interest is in spectral models for synthesis; (3) You want an example of MIDI usage in CLAM; (4) You are into multithreading handling; (5) You play saxophone or trumpet and happen to have a wind-controller; (6) You just want to use a MIDI-controlled synthesizer and play some music.

105 Spectral Delay

Functionality: Real-time audio effect that splits the input sound into three different bands and applies a delay to each of them.

Focus on this example if: (1) You are into real-time processing; (2) You want to see an easy-to-understand application of spectral processing; (3) You want to see an example with real-time GUI interaction.

106 Rappid

Functionality: Real-time amplitude modulation between two input sounds.

Focus on this example: If you are interested in robust, real-time applications in Linux.

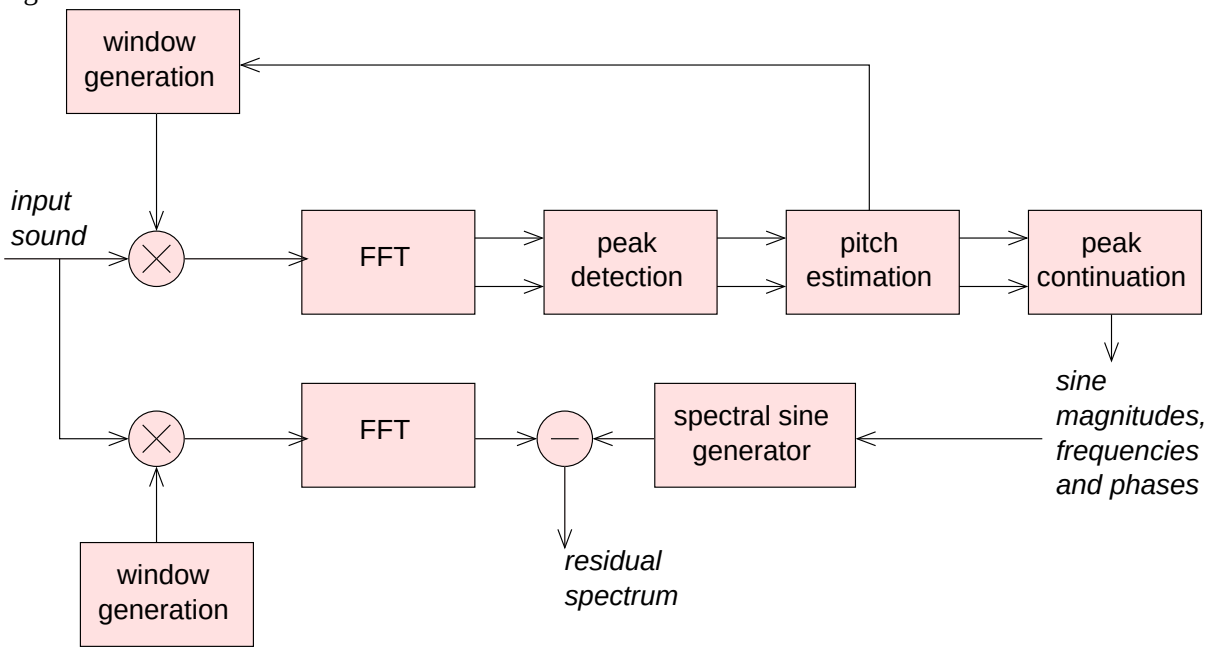
XXIII SMS Example

(Found at /examples/SMS)

The MTG's flagship applications have been the SMSCommandLine and SMSTools, developed 5 years ago but currently discontinued. As a matter of fact, one of the main goals when starting CLAM was to develop the substitute for those applications.

The example you will find in this first release is a beta version and efforts are being put forward to include a more ambitious tool soon.

The application analyzes, transforms and synthesizes back a given sound. For doing so, it uses the Sinusoidal plus Residual model (usually referred to as SMS). Next picture depicts the analysis algorithm:



If you need more information about the signal processing details involved in the process you may have to navigate through the MTG recommended literature (at http://www.iua.upf.es/mtg) or visit the SMS homepage at http://www.iua.upf.es/~sms.

107 Introduction

The application has three different versions: SMSTools - which has a FLTK graphical user interface-, SMSConsole- which is a command-line based version-, and SMSBatch - which can be used for batch processing a whole directory. All you have to do to compile one or the other is to compile the file SMSTools.cxx (if you want GUI), SMS.cxx (if you want the Command Line versrion) or SMSBatch (if you want to batch process). It is strongly recommended that you start with the graphical SMSTools version and only use the others only if you have more specific requirements. The rest of this document will suppose that you are using the graphical version and only mention some differences with the other versions where strictly necessary.

The application has a number of possible different inputs:

1. A configuration xml file. This configuration file is used to configure both the analysis and synthesis processes.
2. An analysis xml file. This file will be the result of a previously performed and stored analysis. The xml parser (xerces) features include a rather slow parsing/storing of documents. On the other hand,

xml is a text-based format and thus you can expect a 1 MB sound file become 10 MB of xml analysis data. For all those reasons the storing/loading of analysis data, although fully working, is not recommended unless you want to have a textual/readable representation of your analysis result, else you will be better off using the SDIF format (see next paragraph).

OR

An analysis SDIF file. This file will be the result of a previously performed and stored analysis. This format is as complete as the XML passivation, but it is dramatically more compact (i.e. smaller files), since data is not stored in a "human readable" format . See XII for more details about SDIF file format.

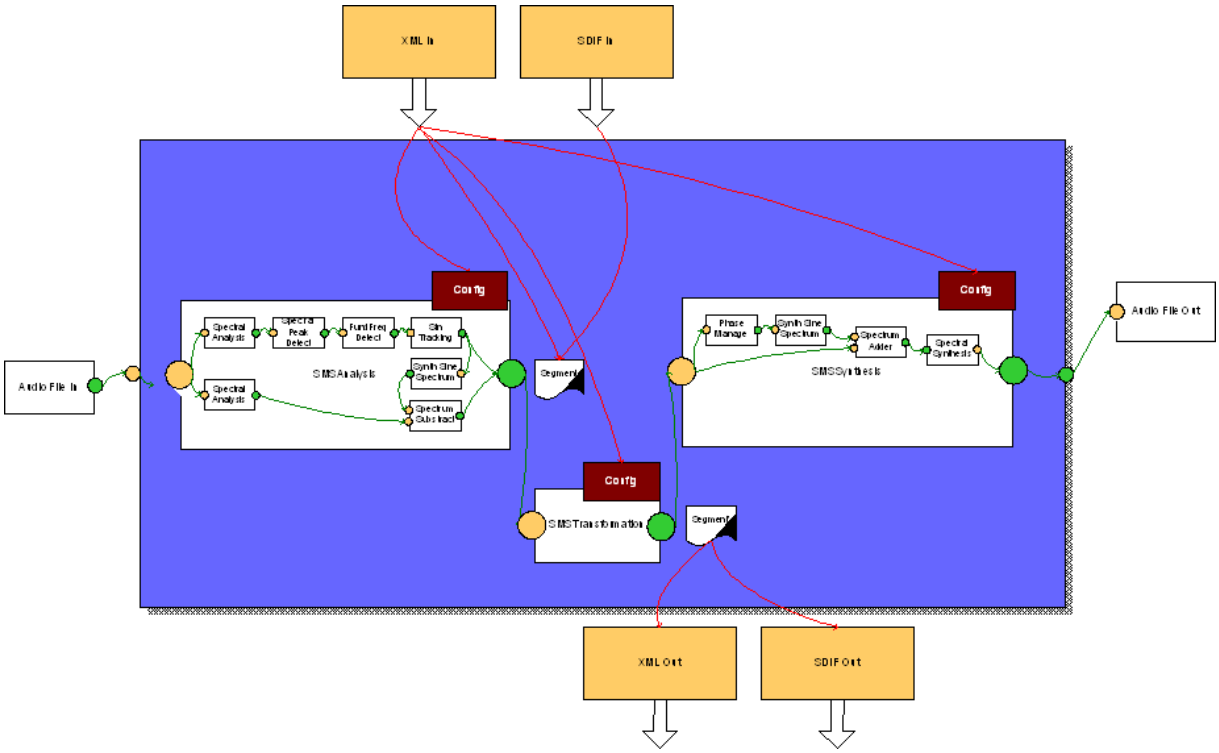
3. A Transformation score in xml format. This file includes a list of all transformations that might be applied to the result of the analysis and the configuration for each of the transformations.

Note that all of them can be selected and generated on run-time from the user interface if you use the SMSTools version.

From these inputs, the application is able to generate the following outputs:

1. Analysis data xml or sdif file.
2. Melody xml file.
3. Output sound: global sound, sinusoidal component, residual component

The following picture illustrates the main blocks of the application:



108 Building the application

If you got a binary precompiled for your platform you may skip this section. Else, if you are building the application from the CLAM source itself, here you will find the necessary information

The example requires the following external libraries: Xercesc 1.7.0, for the XML support, FLTK 1.1.4, for the graphical user interface, the fftw v.2.1.3 library, and either Mesa (3.4.2 or higher) library or vendor specific OpenGL binaries, for the flashy graphics. You should have them installed and fully working on your system before trying to build the application from the library sources.

For building the application:

- On **GNU Linuxes**

First follow the instructions found in the manual sections "Deploying CLAM in your system" and "CLAM Build System Documentation", to have both deployed the library and set up the build system. Once this is accomplished, get into SMS Tools build folder, *<CLAM SOURCE DIR>/build/Examples/SMS/Tools*, and issue the following command

```
$ make CONFIG=release
```

this will trigger SMS Tools building. If everything goes smooth, you will obtain the SMS Tools application binary in the same folder you issued the *make CONFIG=release* command called *SMSTools*.

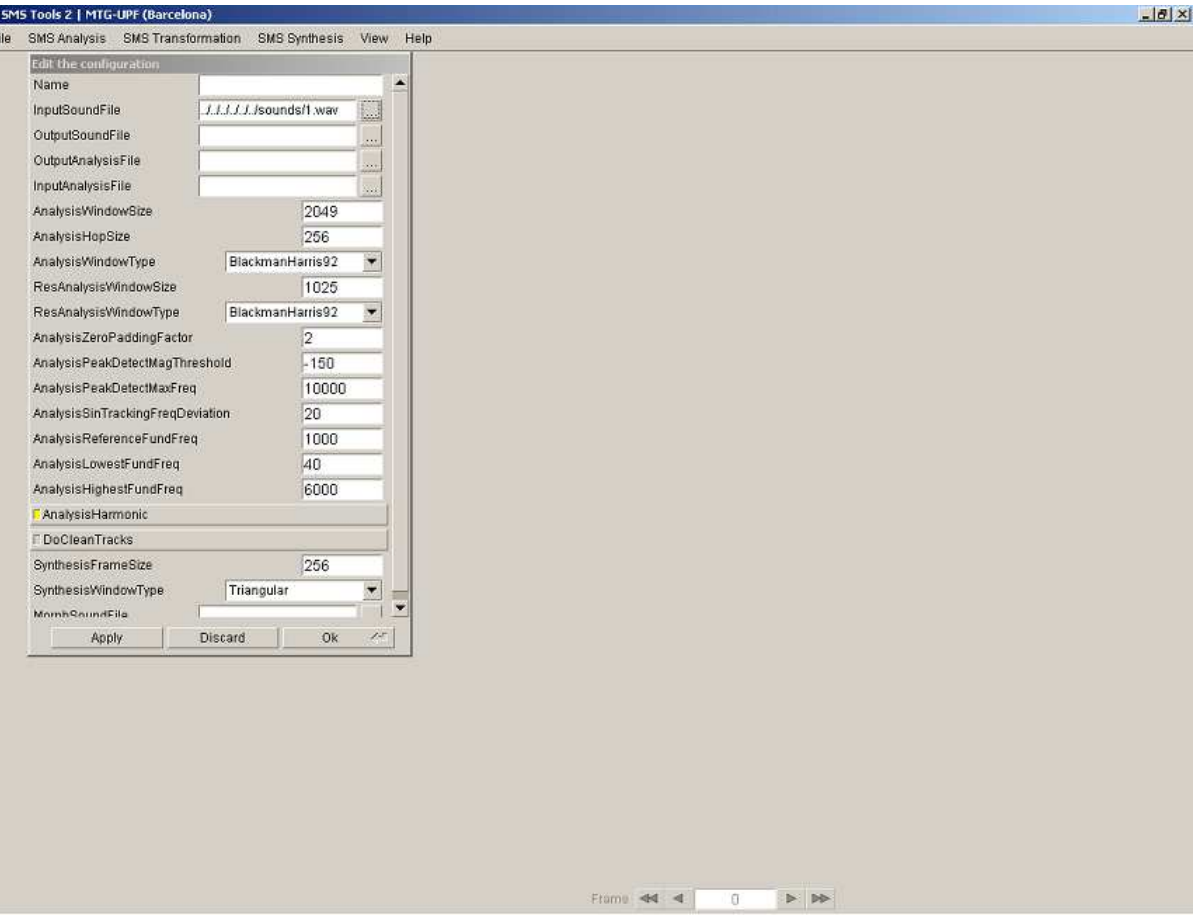
- On **Microsoft Windows**

at *<clam sources dir>/build/Examples/SMS/Tools* folder you will find a Microsoft Visual C++ 6.0 project file (.dsp) already configured for building the application.

109 An SMSTools walkthrough

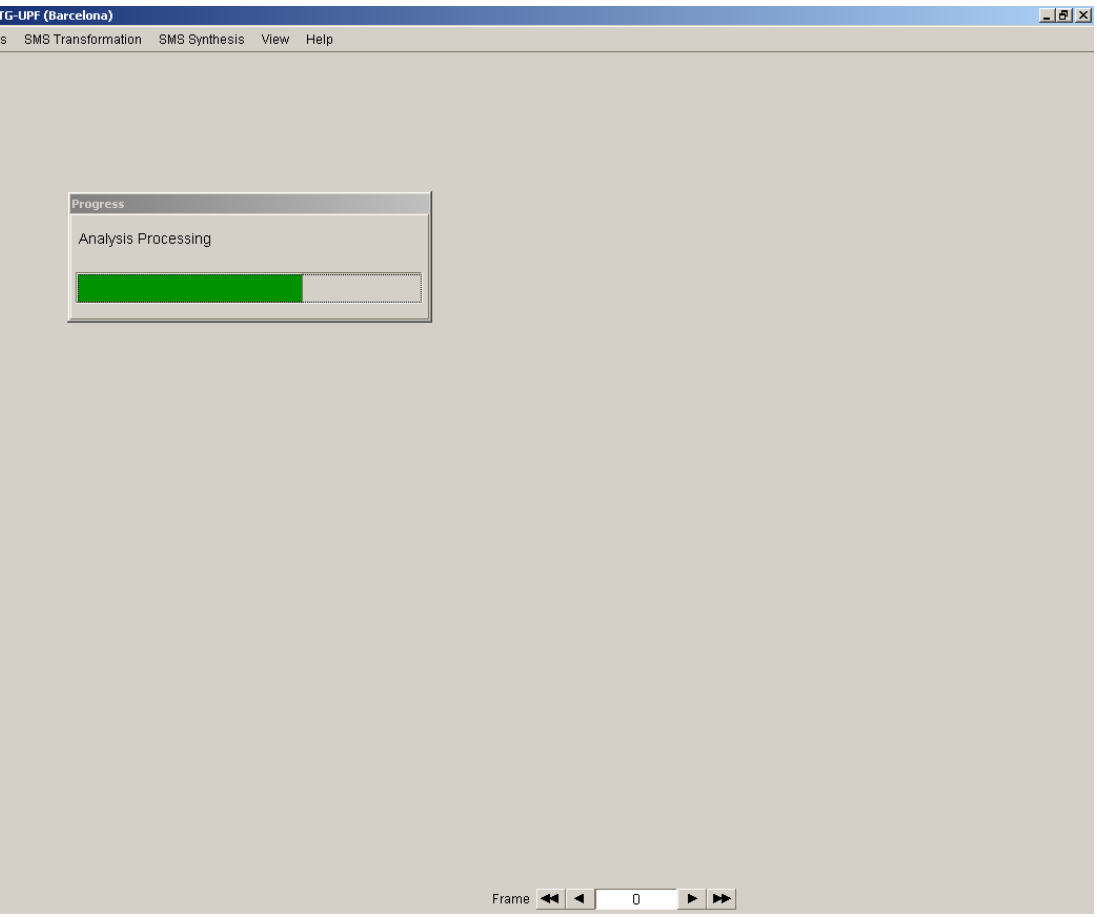
[i1] Once you have built the Application you may follow the steps below to both get a first impression of the application capabilities and check that everything is right:

1. First you have to configure the workspace. For doing so you have two options: (1) Load an xml configuration file (like the one located in *<clam sources dir>/build/Examples/SMS*) or (2) directly edit the default configuration. Let's consider this second option. Go to File->Configuration->Edit. There are many fields you can edit and modify (see Section on Configuration File) but for the sake of simplycity, let us just select an input sound file (.wav, .aiff or .raw) by clicking on the button next to the InputSoundFile field. Your interface should look something like:



(Click to enlarge)

2. If the sound file name you entered is correct, we are now in the position to continue and analyze the sound. Click on SMSAnalysis->Analyze and wait for the progress bar to finish.



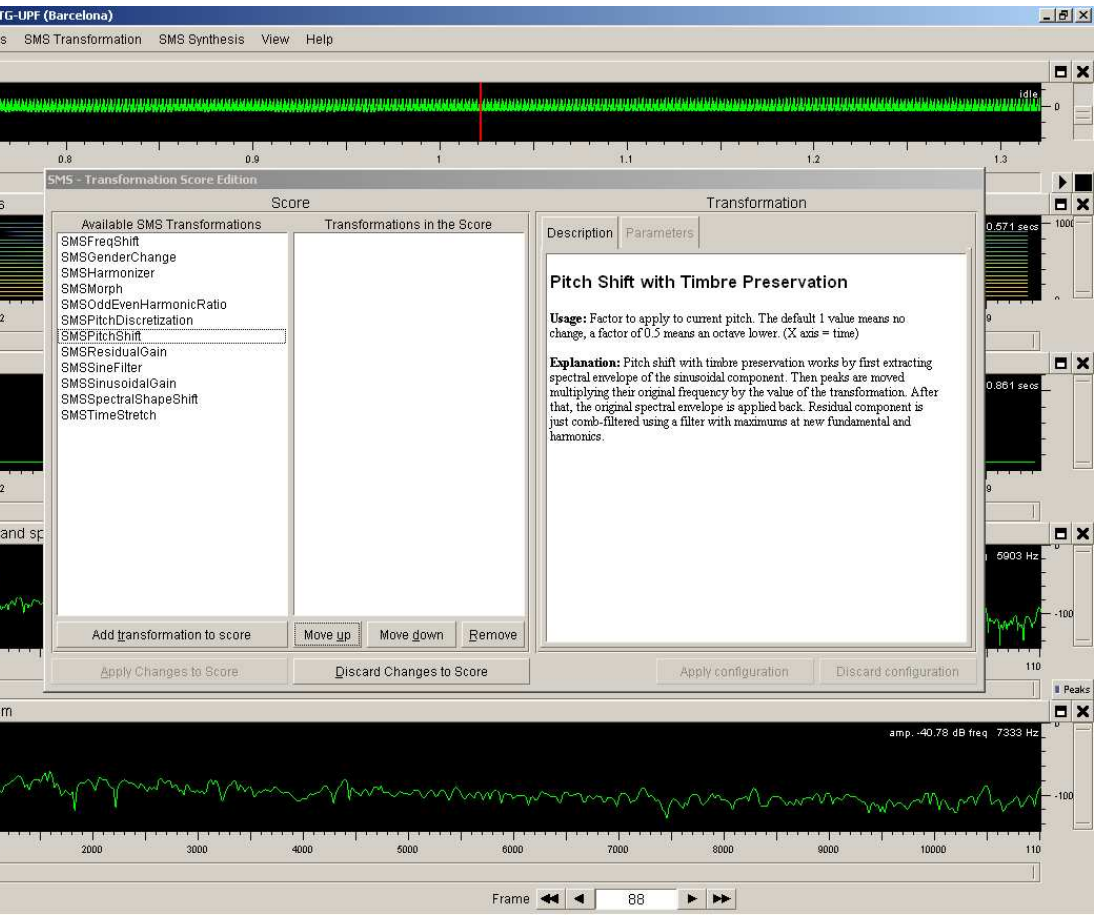
(Click to enlarge)

3. Once the sound is analyzed, we are ready to look at the analysis results. In the View menu you will see active all the available data: original sound, sinusoidal tracks, fundamental frequency, and frame related data (sinusoidal spectrum and peaks and residual spectrum). When you select a frame related view, you can browse through the different frames by clicking on any of the non-frame views or by using the frame browser at the bottom of the interface. You can also listen to the sound by clicking on the play button on the bottom-right of the wave display. Note that the sliders on any view act like a zoom control when clicking on any of the extremes.

(Click to enlarge)

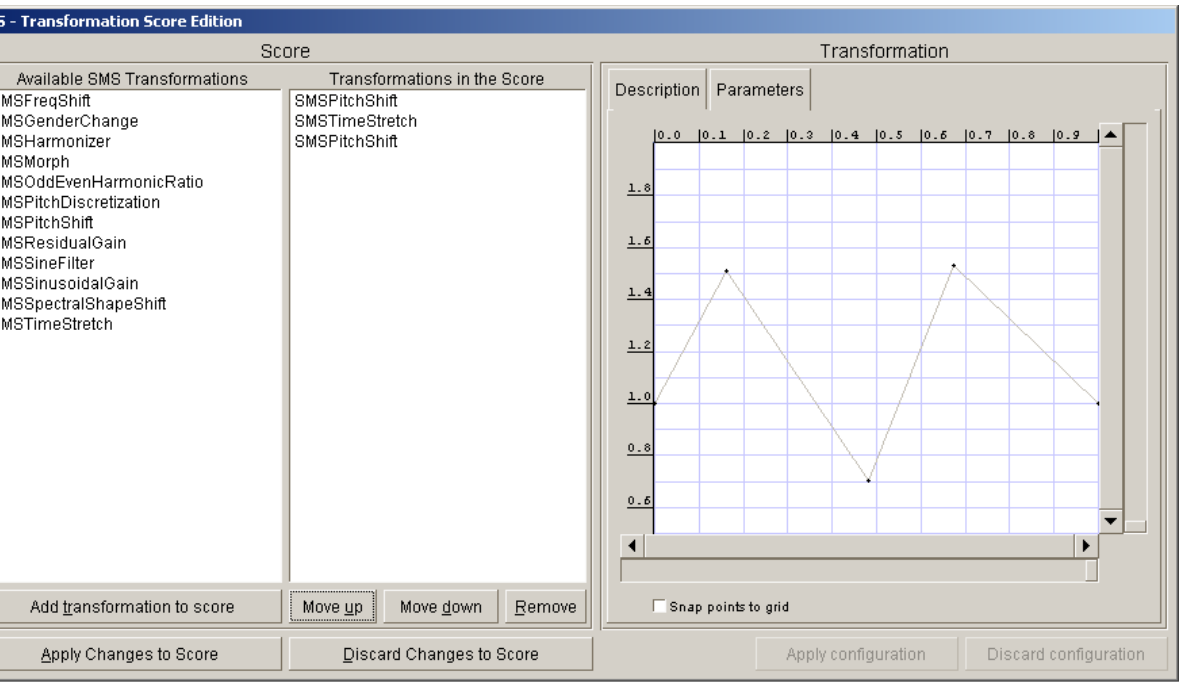
4. Apart from displaying, the result of the analysis can be stored for later uses. It can be stored in xml format or sdif format. Xml is a textual tagged format that can be convenient for debugging and studying results of some analysis but it is very verbose and slow (you should expect an xml file 4 times the size of the original audio). On the other hand SDIF is a binary format that for most uses will be much more convenient. For storing the result of the analysis, choose File->SMS Analysis->Store Analysis File. The application will switch from SDIF format to XML depending on the file extension you choose.

   Note that the result of an analysis can be later loaded selecting File->SMS Analysis->Load Analysis Data.
5. Before synthesizing, we can transform the sound. For doing so, just as it happened with the configuration, we can load an xml transformation score or edit the default one. We will do the latter. Choose File->SMSTransformation->Edit Score. You will see a list of available transformations. Clicking on any of them will open a brief description of the transformation and its usage:



(Click to enlarge)

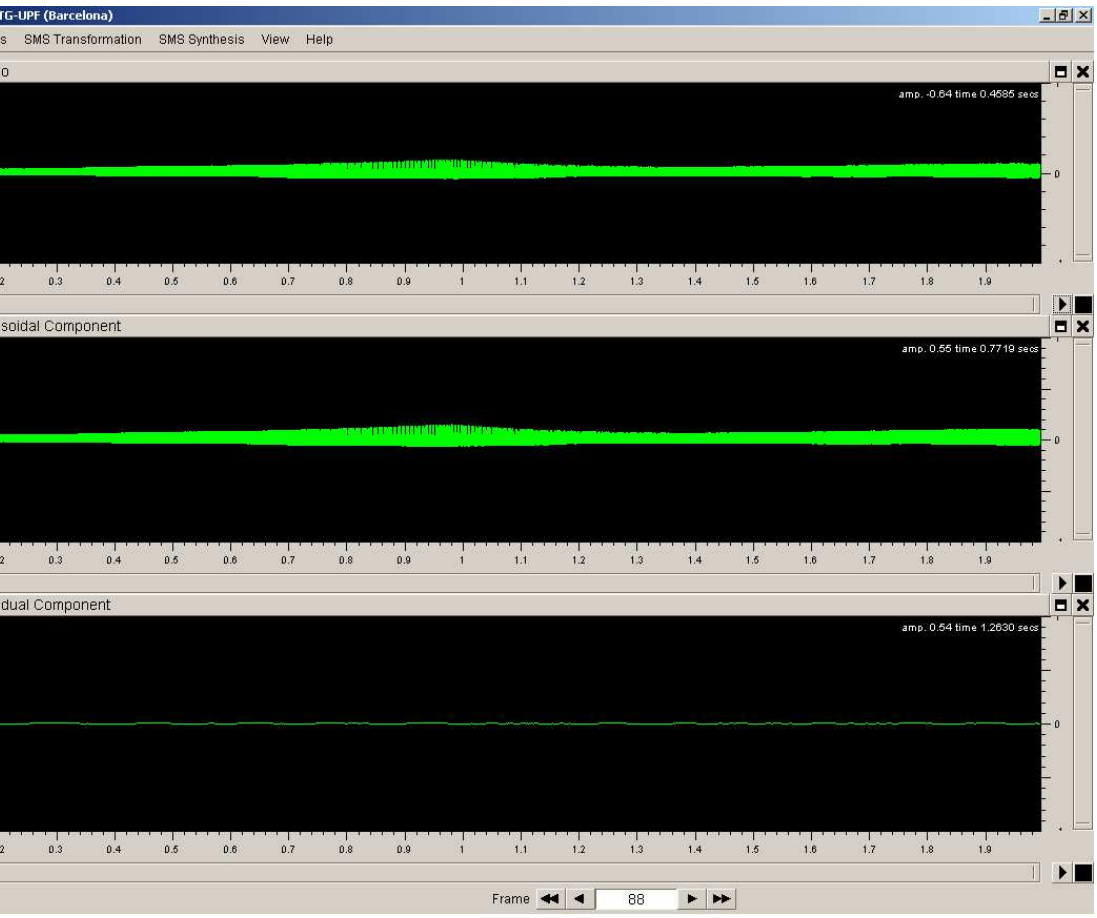
Select any transformation, click on Add Transformation to Score and edit the Parameters on the right. In the break point function editor, duble click to add a point and click to select a point or draw a selection box. Repeat this step for any number of transformations you want to connect one after the other.



(Click to enlarge)

Click on Apply Changes to Score. Now we can transform the sound by selecting SMSTransformation->Apply.

5. We can now synthesize the resulting sound. Select SMSSynthesis->Synthesize. Now visit the View menu again and select the output component you want to display and listen to: Output Sound, Output Sinusoidal and Output Residual.



(Click to enlarge)

6. Finally you can store the results of the synthesis by choosing File->SMS Synthesis->Save...

110 Analysis Output

So, the output of the analysis is a CLAM::Segment that contains an ordered list of CLAM::Frames. Each of these frames has a bunch of attributes, but the most important are: a CLAM::SpectralPeakArray that models the sinusoidal component (including information about sinusoidal track), a residual spectrum and the result of the pitch estimation (or rather fundamental detection).

[i2] The output of this analysis can be (1) stored in xml format, (2) stored in sdif binary format, (3) transformed and (4) synthesized back.

The xml format is rather verbose so the process of storing the result of the analysis in xml can be time-consuming and can result in a large file (you can expect an expansion factor of about 10:1 comparing it to the original sound file). On the other hand, it can be easily read and can be used for debugging purposes. This is a partial example of what your xml file will look like:

```
<Analyzed_Segment>

<BeginTime>0</BeginTime>
<EndTime>0.766258</EndTime>
<prHoldsData>1</prHoldsData>
<FramesArray>
```

<Frame>

<CenterTime>0.00290249</CenterTime>
<Duration>0.00580499</Duration>
<SpectralPeakArray>

<Scale>Log</Scale>
<nPeaks>2</nPeaks>
<nMaxPeaks>50</nMaxPeaks>
<MagBuffer>-30.733 -9.25596e+061</MagBuffer>
<FreqBuffer>343.559 714.887</FreqBuffer>
<PhaseBuffer>1.5182 -0.0999764</PhaseBuffer>
<BinPosBuffer>4.00429 8.33224</BinPosBuffer>
<BinWidthBuffer>5 6</BinWidthBuffer>
<IndexArray>0 1</IndexArray>
<IsIndexUpToDate>0</IsIndexUpToDate>

</SpectralPeakArray>
<Fundamental>

<nMaxCandidates>1</nMaxCandidates>
<nCandidates>1</nCandidates>
<CandidatesFreq>343.559</CandidatesFreq>
<CandidatesErr>-1</CandidatesErr>

</Fundamental>
<ResidualSpec>

<Scale>Linear</Scale>
<SpectralRange>22050</SpectralRange>
<prSize>257</prSize>
<MagBuffer>0.00729711 0.00896791 0.0119354 0.0189819 0.000319389 0.0183698 0.00916167
0.0161299 0.0269896 0.0244382 0.0129489 0.00634384 0.00610172 0.00875066 0.00763111
0.00742003 0.00559243 0.000849196 0.00317517 0.00519037 0.00528536 0.00224068
0.00154297 0.00469072 0.00733866 0.00677861 0.00294284 0.000809419 0.00223092
0.00265733 0.00162793 0.000550603 0.000858541 0.00125168 0.00068071 0.000105767
0.00117828 0.00236095 0.00198881 0.00090993 0.000564566 0.00125519 0.00127909
0.00127066 0.00120286 0.00124451 0.00119085 0.000941523 0.000348921 0.000462266
0.000758022 0.000818371 0.000709333 0.000946096 0.000769061 0.000493843 0.0004842
0.000491863 0.000382826 0.000379278 0.000367886 0.000472623 0.000445563 0.000435359
0.000409167 0.000435712 0.000289131 0.000230931 0.00029841 0.000469285 0.000420821
0.000506603 0.000552621 0.000943916 0.000950009 0.000439976 0.000162634 0.000268673
6.66906e-005 0.000417807 0.000419127 0.000346079 0.000245963 0.000310977 0.000329851
0.000336672 0.000337432 0.000148063 0.000154738 0.000438211 0.000522947 0.000425414
0.000180047 0.000117523 0.000177102 0.000208794 0.000144707 7.62237e-005 9.28201e-005
0.000287346 0.000310371 0.000266661 0.000149645 0.00021754 0.00030985 0.000314559
0.000192857 3.32129e-005 0.000167097 0.000250963 0.000203181 0.000134031 0.000102491
0.000243036 0.000254449 0.000275495 0.000234674 0.000215736 0.000161505 0.000161531
0.000172264 0.00023892 0.000160434 0.000127822 0.000190916 0.000242277 0.000186513
0.000215995 0.000218083 0.000244236 0.000200369 0.000238023 0.000236179 0.000275808
0.000191796 0.000101178 9.95931e-005 0.00016034 0.000175327 0.00027087 0.000222175
0.000191879 0.000384598 0.000499003 0.000365438 0.000181958 0.000113119 0.000311989
0.000328654 0.000263493 0.000163289 0.000115733 4.07026e-005 0.000189727 0.000172219
0.00011008 0.000147785 0.000168118 0.000111638 0.000161761 0.000193647 0.000201768
0.000156759 0.000175221 0.00012852 0.000159159 0.000136581 0.000142458 0.000104311
0.000137866 0.000153131 0.000194957 0.000151109 0.000133012 0.000112813 0.000154199
0.000165131 0.00019192 0.000141609 0.000130629 0.00013765 0.000185932 0.00016096
0.000129 0.000100294 0.000173186 0.000153843 0.000152076 0.000109613 0.000140937
0.000133521 0.000153002 0.000131424 0.000147679 0.000123386 0.000139856 0.000116689
0.000141218 0.000121856 0.00014494 0.000127773 0.000145369 0.000121058 0.000142542
0.000129803 0.000148939 0.000122527 0.00013909 0.000116454 0.000140474 0.000120404

0.000129426 0.000107437 0.000138172 0.000119912 0.000137334 0.000111384 0.000126984
0.000110148 0.000136653 0.000119516 0.00013837 0.000118661 0.000135258 0.000111201 0.000129452 0.000109565
0.000124679 0.000108109 0.000133821 0.000113002 0.000129722 0.000110658 0.000133673 0.000114235 0.000132992
0.000111121 0.000130611 0.000112623 0.000130159 0.000108879 0.00012951 0.000112183 0.000130109 0.00010963
0.000128652 0.000106292 0.00012649 0.000109276 0.000129228 0.000108783 0.000128619 0.000111234 0.000132835
0.000113863 0.000131056 0.000109608</MagBuffer>
<PhaseBuffer>3.14159 3.08208 -2.84775 -2.84702 0.222501 -0.357415 -0.192005 0.857515
0.164266 -0.717075 -1.34692 -1.25306 -0.80304 -1.03352 -1.4255 -1.67493 -2.42337 2.28937 -0.463914 -1.0696 -1.81997
-2.63472 -0.265465 -0.803979 -1.55456 -2.49373 2.90535 -2.02756 -2.19864 -2.99324 2.16832 1.08515 0.158611
-1.07551 -1.91136 -1.9366 0.120491 -0.980804 -2.03769 -3.00178 1.04362 -0.356165 -0.979226 -1.18441 -1.39887
-1.52026 -1.83433 -2.18476 -2.52041 -0.899889 -1.20572 -1.48061 -1.48561 -1.60454 -2.08082 -1.92976 -1.82558
-1.80127 -1.87773 -1.57999 -1.50824 -1.43029 -1.62481 -1.58055 -1.61722 -1.70472 -1.89816 -1.36964 -0.93364
-1.11818 -1.2283 -1.20535 -1.13643 -1.42734 -2.23122 -2.95545 -1.89245 -2.28797 0.792653 -0.796647 -1.49208
-1.64987 -1.6976 -1.35185 -1.64026 -1.63719 -2.06527 -2.53459 -0.410569 -1.05411 -1.79081 -2.55591 2.60431
-0.288706 -1.10392 -1.49526 -1.97181 -2.15656 0.612444 -0.468561 -1.25125 -1.95891 2.80796 0.563956 -0.425114
-1.15636 -1.96053 2.55262 0.369696 -0.400003 -0.842419 -1.04522 0.108873 -0.174641 -0.470307 -0.682711 -0.872543
-0.952344 -0.925881 -0.755265 -0.494537 -0.745278 -1.03808 -0.533305 -0.283193 -0.606887 -0.655821 -0.569462
-0.580754 -0.722371 -0.719113 -0.685507 -0.751142 -0.97438 -1.3813 -1.14374 -0.315467 -0.331636 -0.143219
-0.349771 -0.678538 -0.204147 -0.216003 -0.837882 -1.42846 -1.64832 -0.382795 -0.614104 -1.17802 -1.55154 -1.80173
-1.9215 1.68532 -0.0644832 -0.821967 -0.637113 -0.409722 -0.694729 -0.596954 -0.316125 -0.470586 -0.732607
-0.744035 -0.781622 -0.776868 -0.647869 -0.750809 -0.719915 -0.622058 -0.367594 -0.35117 -0.553486 -0.79245
-0.674985 -0.452145 -0.342092 -0.397661 -0.599614 -0.773513 -0.555672 -0.3547 -0.493991 -0.741259 -0.803371
-0.285476 -0.357441 -0.587013 -0.65025 -0.577342 -0.39602 -0.443145 -0.473738 -0.505712 -0.503574 -0.511424
-0.479682 -0.442623 -0.40107 -0.404894 -0.38004 -0.408274 -0.418635 -0.415458 -0.360539 -0.375894 -0.421767
-0.44643 -0.414257 -0.397415 -0.363917 -0.427264 -0.405524 -0.307071 -0.282502 -0.335477 -0.343677 -0.364971
-0.292669 -0.240198 -0.227494 -0.262238 -0.266188 -0.290855 -0.294307 -0.299723 -0.254931 -0.263727 -0.224682
-0.161851 -0.174198 -0.203569 -0.185546 -0.155451 -0.151441 -0.168252 -0.167166 -0.170164 -0.139481 -0.147306
-0.145664 -0.130857 -0.102859 -0.108977 -0.113821 -0.103976 -0.100063 -0.0870527 -0.0498648 -0.0388209 -0.0417596
-0.0324963 -0.0127574 0.00388132 -0.0023968 -0.016122 -0.0216873 0</PhaseBuffer>

</ResidualSpec>

</Frame> ...

If you prefer to have a more compact representation of your analysis you should choose the SDIF format. SDIF (Sound Description Interchange Format) is a non-propietary format used for exchanging analysis data especially between research teams.

111 Configuration

In order to make the application work, you first need to load a configuration xml file or edit the default one through the graphical interface. This configuration includes all the different parameters for the analysis/synthesis process. If you load an xml file that includes non-valid parameter values (like a path to a non-existing input sound file), the analysis process will remain disabled. (Note: if you try to load an xml file that does not comply to the Configuration schema, the result is unpredictable and may even produce a program crash).

Here is an example of a valid xml configuration:

```
<SMSAnalysisSynthesisConfig>

  <Name />
<InputSoundFile>c:/1_brief.wav</InputSoundFile>
<OutputSoundFile>c:/1_out.wav</OutputSoundFile>
<OutputAnalysisFile>c:/analysis.sdif</OutputAnalysisFile>
<InputAnalysisFile>c:/analysis.xml</InputAnalysisFile>
<AnalysisWindowSize>513</AnalysisWindowSize>
<AnalysisHopSize>256</AnalysisHopSize>
<AnalysisWindowType>Hamming</AnalysisWindowType>
<ResAnalysisWindowSize>513</ResAnalysisWindowSize>
<ResAnalysisWindowType>BlackmanHarris92</ResAnalysisWindowType>
<AnalysisZeroPaddingFactor>0</AnalysisZeroPaddingFactor>
<AnalysisPeakDetectMagThreshold>-120</AnalysisPeakDetectMagThreshold>
<AnalysisPeakDetectMaxFreq>-120</AnalysisPeakDetectMaxFreq>
```

```
<AnalysisSinTrackingFreqDeviation>20</AnalysisSinTrackingFreqDeviation>
<AnalysisReferenceFundFreq>1000</AnalysisReferenceFundFreq>
<AnalysisLowestFundFreq>40</AnalysisLowestFundFreq>
<AnalysisHighestFundFreq>6000</AnalysisHighestFundFreq>
<AnalysisMaxFundCandidates>50</AnalysisMaxFundCandidates>
<AnalysisHarmonic>0</AnalysisHarmonic>
<DoCleanTracks>0</DoCleanTracks>
<SynthesisFrameSize>256</SynthesisFrameSize>
<SynthesisWindowType>Triangular</SynthesisWindowType>
<SynthesisHopSize>-1</SynthesisHopSize>
<SynthesisZeroPaddingFactor>0</SynthesisZeroPaddingFactor>

</SMSAnalysisSynthesisConfig>
```

Let us comment the different parameters involved:

Global Parameters:

<Name>: Particular name you want to give to your configuration file. Not used for anything except the xml parsing.

<InputSoundFile>: path and name of the input sound file you want to analyze (depending if you are running the application in GNU/Linux or Windows that will be a unix path or an msdos path).

<OutputSoundFile>: path and name of where you want to have your output synthesized sound file. The application will add a "_sin.wav" termination to the Sinusoidal component and a "_res.wav" termination the residual file name. In the graphical version of the program (SMSTools) though, this parameter is not used as when the output sound is to be stored, a file browser dialog pops-up.

<OutputAnalysisFile>: path and name of where you want your output analysis data to be stored. The extension of the file can be .xml or .sdif. The application will choose the correct format depending on the extension you give.Not used in the GUI version as it is obtained from the dialog.

<InputAnalysisFile>: path and name of where you want your input analysis data to be loaded from. Not used in the GUI version as it is obtained from the dialog.

(Note to users of previous versions: Sampling Rate is no longer used as it is automatically extracted from the input sound file).

Analysis Parameters

<AnalysisWindowSize>: window size in number of samples for the analysis of the sinusoidal component. (Note: if the value entered is not odd, the program will internally add +1 to it)

<ResAnalysisWindowSize>: window size in number of samples for the analysis of the residual component. (Note: if the value entered is not odd, the program will internally add +1 to it)

<AnalysisWindowType>: type of window used for the sinusoidal analysis. Available: Hamming, Triangular, BlackmannHarris62, BlackmannHarris70, BlackmannHarris74, BlackmannHarris92, KaisserBessel17, KaisserBessel18, KaisserBessel19, KaisserBessel20, KaisserBessel25, KaisserBessel30, KaisserBessel35.

<ResAnalysisWindowType>: type of window used for the residual analysis. Available: Same as in sinusoidal. *Recommended*: as sinusoidal spectrum is synthesized using the transform of the BlackmannHarris 92dB, it is necessary to use that window in the analysis of the residual component in order to get good results.

<AnalysisHopSize>: hop size in number of samples. It is the same both for the sinusoidal and residual component. If this parameter is set to -1 (which means default), it is taken as half the residual window size. *Recommended* values range from half to a quarter of the residual window size.

<AnalysisZeroPaddingFactor> Zero padding factor applied to both components. 0 means that zeros will be added to the input audio frame till it reaches the next power of two, 1 means that zeros will be added to the next power of two etc...

<AnalysisPeakDetectMagThreshold>: magnitude threshold in dB's in order to say that a given peak is valid or not. *Recommended*: depending on the window type and the main-to-secondary lobe relation and the characteristics of the input sound, a good value for this parameter may range between -80 to -150 dB.

<AnalysisPeakDetectMaxFreq>: Frequency of the highest sinusoid to be tracked. This parameter can be adjusted, for example, if you are anlyzing a sound that you know only has harmonics up to a certain frequency. *Recommended*: It depends on the input sound but, in general, a sensible value is 8000 to 10000 Hz.

<AnalysisSinTrackingFreqDeviation>: maximum deviation in hertz for a sinusoidal track.

<AnalyisReferenceFundFreq>: in hertz, reference fundamental.

<AnalyisLowestFundFreq>: in hertz, lowest fundamental frequency to be acknowledged.

<AnalyisHighestFundFreq>: in hertz, highest fundamental frequency to be acknowledged.

<AnalyisMaxFundFreqError>: maximum error in hertz for the fundamental detection algorithm.

<AnalyisMaxFundCandidates>: maximum number of candidate frequencies for the fundamental detection algorithm.

<AnalysisHarmonic>: if 1, harmonic analysis is performed on all segments that have a valid pitch. In those segments the track number assigned to each peak corresponds to the harmonic number. On unvoiced segments, inharmonic analysis is still performed. Set to 1 if the type of transformation you want to perform depends on this harmonic characteristic. It is highly recommended that if you use harmonic analysis you set a low threshold for AnalysisPeakDetectMagThreshold> (i.e. -120). This means that many peaks (more than necessary) will be detected in the peak detection process but will then be removed in the harmonic tracking process.

Synthesis Parameters

<SynthesisFrameSize>: in number of samples, size of the synthesis frame. If set to -1, it is computed as (ResAnalysisWindowSize-1)/2. If any other number is used you are bound to get synthesis artefacts.

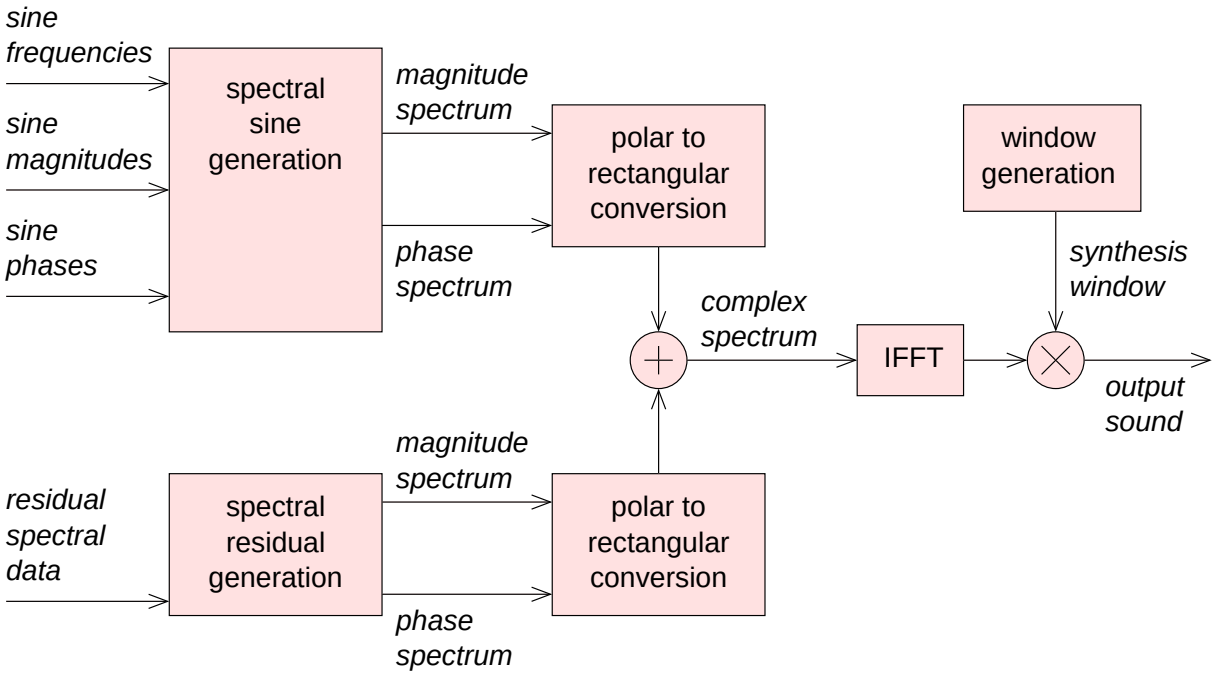
<SynthesisWindowType>: type of window used for the residual analysis. Available: Same as in sinusoidal.

Morph Parameters

<MorphSoundFile>: Optional name of the second file to do a morph on. Only necessary if you want to do a morphing transformation afterwards. Note that the file to morph will be analyzed with the same parameters as the input sound file and that it must have the same sampling rate.

112 Synthesis

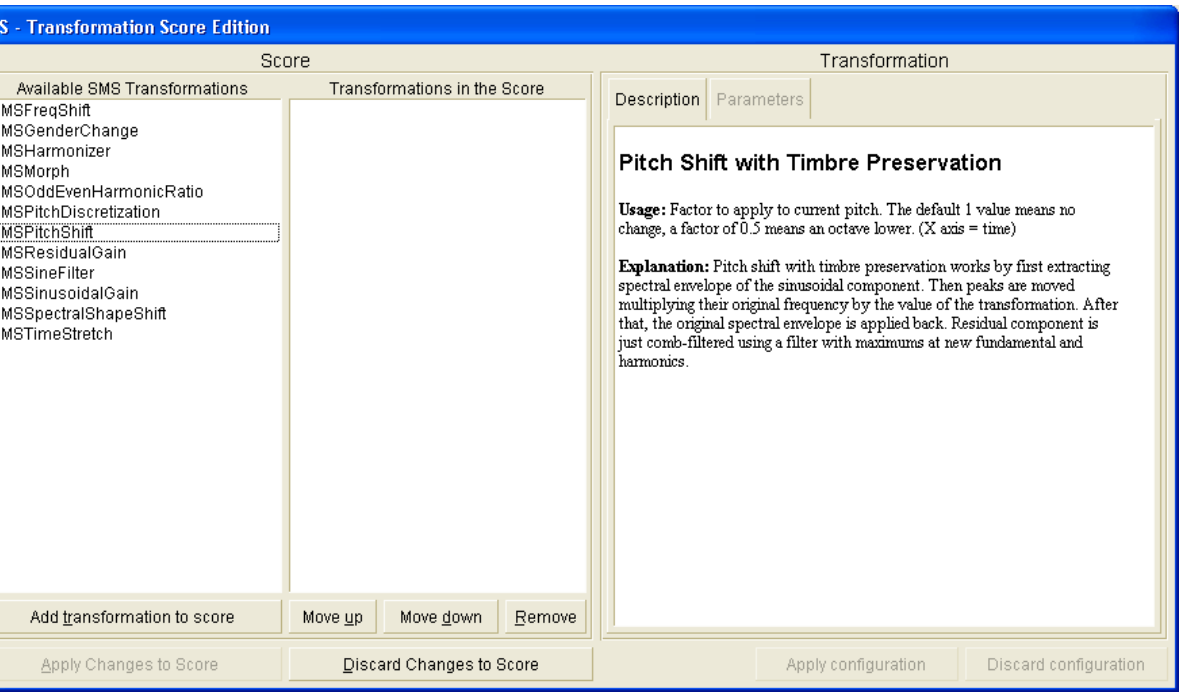
Apart from storing the result of your analysis, you can do more interesting things. The first thing you may like to do is synthesize it back, separating each component: residual, sinusoidal, and the sum of both. The following picture illustrates the SMS synthesis algorithm:



For invoking the synthesis procedure you just have to call the 'synthesize' option (either from the GUI or the menu) and then look or store each of the components.

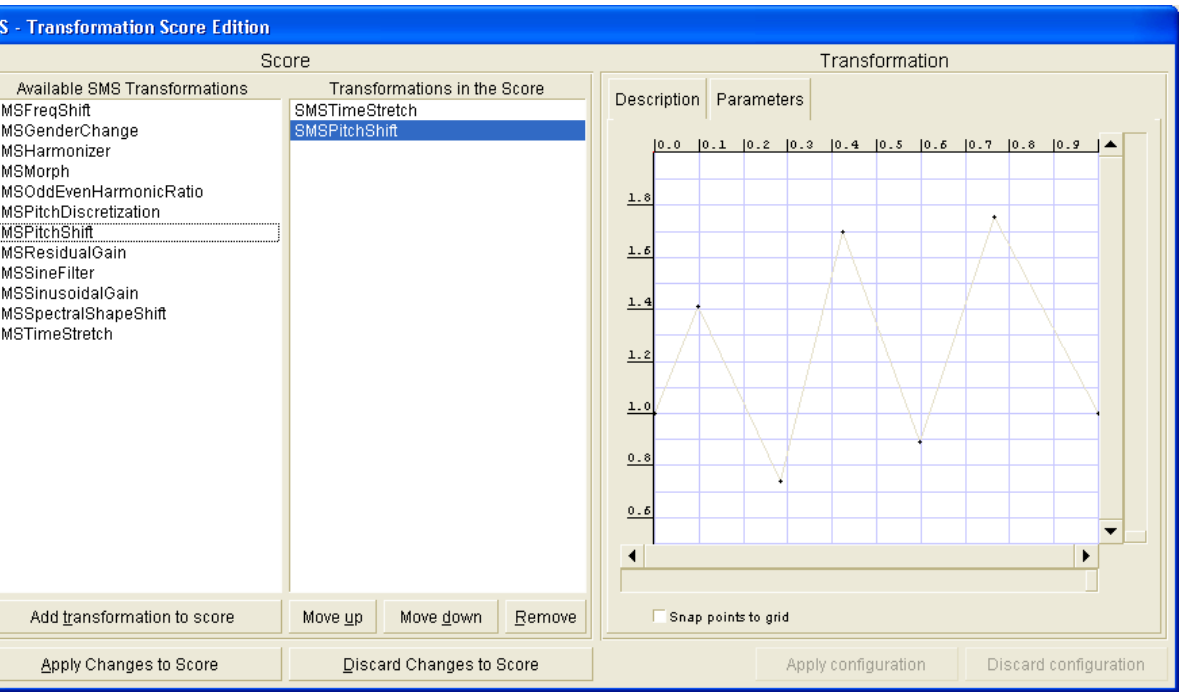
113 Transformation

To transform your sound you first have to load an xml transformation score or use the graphical transformation editor available in SMSTools. It is strongly recommended that you use the graphical interface in order to generate the transformation chain you want to apply to your sound. Click on File->Transformation Score->Edit Score. A new dialog like the one depicted in the next figure will pop-up.



On the left you have a list of available SMS transformations. By clicking on any of them you will get a brief description on the right panel. This description is just intended to give you a very concise explanation on how the transformation works. If you should need a more detailed explanation the recommended source is the book DAFX - Digital Audio Effects, edited by Udo Zoelzer and published by Wiley and Sons Ltd. Once you have decided what transformation you want to use you have click on the "Add to Score" button. The transformation will be added to the central panel, where the list of used transformations is located. Now you are ready to set the parameters for the transformations. Select the "Parameters" tab on the right-hand panel.

Once you have edited a transformation, you can repeat the previous steps to add a new transformations. You may add as many transformations as you need (even more than one instance of the same transformation). Note though that the way the transformations are sorted affects the output (i.e. it is not the same to apply a pitch shift and then a spectral shape shift than doing it the other way around).



Although some transformations have simpler controls, most of them are configured using a break-point-function (or envelope-like) control like the one shown in the previous picture. Note that this break point funtion may represent an evolution over time or an spectral envelope, depending on the type of transformation. On the bpf grid, click to select and move and double click to add a point. You can select a point or a set of points by drawing a selection box. Then you can delete them pressing Ctrl X, you can move them with the mouse or the cursor keys, you can mirror them with the '*' key or increase or decrease the difference to the mean value with '+' and '-' keys.

But if you don't want to use the graphical editor or you are using the non-gui version you will need to understand how the xml score works.This configuration file is basically a composition of concrete configurations that affect specific transformations. You may change the order, activate/deactivate or configure any of them.The way the configuration affects a concrete transformation may vary, you should read the comments before each transformation for learning the particularities of every transformation.All transformations share some common fields. First they need to define the kind of concrete transformation in the ConcreteClassName attribute. Then you need to specify the values for the transformation. This can usually be done in two different ways: using a single-value Amount attribute or a breakpoint function BPFAmount attribute in which you declare a function as a set of points and an interpolation type (note: this function may represent a time envelope or a spectral envelope depending on the transformation).

In any transformation chain there need to be two special transformations called SMSTransformationChainIO as first and last transformation to the chain. These transformations act as input and output, are mandatory but do not affect the result.After the list of transformations there is an 'OnArray' where you have the same number of 1's or 0's as transformations in the Chain. 1 means active and 0 means bypassed (SMSTransformationChainIO's are not affected by this).

Here is a complete commented example of how a transformation score looks like:

```
<SMSTransformationChainConfig>

    <Configurations>

        <!-- Input to the transformation chain. Must always be present. -->
    <Config>
    <ConcreteClassName>SMSTransformationChainIO</ConcreteClassName>
    </Config>
```

<!-- Pitch shift with timbre preservation. You can define a single-value amount or a time envelope. 1 means no change and 0.5 means multiplying current pitch by 0.5-->
<Config>
<ConcreteClassName>SMSPitchShift</ConcreteClassName>
<BPFAmount>
<Interpolation>Linear</Interpolation>
<Points>{0 1.5} {1 1.5}</Points>
</BPFAmount>
</Config>
<!-- Gender change. If amount is 0 it means from male to female. If it is 1 it means from female to male. -->
<Config>
<ConcreteClassName>SMSGenderChange</ConcreteClassName>
<Amount>0</Amount>
</Config>
<!-- Pitch discretization to temparate scale. If active it rounds the pitch to nearest note according to temperate musical scale.-->
<Config>
<ConcreteClassName>SMSPitchDiscretization</ConcreteClassName>
</Config>
<!-- Gain applied to odd harmonics in relation to even harmonic. E.g. A value of 6 means that a 6dB difference will be introduced, thus, odd harmonics will be 3dB higher and even harmonic 3dB lower.-->
<Config>
<ConcreteClassName>SMSOddEvenHarmonicRatio</ConcreteClassName>
<Amount>12</Amount>
</Config>
<!-- Frequency shift applied to all partials expressed in Hz-->
<Config>
<ConcreteClassName>SMSFreqShift</ConcreteClassName>
<Amount>100</Amount>
</Config>
<!-- Filter expressed in bpf format applied to only the sinusoidal components-->
<Config>
<ConcreteClassName>SMSSineFilter</ConcreteClassName>
<BPFAmount>
<Interpolation>Step</Interpolation>
<Points>{0 6} {1 0} {2 0} {100 0} </Points>
</BPFAmount>
</Config>
<!-- Gain in dB’s applied to the residual component-->
<Config>
<ConcreteClassName>SMSResidualGain</ConcreteClassName>
<Amount>6</Amount>
</Config>
<!-- Gain in dB’s applied to the sinusoidal component-->
<Config>
<ConcreteClassName>SMSSinusoidalGain</ConcreteClassName>
<Amount>3</Amount>
</Config>
<!-- Harmonizer. Each point defines a new voice added to the harmonization. The X value is the gain in relation to the original one and the Y value the pitch transposition factor. -->
<Config>
<ConcreteClassName>SMSHarmonizer</ConcreteClassName>
<BPFAmount>
<Interpolation>Step</Interpolation>
<Points>{-3 1.3} {-3 1.5} {-3 1.7}</Points>
</BPFAmount>
</Config>
<!-- Frequency shift applied to the sinusoidal spectral shape expressed in Hz.-->
<Config>
<ConcreteClassName>SMSSpectralShapeShift</ConcreteClassName>
<Amount>50</Amount>

```
    </Config>
      <!-- Morphing between two different sounds. SMSMorph has many different parameters to configure some
    of which are not even implemented. But the basic ones (and already tested) are:
    <HypBPB>: BPF (envelope-like) Parameter. Defines how much of each sound is being used from 0 to 1
    <SynchronizeTime>: BPF (envelope-like) Parameter. Defines temporal relation between input sound and
    sound to hybridize
    <HybPitch: BPF (envelope-like) Parameter. Pitch to use: 0 input, 1 sound to hybridize-->
    <Config>
    <ConcreteClassName>SMSMorph</ConcreteClassName>
    <HybBPF>
    <Interpolation>Linear</Interpolation>
    <Points>{0 0} {1 1}</Points>
    </HybBPF>
    </Config>
    <!-- Output to the transformation chain. Must always be present. -->
    <Config>
    <ConcreteClassName>SMSTransformationChainIO</ConcreteClassName>
    </Config>

  </Configurations>
  <!-- Array defining what transformations are active(1) or bypassed(0). WARNING: This array always has to be the
  same size as the number of previous transformations available.-->
  <OnArray>1 0 0 0 0 0 0 0 0 0 0 1 1</OnArray>

</SMSTransformationChainConfig>
```

114 Implementing your own transformation

If you want to implement a particular transformation (other from the very simple frequency shift included as a sample) you will have to get into a little coding (don't panic, it is very simple!).

These are the steps you have to follow:

1. Look at any of the basic already implemented transformations located at *clam/src/Processing/Transformations/SMS* (SMSFreqShift will do fine as a first step, do not wander into time SMSTimeStretch and SMSMorph as they are much more complicated. Yeah, I bet now that is the first thing you are planning on doing).
2. Write your own MyTransformation.hxx and MyTransformation.cxx files following the same structure. In the header, note that your class MyTransformation must derive from a template class named SMSTransformationTmpl. As the template argument, you must use the part of the analyzed frame you want to transform: SpectralPeakArray if you want to transform the sinusoidal component, Spectrum if you want to transform the residual component or Frame if you want to transform both. Then you need to declare the following methods: GetClassName that must return your classes name, a default constructor, a constructor that takes in an SMSTransformationConfig as argument, and a Do(T,T) method where T is the concrete type on which the transformation is applied (SpectralPeakArray, Spectrum or Frame). Here is how the header of MyTransformation.hxx would look like if we decided we wanted to transform only the residual component:

```
class MyTransformation: public SMSTransformationTmpl<SpectralPeakArray>
{
        const char *GetClassName() const {return "MyTransformation";}
public:
        MyTransformation(){}
        MyTransformation(const SMSTransformationConfig &c):SMSTransformationTmpl<SpectralPeakArray>(c){}
        bool Do(const SpectralPeakArray& in, SpectralPeakArray& out);
};
```

3. Now go to MyTransformation.cxx file and implement the code for your transformation.
   So, if MyTransformation only gets every spectral peak and multiplies its frequency by two, I should only have to write the following code:

```
bool MyTransformation::Do(const SpectralPeakArray& in, SpectralPeakArray& out)
{
   TSize nPeaks=in.GetnPeaks()
   DataArray& inBuffer=in.GetFreqBuffer();
   DataArray& outBuffer=out.GetFreqBuffer();
   for(int i=0;i<nPeaks;i++)
      outBuffer[i]=inBuffer[i]*2;
   return true;
}
```

   Note that the only difference with the code you will find in the SMSFreqShift.cxx file is that here we are using a constant multiplying factor while there it takes the value out of a control by doing mAmountCtrl.GetLastValue(). This CLAM control returns holds the current value from the transformation and it is updated automatically from the SMS application
4. After adding the necessary includes in your files, you are ready to compile your new SMS transformation. It is now ready to use. But, in order to make it automatically available from the SMS application a few more things need to be done to integrate your transformation into what we call the SMSTransformationChain. The steps that follow are in some cases a bit of a hack and we are working on simplifying them in a next release:
   - First we must register the MyTransformation class in the Processing Factory (implementation of the Factory Method pattern for those of you who are into software engineering). For doing so, you must go to the MyTransformation.cxx file and add the following lines:

```
typedef CLAM::Factory<CLAM::Processing> ProcessingFactory;
static ProcessingFactory::Registrator<CLAM::SMSFreqShift> regtMyTransformation( "MyTransformation" );
```

   - Then you need to make the ProcessingChain class that a new transformation has been added and the type of configuration it uses. Open ProcessingChain.cxx file, search for the InstantiateConcreteConfig method and you will see a long and ugly "if" that includes most transformations. Help us make it uglier by adding your transformation over there:

```
if(type=="SMSDummyTransformation"||type=="SMSFreqShift"||type=="SMSPitchShift"||
        type=="SMSOddEvenHarmonicRatio"||type=="SMSSineFilter"||type=="SMSResidualGain"||
        type=="SMSHarmonizer"||type=="SMSSinusoidalGain"||type=="SMSPitchDiscretization"||
        type=="SMSSpectralShapeShift"||type=="SMSGenderChange"||type=="SMSTransformationChainIO"
               ||type=="MyTransformation")
```

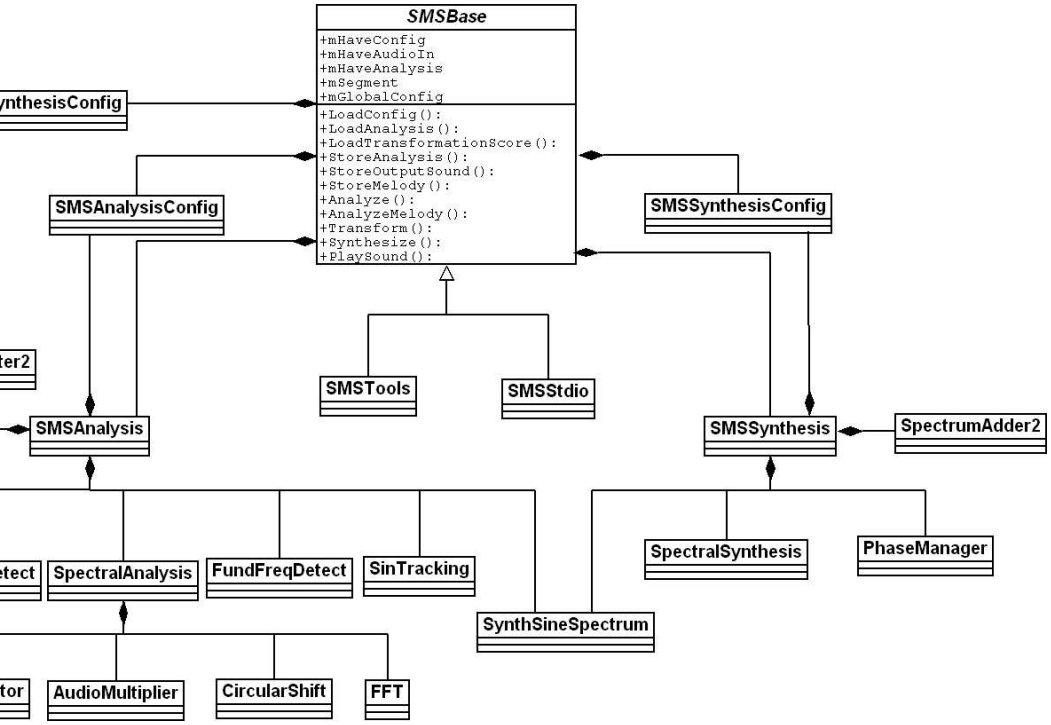
5. Finally, you may want to add a widget to the graphical interface in order to configure your transformation. We recommend you to just follow the example of a pre-existing configurator in */examples/sms/tools/gui*. The SMSFreqShiftConfigurator, for instance, will do fine for a simple configuration.

You are now ready to build the program again and used your configuration from the SMS application. If you are the of curious kind you have surely already wandered into the SMSMorph and SMSTimeStretch transformations. These transformations are much more complex because the (1) use a different configuration instead of the default SMSTransformationConfig, (2) override some of the behaviour of the base SMSTransformation class because they have a more complex logic (SMSTimeStretch, for example processes at a non-constant rate), (3) They are based on other existing Processings. We will leave the exercise of implementing such transformation to the advanced reader (remember you may always get support through the CLAM mailing list clam@iua.upf.es).

115 Internal class structure and program organisation

You should only read this section if you are particularly interested in learning ’what’s inside’ the example or you want to use its structure as a base for another application. Otherwise, if you are using the SMS as an aplication on its own and could not care less about programming details you better skip this part (for your own sake!).

The rest of this section will deal with the main structural aspects with the application. All these aspects are summarized in the following UML diagram:



The main class of our application is the SMSBase class. This is an abstract class (thus cannot be instantiated), but contains the core of the process flow. The two derived classes, SMSTools and SMSStdio are the GUI and Standard I/O versions of the base class.

So let us briefly mention what this base class holds inside. All the methods illustrated in the diagram (LoadConfig, Analyze,...) correspond to functionalities of the program that, in the case of the GUI version, are mapped directly to menu options. Of course the class has other methods but are used for internal convenience but are less important.

The boolean (mHaveConfig, mHaveInputAudio,....) attributes of the class hold important values to control the flow of the program because they inform of whether a previous action has taken place and the desired operation can then be invoked.

The class has two Processing Composite attributes, instances of the SMSAnalysis and SMSSynthesis classes. These Processing Composites are configured when the global configuration is loaded and then run from the Analyze and Synthesize methods. Some intermediate Processing data (a Segment, a Melody and different Audio objects) are used to hold the input/output data generated during the process. These data is then stored/played using the corresponding method (i.e. StoreAnalysis or PlayOutputSound).

You will see that, although this class concentrates most of the functionality of the application and has a great deal of operations, these methods are fairly simple and rarely need more than 10/20 lines of code (as a matter of fact, the longest operation is the Melody analysis, which should not be included in this class but rather in a separate Processing class). Much as the complex logic of some method delegation (as the one existing in the methods Analyze, DoAnalysis, AnalysisProcessing...) is introduced by the needs of the graphical interface, where callbacks need to be assign to GUI commands and methods need to be called on separate threads. But, as an example, let us take a look at the AnalysisProcessing method that implements the actual process for the analysis:

```
1.      void SMSBase::AnalysisProcessing()      {2.         Flush(mOriginalSegment);3.          int k=0;4.          int step=mAnalConfig.GetHopSize();5.                GetAnalysis().Start();
6.              while(GetAnalysis().Do(mOriginalSegment)) {
7.                      k=step*(mOriginalSegment.mCurrentFrameIndex+1);
8.                      mCurrentProgressIndicator->Update(float(k)); }
9.              GetAnalysis().Stop();   }
```

In line 1 we call the Flush method, which initializes the member Segment and deletes previously existing data. In the next two lines we declare and initialize the variables that will be used in the analysis loop: k is the counter for updating the ProgressIndicator and step is the analysis hop size. In

line 5 we Start our SMSAnalysis Processin. In line 6 the loop begins. Note that the output condition is the return value of the call to the Do in the SMSAnalysis. In lines 7-8 we update the progress indicator and finally, in line 9, we Stop the SMSAnalysis Processing.

Note that, if we remove the support for the Progress Indicator the resulting method would only be 5 lines long:

```
1. void SMSBase::AnalysisProcessing() {
   2. Flush(mOriginalSegment);
   3. GetAnalysis().Start();
   4. while(GetAnalysis().Do(mOriginalSegment)) {}
   5. GetAnalysis().Stop(); }
```

If you follow a similar approach you can fairly easily understand all the members of out main application class.

116 SMSSynthesis and SMSAnalysis

We have seen how simple the processing part of our application is: basically a call to the Do method of either SMSAnalysis or SMSSynthesis classes. That is only possible because these Processing Composites hide all the processing complexity.

If we take a look again to the UML diagram we see that these classes contain inside a great deal of other Processing classes. Let us enumerate them and their basic functionality.

Inside the SMSAnalysis we have:

SpectralAnalysis: Performs an STFT of the sound. For doing so, it holds a number of Processing objects inside, namely a WindowGenerator, an AudioMultiplier, a CircularShift (for zero-phase buffer centering) and an FFT. Note that the SMSAnalysis has two instances of this class: one for the sinusoidal component and another one for the residual. This Processing Composite is quite complex in itself but we won't go into details.

SpectralPeakDetect: Implements an algorithm for choosing the spectral peaks out of the previously computed spectrum.

FundFreqDetect: Processing for computing the fundamental frequency.

SinTracking: This Processing performs sinusoidal tracking or peak continuation from one frame to the next one. It implements an inharmonic and harmonic version of the algorithm.

SynthSineSpectrum: Once we have analyzed the sinusoidal component and we have the continued peaks we have to synthesize it back to spectrum in order to compute the residual component. This is the Processing in charge of this synthesis of the sinusoidal component.

SpectrumSubstracter2: Once we have the sinusoidal synthesized spectrum and the original one (coming out from the residual Spectral Analysis), we can subtract them in order to obtain the residual spectrum.

The SMSSynthesis Processing Composite contains:

PhaseManagement: This Processing is in charge of managing phase of spectral peaks, from one frame to the next one.

SynthSineSpectrum: As already commented in the SMSAnalysis, this object is in charge of creating a synthetic spectrum out of the array of spectral peaks.

SpectrumAdder2: Is used to add the spectrum of the residual and the synthesized spectrum.

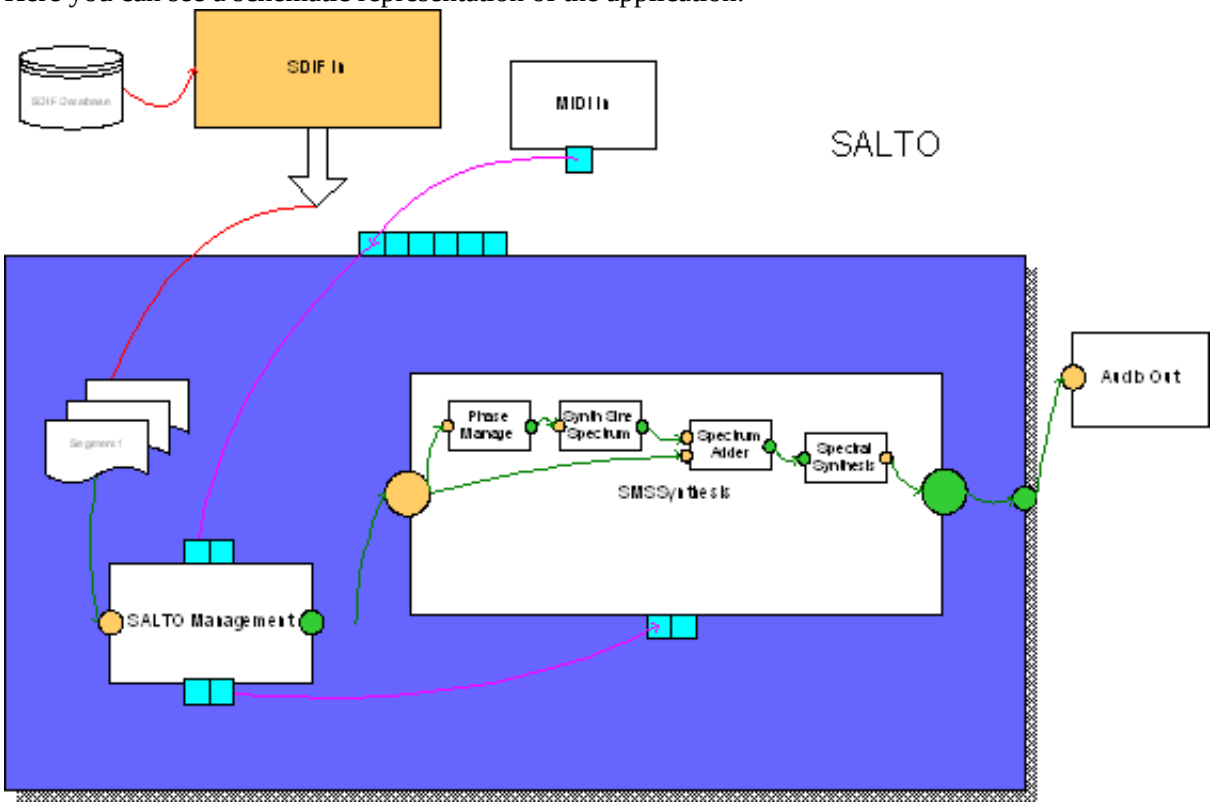
SpectralSynthesis: This processing composite implements the inverse STFT. That is, is the object in charge of computing an output audio frame from an input spectrum.The SMSSynthesis class has three instances of this class: one for the global output sound, one for the residual and one for the sinusoidal component. The SpectralSynthesis Processing Composite has the following processing inside: an IFFT, two WindowGenerators (one for the inverse Analysis window and one for the Synthesis Triangular window), an AudioProduct to actually perform the windowing, a CircularShit to undo the CircularShift introduced in the Analyisis and an OverlapAdd object to finally apply this

process to the output windowed audio frames. As you can see, it is fairly complex in itself and we would need to go into too many signal processing details in order to explain it completely.

So, that is about all. If you feel you still need to know more about the application you can read the above mentioned classes. If you think you'd rather learn more about signal processing details you can browse through the MTG bibliography.

XXIV SALTO

SALTO is an SMS synthesizer that is designed to synthesize high quality Sax and Trumpet sounds. Here you can see a schematic representation of the application:

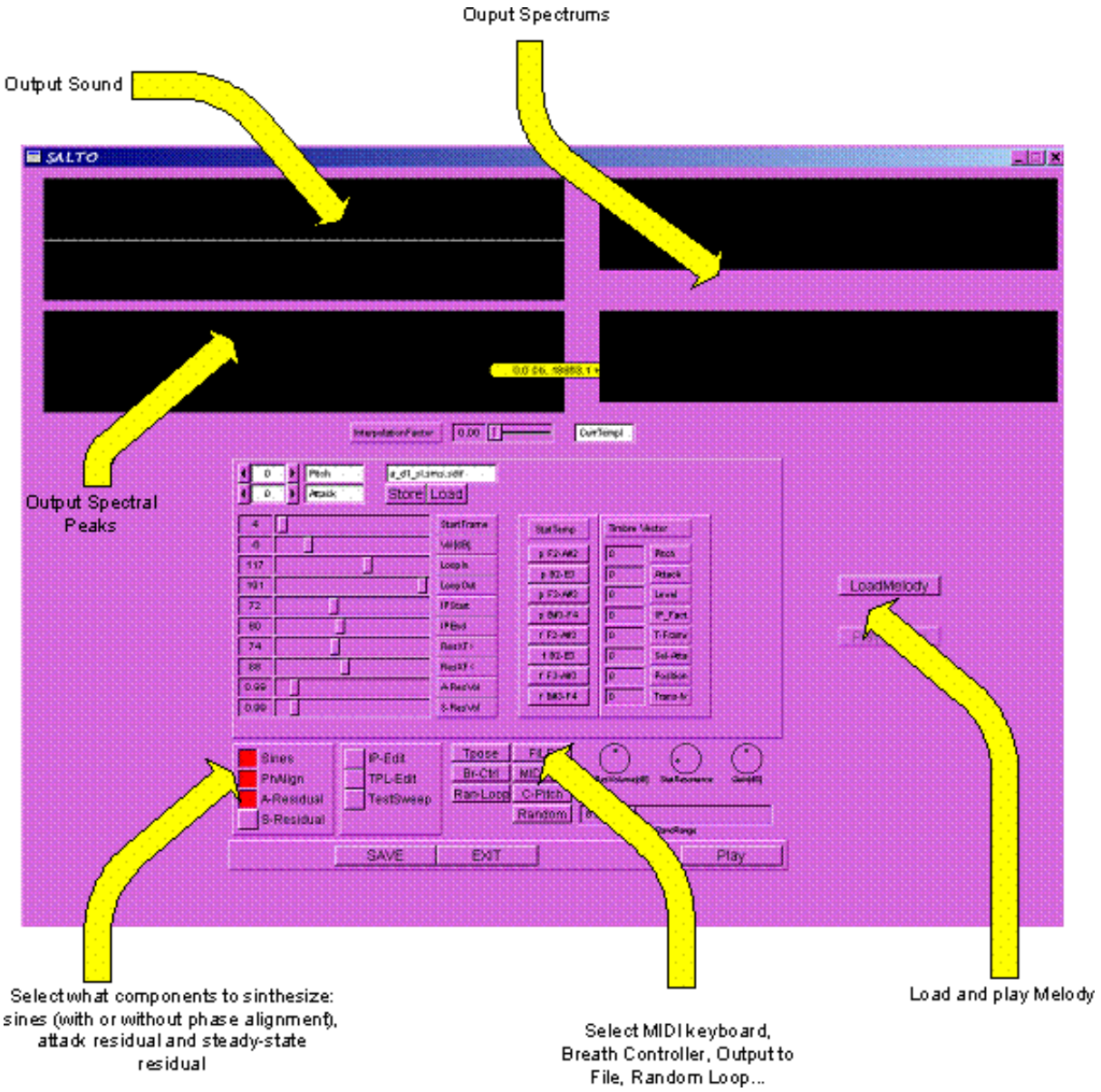


In order to make it work you need to download the SALTO database from the CLAM [i5] website (www.iua.upf.es/mtg/clam). This database is a set of SDIF files that contain the result of previous SMS analysis. To put it in short, these SDIF files contain spectral analysis samples for the steady part of some notes, the residual and the attack part of the notes. If you want to look at these files you can use the SDIF Display application included with CLAM. If you want to synthesize them one by one and even add some transformations, you can use the Analysis/Synthesis Example application.

Apart from this SDIF input, SALTO has three other inputs: MIDI, an XML Melody, and the GUI. Using MIDI as an input you can use SALTO as a regular MIDI synthesizer on real-time. SALTO is prepared to accept incoming MIDI messages coming from a regular MIDI keyboard or a MIDI breathcontroller.

If you use an XML melody as an input you will be able to synthesize it back. It is the easiest way to try that SALTO is working correctly. You can use the melody included in the CLAM repository, write your own one following the same structure or use the Analysis/Synthesis Example application to generate one from an input sound.

Finally, the GUI is still under development and can be basically used to control the way the synthesis is going to work.

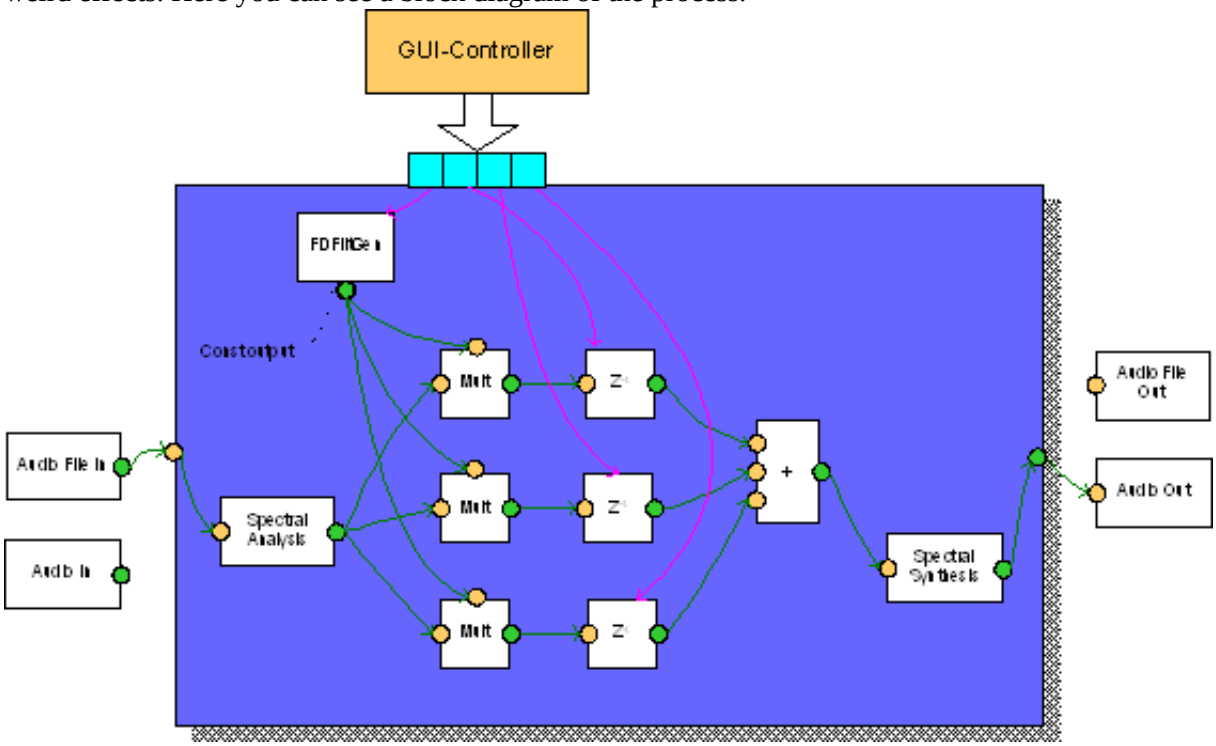


As seen in the previous screenshot, the most important part of the interface is on the lower left: the buttons to select what part of the sound you would like to synthesize. The upper part of the interface is just a graphical display of the output. On the right there are two buttons for loading and playing an xml melody. Finally, the central part is designed to manage the database but is not functional at this moment.

XXV Spectral Delay

SpectralDelay is also known as CLAM's Dummy Test. In this application it is no important to actually implement an impressive application but rather to show what can be accomplished using the CLAM framework. Especial care has been taken on the way things are done and why they are done.

The SpectralDelay implements a delay in the spectral domain. What that basically means is that you can divide your input audio signal in three bands and delay them separately, obtaining interesting or weird effects. Here you can see a block diagram of the process.



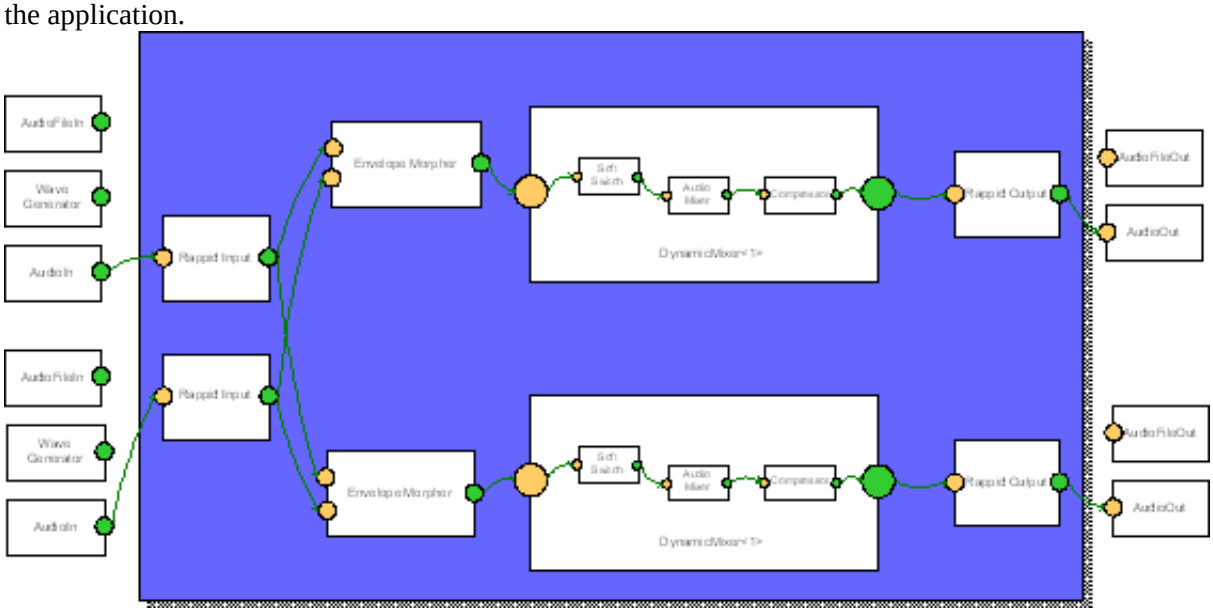
As you can see, the basic of the processing is an STFT (Spectral Analysis) that performs the analysis of the input signal and converts it to the spectral domain. This processing composite contains a bunch of other processing (i.e. WindowGenerator or FFT, see AnalysisSynthesis UML diagram). The signal is synthesized using a SpectralSynthesis Processing that implements the inverse process. It is thus transformed, in between these two steps, in the spectral domain.

The output data of the SpectralAnalysis is read by three AudioMultiplier Processings that also take as input the spectral transform function of a pre-defined filter. As a matter of fact we apply three different filters: a low-pass, a band-pass and a high-pass. We then have the signal divided into three different bands. Each of them is delayed with a different delay time. Finally, and before the synthesis, these three bands are summed up again.

The graphical interface controls the frequency cut-offs and gains of the filters and the delay times of the delays.

XXVI Rappid

Rappid is a testing workbench for the CLAM framework in high demanding situations. The first version of Rappid implements a quite simple time-domain amplitude modulation algorithm. Any other CLAM based algorithm, though, can be used in its place. Next picture illustrate the basic diagrams of the application.



The most interesting thing about Rappid is the way that multithreading issues are handled, using a watchdog mechanism. The current implementation works only under GNU/Linux.

Rappid has been tested in a live-concert situation. Gabriel Brinic used Rappid as a essential part of his composition for harp, viola and tape, presented at the Multiphonies 2002 cicle of concerts in Paris.

XXVII Combining CLAM with LADSPA plugins

117 The LADSPA Toolkit and CLAM, a brief introduction

LADSPA (Linux Audio Developer's Simple Plugin) is a sound API that aims at giving an easy interface to program audio plugins for GNU/Linux. These plugins are designed to be used as shared objects inside any host programs that knows its interface.

The API is written in C, although there are some wrappers to C++ (like the **Computer Music Toolkit** by Richard Furse). In this way,it allows to use LADSPA plugins inside C or C++ host applications.

Seeing its purpose, is logical to think about the possibility of using **CLAM** to develop **LADSPA** plugins.

118 Using CLAM Processings as LADSPA plugins

The only code you need to implement in order to create a ladspa shared object using CLAM are these lines:

```
#include "Oscillator.hxx"
#include "LADSPA_ProcessingBridge.hxx"

LADSPABridge* Instance(void)
{
        int id = 3000;
        return LADSPABridgeTmpl<CLAM::SimpleOscillator>::Create(id);
}
```

As you can see, we declare a function called Instance that returns a pointer to an object of the class LADSPABridge. This class is, as its name says, a bridge between CLAM C++ Processing Objects and LADSPA C Plugins Instances. You need to call the "Create" method of this class templatized by the processing you want to use as LADSPA plugin. Next stage is dedicated to create the settings.cfg of the project, in order to create a shared object. To do it, just put the next line in settings.cfg:

```
IS_LIBRARY = 1
```

The other lines of the config file are the same as any other CLAM project files. You can see some examples of them in build/Examples/CLAM-LADSPA_Toolkit/ (the source is in examples/CLT/).

119 Using LADSPA Plugins as CLAM Processings

There is also the possibility of using LADSPA shared objects inside CLAM, as if they were normal Processings. In order to acomplish this task, we provide the user with a CLAM Processing class called LadspaLoader. This processing has a Filename type attribute in its configuration, where you can assign which shared object you want to use. Another attribute is the index of the plugin ladspa you want to load from this shared object. After setting the correct values to its config, the LadspaLoader will create a correct interface for this plugin (with its ports and controls). This processing only works with a supervised version of Do (without parameters and using ports), and will use the algorithm encapsulated in the plugin to process its input.

MIGRATION GUIDELINES

This section contains a summary of things to change in your code when upgrading CLAM. For a detailed change log see the ChangeLog.txt file on the tarball. Please, if upgrading your code implies more changes than the ones below, warn the CLAM team to update the document (or fix the bug).

120 From 0.6.1 to 0.7.0

- Visualc++ 6 is no longer supported. Upgrade to VisualStudio 7.1. Note that VS 7.0 is neither supported.
- **Processing classes**
  - Processing classes that uses in/out ports must `#include "InPort.hxx"` and `"OutPort.hxx"` instead of `"InPortTmpl.hxx"` and `"OutPortTmpl.hxx"`
  - Processing classes that had in/out audio ports attributes declared like this: `InPortTmpl<Audio> mIn;` or `OutPortTmpl<Audio> mOut;` now must change to `AudioInPort in;` or `AudioOutPort out`, and thereof must `#include "AudioInPort.hxx"` or `"AudioOutPort.hxx"`
  - In the processing sub-classes constructor:
    Till now, a processing sub-class constructor called the in/out port constructor with 3 parameters like this: `mInPort("name", this, 1)` where the third argument was the port window size. Now this third parameter has been removed. Just declare the it like this: `mInPort("name, this);`
- **Ports linking interface**
  The ports method `AttachData` has been deprecated. Now ports can only be linked with ports (always: in -> out) and not with data. A brief rationale about this is that ports are prepared to deal with streams of tokens (any data type), so they do things like: mapping a window of the stream, consuming from the stream and producing to the stream. On the other hand if the user wants to make his data objects explicit, he can use the processings `Do` with parameters, thus reserving the ports interface for managing the processing data automatically.
- **AudioFile class**
  Change in the public interface. Now is a lot clearer and higher level.
  - `SetHeader()` depracated (made private)
  - Now use `OpenExisting(filename)` for reading a file
  - Or `CreateNew( filename, outputFileHeader )` for writing a new file
- **Low-level descriptors**
  Because the review and reimplementation of most of the low level descriptors (AudioDescriptors, SpectralDescriptors, SpectralPeakDescriptors, FrameDescriptors...) you should be very cautious if you rely on the computed values being the same. Most changes are related to:
  - Conformance to the formula on the literature
  - Normalization to meaningfull/comparable units
  - Singularities solving

  Those modifications are reflected on the doxygen documentation. Some other descriptors have been renamed or removed:
  - `SpectralDescriptors::Skewness` has been renamed as `MagnitudeSkewness`, reserving the former name for the real MPEG7 one (to be implemented).
  - `SpectralDescriptors::Kurtosis` has been renamed as `MagnitudeKurtosis` reserving the former name for the real MPEG7 one (to be implemented).
  - `SpectralTilt` and `HarmonicTilt` have been droped because the results were no reliable

(very unstable, numerically).
  - Removed uncomputed AudioDescriptors: `Attack`, `Sustain`, `Release` and `Decay`.
  - Removed uncomputed SpectralDescriptors: `StrongPeak`, `Irregularity`, `BandEnergy`, `HFC`
- CLAM big examples (SMSTools, NetworkEditor, SpectralDelay and Salto) now are apart from the main repository
  So in the strange case that your code does an #include of some header of the named examples it will not compile. Fixing this is easy: download the "used" example and add entries into the include dirs var of your `settings.cfg` file.

121 From 0.5.5 to 0.5.6

- XML related changes
  - Update the xerces version up to 2.3 (2.4 is not supported at this time)
  - Component::StoreOn methods is now const. Every definition of this method in subclasses must change. Be carefull not to forget any definition, since some classes like processings, have the pure virtuality fullfilled on its base class and will not complaint if your definition is wrong.
  - XMLStorage::Dump and XMLStorage::Restore are static methods now. There is no need to instantiate an XMLStorage object. Using them in a non-static way will work but your will be instantiating two XMLStorages. See the updated documentation.
  - XMLStorage methods Store and Load receive references to XML adapters so every time you see:

```
storage.Store(&adapter);
// or
storage.Load(&adapter);
```

    ...must be substituted by:

```
storage.Store(adapter);
// or
storage.Load(adapter);
```

- The way of doing LADSPA has been simplified a LOT. Take a look to the updated documentation.
- Processing and ProcessingConfig interface.
  - The mandatory ProcessingConfig attribute Name is not useful anymore, so you don’t need to declare it. If your code use SetName/GetName from CLAM Processings, please remove it in order to compile because they are not declared now.
  - The interface to access Ports/Controls of a Processing has been changed. You can get a reference to the needed Port or Control by name with the methods Get(In/Out)(Ports/Controls).Get(name), or by number with Get(In/Out)(Ports/Controls).GetByNumber(id). Some examples: GetInControls().Get("first in control"); GetOutPorts().GetByNumber(1);...
- Controls linking interface. The free functions related to connecting controls have been suppresed, because it was redundant interface. Each control have methods to do so, like AddLink and RemoveLink.

122 From 0.5.4 to 0.5.5

- TODO: Explain needed changes for BasicOps and Stats

123 From 0.4.2 to 0.5.0

- Remember that CLAM 0.5.0 now uses fltk 1.1.4RC1 or greater
- You should port your projects based on modules files (GNU/UNIX) or .dsp (VisualC/Windows). Its very easy, just read the build system chapter.
- If you used `roundInt`, you should use `Round` now.
- Snapshots are now called Plots, see Plots.hxx header.

124 From 0.2 to 0.3

124.1 Dynamic Types

1. **The new style of macros**. It you need to migrate from the old macros style, follow these guidelines:

- Change the macro names, this is:
  - REG_NUM_ATTR -> DYN_CLASS_TABLE
  - REG_NUM_ATTR_USING_INTERFACE
  - DYN_CLASS_TABLE_USING_INTERFACE
  - REGISTER -> DYN_ATTRIBUTE
  - REGISTER_PTR -> Not Present, warn us
  - Not present before -> DYN_CONTAINER_ATTRIBUTE

  Add the access specifier for each dyn attribute as first parameter of the `DYN_ATTRIBUTE` macros.

- Swap the parameter positions between attribute type and name
- Place an access specifier after the macros that will assure that the following members will have the proper access (public, private, protected)

2. **UpdateData() is safe and efficient:** it never throws exceptions and returns a bool telling whether it has done any memory update. Also, is efficient, in the sense of not traversing the attributes information but consulting a global modified flag.

3. **AddXXX and RemoveXXX are also safe:** this means that they don't throw exceptions even. If you add an already added attribute, just does nothing. Furthermore, if you remove an existing attribute and later you add it again without having done an update data, it will just have a null effect: the attribute will remain untouched.

Dynamic attributes declarations are *Typedef* safe: Now we can declare dynamic attributes with customized alias types (i.e. typedef int `TInt`) and they will also be supported by the XML store and load.

4. **Support for pointers as dynamic attributes has been eliminated.**

124.2 Processing Data

If you are already familiar with CLAM's library (formerly MTG-Classes) this are the basic things you should be aware of:

- DynamicTypes interface has changed: names of initialization methods and register macros have been updated (you can find more information on this topic in previous section, section 49 section 50 for PD usage of DT and in all sections about Dynamic Types).

- The use of configurations in PD classes is not mandatory and not even recommended (please read through section 52). Only Spectrum preserves its configuration.

124.3 Error handling

Until the publication of this document other error reporting mechanisms have been used, not in a very systematic way. There is a list of deprecated practices and their alternatives.

124.4 PARANOID macro

The use of PARANOID macro is deprecated. You must use the above mechanisms instead mainly assertions and checks.

124.5 Using exceptions as error message generators

Before this document was published the code was full of exceptions used to give the developer an error messages like in:

```
if (someWeirdCondition)
    throw MyException("This error has happend");
```

This may be correct if the error is published and clients are able to catch the exception and handling it. But in most cases this is only a way of reporting the developer that an implementation error has ocurred. Then asserts are more practical and useful because they keep the stack on the correct place on debug mode and throws an exception (or does nothing) on release.

```
CLAM_ASSERT(!someWeirdCondition,"This error has happend");
```

If it is enbraced in a `#ifdef PARANOID` or some other compiler mode dependant structure it is very sure to be the wrong way of doing. Use asserts and check statements.

As a general rule when code Exceptions objects are used to report an error message, and the exception can not be handled by the client code, they must be changed onto assertions.

124.6 Using _DEBUG, NDEBUG and so

Because the use of those macros is not standard along the differents C++ implementations, its use is centralized on the Assert files. If assertions and exceptions does not fullfill your needs consider the proposal of creating a new structure.

124.7 Miscellaneous

The name of the *namespace* has been changed to CLAM. Compile flags have also been changed so you should check your private workspaces or makefiles (more information about flags in chapter XXI).

The classes called ’ProcessingObject’ in previous releases have been renamed to ’Processing’ in this release (the reason is a bit philosophycal so we won’t go into details here).

Apart from this, a number of other ’details’ may have changed form previous releases so you are encouraged to take a look at the main document, specially those parts that mostly relate to your work with CLAM. Anyhow, this is the first time we are able to offer you so complete documentation so we are sure that it will help clarifying doubts about CLAM that you have had since your first contact with the framework.