Add new modules in the CASToR code

September 18, 2020

Foreword

CASToR is designed to be flexible, but also as generic as possible. Any new implementation should be thought to be usable in as many contexts as possible; among all modalities, all types of data, all types of algorithms, etc.

Before adding some new code to CASToR, it is highly recommended to read the general documentation *CASToR_general_documentation.pdf* to get a good overview of the project, as well as the programming guidelines *CASToR__programming_guidelines.pdf*. The doxygen documentation is also a very good resource to help understanding the code architecture.

1 Summary

This HowTo guide describes the philosophy of the modular organization of the CASToR code. It explains how it is built and how to add your own modules by simply observing a few rules and implementing a few mandatory functions.

Adding a module can be done in any part of the code where a manager class makes use of an abstract class (prefixed by the letter 'v', *e.g. vProjector*). To add your own module, the idea is to build a specific class that inherits from this abstract class. In this abstract class, a few pure virtual functions are declared and used in the rest of the code. These pure virtual functions are empty and have to be implemented by your specific class defining your own module.

To make things even easier, we provide an example of template class associated to each abstract class that already implements the whole squeleton. Basically, you will have to change the name of the class and fill the functions up with your own code. Considering an abstract class *vClass* located in a thematic sub-folder *theme*, the actual template files will be *include/theme/iClassTemplate.hh* and *src/theme/iClassTemplate.cc*. These files are actually already part of the source code and can be used as a starting point for your own development.

Right below are the instructions to help you fill the mandatory part of these template files according to the CASToR architecture. As a matter of illustration, the example of creating a new projector is used throughout this guide. It can be replaced by any type of module as the philosophy is the same in the whole CASToR project: *e.g.* projectors, optimizers, convolvers, processing, etc.

2 Constructor and destructor

As specified in the *CASToR_HowTo__programming_guidelines.pdf*, using parameters in a constructor or destructor is forbidden. The aim of the constructor is strictly to set a default value to all members. All parameters will be set using a different mechanism explained below. The destructor should only destroy what was constructed by the class, still checking the validity of any pointer before calling a *delete* or *free* function.

3 Auto-inclusion mechanism

When you want to add a new module in CASToR, you only have to implement the associated class inheriting from the corresponding abstract class. In order to get this new class operating in the CASToR code, there is an automatic mechanism that do the trick for you. What you have to do is adding two mandatory lines of code in the header file associated to your own class. The macro definitions associated to these lines can be found at the end of the abstract class header (*vProjector.hh* for example in the case of a projector). It makes use of the sAddonManager class that manages the modules, as the name suggests (can be found in the *management* sub-folder). It is inspired by the same mechanism used in the GATE simulation software (www.opengatecollaboration.org).

The first line has to be added inside the corpus of the class definition in the header file, as a public declaration, as shown in the sample code below. You simply have to change the name of the macro function by the one used inside the abstract class, and change the parameter of the macro function by the name of your class. As a matter of organization, keep the name of the abstract class as a suffix (replacing the first letter 'v' by 'i') and add the suffix of your choice. Here is an example for the case of a projector.

Listing 1: The function that creates the projector.

```
1 public:
2   FUNCTION_PROJECTOR( iProjectorYourName )
```

The second line has to be added outside of the corpus of the class definition in the header file, as shown in the sample code below. This is where the association of your class with a parameter's name (to call it from the command line) is made. You simply have to change the first parameter with the command line name of your projector, and the second parameter by the actual name of your class.

Listing 2: The singleton class that uses the maker function to create the projector.

```
1 CLASS_PROJECTOR( myProjector, iProjectorYourName )
```

Be sure not to add semi-colon at the end of those lines.

4 Mandatory functions to be implemented

In order to try to keep the code as robust as possible, the addition of modules in CASToR has to follow some pre-defined rules. To constrain the user/developer to respect these rules and make the code robust, abstract classes with pure virtual functions are used. This means that if those functions are not implemented within the module inheriting from the abstract class, the compiler will crash and ask you to implement these functions. These pure virtual functions can be separated in three groups: interface-related functions, management functions and specific functions. Below is a description of the functions included into these three groups.

4.1 Interface-related functions

CASToR is designed to be user-friendly (understand convenient command-line options with detailed command-line help). For a given family of modules (*e.g.* the projectors), the choice of the module (*i.e.* the actual projector) is made through the use of a command-line option dedicated to this family (*-proj* for the projectors), followed by the name of the chosen module (*i.e. myProjector* in this example). To customize the parameters peculiar to the module, CASToR offers two possibilities that both should be implemented: read parameters from a configuration file, or read parameters from a list. One additional function should provide some help on to how to use this projector

and provide its associated parameters. There are three possible ways to call a module through the command-line options (example for the projector):

-proj myProjector : No argument is given, then the default configuration file while be used to get the values associated to the specific parameters. The default configuration file has to be located inside the *projector* sub-folder of the CASToR configuration folder. Its name has to be *myProjector.conf*. By construction, a default configuration file is mandatory, even without parameter, just leave an empty file.

-proj myProjector:path/to/configuration/file.conf : In this case, the configuration file provided along with the chosen projector will be used instead of the default configuration file.

-proj myProjector,parameter1,parameter2,...,parameterN : In this case, the specific parameters are directly read from the provided list of parameters. No configuration will be read. The order of the provided parameters and their syntax must be in agreement with the help provided by the mandatory *ShowHelp()* function (see below).

For families of modules that can be used in different parts of the program, the option is completed by the sign *::* followed by a list of keywords describing when to apply the module, separated by commas. As an example, here is how to write the option to use a stationary Gaussian image convolver within iterations as well as a post-processing, with a transaxial FWHM of 3mm, an axial FWHM of 4mm and 3.5 sigmas included in the convolution kernel:

Listing 3: A command line option to use a gaussian convolver within iterations as well as a post-processing, with a transaxial FWHM of 3mm, an axial FWHM of 4mm and 3.5 sigmas included in the convolution kernel.

```
1 -conv gaussian,3.,4.,3.5::intra,post
```

The reading of a configuration file associated to the module is made by the following function:

Listing 4: Function to read a configuration file for your module.

```
1 int ReadConfigurationFile(const string& a_configurationFile);
```

A configuration file is basically a text file containing on each line, the name of a parameter, a colon (*i.e. ':'*), and the value of the parameter. To ease the implementation, some functions designed to read parameters from a text file or a string are already implemented within the *include/management/gOptions.hh* file. If you want to see examples on how to use these functions, have a look at some other modules already implemented. The parameters' name read into the configuration file have to be consistent with the help provided by the *ShowHelp()* function (see below). This function returns 0 if no issue was encountered and a positive value otherwise, so that errors can be caught.

The reading of the parameters through a list is made by the following function:

Listing 5: Function to read a list of options for your module.

```
1 int ReadOptionsList(const string& a_optionsList);
```

All parameters' values are separated by commas. To ease the implementation, some functions designed to read parameters from a text file or a string are already implemented within the *include/management/gOptions.hh* file. If you want to see examples on how to use these functions, have a look at some other projectors already implemented. The parameters' values read into the list and their order have to be consistent with the help provided by the *ShowHelp()* function (see below). This function returns 0 if no problem was encountered and a positive value otherwise, so that errors can be caught.

The function that provides some help about how to use your module (when using the *-help-proj* option in the case of projectors) is the following:

Listing 6: Function to give specific help on how to use your module.

```
1 void ShowHelpSpecific();
```

or sometimes simply:

Listing 7: Function to give specific help on how to use your module.

```
1 void ShowHelp();
```

This function should provide some clear and detailed description and instructions about what your module does and how to use it through its specific parameters. The list of all parameters should be detailed and well documented. Their order when reading a parameters list from the *ReadOptionsList()* function should also be clear. If the method you use has already been published, the relevant references must be provided. This function is called when using the help option associated to the family of modules in the command-line (*-help-proj* in the case of projectors). More precisely, the *sAddonManager* will ask all modules to describe themselves through their specific *ShowHelp()* or *ShowHelpSpecific()* functions.

4.2 Management functions

CASToR makes its best to keep a standardized and robust code. For all modules, once the object is constructed and the parameters are set, one has to check the parameters' values and initialize what needs to be initialized. To do that, two pure virtual functions have to be implemented.

First, a function to check all parameters pecular to the module must be implemented. This function must return 0 if no problem was encountered and a positive value otherwise, so that errors can be caught. It has the following definition:

Listing 8: Function to check the value of all parameters pecular to your module.

```
1 int CheckSpecificParameters();
```

Second, a function to initialize all specific parameter that has to be initialized in order to be usable. This function must also return 0 if no problem was encountered and a positive value otherwise, so that errors can be caught. It has the following definition:

Listing 9: Function to initialize all pecular stuff to your module.

```
1 int InitializeSpecific();
```

4.2.1 Specialized functions

The specialized functions are those whose purpose is specific to the family the module belongs to. These functions are also pure virtual and their definitions can be found in the abstract class associated to the module family. For any module family, a dedicated guide can be found that describes the purpose of these functions and the tools to implement them. The doxygen language is also used to directly document the code, *i.e.* purpose and aim of each class/function. So please refer to these sources of documentation.