The **Black Box Toolkit**

Package Developers' Guide

`bbtk` version 0.9.0 (15/10/2008)

Generated on : November 12, 2008

Eduardo Dávila, Laurent Guigues, Jean-Pierre Roux

CREATIS-LRMN, Centre de Recherche en Imagerie Médicale
CNRS UMR 5220, INSERM U620, INSA Lyon, Université Claude-Bernard Lyon 1
http://www.creatis.insa-lyon.fr/creatools/bbtk

Contents

List of Tables

List of Figures

1 Introduction

This guide describes how to create new bbtk packages and black boxes. How to use them is described in bbtk Users' guide.

Any black box must be included in a bbtk package, that is in a particular shared library which can be loaded dynamically by bbtk , either in C++ code or in bbs scripts with the commands include or load. The steps to create new boxes are thus to :

1. **Create a new package.** This is described in section 2.
2. **Describe your new box.** You can do it either :
   - In C++ code. You will have to write the class for your box, mostly using bbtk macros.
   - In xml code. When configuring your project with cmake , the utility bbfy will then generate the corresponding C++ code.

   This is described in section 3.

2 Creating a new package

2.1 Creating the file tree

Before defining any black box you have to create a package, or more precisely the source files which will allow you to generate the package (compile and link the shared library) and may be install it.

The bbtk command line application bbCreatePackage allows to create the basic file architecture to start the development of a new black box package. Type bbCreatePackage in a console to get its usage :

```
bbCreatePackage <package-path> <package-name> [author] [description]
```

bbStudio also offers a graphical interface to the bbCreatePackage application. You can run it with the menu Tools > Create Package.

In both cases (using the command line tool or bbStudio interface), you have to choose :

- The **directory** of your new package. Two cases occur :
  - The black boxes you want to create are based on a processing code (C++ classes or C functions) which is in an existing project handled by cmake and you want the new package to be part of your existing project. You will have to create your new package into the source tree of your project and add a SUBDIRS command in the CMakeLists.txt file of the parent directory of your package.

  - You do not have an already existing project (you want to create the new boxes from scratch) or you want/are imposed that the existing project remain external to the package project. You will have to create your new package in a new location and may be include/link against existing libraries.

- The **name** of your new package. This name will be used to load the package in `C++` and `bbs` scripts.

You must also provide the `author` list and a `description` which will be used for your package documentation.

After running `bbCreatePackage` or clicking 'Run' in `bbStudio` interface you should get a file structure like this (Linux users can verify it with the `tree` command):

```
NEW_PACKAGE
|-- CMakeLists.txt
|-- Configure.cmake
|-- PackageConfig.cmake.in
|-- README.txt
|-- UsePackage.cmake.in
|-- bbs
|   |-- CMakeLists.txt
|   |-- appli
|   |   `-- README.txt
|   `-- boxes
|       `-- README.txt
|-- data
|   `-- CMakeLists.txt
|-- doc
|   |-- CMakeLists.txt
|   |-- bbdoc
|   |   |-- CMakeLists.txt
|   |   `-- header.html.in
|   `-- doxygen
|       |-- CMakeLists.txt
|       |-- DoxyMainPage.txt.in
|       `-- Doxyfile.txt.in
`-- src
    `-- CMakeLists.txt
```

You can then :

- Edit the root CMakeLists.txt file to customize your package build settings (see 2.2 below)

- Put your c++/xml boxes sources in 'src'.

  Please use the convention : If the name of your package is Pack and the name of your box is Box then name the source files bbPackBox.{h;cxx;xml}.

- Put your script-defined boxes (complex boxes) in 'bbs/boxes'.

  Please use the convention : If the name of your box is 'Box' then call the file 'bbBox.bbs' to let others know that the script defines a complex black box type.

- Put your script-defined applications in 'bbs/appli'.

  Please use the convention : Do not prepend 'bb' to the files.

- Put your data in 'data'. Any data put there will be installed and accessible in your scripts : the package data path is provided by the box `std::PrependPackageDataPath`.

- You can customize the header of your package html doc by editing the file 'doc/bbdoc/header.html.in'. You must put html code in this file (or edit it with an html editor). You can include images or links to other html pages. The images and pages must be put in the folder 'doc/bbdoc' and will be properly installed. The same way, you can link to these images or pages in your boxes descriptions without giving any path. If you create subdirs for your material then you have to install the materials yourself by editing the CMakeLists.txt and links must use paths relative to 'doc/bbdoc'.

- You can customize the main page of your doxygen doc by editing the file 'doc/doxygen/DoxyMainPage.txt.in'.

2.2 Configuring the root `CMakeLists.txt`

First you must configure your new package build settings, by editing the file `CMakeLists.txt` in the package root directory. This file contains :

File CMakeLists.txt

```
#============================================================================
# CMAKE SETTINGS FOR BUILDING A BBTK PACKAGE
#============================================================================

#============================================================================
# THE NAME OF THE BBTK PACKAGE
SET(BBTK_PACKAGE_NAME MyPackage)
#============================================================================

#============================================================================
# IF IT IS A STANDALONE PROJECT UNCOMMENT NEXT LINE TO DECLARE YOUR PROJECT
# PROJECT(bb${BBTK_PACKAGE_NAME})
```

```
#============================================================================

#============================================================================
# PACKAGE AUTHOR
# !!! NO COMMA ALLOWED !!!
SET(${BBTK_PACKAGE_NAME}_AUTHOR "myself")
#============================================================================

#============================================================================
# PACKAGE DESCRIPTION
SET(${BBTK_PACKAGE_NAME}_DESCRIPTION "The kinkiest stuff you ve ever seen.")
#============================================================================

#============================================================================
# PACKAGE VERSION NUMBER
SET(${BBTK_PACKAGE_NAME}_MAJOR_VERSION 1)
SET(${BBTK_PACKAGE_NAME}_MINOR_VERSION 0)
SET(${BBTK_PACKAGE_NAME}_BUILD_VERSION 0)
#============================================================================

#============================================================================
# UNCOMMENT EACH LIBRARY NEEDED (WILL BE FOUND AND USED AUTOMATICALLY)
# SET(${BBTK_PACKAGE_NAME}_USE_VTK  ON)
# SET(${BBTK_PACKAGE_NAME}_USE_ITK  ON)
# SET(${BBTK_PACKAGE_NAME}_USE_GDCM ON)
# SET(${BBTK_PACKAGE_NAME}_USE_GSMIS ON)
# SET(${BBTK_PACKAGE_NAME}_USE_WXWIDGETS ON)
#============================================================================

#============================================================================
# LIST HERE THE OTHER bbtk PACKAGES NEEDED
# (WILL BE FOUND AND USED AUTOMATICALLY)
SET(${BBTK_PACKAGE_NAME}_USE_PACKAGES
  # std
  # wx
  # itk
  # vtk
  # ...
  )
#============================================================================

#============================================================================
# THE SOURCES OF THE PACKAGE
# EITHER UNCOMMENT NEXT LINE TO COMPILE ALL .cxx OF THE src DIRECTORY :
SET(${BBTK_PACKAGE_NAME}_COMPILE_ALL_CXX ON)
# ... OR LIST THE FILES TO COMPILE MANUALLY :
#SET(${BBTK_PACKAGE_NAME}_SOURCES
# LIST HERE THE FILES TO COMPILE TO BUILD THE LIB
# E.G. TO COMPILE "toto.cxx" ADD "toto" (NO EXTENSION)
```

```
# THE PATH MUST BE RELATIVE TO THE src FOLDER
#    )
#============================================================================

#============================================================================
# THE xml SOURCES OF THE PACKAGE
# EITHER UNCOMMENT NEXT LINE TO bbfy ALL .xml OF THE src DIRECTORY :
SET(${BBTK_PACKAGE_NAME}_COMPILE_ALL_XML ON)
# ... OR LIST THE FILES TO COMPILE MANUALLY :
#SET(${BBTK_PACKAGE_NAME}_XML_SOURCES
# LIST HERE THE FILES TO bbfy TO BUILD THE LIB
# E.G. TO bbfy "toto.xml" ADD "toto" (NO EXTENSION)
# THE PATH MUST BE RELATIVE TO THE src FOLDER
#    )
#============================================================================

#============================================================================
# THE SCRIPT-DEFINED BOXES OF THE PACKAGE (bbs)
# EITHER UNCOMMENT NEXT LINE TO INCLUDE ALL .bbs OF THE bbs/boxes DIRECTORY :
SET(${BBTK_PACKAGE_NAME}_INCLUDE_ALL_BBS_BOXES ON)
# ... OR LIST THE FILES TO INCLUDE MANUALLY :
# SET(${BBTK_PACKAGE_NAME}_BBS_BOXES
# LIST HERE THE bbs FILES TO INCLUDE
# E.G. TO INCLUDE "boxes/bbtoto.bbs" ADD "boxes/bbtoto" (NO EXTENSION)
# !! THE PATH MUST BE RELATIVE TO THE bbs FOLDER !!
#)
#============================================================================

#============================================================================
# THE SCRIPT-DEFINED APPLICATIONS OF THE PACKAGE (bbs)
# EITHER UNCOMMENT NEXT LINE TO INCLUDE ALL .bbs OF THE bbs/appli DIRECTORY :
SET(${BBTK_PACKAGE_NAME}_INCLUDE_ALL_BBS_APPLI ON)
# ... OR LIST THE FILES TO INCLUDE MANUALLY :
# SET(${BBTK_PACKAGE_NAME}_BBS_APPLI
# LIST HERE THE bbs FILES TO INCLUDE
# E.G. TO INCLUDE "appli/testToto.bbs" ADD "appli/testToto" (NO EXTENSION)
# !! THE PATH MUST BE RELATIVE TO THE bbs FOLDER !!
#)
#============================================================================

#============================================================================
SET(${BBTK_PACKAGE_NAME}_INCLUDE_DIRS
  # LIST HERE YOUR ADDITIONAL INCLUDE DIRECTORIES
  # EXCEPT :
  #  - src
  #  - bbtk dirs
  #  - automatically handled libraries or packages : wx, vtk... (see above)
  #  - the dirs automatically set by other libraries found by FIND_PACKAGE
  )
```

```
#=============================================================================

#=============================================================================
SET(${BBTK_PACKAGE_NAME}_LIBS
  # LIST HERE THE ADDITIONAL LIBS TO LINK AGAINST
  # EXCEPT : the same libs than for INCLUDE_DIRS
  )
#=============================================================================

#=============================================================================
# IF NEEDED : UNCOMMENT NEXT LINE
# AND LIST ADDITIONNAL DIRECTORIES
# IN WHICH TO LOOK FOR LIBRARIES TO LINK AGAINST
# LINK_DIRECTORIES()
#=============================================================================

#=============================================================================
# SET TO TRUE TO HAVE INFORMATION ON LIBRARIES FOUND DURING CMAKE CONFIGURE
SET(FIND_PACKAGE_VERBOSE TRUE)
#=============================================================================

#=============================================================================
# END OF USER SECTION
#=============================================================================

#=============================================================================
# Include configuration script
INCLUDE(Configure.cmake)
#=============================================================================

#=============================================================================
# EOF
#=============================================================================
```

End of file

The comments in the file should be easily understandable ! In this file, you can see some of the informations you supplied in previous step:

- The **name** of your package. This will be the name used to load it in `bbi` . The shared library however will be called `bb`name hence on *Linux* the object file will be called `libbb`name`.so` and on *Windows* it will be called `bb`name`.dll`.
- The **author(s)** of the package. Preferably provide e-mail adresses.
- A **description** of the package, which will appear in the help of your package or in its html documentation automatically generated by `bbtk` .

You can additionaly set :

- The **version** of the package.
- The **libraries used** by the package : `vtk` , `itk` , `gdcm` , `gsmis` , `wxWidgets` . The mecanisms to find these libraries, their sources and to link against them are automatically handled by the `cmake` files installed by `bbCreatePackage` . You just have to uncomment a line to use one of these libraries.
- The **core `bbtk` packages used** by the package as `C++` libraries (if you need to use the black boxes of these packages in your `C++` code, i.e. include some header and link with the library). The mecanisms to find these libraries, their sources and to link against them are automatically handled by the `cmake` files installed by `bbCreatePackage` . You just have to uncomment a line to use one of these libraries.
- The **`C++` sources** of the package : you can list each input `C++` file explicitly or tell `cmake` to include in the project all the `C++` files of the 'src' directory (default).
- The **`xml` sources** of the package : you can list each input `xml` file explicitly or tell `cmake` to include in the project all the `xml` files of the 'src' directory (default).
- The **boxes `bbs` sources** of the package : you can list each input `bbs` file explicitly or tell `cmake` to include in the project *all* the `bbs` files of the 'bbs/boxes' directory (default, recommanded).
- The **appli `bbs` sources** of the package : you can list each input `bbs` file explicitly or tell `cmake` to include in the project *all* the `bbs` files of the 'bbs/appli' directory (default, recommanded).
- **Additional include directories**. Set it if your package needs to include source files which are not in the package directory, typically if it depends on another library which is not one the libraries automatically handled (`vtk` , `itk` ...) and which you did not find with the `FIND_PACKAGE` mechanism of `cmake` .
- **Additional libraries** to link against. Set it if your package needs to link against another library which is not one the libraries automatically handled (`vtk` , `itk` ...) and which you did not find with the `FIND_PACKAGE` mechanism of `cmake` .
- **Additional link directories** in which to find libraries not automatically handled and which you did not find with the `FIND_PACKAGE` mechanism of `cmake` .

Of course, this is only a framework and you can add any other `cmake` commands in the file.

3 Creating a new box

3.1 Principles

3.1.1 C++ or XML ?

There are two ways to create a new black box in an existing package :

- Write an xml description file which will be automatically translated in C++ by the bbfy application during build (recommanded).
- Write the C++ code of the box using bbtk macros.

3.1.2 From which bbtk class inherit ?

Apart from the choice of the description langage to use, there is an important choice to do concerning the implementation of the box. In C++ , a black box is nothing but a class which has the standard interface of all black boxes : what's its name ? inputs ? outputs ? and so on.

The abstract description of this interface is done in the class bbtk::BlackBox of the bbtk library and is implemented in its children classes : bbtk::AtomicBlackBox and bbtk::WxBlackBox [1].

To create a new black box, you have to inherit one of these two concrete classes in order to inherit the black box interface and a particular implementation of this interface.

If your black box is a *Widget* black box, that is a black box which has (or is) a piece of a graphical interface based on the wxWidgets library, then it must inherit the class bbtk::WxBlackBox.

Concretely, a bbtk::WxBlackBox is associated a wxWindow and must be able to return a pointer to it. If your black box is not a widget black box (that is : doesn't returns a pointer to a wxWindow), it must inherit from bbtk::AtomicBlackBox. NOTE : *modal dialogs* which are created and destroyed at the end of the process method of the box are NOT WxBlackBoxes : they do not return a wxWindow, see the code of wx::FileSelector for example.

3.1.3 Inherit or encapsulate ?

Now, your black box will do something (hopefully !). When you decide to write a new black box, you should be in one of these three cases :

1. You already have a C-like function which does the processing that you wish to 'blackboxify' (bbfy in short).
2. You already have a C++ class which does the processing that you wish to 'blackboxify'

[1] all the classes of the bbtk library are in a *namespace* called bbtk and the C++ header of a class called NameOfAClass is in the file called bbtkNameOfAClass.h

3. You start from scratch without any existing code

The idea of The **Black Box Toolkit** is to embed processing codes into `C++` objects which have a standard and generic interface - namely black boxes - to be able to chain arbitrary processes afterwards.

In `C++` , in order to embed an existing processing *class* into a standard interface you only have two possibilities :

1. **Inherit** the existing processing class *and* the interface class (e.g. `bbtk::AtomicBlackBox`). In this case you have to :
   (a) make the link between the inputs and outputs of the black box and the interface of the inherited class
   (b) call the processing method of the inherited class in the processing method of the black box.
2. **Encapsulate** the existing processing class in a class inherited from the interface class (e.g. `bbtk::AtomicBlackBox`). In this case you have to :
   (a) declare an instance of the processing class as a member of the black box,
   (b) instantiate it at the right time (either in the constructor or in the processing method of the black box)
   (c) in the processing method of the black box :
      i. set the inputs of the member processing class with the inputs of the black box,
      ii. call the processing method of the encapsulated class
      iii. set the ouputs of the black box with the outputs of the encapsulated class.

If you wish to 'blackboxify' a C-like *function*, you do not have the choice, you can only use the second mechanism, namely encapsulation.

Obviously, the inheritance mechanism is more powerfull and - when it is possible to use it - it demands less effort because, as we will see, in `bbtk` you can directly link the accessors to the input and output data of the box to the accessors of the inherited processing class, as well as the procesing method of the black box to the processing method of the inherited processing class, very much like a callback mechanism.

3.1.4 Input and output accessors

When you encapsulate a processing class or a C function or when you write down a black box from scratch, you must access the inputs and outputs of the black box, in order to interface it manually with your processing method or simply write your processing code (there are other cases in which you also need to access the inputs and outputs, we will talk about them later).

The only thing you must know about the C++ code generated from your xml or your C++ macro-based description is that when you declare an input or an output of a black box then two *accessors* for this input or output are generated : one to *get* the value of the input or output and one to *set* it. These accessors have normalized names :

- The declaration of an **input** called NAME and of type TYPE generates the two accessors [2]:
  - `void bbSetInput<NAME>(<TYPE>);`
  - `<TYPE> bbGetInput<NAME>();`
- The declaration of an **output** called NAME and of type TYPE generates the two accessors:
  - `void bbSetOutput<NAME>(<TYPE>);`
  - `<TYPE> bbGetOutput<NAME>();`

For example, declaring an input called Image would generate the two accessors bbSetInputImage and bbGetInputImage.

Note that:

- All bbtk methods are prefixed by bb to avoid conflicts with potential inherited methods.
- An input and an output can have the same name (e.g. 'Image'). No conflict between accessors occur (e.g. four distinct accessors are created : bbSetInputImage, bbGetInputImage, bbSetOutputImage and bbGetOutputImage).

3.2 Generate the black box skeleton

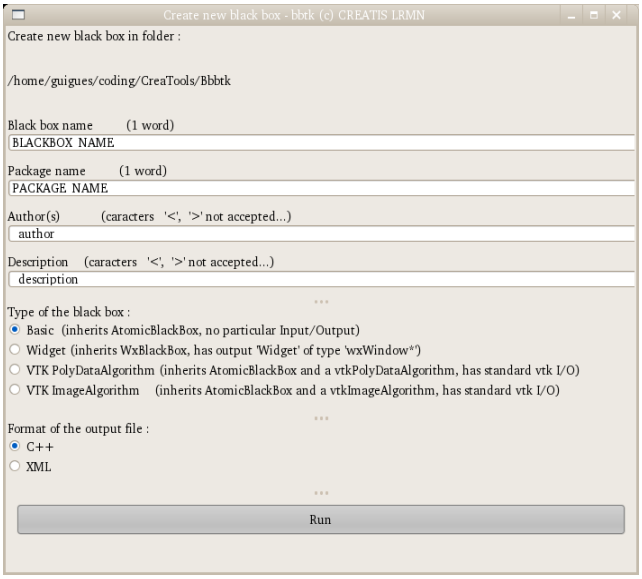
The command line application bbCreateBlackBox allows to create a skeleton C++ or xml files for a new black box. It has a rather complex usage, we recommand you use the graphical interface to it which is accessible with bbStudio menu Tools > Create black box. The interface looks like in fig. 1.

You will have to give :

1. The **name** of the box
2. The **package** to which the box belongs (can we do it automatically ? LG : think about it)
3. The **author**(s) of the box

[2] For the sake of simplicity, the parameters and return value are shown here as if they were all passed by value. However the actual code can use references. The same way, the issue of const or non const methods is eluded here. Different cases occur in practice.

Figure 1: Create Black Box interface



4. A **description** of the box
5. Its **type**, either
   (a) Basic (inherits `AtomicBlackBox`, no particular Input/Output)
   (b) Widget (inherits `WxBlackBox`, has output 'Widget' of type 'wxWindow*')
   (c) VTK PolyDataAlgorithm (inherits `AtomicBlackBox` and a vtkPolyDataAlgorithm, has standard vtk I/O)
   (d) VTK ImageAlgorithm (inherits `AtomicBlackBox` and a vtkImageAlgorithm, has standard vtk I/O)
6. The output format of the file, either a C++ file or an XML file.

3.3 `XML` description of a box

3.3.1 General `xml` tags

Let us examine the `xml` file describing the `Add` box of the `std` package :

File `packages/std/src/bbAdd.xml`

```
<?xml version="1.0" encoding="iso-8859-1"?>

<blackbox name="Add">
```

```
  <author>laurent.guigues@creatis.insa-lyon.fr </author>
  <description>Adds its inputs                 </description>
  <category>math                               </category>

  <input name="In1"  type="double" description="First number to add"/>
  <input name="In2"  type="double" description="Second number to add"/>
  <output name="Out" type="double" description="Result"/>

  <process><PRE>
    bbSetOutputOut( bbGetInputIn1() + bbGetInputIn2() );
  </PRE></process>

  <constructor><PRE>
    bbSetInputIn1(0);
    bbSetInputIn2(0);
    bbSetOutputOut(0);
  </PRE></constructor>

</blackbox>
```

End of file

The tags and their role are easily understandable.

As the box is not a widget, we inherit implicitely from `bbtk::AtomicBlackBox` (the default).

The only part of the file which needs a bit of explaination is the body of the `process` tag, which describes the actual code to execute in the box. This code must be enclosed in a `<PRE></PRE>` tag to tell the `xml` parser not to interpret it as `xml` instructions. This is necessary to be able to use any symbol, like the `<` and `>` which have a special meaning in `xml` . In the case of the `Add` box, the process code is very simple : remember that `bbGetInputIn1()` is the accessor to the input `In1` declared above and `bbGetInputIn2()` is the accessor to the input `In2`; the code simply adds the values of the two inputs and sets the output `Out` with the resulting value.

To describe your own black boxes in `xml` code, you must modify the xml file generated by `bbCreateBlackBox` :

1. Complete the description and author tags if you feel like.
2. Add the `#include` directives to be put in the generated `.h` file
3. Create your inputs and outputs
4. Fill in the process tag
5. Fill in the constructor tag
6. Fill in the copyconstructor tag
7. Fill in the destructor tag

3.3.2 Writting new widget boxes in `xml`

See the example `packages/wx/src/bbwxOutputText.xml`

File `packages/wx/src/bbwxOutputText.xml`

```
<blackbox name="OutputText" widget>

  <author>laurent.guigues at creatis.insa-lyon.fr</author>
  <description>Text zone to be inserted into a window (wxStaticText)</description>
  <category></category>

  <input name="Title" type="std::string" description="Title prepended to the text"/>
  <input name="In" type="std::string" description="Text"/>

  <createwidget><PRE>
   bbSetOutputWidget( new wxStaticText ( bbGetWxParent() , -1 , _T("") ) );
   Process();
  </PRE></createwidget>

  <process><PRE>
   std::string msg;
    if (bbGetInputTitle()!="")
      {
msg = bbGetInputTitle()+": " + bbGetInputIn();
      }
    else
      {
msg = bbGetInputIn();
      }
   ((wxStaticText*)bbGetOutputWidget())->SetLabel( bbtk::std2wx( msg ) );
  </PRE></process>

  <constructor><PRE>
    bbSetInputIn("");
    bbSetInputTitle("");
  </PRE></constructor>


</blackbox>
```

End of file

Explainations:

- The attribute `widget` of the `blackbox` tag instructs `bbfy` that the box inherits from `bbtk::WxBlackBox`.
- An output called `'Widget'` of type `wxWindow*` is automatically declared (you do not have to do it).

- The tag `createwidget` provides the body of the method which creates the widget. At the end of this method the output `'Widget'` must have been set with the newly created `wxWindow`. Here we create a new `wxStaticText`. The parent of the widget to create MUST BE the one provided by the method `bbGetWxParent()` which returns a `wxWindow*`. To update the static text after creation we simply call the `Process` method.

- The body of the `process` method simply concatenates the input `'Title'` (if non empty) and the input `'In'` and updates the `wxStaticText`. Remark that to get it, we use the `bbGetOutputWidget()` method which returns a `wxWindow*` which we cast into a `wxStaticText*` to use its specific method `SetLabel`.

More complex examples can be found in the `package/wx/src` folder.

3.3.3 Specific `xml` tags for `vtkImageAlgorithm` classes bbfication by inheritance

If you wish to bbfy a `vtk` object which is a `vtkImageAlgorithm` (such as `vtkImageGaussianSmooth`, `ImageAnisotropicDiffusion3D`, ...) we recommand you do it in `xml` (you can have a look at the examples in the `vtk` core package 'src' folder). The bbfication mechanism is inheritance.

You have to add the attribute `type="VTK_ImageAlgorithm"` to the `blackbox` tag :

```
<blackbox name="..." type="VTK_ImageAlgorithm">
```

You have to had an include tag which includes the vtk parent header, such as :

```
<include> vtkImageAnisotropicDiffusion3D.h </include>
```

You have to add the tag `vtkparent` which gives the `vtk` parent of the box, e.g.:

```
<vtkparent> vtkImageAnisotropicDiffusion3D </vtkparent>
```

The `vtk` algorithm input/ouput are wrapped directly using the `special` attributes of the input and output tags. A typical example is :

```
<input name="In"   type="vtkImageData*" special="vtk input"
       description="Input image"/>
<output name="Out"  type="vtkImageData*" special="vtk output"
        description="Output image"/>
```

The attribute `special="vtk input"` of the input 'In' definition directly connects it to the input of the vtk object the box inherits. No additional code is needed, the vtk object will directly receive the value of this input. The same mechanism hold for the output.

The parameters of the vtk object which are declared using `vtkSetMacro` and `vtkGetMacro` can also be directly wrapped using the attribute `special="vtk parameter"` of the input tag, e.g. :

```
<input  name="DiffusionThreshold"  type="double" special="vtk parameter"
        description="Difference threshold that stops diffusion"/>
```

The attribute `special="vtk parameter"` of the input called `DiffusionThreshold` instructs `bbfy` to directly call the `SetDiffusionThreshold` and `GetDiffusionThreshold` methods of the vtk parent when needed.

NOTE : For this mechanism to work, the name of the `bbtk` input MUST be the same than the name of the `vtk` parent parameter.

No `process` method has to be given, `bbfy` generates a process body for you, which simply calls the `Update()` method of the vtk parent.

NOTE : you can write your own `process` code which will overload the default. Don't forget to call Update(). See `packages/vtk/src/bbvtkConeSource.xml` for an example.

3.3.4 Specific `xml` tags for `vtkPolyDataAlgorithm` classes bbfication by inheritance

If you wish to bbfy a `vtk` object which is a `vtkPolyDataAlgorithm` (such as `vtkConeSource`, ...) we recommand you do it in `xml` (you can have a look at the examples in the `vtk` core package 'src' folder). The bbfication mechanism is inheritance.

You must use the same `xml` tags and attributes than for wrapping a `vtkImageAlgorithm` (see above) :

```
<blackbox name="..." type="VTK_PolyDataAlgorithm">

<vtkparent>the vtk Polydata class it inherits from</vtkparent>
<input  name="..."  type="vtkPolyData*" special="vtk input"
        description="..."/>
<output name="..."  type="vtkPolyData*" special="vtk output"
        description="..."/>
<input  name="..."  type="double"       special="vtk parameter"
        description="..."/>
```

3.3.5 Specific `xml` tags for `itk::ImageToImageFilter` classes bbfication by inheritance

to be written...

3.3.6 `bbfy xml` tags reference

See tables 1, 2

Table 1: `bbfy xml` tags reference (part 1)

Tag	Attributes	Condition	Multiplicity	Description
`<blackbox>`	`name`	-	1	The name of the box
	`type`	-	1	The type of the box. In: {`standard` (default), `ITK_ImageToImageFilter`, `VTK_ImageAlgorithm`, `VTK_PolyDataAlgorithm`}
	`generic`	a)	0-1	Generate the generic filter (see text)
	`widget`	-	1	If present then the box inherits from `WxBlackBox` (`AtomicBlackBox` if absent)
`<description>`	-	-	0-n	The description of the box. Multiple occurrence are concatenated
`<author>`	-	-	0-n	The author of the box. Multiple occurrence are concatenated
`<category>`	-	-	0-1	The box category (if more than one, they are separated with commas) see Tab 4
`<namespace>`	-	-	0-1	The namespace of the box. Use `bbPACKAGE`, where `PACKAGE` is the name of the package
`<include>`	-	-	0-n	Additionnal file to include (generates : `#include 'value'`)
`<template>`	-	-	0-n	Template parameter of the box. The template parameter list is generated in the order of appearance of the tag.
`<itkparent>`	-	a)	1	The parent itk class (with namespace)
`<vtkparent>`	-	b)	1	The parent vtk class
`<input>`	`name`	-	1	The name of the input
	`type`	-	1	The type of the input
	`special`	-	0-1	In: {`'itk input'`, `'vtk input'`, `'itk parameter'`, `'vtk parameter'`} (see below).
	`generic_type`	c)	0-1	The "generic" type of the input (see text).

Table 2: `bbfy xml` tags reference (part 2)

Tag	Attributes	Condition	Multiplicity	Description
`<output>`	`name`	-	1	The name of the output
	`type`	-	1	The type of the output
	`special`	-	0-1	In: {`'itk output'`, `'vtk output'`} (see below).
	`generic_type`	c)	0-1	The "generic" type of the output (see text).
	`nature`	c)	0-1	The "nature" of the output (used for automatic GUI generation).
`<process>`	-	-	0-1	The code of the processing method of the box. Must be put between clear tags : `<PRE></PRE>`
`<createwidget>`	-	d)	0-1	The code of the widget creation method of the box. Must be put between clear tags : `<PRE></PRE>`
`<constructor>`	-	-	0-1	The code of the user Constructor of the box (may contains default initialisations). Must be put between clear tags : `<PRE></PRE>`
`<copyconstructor>`	-	-	0-1	The code of the user Copy Constructor of the box . Must be put between clear tags : `<PRE></PRE>`
`<destructor>`	-	-	0-1	The code of the user Destructor of the box. Must be put between clear tags : `<PRE></PRE>`

Table 3: `bbfy xml` tags conditions

a)	`<blackbox type == 'ITK_ImageToImageFilter'>`
b)	`<blackbox type == 'VTK_ImageAlgorithm' or 'VTK_PolyDataAlgorithm'>`
c)	`<blackbox type == 'ITK_ImageToImageFilter'>` and `<blackbox generic>` is present.
d)	`<blackbox widget>` is present

.
.
.
.
.

Table 4: `Black Box` categories

`Category name`	: Meaning
`adaptor`	: Adaptor box
`application`	: Final application, end user intended
`atomic box`	: System category. Automatically assigned to Atomic Black Boxes (c++ defined)
`complex box`	: System category. Automatically assigned to Complex Black Boxes (script defined)
`command line`	: Script which defines a command line application (no embedded GUI, but command line imput parameters)
`demo`	: Demonstration
`dicom`	: DICOM aware box
`example`	: Example script showing a box use-case
`filter`	: Filtering box
`image`	: Image processing related box
`math`	: Mathematical operations
`mesh`	: Mesh processing related box
`misc`	: A box that cannot be put in other category !
`read/write`	: Box that read or write data from or to disk
`viewer`	: Box which displays some data
`widget`	: Piece of graphical interface
`3D object creator`	: Sophisticated 3D widget
`toolsbbtk`	: `bbtk` development tools (GUICreatePackage, GUICreateBlackBox,...)

Table 5: `Black box` kinds

`Kind`	Use as :
`ADAPTOR`	
`DEFAULT_ADAPTOR`	
`WIDGET_ADAPTOR`	
`DEFAULT_WIDGET_ADAPTOR`	
`GUI`	
`DEFAULT_GUI`	
`ALL`	If kind='ALL' then sets the level for all kinds

Table 6: Input/output `natures`

`Nature`	: Associated `DEFAULT_GUI` box
`'file name'`	`wx::FileSelector`
`'directory name'`	`wx::DirectorySelector`
`'colour'`	`wx::ColourSelector`

3.4 C++ description of a box

Almost everything is performed using macros.

For a quick start, the best you have to do is to run `bbStudio`, then in the menu `Tools`, choose the item `Create black box`, click on `C++`, and have a look to the generated files, or have a look at the source files of `bbtk` core packages.

3.4.1 Black box basic header file (.h)

Let's have a look at the file `packages/std/bbstdMakeFileName.h`

File `packages/std/bbstdMakeFileName.h`

```
#ifndef __bbstdMakeFileName_h_INCLUDED__
#define __bbstdMakeFileName_h_INCLUDED__

#include "bbtkAtomicBlackBox.h"

namespace bbstd
{
  class MakeFileName : public bbtk::AtomicBlackBox
  {
    BBTK_BLACK_BOX_INTERFACE(MakeFileName,bbtk::AtomicBlackBox);
    BBTK_DECLARE_INPUT(Directory, std::string);
    BBTK_DECLARE_INPUT(File,      std::string);
    BBTK_DECLARE_INPUT(Extent,    std::string);
    BBTK_DECLARE_OUTPUT(Out,      std::string);
    BBTK_PROCESS(DoProcess);
    void DoProcess();
  protected:
    virtual void bbUserConstructor();
  };

  BBTK_BEGIN_DESCRIBE_BLACK_BOX(MakeFileName,bbtk::AtomicBlackBox);
  BBTK_NAME("MakeFileName");
  BBTK_AUTHOR("jpr@creatis.insa-lyon.fr");
  BBTK_CATEGORY("misc");
  BBTK_DESCRIPTION("Makes a kosher file name");
  BBTK_INPUT(MakeFileName,Directory,"Directory Name",std::string,
             "directory name");
  BBTK_INPUT(MakeFileName,File,     "File Name",     std::string,
             "file name");
  BBTK_INPUT(MakeFileName,Extent,   "Extention",     std::string,
             "file extension");

  BBTK_OUTPUT(MakeFileName,Out,"Full File Name",std::string,"file name");
  BBTK_END_DESCRIBE_BLACK_BOX(MakeFileName);
```

```
}
// EO namespace bbstd

#endif //  __bbstdMakeFileName_h_INCLUDED__
```

End of file

It includes `bbtkAtomicBlackBox.h`. The box class is `MakeFileName`. It inherits `bbtk::AtomicBlackBox`. It is in the `bbstd` namespace : each box of a given package, say PACK, must be inserted into the namespace `bbPACK`.

The macro `BBTK_BLACK_BOX_INTERFACE` declares the interface of the class : constructor, destructor, standard methods (e.g. New), etc. The following macros then declare inputs and outputs of the box, with their types. The macro `BBTK_PROCESS` then declares which method to call when processing the box (the process callback). The callback itself is declared just below.

The line `virtual void bbUserConstructor();` then overloads the virtual method `bbUserConstructor` which is used to perform specific things at construction time. You can also overload `bbUserCopyConstructor` and `bbUserDestructor` with the same signature. The black box interface macros are summarized in table 7.

After the black box class declaration then comes a zone in which you describe your black box, between the macros `BBTK_BEGIN_DESCRIBE_BLACK_BOX` and `BBTK_END_DESCRIBE_BLACK_BOX`.

The macro `BBTK_BEGIN_DESCRIBE_BLACK_BOX` actually starts the declaration of another class, called `<BOXNAME >Descriptor` (in our case `MakeFileNameDescriptor`). The descriptor of a black box :

- has only one instance, which is stored in the package
- provides information about the box type (author, description, ...) which is used for documentation.
- provides information about the box I/Os, mainly their types (uses RTTI : `std::type_info`).
- is responsible for creating new instances of the box it describes.

As you can see, the macros which are between `BBTK_BEGIN_DESCRIBE_BLACK_BOX` and `BBTK_END_DESCRIBE_BLACK_BOX` provide the box name (the string), its authors, description, category, the descriptions of its inputs and outputs. Black box descriptor related are described in table 8.

3.4.2 Black box basic implementation file (.cxx)

Now let's have a look at the file `packages/std/bbstdMakeFileName.cxx`

File `packages/std/bbstdMakeFileName.cxx`

Table 7: Black box interface C++ macros

BBTK_BLACK_BOX_INTERFACE(BOX_NAME,BBTK_PARENT) : Yes, we know the bbtk parent is redundant with the inheritance list... That's why we allow you to describe your class in xml format!
BBTK_VTK_BLACK_BOX_INTERFACE(CLASS,BBTK_PARENT,VTK_PARENT) : Black box interface for vtk object inherited boxes ...
BBTK_ITK_BLACK_BOX_INTERFACE(CLASS,BBTK_PARENT,ITK_PARENT) : Black box interface for itk object inherited boxes ...
BBTK_DECLARE_INPUT (NAME,TYPE) : Declares an input of the box, with NAME : the input name (as it will appear to the users of your black box) and TYPE : C++ type of the input (e.g. double, std::string, vtkImageData*, ...).
BBTK_DECLARE_INHERITED_INPUT(NAME,TYPE,GETMETHOD,SETMETHOD) : Declares an input of the box which wraps the GETMETHOD / SETMETHOD accessors
BBTK_DECLARE_VTK_INPUT(NAME,TYPE) : Declares a vtk object-inherited input
BBTK_DECLARE_VTK_IMAGE_ALGORITHM_INPUT(NAME,TYPE) : Declares a vtkImageAlgorithm-inherited input
BBTK_DECLARE_VTK_POLY_DATA_ALGORITHM_INPUT(NAME,TYPE) : Declares a vtkPolyDataAlgorithm-inherited input
BBTK_DECLARE_ITK_INPUT (NAME,TYPE) : Declares a itk object-inherited input
BBTK_DECLARE_OUTPUT (NAME,TYPE) : Declares an output of the box
BBTK_DECLARE_INHERITED_OUTPUT(NAME,TYPE,GETMETHOD,SETMETHOD) : Declares an output of the box which wraps the GETMETHOD / SETMETHOD accessors
BBTK_DECLARE_VTK_OUTPUT(NAME,TYPE) : Declares a vtk object-inherited output
BBTK_DECLARE_ITK_OUTPUT(NAME,TYPE) : Declares a itk object-inherited output
BBTK_DECLARE_VTK_PARAM(VTK_PARENT,NAME,TYPE) : Declares an input corresponding to an inherited vtk parameter (you know, the ones that are declared by vtkSetMacro/vtkGetMacro). Its name must be the same than the vtk parameter name.
BBTK_DECLARE_ITK_PARAM(NAME,TYPE) : Declares an input corresponding to an inherited itk parameter
BBTK_PROCESS(METHOD_NAME) : Defines the method to call when the box is processed.
BBTK_VTK_PROCESS : Defines AND implements the default processing method for vtk inherited black boxes (calls vtkParent::Update)
BBTK_ITK_PROCESS : Defines AND implements the default processing method for itk inherited black boxes (calls itkParent::Update)

```
#include "bbstdMakeFileName.h"
#include "bbstdPackage.h"

namespace bbstd
{

  BBTK_ADD_BLACK_BOX_TO_PACKAGE(std,MakeFileName);
  BBTK_BLACK_BOX_IMPLEMENTATION(MakeFileName,bbtk::AtomicBlackBox);
```

Table 8: Black box descriptor C++ macros

BBTK_BEGIN_DESCRIBE_BLACK_BOX(BOX_NAME,BBTK_PARENT) : Yes, we know it's redundant with public inheritance ... That's why we allow you to describe your class in xml format! All the following items will be used in the Help interface; describe them carefully (i.e. in a Human understandable way!).

BBTK_ADAPTOR : Declares that the box is an adaptor

BBTK_DEFAULT_ADAPTOR : Declares that the box is the default adaptor for its I/O types

BBTK_NAME(STRING) : The name of your box

BBTK_AUTHOR(STRING) : The author(s) (better you put e-mail adresses)

BBTK_DESCRIPTION(STRING) : Brief description of what does the box

BBTK_CATEGORY(STRING) : Box categories, semicolon separated (see table 4)

BBTK_INPUT(BOX_NAME,INPUT_NAME,DESCRIPTION,CPP_TYPE,INPUT_NATURE)

- BOX_NAME : The current black box name.
- INPUT_NAME : The input name
- DESCRIPTION (string) : A brief description of what the parameter is used for.
- CPP_TYPE : The C++ type of the input (e.g. double, std::string, vtkImageData*, ...)
- INPUT_NATURE : The 'nature' of the parameter (see table 6) if you wish your box may be used by automatic GUI generator. Supply an empty string ("") if you don't care.

BBTK_OUTPUT(BOX_NAME,OUTPUT_NAME,DESCRIPTION,CPP_TYPE) : The same

BBTK_END_DESCRIBE_BLACK_BOX(BOX_NAME)

```
  void MakeFileName::bbUserConstructor()
  {
    bbSetInputDirectory("");
    bbSetInputFile("");
    bbSetInputExtent("");
  }

  void MakeFileName::DoProcess()
  {
    ...
  }
}
// EO namespace bbstd
```

End of file

The first line includes the header file. The second one includes the `std` package header file. This file is automatically generated during cmake configuration : for a package named <PACK >, `cmake` creates the files `bb`<PACK >`Package.h` and `bb`<PACK >`Package.cxx`. The header is to be included in any box implementation file and the second one is compiled in the package library.

The macro BBTK_ADD_BLACK_BOX_TO_PACKAGE then registers the box MakeFileName into the package std.

The macro BBTK_BLACK_BOX_IMPLEMENTATION is the mirror macro of the macro BBTK_BLACK_BOX_INTERFACE that was used in the header : it implements the methods declared in the header.

We then need to write the body of bbUserConstrutor and of the processing callback (here DoProcess).

That's all we need for a 'basic' black box. The implementation related macros are summarized in table 9.

Table 9: Black box implementation C++ macros

BBTK_ADD_BLACK_BOX_TO_PACKAGE(PACKAGE_NAME,BOX_NAME)
BBTK_BLACK_BOX_IMPLEMENTATION(BOX_NAME,BBTK_PARENT)

3.4.3 Widget black boxes C++ macros

See the example of package/wx/src/bbwxLayoutLine.h|cxx. The only differences with a non-widget black box are :

- The header must include bbtkWxBlackBox.h and the class must inherit bbtk::WxBlackBox.
- The black box interface must declare the widget creation callback with the macro BBTK_CREATE_WIDGET(CALLBACK). The callback must be declared in the interface and implemented.
- You can overload the method void bbUserOnShow() which is called just after the wxWindow has been shown, e.g. to refresh its content. Note that Layout widget *MUST* overload this method and call bbUserOnShowWidget(INPUT_NAME) for all inputs which correspond to an 'embedded' window (the 'Widget1' ... 'WidgetN' inputs, see package/wx/src/bbwxLayoutLine.cxx)

3.4.4 VTK black boxes C++ macros

See the example of package/wx/src/bbvtkMarchingCubes.h|cxx. The macros are summarized in table 7.

3.4.5 Template black boxes C++ macros

You can write down black box classes *templates*. However, only *actual* classes, that is instanciated templates, can be inserted into a package.

The files package/std/src/bbstdStringTo.h|cxx provide an example of a class template with one template parameter.

The files package/std/src/bbstdCast.h|cxx provide an example of a class template with two template parameters.

Class templates related macros are summarized in table 10.

Table 10: Black box templates-related C++ macros

BBTK_TEMPLATE_BLACK_BOX_INTERFACE(BOX_NAME,BBTK_PARENT, TEMPLATE_PARAMETER)
BBTK_TEMPLATE2_BLACK_BOX_INTERFACE(BOX_NAME,BBTK_PARENT, TEMPLATE_PARAMETER_1, TEMPLATE_PARAMETER_2)
BBTK_BEGIN_DESCRIBE_TEMPLATE_BLACK_BOX(BOX_NAME,BBTK_PARENT) : Note that in the descriptor, the template parameter name is T
BBTK_BEGIN_DESCRIBE_TEMPLATE2_BLACK_BOX(BOX_NAME,BBTK_PARENT) : Note that in the descriptor, the template parameters names are T1 and T2
BBTK_END_DESCRIBE_TEMPLATE_BLACK_BOX(BOX_NAME)
BBTK_END_DESCRIBE_TEMPLATE2_BLACK_BOX(BOX_NAME)
BBTK_TEMPLATE_INPUT(BOX_NAME,INPUT_NAME,DESCRIPTION,CPP_TYPE, INPUT_NATURE) : Same than for non-templates, except that the CPP_TYPE can be the template parameter.
BBTK_TEMPLATE2_INPUT(BOX_NAME,INPUT_NAME,DESCRIPTION,CPP_TYPE, INPUT_NATURE) : Same remark
BBTK_TEMPLATE_OUTPUT(BOX_NAME,OUTPUT_NAME,DESCRIPTION,CPP_TYPE) : Same remark
BBTK_TEMPLATE2_OUTPUT(BOX_NAME,OUTPUT_NAME,DESCRIPTION,CPP_TYPE) : Same remark
BBTK_BLACK_BOX_TEMPLATE_IMPLEMENTATION(BOX_NAME,BBTK_PARENT)
BBTK_BLACK_BOX_TEMPLATE2_IMPLEMENTATION(BOX_NAME,BBTK_PARENT)
BBTK_ADD_TEMPLATE_BLACK_BOX_TO_PACKAGE(PACKAGE_NAME,BOX_NAME, TEMPLATE_PARAMETER_VALUE) : Adds the black box template instanciated on a certain value of its template parameter to the package. You can put as many such lines with different template parameter values as you want (see e.g. package/std/src/bbstdStringTo.cxx)
BBTK_ADD_TEMPLATE2_BLACK_BOX_TO_PACKAGE(PACKAGE_NAME,BOX_NAME, TEMPLATE_PARAMETER_1_VALUE, TEMPLATE_PARAMETER_2_VALUE) : The same for two template parameters (see e.g. package/std/src/bbstdCast.cxx)

IMPORTANT NOTE ON TEMPLATE BLACK BOXES NAMES:

Two different boxes registered in a package must have two different names. Hence when using black box classes templates, one must give different names to two instanciations of the template on two different types. This is typically done with inserting the template parameter type name in the black box class name. An example is provided in package/std/src/bbstdStringTo.h :

```
BBTK_BEGIN_DESCRIBE_TEMPLATE_BLACK_BOX(ToString,bbtk::AtomicBlackBox);
BBTK_NAME(bbtk::HumanTypeName<T>()+"ToString");
 ...
BBTK_END_DESCRIBE_TEMPLATE_BLACK_BOX(ToString);
```

To get the string corresponding to the name of a C++ type (here the template

parameter T) one must use the template bbtk function bbtk::HumanTypeName<T>() [3]. It is then concatenated to the name ToString. This thus gives the name IntToString to the black box ToString<int >, DoubleToString to the black box ToString<double >, etc.

You can also use bbtk::HumanTypeName<T>() in the macro BBTK_DESCRIPTION, like for example:

```
BBTK_DESCRIPTION("Converts a "+bbtk::HumanTypeName<T>()+" ("
 +bbtk::TypeName<T>()+") into a string");
```

3.4.6 ITK black boxes C++ macros

It is a special cas of black box templates with also special macros for itk object inherited black boxes.

See the example of package/wx/src/bbitkBinaryThresholdImageFilter.h|cxx, the tables 7 and 10.

Note that there is also a mechanism for making "generic" untemplatized itk black boxes. See the example in the file above.

[3] HumanTypeName returns a human readable type name, without special chars such as :: or <. For example the human readable type name of std::vector<std::string > is VectorOfString. The 'inhuman' type name is given by the function bbtk::TypeName<T>().