Object-oriented application development with MeVisLab and Python

Frank Heckel, Michael Schwier and Heinz-Otto Peitgen

Fraunhofer MEVIS
Universitaetsallee 29
28359 Bremen, Germany
frank.heckel@mevis.fraunhofer.de
michael.schwier@mevis.fraunhofer.de
heinz-otto.peitgen@mevis.fraunhofer.de

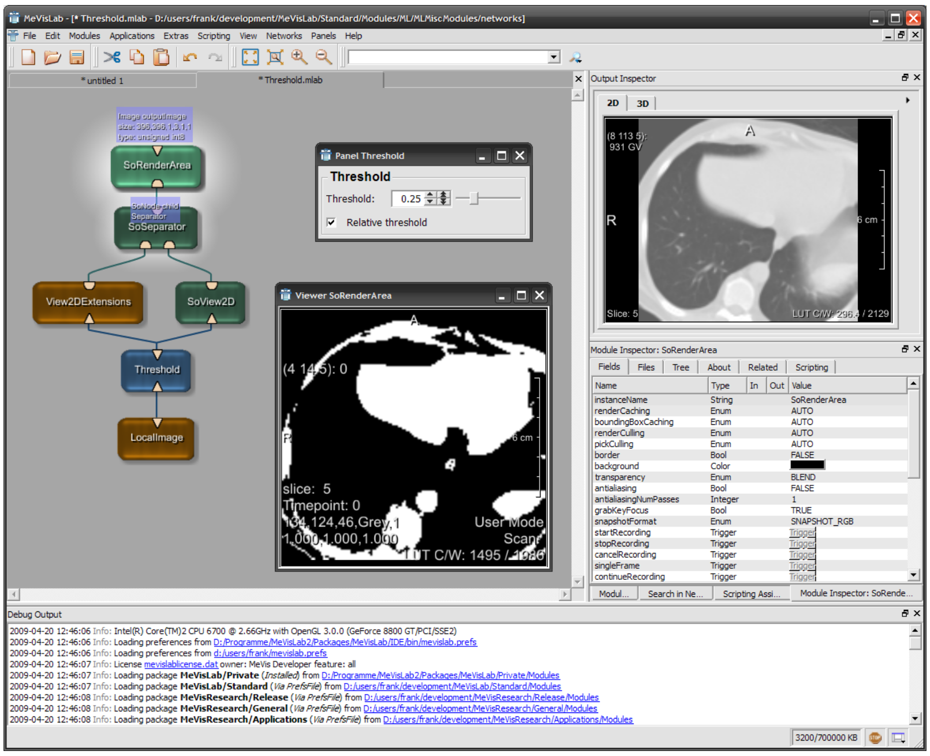
Abstract: MeVisLab is a research and rapid prototyping platform for medical image processing developed by MeVis Medical Solutions AG and Fraunhofer MEVIS. We present an object-oriented approach for developing applications using MeVisLab and Python. Our approach simplifies the development of scripting intensive and complex applications for medical image processing. The program flow becomes clearer and the resulting modules are easier to maintain compared to the common functional approach thus making development less error prone.

1 Introduction

MeVisLab is a research and rapid prototyping platform for medical image processing [MeV09]. It has been developed by *MeVis Medical Solutions AG* and *Fraunhofer MEVIS*[1]. MeVisLab is written in C++ and uses Qt[2] for graphical user interfaces. It is available for Microsoft Windows, Linux and MacOS X. MeVisLab comes with a comprehensive set of image processing and visualization methods via the MeVis Image Processing Library (ML) and Open Inventor. Furthermore, it also includes both the Insight Segmentation and Registration ToolKit (ITK) and the Visualization ToolKit (VTK) [RJS$^+$05, KSR$^+$06]. Algorithms and applications are developed in a visual programming manner by building a network of functional units (*modules*) where a module represents an algorithm or visualization method for example. Networks can be hierarchically encapsulated into *macro modules* to form new algorithms or reusable parts of an application (see Fig. 1).

Moreover, MeVisLab offers the possibility to create a graphical user interface (GUI) for an application consisting of various modules using an abstract module definition language (*MDL*). Dynamic functionality can be added to an application via scripting using JavaScript and Python. This way it is for example possible to react on user interactions, to manipulate the processing pipeline (modules, networks or the user interface) or even

[1]`http://www.mevis.de`
[2]`http://www.qtsoftware.com`


MeVisLab - [* Threshold.mlab - D:/users/frank/development/MeVisLab/Standard/Modules/ML/MLMiscModules/networks]
File Edit Modules Applications Extras Scripting View Networks Panels Help
* untitled 1
* Threshold.mlab
SoRenderArea
SoSeparator
View2DExtensions
SoView2D
Threshold
LocalImage
Panel Threshold
Threshold
Threshold: 0.25
Relative threshold
Viewer SoRenderArea
Output Inspector
2D
3D
Slice: 5
LUT C/W: 296.4 / 2129
Module Inspector: SoRenderArea
Fields Files Tree About Related Scripting
Modul... Search in Ne... Scripting Assi... Module Inspector: SoRende...
Debug Output
3200/700000 KB

Name	Type	In	Out	Value
instanceName	String			SoRenderArea
renderCaching	Enum			AUTO
boundingBoxCaching	Enum			AUTO
renderCulling	Enum			AUTO
pickCulling	Enum			AUTO
border	Bool			FALSE
background	Color			■
transparency	Enum			BLEND
antialiasing	Bool			FALSE
antialiasingNumPasses	Integer			1
grabKeyFocus	Bool			TRUE
snapshotFormat	Enum			SNAPSHOT_RGB
startRecording	Trigger			Trigger
stopRecording	Trigger			Trigger
cancelRecording	Trigger			Trigger
singleFrame	Trigger			Trigger
continueRecording	Trigger			Trigger

```
2009-04-20 12:46:06 Info: Intel(R) Core(TM)2 CPU 6700 @ 2.66GHz with OpenGL 3.0.0 (GeForce 8800 GT/PCI/SSE2)
2009-04-20 12:46:06 Info: Loading preferences from D:/Programme/MeVisLab2/Packages/MeVisLab/IDE/bin/mevislab.prefs
2009-04-20 12:46:06 Info: Loading preferences from d:/users/frank/mevislab.prefs
2009-04-20 12:46:07 Info: License mevislablicense.dat owner: MeVis Developer feature: all
2009-04-20 12:46:07 Info: Loading package MeVisLab/Private (Installed) from D:/Programme/MeVisLab2/Packages/MeVisLab/Private/Modules
2009-04-20 12:46:07 Info: Loading package MeVisLab/Standard (Via PrefsFile) from D:/users/frank/development/MeVisLab/Standard/Modules
2009-04-20 12:46:08 Info: Loading package MeVisResearch/Release (Via PrefsFile) from D:/users/frank/development/MeVisResearch/Release/Modules
2009-04-20 12:46:08 Info: Loading package MeVisResearch/General (Via PrefsFile) from D:/users/frank/development/MeVisResearch/General/Modules
2009-04-20 12:46:08 Info: Loading package MeVisResearch/Applications (Via PrefsFile) from D:/users/frank/development/MeVisResearch/Applications/Modules
```

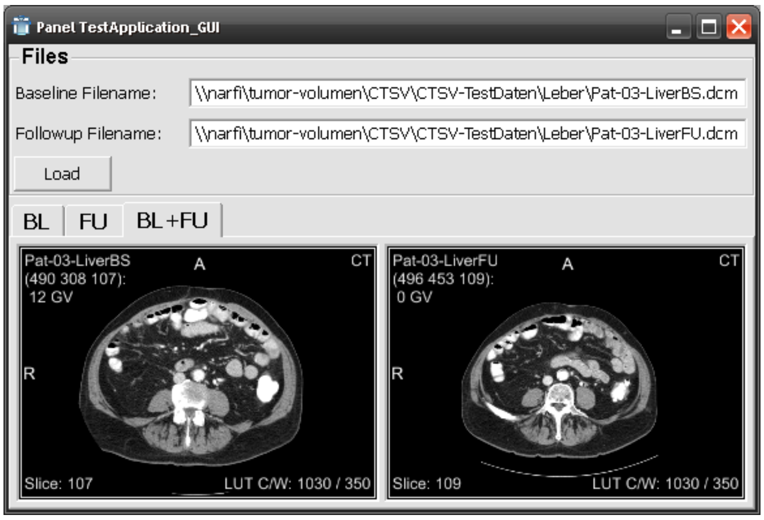
Figure 1: A simple example for visual network based programming using an alpha version of MeVisLab 2.0. The network consists of a image processing module (blue), some visualization modules (green) and some macro modules (orange).

to calculate and save results. Furthermore, the application logic is typically implemented using scripting.

A comprehensive introduction to MeVisLab's predecessor, ILAB 4, is given by Hahn *et al.* in [HLP03]. An introduction to application development with MeVisLab is given by Rexilius *et al.* in [RKHP06] and Merkel *et al.* in [MHK$^+$08]. A comparison between MeVisLab and other integrated development environments and libraries that use ITK is given by Bittner *et al.* in [BvUW$^+$07].

In this paper we focus on MeVisLab's scripting capabilities. We will outline the drawbacks of the common functional development approach. After that, we present a novel object-oriented approach for scripting of complex applications in Python which overcomes those drawbacks. Our programming concept simplifies scripting of large and complex applications. Communication between modules becomes easier and the program flow becomes clearer. Thus, the resulting code is less error prone and easier to maintain.

We will neither give an introduction to Python nor to object-oriented programming. For an introduction to both topics have a look in the official Python documentation [Pyt09].


Panel TestApplication_GUI

Files

Baseline Filename: \\narfi\tumor-volumen\CTSV\CTSV-TestDaten\Leber\Pat-03-LiverBS.dcm

Followup Filename: \\narfi\tumor-volumen\CTSV\CTSV-TestDaten\Leber\Pat-03-LiverFU.dcm

Figure 2: A simple application where the user can load a baseline scan, a follow-up scan or both. The GUI contains various viewer configurations to show the user the data in an appropriate way.

2 Motivation

In the following sections we give a brief overview on application development and scripting in MeVisLab based on Python and the common functional approach. Moreover, we introduce an example application using this development method which, although being simple, already shows the advantages of our object-oriented approach.

2.1 A simple example application

Consider the following, simple scenario: An application is able to load a baseline scan, a follow-up scan or both. First, the clinician specifies the filenames for baseline and/or follow-up in the GUI. Then he presses the load button. As soon as all the data is loaded, the application should automatically switch to a layout that contains the appropriate viewers which allows the clinician to view the single scans or both in parallel. Figure 2 shows the GUI of such an application. The application should be implemented by the modular, hierarchical structure shown in Figure 3 which allows to reuse certain functional units. The orange boxes represent the various macro modules (i.e. parts of the application).

This simple functionality already requires some communication between the modules. In particular, parameters and results have to be passed between them.

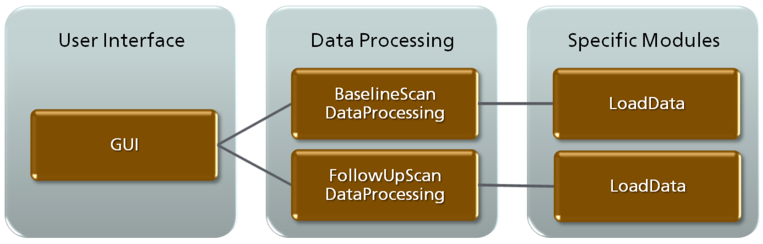


Figure 3: The hierarchical structure of the simple example application consisting of some macro modules (orange boxes).

2.2 Functional application development in MeVisLab

2.2.1 Scripting in MeVisLab

Scripting in MeVisLab is done by reacting on certain events (e.g. application initialization) or on *field* changes via so called *field listeners*. Fields are basically parameters of various types, triggers (e.g. "buttons") and input/output elements of a module. If such an event occurs, a user defined scripting function is called where the event is handled. This functionality is achieved using the following MDL module definition which is typically located in a module's `.script` file:

Code Listing 1: MDL module definition example

```
Commands {
  // tell MeVisLab where additional Python modules are located
  importPath = "$(LOCAL)"
  // tell MeVisLab which file contains the Python script functions
  source = "$(LOCAL)/Module.py"
  // tell MeVisLab to call the "init" function on initialization
  initCommand = init

  // tell MeVisLab to call "reactOnFieldChange" if "someField" changes
  FieldListener someField {
    command = reactOnFieldChange
  }
}
```

The module's Python script file `Module.py` might look like this:

Code Listing 2: Python script example

```
def init():
  # do some initialization ...

def reactOnFieldChange(field):
  # do something based on the new field value ...
```

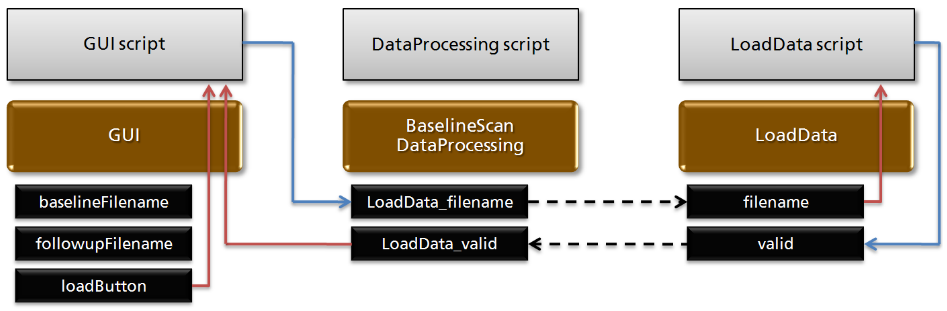


Figure 4: Typical communication between modules and their scripts using the common functional development approach: Field listener calls (red arrows), field connections (dashed black arrows) and scripting commands (blue arrows). For simplicity reasons the figure only shows the baseline scan.

Furthermore, MeVisLab offers the possibility to pass values between fields via so called *field connections* which means that the value of one field is automatically forwarded to the connected fields.

2.2.2 Implementing the example the common way

In the common functional approach parameters and results are passed between modules by fields. These fields can therefore be seen as "communication" fields. The functionality described in Section 2.1 would be implemented by setting the values of those fields and by defining field listeners on the trigger (`loadButton`), some parameters (`filename`) and some "result fields" (`LoadData_valid` in `BaselineScan` and `FollowUpScan`). This way scripting functions of other modules will implicitly be called by setting the values of the fields for which a field listener has been defined. Furthermore, some part of the communication might be done via field connections (e.g. between `filename` and `LoadData_filename`). Figure 4 shows the communication between the modules using the described implementation. The `.script` and `.py` files for this implementation might look as follows. Notice that one logical coherent step (loading the files) is split into three scripting functions on GUI level.

Code Listing 3: Functional LoadData MDL definition

```
...
FieldListener filename {
  command = filenameChanged
}
...
```

Code Listing 4: Functional LoadData Python script

```
def filenameChanged(field):
  loadSuccessful = False
  # load the file and set loadSuccessful ...

  # setting valid results in an implicit field listener call
  ctx.field("valid").value = loadSuccessful
```

Code Listing 5: Functional GUI MDL definition

```
...
FieldListener loadButton {
  command = loadButtonPressed
}

FieldListener BaselineScan.LoadData_valid {
  command = baseline_LoadDataDone
}

FieldListener FollowUpScan.LoadData_valid {
  command = followup_LoadDataDone
}
...
```

Code Listing 6: Functional GUI Python script

```
# the state of the GUI (BL, FU, BL_AND_FU)
state = ""

## called if the load button has been pressed
def loadButtonPressed(field):
  global state

  if ctx.field("baselineFilename").value != "" and \\
  ctx.field("followupFilename").value != "":
    state = "BL_AND_FU"
    # setting the filename results in an implicit field listener call
    ctx.field("BaselineScan.LoadData_filename").value = \\
    ctx.field("baselineFilename").value
    # setting the filename results in an implicit field listener call
    ctx.field("FollowUpScan.LoadData_filename").value = \\
    ctx.field("followupFilename").value

  elif ctx.field("baselineFilename").value != "":
    state = "BL"
    # setting the filename results in an implicit field listener call
    ctx.field("BaselineScan.LoadData_filename").value = \\
    ctx.field("baselineFilename").value

  elif ctx.field("followupFilename").value != "":
    state = "FU"
    # setting the filename results in an implicit field listener call
    ctx.field("FollowUpScan.LoadData_filename").value = \\
    ctx.field("followupFilename").value

  else:
    print "invalid filenames"
```

```
## called if the baseline scan's LoadData module finished loading
def baseline_LoadDataDone(field):
  global state

  successful = field.valid

  if state == "BL_AND_FU":
    if not successful:
      print "unable to load baseline"
      # the followup is still loaded although it is not necessary
  elif state == "BL":
    if successful:
      # switch to bl layout ...
    else:
      print "unable to load baseline"

## called if the follow-up scan's LoadData module finished loading
def followup_LoadDataDone(field):
  global state

  successful = field.valid

  if state == "BL_AND_FU":
    if successful:
      # switch to bl+fu layout ...
    else:
      print "unable to load followup"
  elif state == "FU":
    if successful:
      # switch to fu layout ...
    else:
      print "unable to load followup"
```

2.2.3 Drawbacks for development of large applications

The functional approach has some drawbacks for development of large applications. The modules of an application consist of many fields and field listeners of which most are for communication purposes only. Because of the way parameters and results are passed between modules, the program flow of the application is driven by implicit function calls via field listeners. This results in both, a program flow that is hard to track and in spreading logical coherent processes over various functions. In conclusion the resulting code of an application becomes confusing and unreadable. Moreover, it gets hard to maintain and extending it is very error-prone.

3 Object-oriented application development

To overcome the drawbacks of the functional approach for development of large and complex applications, we developed an object-oriented approach that we will describe in the following sections.

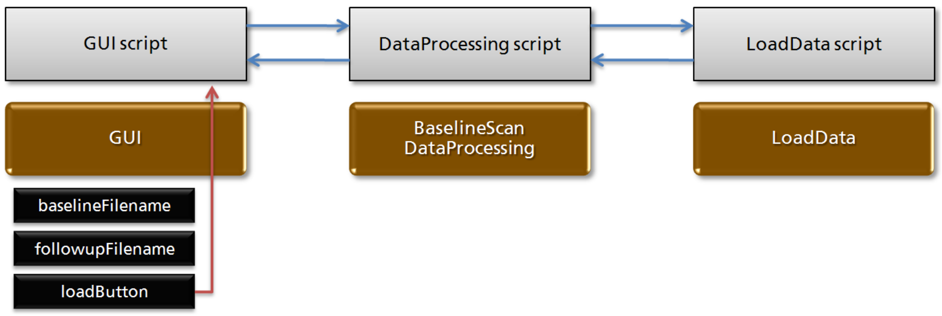


Figure 5: Communication between modules and their scripts using our object-oriented development approach: Field listener calls (red arrows) and scripting commands (blue arrows). For simplicity reasons the figure only shows the baseline scan.

3.1 Concept

In our approach a unique instance of a Python class is assigned to a module. The class encapsulates a module's scripting functionality in its member functions. By assigning unique class instances to modules it is possible to save references to other modules (respectively their Python class instances). These can be:

- Parent (a module's "surrounding" macro module)
- Children (modules within the macro module)
- Siblings (other modules within the parents internal network)

Using these references it is possible to perform communication between modules directly on the Python script level via explicit member function calls instead of using fields and field listeners (see Fig. 5). Parameters and results can now be passed as parameters and return values of functions. As a consequence, using this concept the indirection in communication between modules is resolved. Furthermore, communication-only-fields are avoided and fields are only used where they are necessary: for parameterization of algorithms or application modules.

3.2 Implementation

For implementing the object-oriented concept in MeVisLab a module's class definition looks like this:

Code Listing 7: Module class definition in Python

```python
class Module(object):

  ## the constructor
  def __init__(self, ctx, child1, child2):
    # the context of this module
    self.__ctx = ctx
    # reference to the parent of this module
    self.__parent = None
    # reference to the 1st child module
    self.__child1 = child1
    # reference to the 2nd child module
    self.__child2 = child2
    # reference to a sibling
    self.__sibling = None

    # set the parent in the child modules
    self.__child1.setParent(self)
    self.__child2.setParent(self)
    # the parent also tells its children about siblings
    self.__child1.setSibling(self.__child2)
    self.__child2.setSibling(self.__child1)

  ## set the parent
  def setParent(self, parent):
    self.__parent = parent

  ## set the sibling
  def setSibling(self, sibling):
    self.__sibling = sibling

  ## return the field of the given name
  def getField(self, fieldName):
    return self.__ctx.field(fieldName)

  ## a field listener callback
  def _reactOnFieldChange(self, field):
    # do something based on the new field value ...
```

For generating, saving and accessing the module's unique class instance the following code has to be added to the Python script. For avoiding confusion with the `self`-reference used in Python, we call a module's class instance "`this`".

Code Listing 8: Class access and instantiation in Python

```python
## global reference to this modules unique instance
this = None

## access method to the global this object
def getThis():
  global this
  return this

## the init method creates the unique global class instance
def init(ctx):
  global this
```

	Functional	Object-oriented
initCommand	*optional*	initCommand = ”*py: init(ctx)*”
FieldListener	command = function	command = ”*py: this._function(args[0])*”
Function definition	def function(...):	def function(self, ...):
Function call	function(...)	self.function(...)
ctx access	ctx. ...	self.__ctx. ...
Fields of internal modules	ctx.field(”module.field”)	self.moduleReference. getField(”field”)

Table 1: Comparison between functional and object-oriented application development in MeVisLab

```
  # access the children's unique class instances
  child1 = ctx.module("Child1").call("getThis")
  child2 = ctx.module("Child2").call("getThis")
  # create this modules class instance and tell it about children
  this = Module(ctx, child1, child2)
```

In addition the module's MDL definition has to be adapted to the object-oriented implementation. In the following code `args` is the parameter list given by MeVisLab when a field listener is called where `args[0]` is the field.

Code Listing 9: MDL module definition for object-oriented implementation

```
Commands {
  // tell MeVisLab where additional Python modules are located
  importPath = "$(LOCAL)"
  // tell MeVisLab which file contains the Python script functions
  source = "$(LOCAL)/Module.py"
  // tell MeVisLab to call the "init" function on initialization
  initCommand = "*py: init(ctx) *"

  // tell MeVisLab to call "reactOnFieldChange" if "someField" changes
  FieldListener someField {
    command = "*py: this._reactOnFieldChange(args[0]) *"
  }
}
```

The above implementation assumes that internal modules are created before the parent's init command is called which applies to MeVisLab. Because MeVisLab instantiates the scripts of each module and because Python is a dynamic programming language, no import statements are necessary in the script files of the modules for accessing other modules functions or variables.

A comparison between functional and object-oriented development of applications using MeVisLab is given in Table 1.

3.3 Calling conventions

There are typically three types of functions: interface functions, internal helper functions and field listener callbacks. Interface functions are “public” by definition because they shall be called by other modules. In contrast internal functions are “private” because they shall only be called within one module. In Python a function (or variable) is defined as private by putting “__” in front of its name. Field listeners are logically private, because they shall not be called by other modules but only if the modules field has changed. However, by making them private MeVisLab would not be able to call them, because a private function can only be called from within the class in which it has been defined. The solution is to make them “protected” which is done by putting “_” in front of the name. This is only a convention which is not checked by Python. Thus, the function can still be called by other modules and by MeVisLab, but its name indicates that it actually should not be used by other modules. Using this convention a developer can easily see the purpose of a function based on the prefix of its name.

Code Listing 10: Calling conventions

```
class Module(object):
  ...
  ## a private helper function
  def __helperFunction(self):
    pass

  ## a "protected" field listener callback
  def _fieldListenerCallback(self, field):
    pass

  ## a public interface function
  def interfaceFunction(self):
    pass
```

3.4 Implementing the example the object-oriented way

Using the object-oriented approach, most of the fields can be removed from the internal modules because communication is now done via member function calls (see Fig. 5). For example the `LoadData` module now has a function `loadFile(self, filename)` that can be called by the parent to load the given file. When finished, the function simply returns `True` if loading was successful or `False` otherwise. Moreover, loading the data and switching to the proper layout is now done in only one function.

Code Listing 11: Object-oriented LoadData Python script

```
class LoadData(object):
  ...
  ## load the given file
  def loadFile(self, filename):
    loadSuccessful = False
    # load the file and set loadSuccessful ...

    return loadSuccessful
```

Code Listing 12: Object-oriented DataProcessing Python script

```
class DataProcessing(object):
  ...
  ## load the given file
  def loadFile(self, filename):
    return self.__loadData.loadFile(filename)
```

Code Listing 13: Object-oriented GUI MDL definition

```
...
FieldListener loadButton {
  command =  *py: this._loadData() *
}
...
```

Code Listing 14: Object-oriented GUI Python script

```
class GUI(object):
  ...
  ## called if the load button has been pressed
  def _loadData(self):
    if self.getField("baselineFilename").value != "" and \\
    self.getField("followupFilename").value != "":
      # call the loading function and react on returned value
      if self.__baselineScan.loadFile( \\
      self.getField("baselineFilename").value ):
        # call the loading function and react on returned value
        if self.__followupScan.loadFile( \\
        self.getField("followupFilename").value ):
          # switch to bl+fu layout ...
        else:
          print "unable to load followup"
      else:
        print "unable to load baseline"

    elif self.getField("baselineFilename").value != "":
      # call the loading function and react on returned value
      if self.__baselineScan.loadFile( \\
      self.getField("baselineFilename").value ):
        # switch to bl layout ...
      else:
        print "unable to load baseline"

    elif self.getField("followupFilename").value != "":
      # call the loading function and react on returned value
      if self.__followupScan.loadFile( \\
      self.getField("followupFilename").value ):
         # switch to fu layout ...
      else:
        print "unable to load followup"

    else:
      print "invalid filenames"
```

Using this implementation, the developer can easily see which functions of which modules are called and which parameters they need. Moreover, the script can directly react on the returned value of the internal module without the indirection of fields.

4 Conclusion

Our novel object-oriented approach simplifies development of large, complex applications. By holding references to class instances of other modules communication between modules becomes easier, because it can be done via explicit function calls which makes the program flow easier to read for the developer. Parameters and results are passed via function parameters and return values. Thus, the modules "field-interfaces" are reduced. Using object-oriented development, the resulting code is less error-prone, a module's functionality is clearer and it is easier to maintain and to extend.

The main drawback of our novel approach is the additional slight overhead in module development. But this overhead is small compared to the overhead of communication via fields and field listeners. Another disadvantage is that MeVisLab's `callLater` functionality cannot handle class member functions. Here the functional approach still has to be used. But this might change in future version of MeVisLab.

Our object-oriented development concept is already used in various applications and prototypes based on MeVisLab. Although it is rather a programming concept than a framework by now, i.e. it does not offer common functionalities used in medical applications itself, it is a suitable base for such a framework in MeVisLab and there are efforts in building an application framework based on the proposed programming approach.

References

[BvUW$^+$07] I. Bitter, R. van Uitert, I. Wolf, L. Ibáñez, and J.M. Kuhnigk. Comparison of Four Freely Available Frameworks for Image Processing and Visualization That Use ITK. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):483–493, 2007.

[HLP03] H.K. Hahn, F. Link, and H.-O. Peitgen. Concepts for Rapid Application Prototyping in Medical Image Analysis and Visualization. In *SimVis – Simulation und Visualisierung*, pages 283–298, Ghent, 2003. SCS.

[KSR$^+$06] M. Koenig, W. Spindler, J. Rexilius, J. Jomier, F. Link, and H.O. Peitgen. Embedding VTK and ITK into a visual programming and rapid prototyping platform. In *Proc. SPIE Medical Imaging*, volume 6141 of *Image Processing*, pages 796–806, 2006.

[MeV09] MeVisLab homepage, 2009. http://www.mevislab.de.

[MHK$^+$08] B. Merkel, M.T. Harz, O. Konrad, H.K. Hahn, and H.-O. Peitgen. A novel software assistant for clinical analysis of MR spectroscopy with MeVisLab. In *Proc. SPIE Medical Imaging*, volume Vo. 6915 of *Progress in Biomedical Optics and Imaging*, pages 69152R–1 – 69152R–9, 2008.

[Pyt09] Python documentation, 2009. http://www.python.org/doc.

[RJS$^+$05] J. Rexilius, J. Jomier, W. Spindler, F. Link, M. König, and H.O. Peitgen. Combining a Visual Programming and Rapid Prototyping Platform with ITK. In *Bildverarbeitung für die Medizin*, pages 460–464, Bvm 2005, 2005. Springer-Verlag Berlin/Heidelberg.

[RKHP06] J. Rexilius, J.M. Kuhnigk, H.K. Hahn, and H.-O. Peitgen. An Application Framework for Rapid Prototyping of Clinically Applicable Software Assistants. In *Proc Workshop Softwareassistenten*, volume 93 of *Lecture Notes in Informatics*, pages 522–528. Springer, 2006.