# How the GUIB works

Anton Kokalj

February 24, 2004

**Abstract**

This document describes briefly the GUIB: (i) how it works, and (ii) how to make a GUI application by using the GUIB engine. It is highly recommended that the document from the GUIB web page entitled *Description* is read first.

## Contents

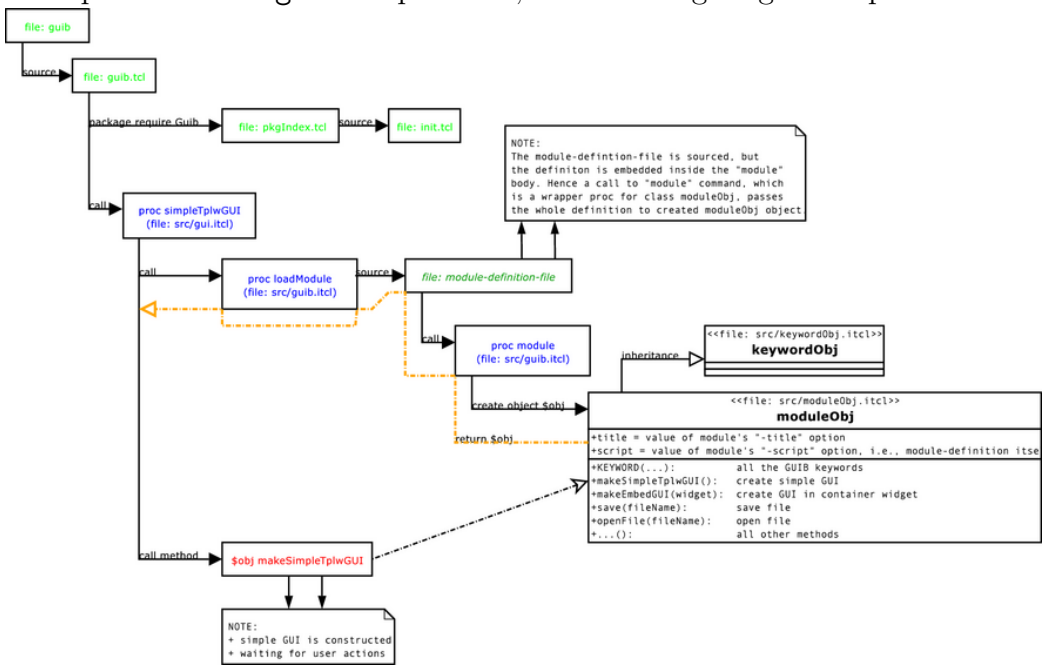


## 1 The easiest way

The easiest way to create a simple GUI application, is to use the `guib` script that comes with GUIB package. Let say that your module definition-file is named `myGUI.def`, then a simple GUI will be constructed by the following execution:

```
guib myGUI.def
```

To explain how the `guib` script works, the following diagram is provided:




This diagram shows that the module definition-file is source from the **loadModule** procedure. Technically, the module definition-file is a Tcl script, which uses the GUIB library. The whole script is embedded inside the **module** keyword, which is a wrapper for the **moduleObj** class. This means that by a call to **module** an object of **moduleObj** type is constructed. This is a complex object, and it holds the whole GUI definition.

To illustrate—from programming point of view—how the definition of the GUI is stored in the **moduleObj** type object, let us consider the following example:

```
module \#auto -title "Testing" -script {
    page p1 -name "Page No.1" {
        line l1st -name "1st line" {
            var title -label "Title:"
            var code  -label "Code:"
        }
        line l2nd -name "2nd line" {
            var description -label "Description:"
        }
    }
```

```
    line llast -name "last line" {
        var conclusion -label "Conclusion:"
    }
}
```

Here the following GUIB keywords are used: **module**, **page**, **line**, **var**. The GUIB keywords can be divided into two groups: objects and items. The keywords such as **page** and **line** are object-keywords as they construct new object of a given type, while the **var** keyword is an example of the item-keyword, i.e., item keywords do not create new objects, but their content is stored in corresponding object-keyword. Which GUIB keyword is object- or item-like can be guessed from the syntax. Consider the following:

```
line l1st -name "1st line" {
    var title -label "Title:"
    var code  -label "Code:"
}
```

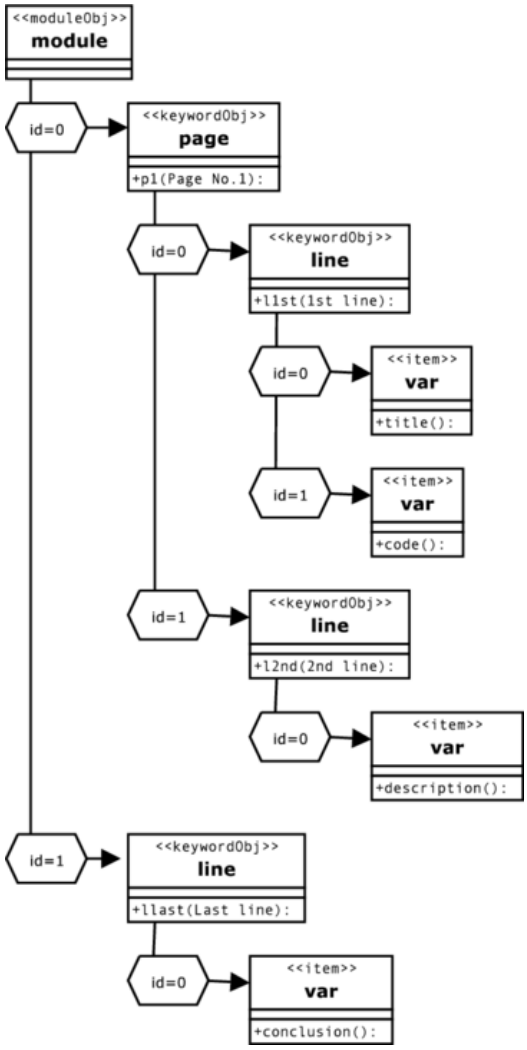
Obviously, the **line** keyword is an object-keyword, and in this examples it holds two **var** item-type keywords.

The base-class of keyword objects is **keywordObj** class. This class possesses a mechanism for storing the information of the module definition file in hierarchical fashion. Let say that we want to retrieve all information of the module definition file that was already stored in **moduleObj** type object. This will be done recursively and will look like this:

```
proc retrieve_and_DoWhatever {obj} {

    set NItem [$obj getID]
    for {set id 0} {$id <= $NItem} {incr id} {

        # check if $id keyword is object-type !!!
        set childObj [$obj getChild $id]

        if { $childObj != "" } {
            #
            # keyword is OBJECT-type, manage it recursively
            #
            ...
            retrieve_and_DoWhatever $childObj
            ...
```

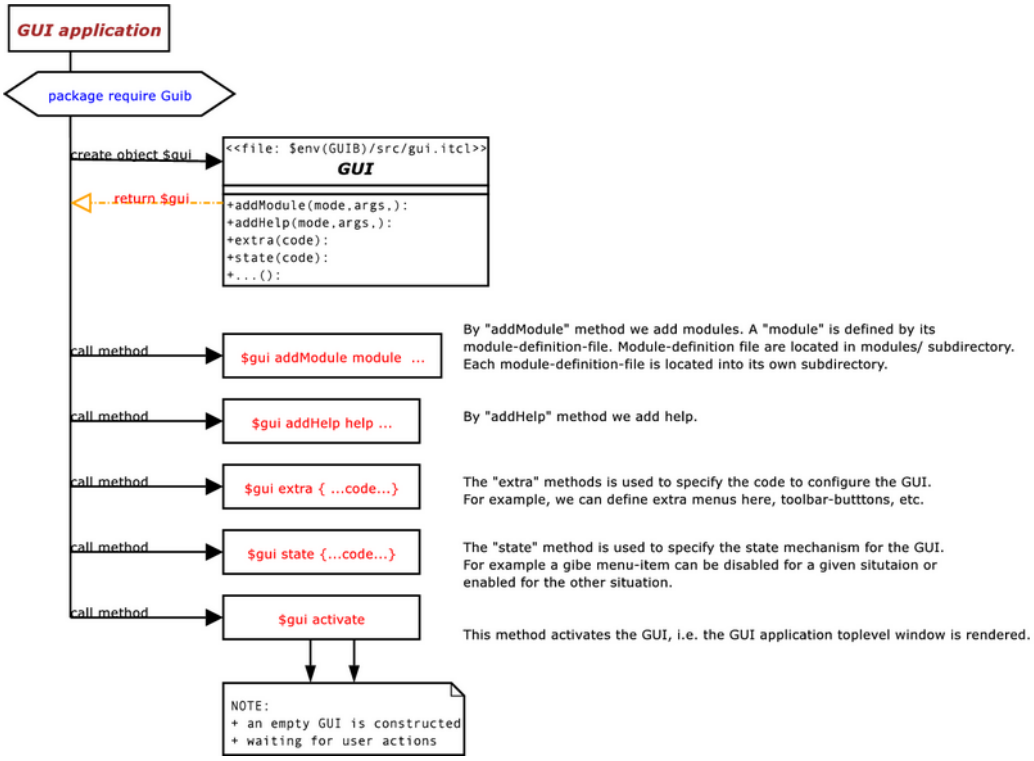
```
        } else {
            #
            # keyword is ITEM-type
            #
            ...
        }
    }
}
```

To illustrate more schematically, how the information about the module-definition file is stored, the following diagram shows this graphically for the above **module #auto -title "Testing" -script {...}** example.




# 2 A more elaborate way

The GUI constructed by the `guib` script can only hold one module definition file. Moreover no configuration is possible. If we want to create a configurable GUI that will be able to create/manipulate several different input files, than a **GUI** class provided by the GUIB package can be used. Below scheme shows the idea in diagrammatic fashion:



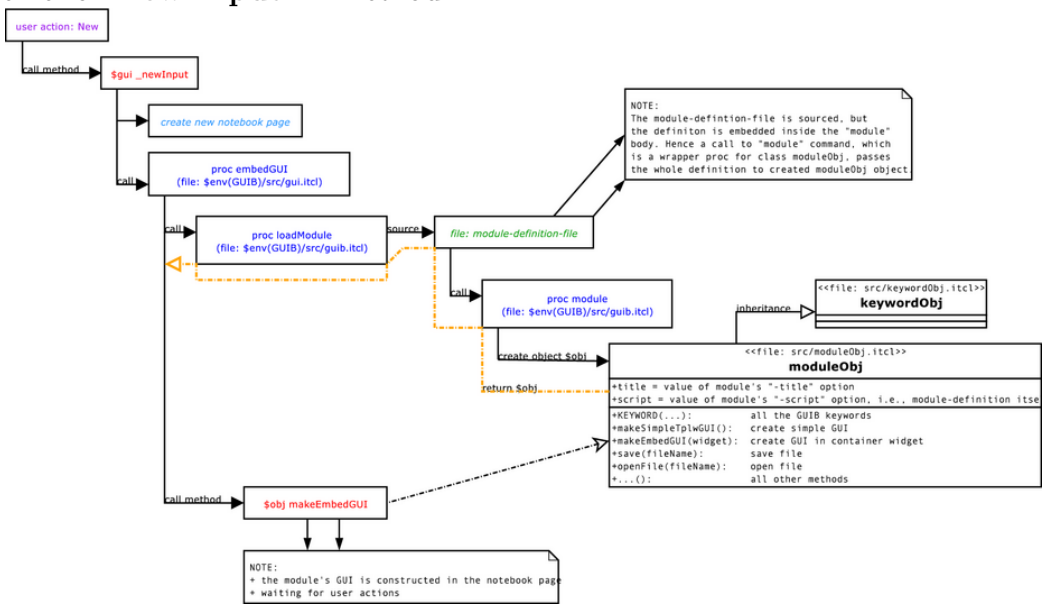

Now let us do a sample GUI application that will use the **GUI** class to construct a configurable GUI, which will be able to create/manipulate several input files. Let's say that the module definition files for these inputs are stored in files `myInput-1.def`, and `myInput-2.def`. Let's say we also have help files with description of both input formats stored in files `myInput-1.html`, and `myInput-2.html`. Now we have to create a GUI script, for example:

```
#!/bin/sh
# next line restarts wish \
exec wish

# ------------------------------------------------------------------------
#  INITIALIZATION
# ------------------------------------------------------------------------

# check if GUIB environmental variable is defined

if { [info exists env(GUIB)] } {
    # instruct the Tcl where to search for GUIB package
    lappend auto_path [file join $env(GUIB)]
```

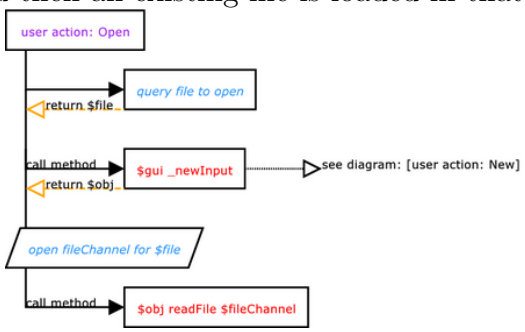
```
} else {
    # we assume that GUIB package is on some "standard" path and Tcl
    # will find it.
}

# load a Guib package
package require Guib 0.1.1

# withdraw the "." toplevel window, and bind the <Destroy> event
# to ::guib::exitApp
wm withdraw .
bind . <Destroy> ::guib::exitApp


# ----------------------------------------------------------------------
#  GUI construction
# ----------------------------------------------------------------------

# construct the GUI object
set gui [::guib::GUI \#auto -title "My 1st GUI" -appname MyGUI]

# Add modules. The syntax is:
# ------------
# $gui addModule module $moduleID $moduleLabel $moduleFile
#
$gui addModule module inp1 "My Input-1" myInput-1.def
$gui addModule module inp2 "My Input-2" myInput-2.def

# Add help. The syntax is:
# ---------
# $gui addHelp help $helpID $helpLabel $helpFile
#
$gui addHelp help help1 "Help for Input-1" myInput-1.html
$gui addHelp help help2 "Help for Input-2" myInput-2.html


#
# some extra configuration of the GUI
#
$gui extra {
    #
    # add a logo to toolbar panel
    #
    image create photo myLogo -format gif -file myLogo.gif
```

```
    set tb   [component toolbar]
    set logo [$tb add label logo -image myLogo]
    pack configure $logo -side right
}
```

When GUI defined in above file will be launched, an application toplevel window with menubar and toolbar will appear. But none of the GUI will be rendered. This can be done by selecting either **File** → **New Input ...** or **File** → **Open Input ...** menu. The following diagram shows a programming scheme for **New Input ...** method:

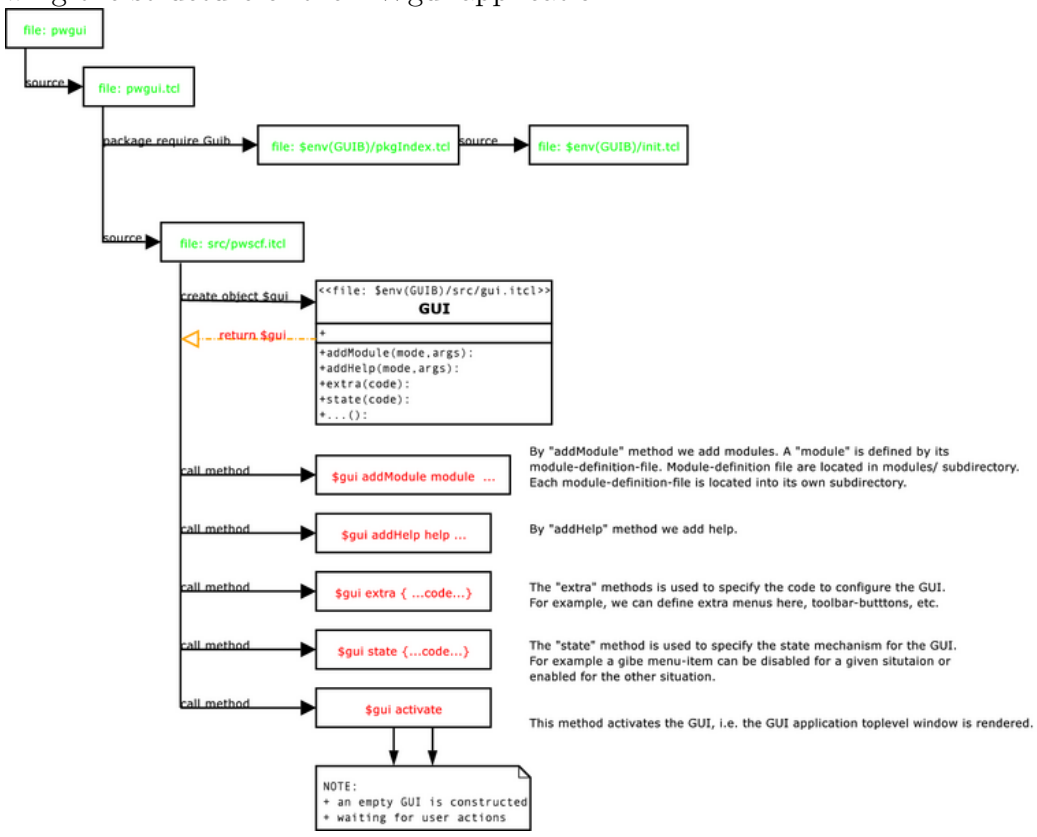



The scheme for the **Open Input ...** is very similar. In fact the "open" method first call the "new" method, which constructs an appropriate GUI, and then an existing file is loaded in that GUI. The scheme is shown here:



# 3 An example of real GUI application: PWgui

PWgui is a GUI for PWscf set of numerical programs for electronic structure calculations. It is a real GUI application that uses the GUIB engine. It is only a slightly more complicated than above GUI example. Here is a diagram showing the structure of the PWgui application:




# 4 File and directory structure of GUIB

**Directory structure:**

- `lib/` auxiliary Tcl/Tk routines, i.e., tclUtils.tcl, tkUtils.tcl
- `doc/` programming documentation generated from source-code
- `examples/` a few examples (definition files: *.tcl; input files: *.inp)
- `external/` external **cmdline** library
- `images/`
- `src/` GUIB source code directory

**Description of some src/ files:**

| | |
|---|---|
| • `moduleObj.itcl` | implementation of **moduleObj** class |
| • `keywordObj.itcl` | implementation of **keywordObj** class |
| • `guibKeywords.itcl` | implementation of all GUIB keywords |
| • `build.itcl` | build Tk-based GUI defined by module definition file. |
| • `open.itcl` | open an input file |
| • `save.itcl` | save edited input file |
| • `guib-keywords-def.tcl` | definition of options of GUIB-keywords |
| • `gui.itcl` | application GUI: procs for "simple-GUI" and implementation of more elaborate **GUI** class |