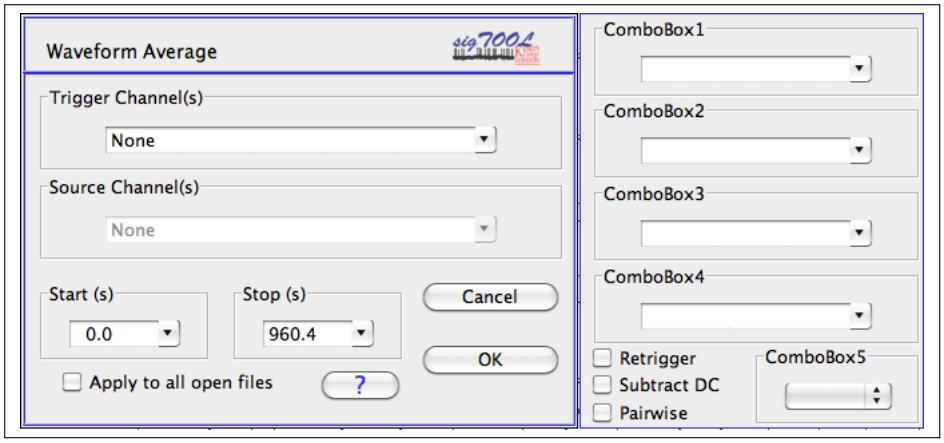
KING'S College LONDON



Making MATLAB® Swing More

Project Waterloo GUI Import Support

Malcolm Lidierth
Wolfson Centre for Age-Related Diseases

`http://sigtool.sourceforge.net/`

Revised: 9th March 2012

Acknowlegements:

MATLAB code in this document was styled using Florian Knorn's
M-code LaTeXPackage.
http://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package

The document was prepared using Pascal Brachet's TeXMaker
http://www.xm1math.net/texmaker/

Contents

Purpose

The *GImport* class enables easy import of graphical user-interfaces (GUIs) designed using Java Swing. With this class you can import GUIs designed in external GUI-designers such as those in NetBeans[1], Eclipse[2] and IntelliJ IDEA[3] and display them in standard MATLAB figures and uipanels.

The *GImport* class is part of Project Waterloo and requires the *jcontrol* class from that project which is available from `http://sigtool.sourceforge.net/`.

This document explains the use of *GImport*. It is not a guide to Java use in MATLAB. For that, I recommend Yair Altman's *Undocumented Secrets of MATLAB Java Programming*, (ISBN 9781439869031) Chapman & Hall/CRC Press, 2011.

[1] `http://netbeans.org/`
[2] `http://www.eclipse.org/`
[3] `http://www.jetbrains.com/idea/`

GImport use is extremely easy. For example, given a custom MATLAB class called *SimplePanel* in a package called *examples*, install the GUI into a MATLAB figure using

```
g=GImport(figure(), examples.SimplePanel());
```

The imported GUI and all of its components will produce 'normalized' resize behaviour just like standard MATLAB uicontrols. All components of the GUI will be programatically available in MATLAB. Their properties can be altered on-the-fly and you can add and remove components just as you might in Java. The panel is populated with 10 Swing check boxes in a simple grid. Figure 1 show the panel displayed in a MATLAB figure window using the code above.

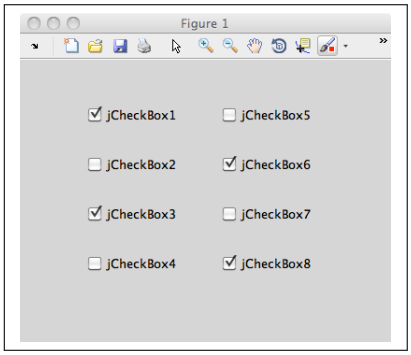


Figure 1: A simple GUI

Working with imported GUIs in MATLAB

To work with imported components in MATLAB requires access to their handles. The *GImport* class has several convenience methods to make this easy: *getHandles*, *getComponents* and *find*. Additionally, the *getValues* method returns the values of the GUI components.

The getHandles method

To get the handles of the buttons as a structure, call the *getHandles* method:

```
s=g.getHandles()
```

which, for the GUI shown above, will display the following

```
SimplePanel: [1x1 javahandle_withcallbacks.examples.SimplePanel]
  jCheckBox1: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox2: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox3: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox4: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox5: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox6: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox7: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
  jCheckBox8: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
```

The names of fields of the structure are shown to the left of the ":" on each line and the contents to the right. Field names are derived from the GUI as described further below in . To access individual items use standard syntax such as:

```
>> s.jCheckBox4.getText()
ans =
jCheckBox4
```

The getValues method

The values of all of the GUI items can be retrieved using the *getValues* method which returns a strucure with the values returned by each component in the *getHandle*s structure. Fot the checkbox example above

```
g.getValues()
```

returns

```
SimplePanel: []
   jCheckBox1: 1
   jCheckBox2: 0
   jCheckBox3: 1
   jCheckBox4: 0
   jCheckBox5: 0
   jCheckBox6: 1
   jCheckBox7: 0
   jCheckBox8: 1
```

The value for the panel is empty. Other fields are set true (1) or false (0) depending on the state of the checkbox as seen in Figure 1. Where a component returns a string, the string will be converted to numeric values where possible. The *getValues* method returns an empty value for components that have no value such a JPanel and may also return empty for non-standard Swing components. In those cases, user-code will need to retrieve the value explicitly via the handle.

The getComponents method

While *getHandles* returns a structured list of components and *getValue* returns a matching structure of their values, *getComponents* returns a flattened vector of components as a cell array.

```
g.getComponents();
```

produces:

```
[1x1 javahandle_withcallbacks.examples.SimplePanel ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
```

As described more fully below, the *GImport* constructor has an optional input that can be used to simplify the structure returned by the *getHandles* method and to include only those components that the user selects at the GUI design stage. In contrast, *getComponents* always returns a full list of components in the hierarchy.

Adding OK and Cancel buttons

Often, MATLAB code execution will need to be stopped while a GUI is displayed to allow user interaction. Figure 2 shows a modified simple panel that has 2 added buttons: OK and Cancel. Take a look at the structure, s, returned by *s=getHandles()* method:

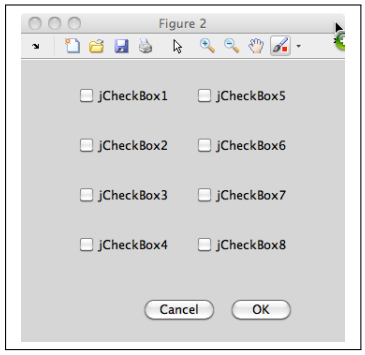


Figure 2: The GUI from Fig. 2 with OK ansd Cancel buttons added

```
SimplePanelOK: [1x1 javahandle_withcallbacks.examples.SimplePanelOK]
  Cancel: [1x1 javahandle_withcallbacks.javax.swing.JButton]
  OK: [1x1 javahandle_withcallbacks.javax.swing.JButton]
    jCheckBox1: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox2: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox3: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox4: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox5: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox6: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox7: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
    jCheckBox8: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
```

The OK and Cancel buttons now appear in the list. Their callbacks can be programmed as usual but the *GImport* class provides convenience methods for this. Construct the instance using:

```
g=GImport(figure(), examples.SimplePanelOK());
uiwait();
```

setOK, setCancel and setQueryOnCancel methods

MATLAB code execution will be suspended by the uiwait() command until the Cancel or OK buttons are clicked because the *GImport* constructor automatically recognises buttons labelled "OK" and "Cancel" in the top level container and sets up the callbacks accordingly. To program these explicitly use the *setOK*, and *setCancel* or *setQueryOnCancel* methods e.g.

```
g.setOK(g.getHandles().OK);
g.setQueryOnCancel(g.getHandles().Cancel);
```

OR

```
g.setOK(g.getHandles().OK);
g.setCancel(g.getHandles().Cancel);
```

These set up the MouseClickedCallbacks of the specified components as follows:

1. Cancel
   The GUI will be deleted together with the *GImport* instance and a *uiresume()* will be issued.

2. QueryOnCancel
   An option pane will appear and the user will be asked to confirm that they meant to press Cancel[4] . If the answer is Yes, the GUI will be

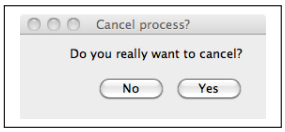


Figure 3: TheQuery on Cancel option pane

   deleted together with the *GImport* instance and a *uiresume()* will be issued.

3. OK
   The *getValues()* method will be called to retrieve the values in all components. These will be stored in the *GImport* instance. The GUI will be deleted and all handles in the *GImport* instance will be cleared. A *uiresume()* will be issued. The *GImport* instance will not be deleted so can be queried as before with the *getValues* method which will return the values stored when OK was clicked. The instance will then be deleted by the *getValues* method.

[4]A static method, *GImport.setIcon*, allows the user to set an icon that will be displayed to the left in this option pane

Linking instances

Often, there will be only one OK or Cancel button but several *GImport* instances containing seperate, related GUIs. The createLink and removeLink methods allow additional GUIs to be linked.

```
g.createLInk(GImportInstance);
```

marks *GImportInstance* to be deleted along with *g*. When OK is clicked, *getValues* will be run on all linked linked *GImport* instances before the display is removed.

```
g.removeLInk(GImportInstance);
```

removes any link to *g*. Maintain links in a the main *GImport* instance contining the OK and Cancel buttons. Links are unidirectional.

Custom GUI design

By default, the *GImport* constructor adds all components to the structure returned by the *getHandles* and *getValues* methods which then mimics the Swing component hierarchy. These structures can be cumbersome with even moderately complex GUIs. Also, if the GUI is edited, the path to a field in the returned structure may change. The *GImport* constructor has an optional flag which causes only selected components to be added to the hierarchy.

```
g=GImport(MATLABcontainer, javaobject, includeFlag);
```

Setting *includeFlag* to *false*, causes the constructor only to include specified components in the structure. Specify them by setting the Name property of individual components at the GUI design stage. Components that have an empty Name property will not be included. For those where the Name property is set, the first character determines whether the component will be included and can be '$' or '+' :

1. '$' The component will be included
2. '+' The component will be included together with all of its subcomponents

Note that this has no effect on the layout of components: only of the structure returned by the *getHandles* and *getValues* methods. The effect is to

flatten, or partially flatten, the structures returned by *getHandles* and *getValues* making further programming easier and making it likely that minor revisions of a GUI design will not substantially alter the structures returned - thus not breaking MATLAB code dsigned to deal with them.

This is best illustrated with a real GUI. Fig. 4 shows one from the sigTOOL Project that this code was developed to support. A JPanel contains two other JPanels. That at the top just contains a title and logo. That beneath has a set of JComboBoxes and JButtons. Each of the JComboBoxes is contained in a JPanel with a TitledBorder.

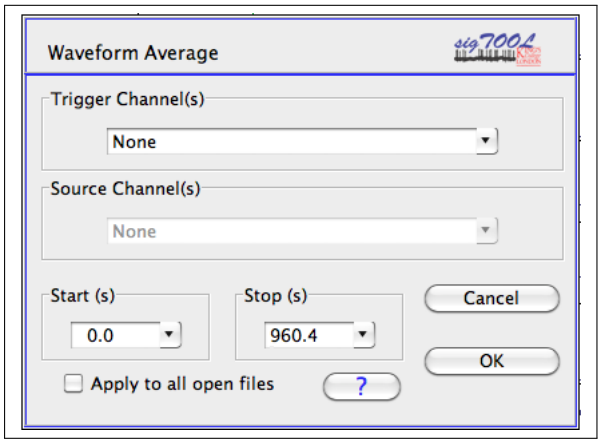


Figure 4: A GUI from the sigTOOL project

If this were imported with the includeFlag set to true e.g. with:

```
g=GImport(uipanel(figure(1), 'Units', ...
    'normalized', 'Position', [0.2 0.2 0.2 0.7]), ...
    kcl.sigtool.gui.DefaultPanel(), true);
```

the structure returned by getHandles() would look as follows:

```
DefaultPanel: [1x1 javahandle_withcallbacks.kcl.sigtool.gui.DefaultPanel]
       JPanel: [1x1 struct]
     JPanel_0: [1x1 struct]
```

JPanel_0 is the lower panel. To access the combobox labelled "Source Channel(s)" we would need to access it via its titled JPanel and use the rather cumbersome:

```
g.getHandles().JPanel_0.JPanel_1.ChannelB
```

to dig through the nested panels. This would return all the components that make up the JComboBox which is a compound component:

```
ChannelB: [1x1 javahandle_withcallbacks.javax.swing.JComboBox]
AquaComboBoxButton: [1x1 javahandle_withcallbacks.com.apple.laf.AquaComboBoxButton]
AquaComboBoxUI0x24AquaCustomComboTextField: ...
[1x1 javahandle_withcallbacks.com.apple.laf.AquaComboBoxUI$AquaCustomComboTextField]
CellRendererPane: [1x1 javahandle_withcallbacks.javax.swing.CellRendererPane]
```

The only component of real interest is the JComboBox which has the Name property of "$ChannelB" set by the GUI designer. Things are much simpler when the includeFlag is set to false:

```
g=GImport(uipanel(figure(1), 'Units', ...
    'normalized', 'Position', [0.2 0.2 0.4 0.4]), ...
    kcl.sigtool.gui.DefaultPanel, false);
```

Now *getHandles* returns a flattened structure that only has those components *Name*d with a string starting with "$" (or "+", but that is not used here):

```
g.getHandles()
```

returns

```
DefaultPanel: [1x1 javahandle_withcallbacks.kcl.sigtool.gui.DefaultPanel]
ApplyToAll: [1x1 javahandle_withcallbacks.javax.swing.JCheckBox]
Cancel: [1x1 javahandle_withcallbacks.javax.swing.JButton]
ChannelA: [1x1 javahandle_withcallbacks.javax.swing.JComboBox]
ChannelB: [1x1 javahandle_withcallbacks.javax.swing.JComboBox]
Help: [1x1 javahandle_withcallbacks.javax.swing.JButton]
Logo: [1x1 javahandle_withcallbacks.javax.swing.JButton]
OK: [1x1 javahandle_withcallbacks.javax.swing.JButton]
Start: [1x1 javahandle_withcallbacks.javax.swing.JComboBox]
Stop: [1x1 javahandle_withcallbacks.javax.swing.JComboBox]
Title: [1x1 javahandle_withcallbacks.javax.swing.JLabel]
```

Note how all the components that have been appropriately named are now in the list. This includes the *JButton* that carries the project logo (so the icon can be set on the fly), and the OK and Cancel buttons which will be automatically detected and supported by the *GImport* constructor. To access the combo box above now just needs

```
g.getHandles().ChannelB
```

Access to all components is still available through the *getComponents* and *find* methods but the structure returned by *getHandles* will often prove easier to use. If we want to customise a feature not in the structure, just use standard Java: to change the title of the JPanel that houses the combo box for example:

```
combobox=g.getHandles().ChannelB;
combobox.getParent().getBorder().setTitle('Change ...
    the title to this');
```

This change will be done on the EDT. A single GUI can then be designed but customised on-the-fly in MATLAB to suit different analyses. The GUI in Fig. 4 is used repeatedly in sigTOOL. To keep things simple, a second GUI allowing optional arguments for different analyses is kept separate by associated with the principle GUI as required. Figure 5 shows the result - in this case with the generic combobox titles still showing: they would normally be customised on-the-fly for each analysis type.

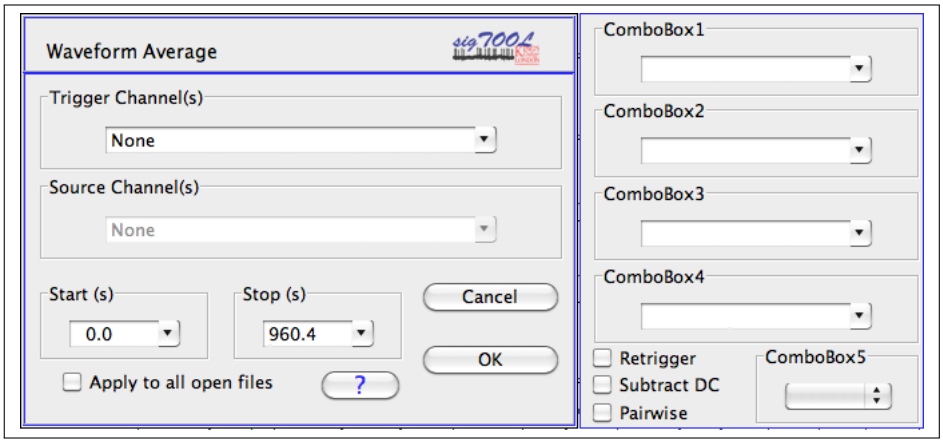


Figure 5: The GUI of Fig. 4 extended with the standard options panel in sigTOOL

alignToRight and alignToSouth

In Fig. 5 , two GUIs have been designed and aligned alongside each other using the *alignToRight* method which simply sizes the second (right side) GUI so that it has the same height as the first - then re-centers the combined GUI.

```
g1=GImport(uipanel(figure(1), 'Units', ...
    'normalized', 'Position', [0.2 0.2 0.5 0.7]), ...
    kcl.sigtool.gui.DefaultPanel,false);
g2=GImport(uipanel(figure(1), 'Units', ...
    'normalized', 'Position', [0.2 0.2 0.2 0.4]), ...
    kcl.sigtool.gui.DefaultPanel,false);
```

```
g1.alignToRight(g2);
```

Note that the height of *g2* will set to the height of *g1* following the *alignToRight* call.

A similar method, *alignToSouth* aligns two GUIs one above the other.

Both GUIs will be represented as separate *GImport* instances but a link can be created between the two using the *createLink* method as described below.

createLink and removeLink

In the example above, both GUIs were represented in separate *GImport* instances. A link can be created between the two using the *createLink* method:

```
g1.createLink(g2);
```

This affects only the behaviour of the OK and Cancel buttons: Cancel will delete both GUIs while OK will call *getValues* on both GUIs before deleting them. To dissociate the GUIs, use the *removeLink* method:

```
g1.removeLink(g2);
```

Assigning fieldnames

The *GImport* constructor assigns field names in the structure return by *getHandles* using the following rules:

1. If any object returns a name from its *getName* method, that name will be used to derive the field name
2. If *getName* returns empty, *getLabel* will be tried followed by *getText* until a non-empty result is returned.
3. If all fails, the class of the item will be used to form the field name with a number added to the end of the string.

1 and 2 above allow the field names to be determined at the GUI design stage. The resulting field name will have any white space removed. Also, any text in brackets will be removed e.g. 'Range (V)' will give a field name of 'Range'. If the name has a leading '$' or '+' character, this will be removed.

In all cases, a check is made for duplication of field names. If a name is a duplicate, a number will be appended to the string[5]. This check is made only for the current level in the tree - names can be duplicated between sub-structures of a nested structure.The resulting name will be checked for validity as a MATLAB field name using *genvarname*, and the output of *genvarname* will be used to name the field.

When a component has included sub-components, its field name appears twice: once as the name of a field containing a structure and again within that structure as the name of the field containg the handle (or value) of the component.

Sorting field names

The top level component is always listed as the first field in the getHandle structure and any components with subcomponents appear as the first field in any sub-structure describing their contents. Remaining fields are sorted alphabetically at each level.

Customising a GUI in the GImport constructor

A final optional input to the *GImport* construtor allows a user-specified function to be run once a GUI has been imported. All GUIs are initially rendered at the size specified by the top component's PreferredSize if it is set, or at the size of the parent MATLAB container if not. A user-specified function will be run at this stage and before any new layouts are installed. The constructor then takes the form

```
g=GImport(MATLABcontainer, javaobject, ...
    includeFlag, fcn);
```

OR

```
g=GImport(MATLABcontainer, javaobject, ...
    includeFlag, cell array);
```

OR

[5]Numbering begins at zero and the number is incremented each time a number is assigned to a field name**. Numbers are pre-pended with '_' so that the letter 'l' and numbers '1' are not confused. (**or would be assigned if *includeFlag* were *true* - so numbering and field name assignments are more likely to be consistent regardless of the *includeFlag* setting).

```
g=GImport(MATLABcontainer, javaobject, ...
    includeFlag, string);
```

When specified, *fcn* should be a function handle that will be invoked with the top level java component (at this stage housed in a *jcontrol* object) as its input. If a cell array, element 1 should contain the function handle with elements 2 onward representing user-specified inputs to the function. If a string is supplied, this must be the name of a method for the java component requiring no inputs (use a cell array and a MATLAB function to invoke methods with inputs).

Setting GUI behaviour in MATLAB

Using uiwait and uiresume

Once a GUI is initialised, it can be customised before the *uiwait* function is invoked, e.g. by adding items to a *JList* or *JComboBox*. In general therefore, to *GImport* a GUI use:

```
g=GImport(figure(), examples.SimplePanel());
% Place user-code here
...
uiwait();
s=g.getValues();
% Place user-code here to act on the results ...
    returned by the GUI
...
```

The default behaviour of the OK and Cancel buttons can be overridden if a user-specified callcack is declared via the *setOK*, *setCancel* or *setQueryOnCancel* methods:

```
g.setOK(g.getHandles().OK,    @fcn);
g.setCancel(g.getHandles().Cancel,    @fcn);
% OR
g.setQueryOnCancel(g.getHandles().Cancel,    @fcn);
```

In this case the function fcn should be defined to accept 3 inputs, the handle of the button and the event object (as standard for MATLAB) together with the GImport instance:

```
function fcn(jObject, EventData, ...
    GImportObjectInstance)
% User code goes here
```

```
return
end
```

The *setOK* etc. methods set up the callbacks by calling the *GImport setCallback* method. This can also be called explicitly by the user to set up any callback where the *GImport* instance is to be passed as an argument as described below.

The setCallback method

User-specified callbacks can be set up for any of the components in the hierarchy using the MATLAB *set* command as usual. Note however, that if the *GImport* object is to be passed as an input to the callback function, as it was above, this will lead to leaked references to the object when it is destroyed. To avoid these leaks, use the *setCallback* method to set up the callbacks.

```
g.setCallback(jObject, callbackstring, @fcn, ...
    arg0, arg1, ...)
```

Here, *jObject* is the component whose callback is being set and *callbackstring* specifies the callback, eg. 'MouseClickedCallback'. The callback, *fcn*, should be declared as:

```
function fcn(jObject, Eventdata, arg0, arg1...)
% User code goeds here
return
end
```

Each *GImport* instance maintains a record of the callbacks set up via *setCallback* and clears those references in its destructor method before deleting the object. This will avoid reference leaks when the instance is deleted.

Customising the GUI on-the-fly

As described below, *GImport* can deal with far more complex GUIs than those above. Any Swing component can be included and components can be nested. Nonetheless, it will often be simplest to include several separate GUIs in a figure, each contained in a separate *GImport* instance. To look good, these will need to share the same fonts, color schemes etc. While this

will often best be achieved at the design stage, the *GImport* class provides methods to alter the properties of all components on-the-fly

invokeEDT

The *invokeEDT* method calls a specified Java method on all components of a *GImport* instance. The method is always invoked on the Java Event Despatch Thread (EDT). The *invokeEDT* method takes the form

obj.invokeEDT(methodname, arg0, arg1....)

where *arg0* etc are optional input arguments passed to the method. For example:

```
g=GImport(figure(), examples.SimplePanelOK());
g.invokeEDT('setFont', java.awt.Font('Arial', ...
    java.awt.Font.PLAIN, 16));
```

will cause all components to use a 16 point plain Arial font.

invoke

The *invoke* method is similar to *invokeEDT*, but can return values. In addition, the invoke method does not explicitly cause code to execute on the EDT. However, as the *GImport* constructor refers all components to the EDT using the MATLAB *javaObjectEDT* function, this will not usually be an issue[6] . To return values from a call to *invoke* use e.g.

```
colors=g.invoke('getBackground');
```

This will return a cell array with one value for each component in the hierarchy. For the panel above this might be:

```
  [1x1 java.awt.Color                                          ]
 [1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
 [1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
 [1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
 [1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
 [1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
 [1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
```

[6]Thread-safety is discussed further below

```
[1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
[1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
[1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
[1x1 com.apple.laf.AquaNativeResources$CColorPaintUIResource]
```

If the method fails on any component, the corresponding cell array element will be empty.

Note, as explained below, that the number of components can be greater than the number returned by *getHandles*. To retrieve all components, in the same order returned by *invoke* use the *getComponents* method:

```
g.getComponents()
```

returns

```
[1x1 javahandle_withcallbacks.examples.SimplePanelOK]
[1x1 javahandle_withcallbacks.javax.swing.JButton   ]
[1x1 javahandle_withcallbacks.javax.swing.JButton   ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
[1x1 javahandle_withcallbacks.javax.swing.JCheckBox ]
```

Static runEDT and run methods

Static methods *runEDT* and *run* work in the same way as the class methods *invokeEDT* and *invoke* but allow the programmer to supply a list of Java component handles on input. The components may or may not be components of a *GImport* instance. For example, to achieve the same result as with *invoke* above:

```
comp=g.getComponents();
colors=GImport.run(comp, 'getBackground');
```

The input *comp* can be a scalar instance, array or cell array of components.

The revalidate method

Components can be added/removed from the GUI hierarchy using standard Java. To refresh the handles structure and update management of the GUI from the GImport class methods simply call the *revalidate* method.

```
g.revalidate()
```

Note that this can alter the structure returned by *getHandles* and the update will be reflected in the cell array returned by *getComponents*.

The find method

As GUIs get more complex, perhaps with panels nested in panels it may be useful to retrieve a handle or handles using the *find* method. Figure 6 shows a GUI with several panels nested inside a main panel. Each panel

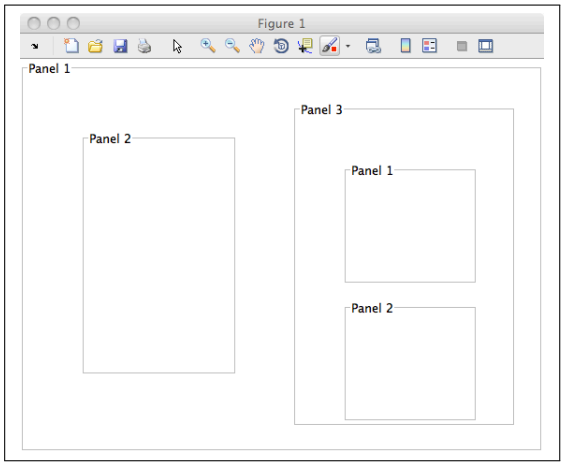


Figure 6: A GUI with some nested JPanels

has a titled border and the Name of each panel has been set to match these titles. Note that the titles "Panel 1" and "Panel 2" are duplicated: there is a Panel 2 in Panel 1, and Panels 1 and 2 inside Panel 3. Calling *getHandles* for this *GImport* instance gives:

```
>> s=g.getHandles()
s =
    Panel1: [1x1 javahandle_withcallbacks.examples.NestedPanels]
     Panel2: [1x1 javahandle_withcallbacks.javax.swing.JPanel]
```

```
    Panel3: [1x1 struct]
```

Note that blank space is removed from the field names in the structure. As Panel3 has several components, it too is a structure. Examine it with

```
>> s.Panel3
ans =
    Panel3: [1x1 javahandle_withcallbacks.javax.swing.JPanel]
    Panel1: [1x1 javahandle_withcallbacks.javax.swing.JPanel]
    Panel2: [1x1 javahandle_withcallbacks.javax.swing.JPanel]
```

Find takes 4 forms.

1. find(string)
   With a string as input, *find* returns a component or cell array of components for which string is the field name in the *getHandles* structure. For the example above,

   ```
   g.find('Panel1');
   ```

   returns

   ```
   [1x1 javahandle_withcallbacks.examples.NestedPanels]
   [1x1 javahandle_withcallbacks.javax.swing.JPanel   ]
   ```

2. find(propertyValue, propertyName)
   Returns a component or cell array of components for which *propertyValue* is matched by the value stored in *propertyName* (if it has an *is* or a *get* method). For example:

   ```
   g.find(java.lang.String('Test string'), 'Name');
   g.find(java.awt.Color.white, 'Background');
   g.find(true, 'Selected');
   ```

   There are two special case *propertyName*s:

   (a) 'fieldname' produces the same result as *g.find(string)* but can accept a *java.lang.String* on input.

   (b) 'bordertitle' returns objects with a border that returns *propertyValue* from a call to *getTitle()* on the border.

3. find(class)
   With a *java.lang.class* instance as input, *find* returns a component or a cell array of components matching that class. Thus,

   ```
   g.find(javax.swing.JPanel().getClass());
   ```

   returns

   ```
   [1x1 javahandle_withcallbacks.javax.swing.JPanel]
   [1x1 javahandle_withcallbacks.javax.swing.JPanel]
   [1x1 javahandle_withcallbacks.javax.swing.JPanel]
   [1x1 javahandle_withcallbacks.javax.swing.JPanel]
   ```

Note that the top panel, which extends the JPanel class, is not included in the output. Use an optional flag input to *find* to output all sub-classes of a class:

```
g.find(javax.swing.JPanel().getClass(), true);
```

returns

```
[1x1 javahandle_withcallbacks.examples.NestedPanels]
[1x1 javahandle_withcallbacks.javax.swing.JPanel   ]
[1x1 javahandle_withcallbacks.javax.swing.JPanel   ]
[1x1 javahandle_withcallbacks.javax.swing.JPanel   ]
[1x1 javahandle_withcallbacks.javax.swing.JPanel   ]
```

4. find(object)
   With an object specified on input, find returns the path name for the component in the *getHandles* structure. For example,

```
comp=g.getHandles().Panel3.Panel2;
g.find(comp);
```

returns

```
'Panel3'
'Panel2'
```

Plainly, this is intended for use where a component has been identified in some way other by than calling *getHandles*.

Constructor options

The *GImport* constructor accepts optional string arguments that control the settings during and immediately after import.

'guisize'

By default, the GUI is resized to fit the MATLAB container specified on construction. Use the 'guisize' option to resize the MATLAB container to the *PreferredSize* of the GUI.

```
g=GImport(MATLABcontainer, ...
    kcl.sigtool.gui.DefaultPanel(), true, [], ...
    'guisize');
```

show and hide methods and options

GImport provides *show* and *hide* methods that, unsurprisingly, show and hide the GUI. During instantiation, the GUI will be hidden and *show* will be invoked by default to display it. Override this with optional inputs at construction e.g.:

```
g=GImport(MATLABcontainer, ...
    kcl.sigtool.gui.DefaultPanel(), true, [], ...
    'hide');
g=GImport(MATLABcontainer, ...
    kcl.sigtool.gui.DefaultPanel(), true ,[], ...
    'show');%Default behaviour
```

g.show() and *g.hide()* can be invoked, as usual, on-the-fly.

'center'

The 'center' (or 'centre') causes the GUI to be centered within the parent of the MATLAB container specified on construction once displayed, e.g.:

```
g=GImport(MATLABcontainer, ...
    kcl.sigtool.gui.DefaultPanel(), true, [], ...
    'guisize', 'center');
```

If *MATLABcontainer* is a uipanel in a figure it will be resized to the size of the GUI then centered in the parent figure before *show* is called (by default).

'noresize'

Suppresses installation of the resize callback for the imported component.

'nolayout'

Suppresses all layout management. The GUI will be imported as usual but no resizing will be implemented by the *GImport* class methods and instal-

lation of the resize callback for the imported component will be suppressed. Use this if you have built in a MATLAB-friendly layout at the design stage.

Layout Management

To achieve the required MATLAB-like resize behaviour, *GImport* takes over the layout management for selected components in the heirarchy. These components have 'actively managed' layouts. *GImport* installs a SpringLayout[7] for these components, and updates the constraints whenever the component is resized.

The decision on whether to manage a layout actively is done at construction or when *revalidate* is called on a *GImport* instance. The default position is intended to give the behaviour a designer is likely to want so hopefully, users should not need to consider the details with most GUIs. Note that the management is installed where appropriate for each component in the hierarchy - not just for top component. Active management of the layout depends on:

1. Whether the layout installed by the GUI designer for the component is in a list of actively managed layouts
2. Whether the parent container for the component is in a list of excluded components

getManagedLayoutList

Get a list of the actively managed layouts by calling the static *getManagedLayoutList* method which returns a cell array of those layouts as strings:

```
GImport.getManagedLayoutList()
```

The default list (at the time of writing) is:

```
'java.awt.FlowLayout'
'java.awt.BoxLayout'
'java.awt.GridBagLayout'
'javax.swing.GroupLayout'
'javax.swing.SpringLayout'
```

[7] http://docs.oracle.com/javase/tutorial/uiswing/layout/spring.html

```
'com.jgoodies.forms.layout.FormLayout'
'org.jdesktop.swingx.HorizontalLayout'
'org.jdesktop.swingx.VerticalLayout'
'org.netbeans.lib.awtextra.AbsoluteLayout'
'org.jdesktop.layout.GroupLayout'
'com.intellij.uiDesigner.core.GridLayoutManager'
```

Editing the managed layout list

Alter the list of managed layouts using the following static methods

1. *addManagedLayout(string)* to add a layout to the list
2. *removeManagedLayout(string)* to remove a layout from the list
3. *setManagedLayoutList(cell array)* to replace the the list. The input should be a cell array formatted as above for the list returned by *getManagedLayoutList*

Note that, as the decision on whether to manage a layout actively is done at construction, changing the list of managed layouts only affects GImport instances created after the change. If a revalidate is issued on a *GImport* instance after editing the managed layout list, any newly added layouts will become managed but a removed layout will not be restored.

Imported GUIs can be positioned as usual using standard MATLAB simply by positioning the host MATLAB container. Several convenience methods are also provided: Note *vec* is a standard 4-element position vector

```
g.setPosition(vec, units);% units, if not ...
    specified, defaults to normalized
g.getPosition(units);
g.setpixelposition(vec);
g.getpixelposition();
g.setCentral();%Position GUI to center of parent
```

getExcludedClasses

Excluded classes are those whose layout will not be actively managed regardless of the layout installed at GUI-design stage. The decision on whether

to manage a layout actively is done at construction or when *revalidate* is called on a *GImport* instance.

Get a list of excluded classes by calling the static *getExcludedClasses* method which returns a vector of classes as a *java.lang.Class* array.

```
GImport.getExcludedClasses()
```

Excluded classes are those whose default layout behaviour is already as required, so there is no need to manage their layouts. At the time of writing, these are *JSplitPane*, *JTabbedPane* and *JViewport* (so that any component using a view port will be excluded). Note that any subclass of these classes will also be excluded and that other components will be excluded because their default layout managers are not in the list of actively managed layouts (as above).

Exclusion of the *JTabbedPane* is the only default exclusion users are likely to want to change. Tabbed panes uses a specialized layout that is platform-dependent and will not be actively managed although the tabbed pane components can be. How *GImport* performs with a *JTabbedPane* therefore depends on its contents. Exclusion by default allows the user to set layouts at the design stage that will perform well when a MATLAB figure is resized e.g. a FlowLayout. When included, the *JTabbedPane* is likely to give unsatisfactory performance with the some added components.

Alter the list of excluded components using the following static methods

1. *addExcludedClass(clzz)* to add a component to the list

2. *removeExcludedClass(clzz)* to remove a component from the list

3. *setExcludedClasses(clzz or clzz[])* to replace the the list.

In all cases, the input *clzz*, should be a *java.lang.Class* (or *java.lang.Class[]*), *not* a string returned from a call to the MATLAB class function. Note that, as the decision on whether to manage a layout actively is done at construction, changing the list of excluded components only affects GImport instances created after the change. If a *revalidate* is issued on a *GImport* instance after editing the excluded components list, any newly added layouts will become managed but a removed layout will not be restored.

Using GImport with JFrames

Using a MATLAB container allows MATLAB graphics to be added. If you have a pure Java GUI, you may prefer to use a simple *JFrame*. In that case, the GUI must manage its own layout.

If you create a JFrame independently, you can still use *GImport* to access its contents: just call *GImport* setting the target to empty:

```
g=GImport([], javaobjectinstance, includeFlag);
```

where javaobject instance is a Java swing object, typically a *JFrame*.

For convenience, the *GXJFrame* function returns a *JFrame* for use in MATLAB and with *GImport*.

```
frame=GXJFrame(figure, title, javaobject);
```

creates a *JFrame* centered on the specified figure. The *JFrame* is not otherwise associated with the figure. The javaobject, if specified, will be added to, and fill, the *JFrame*.

To use the *GXJFrame* function with *GImport*, create a JFrame with

```
frame=GXJFrame(figure, title);
```

then add the contents with a call to *GImport*:

```
g=GImport(frame, MyPackage.MyClass(), includeFlag);
```

Drag and Drop support

Two static methods and one class method control drag-and-drop support for imported components

putDragEnabled and setDragEnabled

To enable/disable dragging from a component or selection of components use the static *putDragEnabled* method

```
GImport.putDragEnabled(h, flag)
```

where *h* is a Java component handle or an array or cell array of handles and *flag* is *true* or *false*. The handles do not need to be components imported via *GImport*. Default drag behaviour for the components will be activated. In addtion,

```
GImport.putDragEnabled(h, flag, propertyname)
```

sets the *flag* then installs a *javax.swing.TransferHandler* with the specified *propertyname* e.g. 'background'.

As a convenience, the *setDragEnabled* method activates/deactivates dragging from all components of a *GImport* instance for which *setDragEnabled/setTransferhandler* is/are valid methods:

```
g.setDragEnabled(h, flag)
or
g.setDragEnabled(h, flag, propertyname)
```

To restore a default-compatible *TransferHandler* set *propertyname* to 'default'.

```
g.setDragEnabled(h, flag, 'default')
```

The *setDragEnabled* method will use reflection to generate a new *TransferHandler* with the default settings for each class in h.

For components that so not implement the Java *setDragEnabled* method, you will need to use the mouse event callbacks to set up custom drag behaviour (using e.g. the *GImport setCallback*).

putDropCallback

Any components that have drop support activated by default will implement the default drop behaviour. To customise the behaviour, use the

static *putDropCallback* method. Note that this method requires that the associated *DNDAssistant* class is installed on your MATLAB path (this is part of the Project Waterloo distribution and is included in the jar file)[8]. If *DNDAssistant* is not defined, a warning will be issued. To customise a drop from drag-and-drop, call:

```
GImport.putDropCallback(h, fcn);
```

where *h* is a Java component handle or array of handles and *fcn* is a function handle. When a drop occurs, the function will be called with two inputs: the handle of the Java component to which the drop was made and a *DNDAssistant* instance. Thus, the function declaration should take the form:

```
function fcn(jObject, DNDAssistantInstance)
...
return
end
```

The DNDAssistant class

A *DNDAssistant* instance can be used to access the data from a drop. It simply extends the usual AWT *DropTarget* class overriding the usual listeners. *DNDAssistant* and extracts and stores the transfered data before calling the appropriate superclass listener. This overcomes a problem that the DND system is not in the required state to extract information outside of the Java listeners including inside MATLAB callbacks.

DNDAssistant implements the following methods:

1. *getFlavors* returns a list of available data flavors as *a java.lang.Object[]*.
2. *getData* returns the available data as a *java.lang.Object[]*, with one element for each flavor returned by *getFlavors*.
3. *isLocal* returns *true* if the source of the drag was within the same Java Virtual Machine as the drop (and *false* otherwise)

It is left to the user to handle the data as appropriate within the callback function specified with *putDropCallback*.

[8]The *DNDAssistant* class is defined in the Project Waterloo kcl.jar or kcl-matlab.jar file which is provided in the full download from `http://sigtool.sourceforge.net/`

Appendix 1: GUI designers

While GUIs can be programmed by hand, often to good effect, several GUI-based GUI-designers are available in open-source Integrated Development Environments and it can often be convenient to use these. If these GUIs link against external libraries, as they usually will, those libraries will need to be made available at run-time, usually by copying them to the project /lib folder during the build phase.

Eclipse

Although I use NetBeans more extensively, the Eclipse GUI is the best in my view. It also generates Java source code that can be modified by hand and the GUI 'form' design will automatically update. Many other GUI designer tools do not allow the code to be modified outside of the design tool.

To export a MATLAB loadable *jar* file, use File->Export and select the Java/Jar file option.

NetBeans

With NetBeans, design a class that extends a standard Swing container. Typically it will extend the javax.swing.JPanel class. To create a new Project, choose File->New Project->Java Class Library. Activate the 'Use Dedicated Folder for Storing Libraries' checkbox.

Right click again on the project in the Project Manager and choose Properties. Select 'Build->Packaging' and actiavate the 'Copy Dependent Libraries' checkbox.

Click the 'Source Packages' folder in Project Manager, right-click and select New->Java package. Click on the package name in the Project Manager and select New->JPanel Form.

Design the GUI(s), build the project and import the resuting *jar* file into MATLAB using *javaaddpath* at command line.

IntelliJ IDEA

With IntelliJ, design a class that extends a standard Swing container. Typically it will extend the javax.swing.JPanel class. To create a new Project,

choose File->New Project and select 'Create project from scratch'. Name the project as you want, activating the 'Create Module' and 'Create source directory' checkboxes then click Next until the File Editor appears.

Click on the project in the Project Manager and select the 'src' folder. Right-click and select New->Package. Right click on the newly created package and select New->GUI Form. Select a Layout Manager (if new to Java try the 'FormLayout (JGoodies)' layout). Ensure that your design is associated with a top level Swing container.

Select Code->Generation source code editor and add a contructor for your class. Make sure that at least the top level swing container is public so that you can import it into MATLAB (it will be private by default).

Select File->Project Structure, select 'Artifacts' and click the small yellow '+' button at the top of interface. Select 'Jar' and 'From Modules with Dependencies'. You do not need a 'Main' class. Select the 'copy to output directory and link via manifest' checkbox.

Select Build->Build Artifacts to generate a *jar* file. Import the resuting *jar* file into MATLAB using *javaaddpath* at command line.

Note that with IntelliJ the GUI to be imported will typically reside in a property of the developed class e.g. *MyClass.panel1* unless