Golit Functionality in Detail 4

This chapter explains in detail how to use Golit to generate source code and a makefile. It suggests ways to alter the Makefile and the generated callback templates. It also describes how to use libgolit functions to instantiate parts of the generated widget tree and how to use Create Procs to implement your own widget creation routines.

Designing Golit Interfaces in Devguide

Before you use Golit, you must first use Devguide to design an interface and save it as a GIL file. The GIL file stores the interface using a generic description language that identifies each element and lists its properties.

To help keep your files straight during interface editing and compiling, you should create a single directory to store all the source code files for a single program: the `.P` and `.G` files for the interface, the Golit-generated source code files for the interface, your own custom source code files for the software underlying the interface, and the makefile.

Handling Events With Connections

To make your interface handle events, you specify *connections* in Devguide. When an event occurs on one object (called the *Source*) a connection triggers a specified action on a second object (called the *Target*).

Use the Connections Manager window to define connections. Connections are discussed in detail in the *OpenWindows Developer's Guide: User's Guide*.

The When menu items that appear for a specified Source object represent the events that can occur on that object. The Action menu items that appear for a specified Target object represent actions that are valid for that object.

Golitgenerates all the code necessary to execute any action except ExecuteCode and CallFunction. If you select ExecuteCode, a pop-up window which you can enter code in appears. Golit generates a callback function that includes this code in the `_stubs.c` file. If you select CallFunction, you must also specify a function name. Golit generates a template callback under that name in the `_stubs.c` file.

When Menu Items

The When menu items you can choose from are listed below, along with a brief description of when the corresponding connection will be invoked.

- Adjust - the user clicks the OPEN LOOK mouse ADJUST
- AnyEvent - any keyboard or mouse event
- Create - the source object is created (create proc)
- Destroy - the source object is destroyed
- Done - (before) popup window is dismissed
- DoubleClick - SELECT is pressed twice quickly
- DroppedUpon - drop target receives drop
- Enter - the pointer enters a window or pane
- Exit - the pointer exits a window or pane
- Keypress - the user presses any key on the keyboard
- Menu - the user clicks the OPEN LOOK mouse MENU
- Notify - the user selects a control UI element
- Popdown - menu pops down
- Popup - menu pops up
- Repaint - the user repaints a window or pane
- Resize - the user resizes a window or pane
- Select - the user clicks the OPEN LOOK mouse SELECT
- Unselect - the user lets up the OPEN LOOK mouse SELECT

Note that Golit generates the same resource for many of the When items. It is up to the application to distinguish between them. Table 4-1 on page 49 lists the resource generated for each When item.

Table 4-1 When Items and the Resources Generated for Them

When Item	Resource
Adjust	XtNconsumeEvent
AnyEvent	XtNconsumeEvent
Destroy	XtNdestroyCallback
Done	XtNpopdownCallback
DoubleClick	XtNconsumeEvent
DroppedUpon	XtNdndTriggerCallback
Enter	XtNconsumeEvent
Exit	XtNconsumeEvent
KeyPress	XtNconsumeEvent
Menu	XtNconsumeEvent
Notify	XtNselect, XtNsliderMoved, or XtNverification
Popdown	XtNpopdownCallback
Popup	XtNpopupCallback
Repaint	XtNexposeCallback
Resize	XtNresizeCallback
Select	XtNconsumeEvent
Unselect	XtNunselect

Note that the resource generated for Notify depends on the source object.

Action Menu Items

The Action menu items you can choose from are listed below, along with a brief description of their functionality.

- CallFunction - calls the specified function
- Disable - disables the Target object
- Enable - enables the Target object
- ExecuteCode - executes the specified code
- GetLabel - returns the Target object's label

- GetValueNumber - returns the Target object's numeric field value
- Hide - makes the Target object invisible
- LoadTextfile - loads the specified text file
- SetLabel - sets the Target object's label
- SetValueNumber - sets the Target object's numeric field value
- Show - displays the Target object

Setting up Create Procs to Conduct Your Own Widget Creation

To conduct some operations, such as setting fonts or registering help, you must intercept Golit's default widget creation and provide your own widget creation routines. To do this, you use *Create Procs*. A Create Proc is a special type of callback that is invoked when a user interface object is created.

To generate a Create Proc, you specify a connection for it in Devguide. You must make sure the event is set to Create, and the Source and Target are both set to the object for which you want to generate the Create Proc. You must also set the Action to CallFunction. You can specify a single function as the Create Proc for several objects.

Note – If you do not set the Source and Target to the same object when you specify a Create Proc in the Devguide Connections Manager window, Golit will not generate the Create Proc template.

Using Golit to Generate Code

`golit` is the executable program that takes the Devguide GIL (`.G`) or project (`.P`) file and generates C files and the `Makefile`. The format of the `golit` command is as follows:

```
golit [option flags] <UIFileName>
```

Where `UIFilename` is the name of the GIL or project file that you want to generate code for.

GolitCommand-Line Options

Golit provides the following command-line options:

-h (-help)

Displays a message describing Golit usage and options.

l-i or -international

Writes all Leve 3 internationalization-specific resources into a resource file. For more information on internationalizing OLIT applications, see Chapter 6, "Internationalization."

-k or -kandr

Writes K&R C code that doesn't contain function prototypes. Golit produces ANSI C code by default.

-m (-main)

Generates code only for the `main()` program `<project_name>.c` and `<project_name.h>` files. Only works with `the -p` option. Use the `-m` option if you have already run Golit on a project's `.G` files.

-n or -nomerge

Keeps Golit from merging existing and new `_stubs.c` files. If you are prototyping an interface and have never added any code to the `_stubs.c` file, you should run Golit in `-n` mode. This prevents unused callbacks from accumulating in your `_stubs.c` file. If you have added code to your `_stubs.c` file, don't use `-n`, since Golit will omit the code you added.

-p project

Generates code for a project. When you use `-p`, `UIFileName` must be a project name.

-r or -resources

Writes *all* resources into a resource file.

-s or -silent

Instructs Golit to operate silently; no messages will appear.

Files Generated for a Single GIL File

Golit names its generated files after the original GIL filename. It strips off the `.G` extension and adds new extensions to identify each file. If you generate code for a single GIL file, Golit generates the following files:

- `<GIL_filename>_ui.c`
- `<GIL_filename>_ui.h`
- `<GIL_filename>_stubs.c`

For example, if you use Golit to generate source code for a GIL file named `newt.G`, Golit generates the files `newt_ui.c`, `newt_ui.h`, and `newt_stubs.c`. Golit also creates a makefile under the name `Makefile`, if it doesn't already exist. Figure 4-1 provides an overview of the files Golit generates for a single GIL file (`fn`) and the other files necessary to create an executable program.

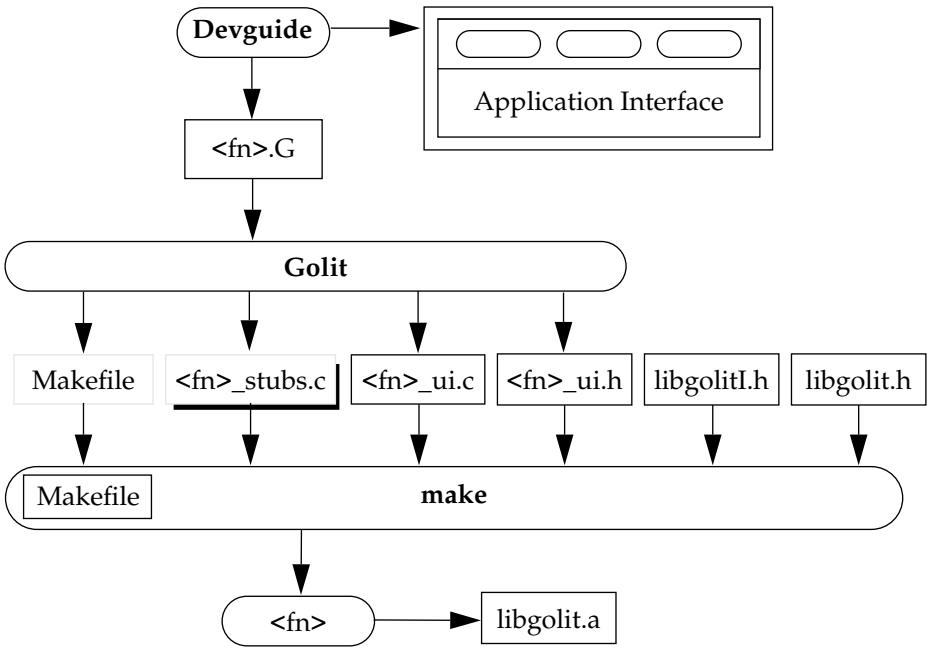

Figure 4-1 Overview of Golit Files for a Single GIL File

In Figure 4-1, the following conventions apply:

Rounded boxes indicate programs.

Square boxes indicate data files.

Boxes with drop shadows indicate files for which <fn>.BAK files are created.

Dashed boxes indicate data files that you can modify.

4

_ui.c File

This file calls macros to create a widget tree that describes the interface in the GIL file. The widget tree comprises statically initialized data structures (*widget descriptors*) that describe the objects in the user interface. Each widget descriptor provides the following information:

- a reference to the object's immediate sibling
- a reference to the object's children
- a reference to the object's popups
- the instance name of the object
- the object's class
- a reference to the object's resources and their values
  - ArgList
  - Typed ArgList
- a reference to the object's callbacks
- a method of instantiating the object

The libgolit routines in the generated `main()` program and your application code use the widget descriptors to instantiate widgets.

Note – Do not modify the `_ui.c` file by hand. Any changes you make to this file will be lost when you regenerate code.

Note that there isn't a one-to-one correspondence between objects you design in Devguide and widget descriptors in the tree. Many Devguide objects consist of a hierarchy of several OLIT widgets. For example, a Text Field object with a label in Devguide is actually two widgets in the Golit generated code: a textField widget and a caption widget.

The `_ui.c` file provides external *widget description pointers* that enable you to access widget descriptors. When Golit generates the widget hierarchy for a Devguide object, it bases the widget description pointer names on the name of the object. If a Devguide object comprises only one widget, Golit creates the widget description pointer by joining the GIL file name and the Devguide object name. For example, the widget description pointer for a Message object called `mymessage` in a GIL file named `mygilfile` will be `Mygilfile_mymessage`. Note that the first letter of the name is automatically capitalized.

If a Devguide object comprises more than one widget, Golit adds different prefixes to the object's name to create the widget descriptors. For example, if you create a Text Field object (with a label) named `mytextfield` in `mygilfile`, Golit names the textfield widget description pointer `Mygilfile_mytextfield` and the caption widget description pointer `Mygilfile_G_Caption_mytextfield`.

Figure 4-2 and Figure 4-3 show examples of widget trees created by Golit. If you create an interface (called `fn`) with a base window (`window1`) that has a single control area (`Controls1`) with a textfield in it (`textfield1`) and a single popup window (`popup1`), Golit generates the widget tree depicted in Figure 4-2. If you create the base window with a footer, Golit generates the widget tree depicted in Figure 4-3. The widget class for each widget in these trees is shown in parentheses. The solid lines indicate a parent-child relationship. The dotted line indicates a sibling relationship.

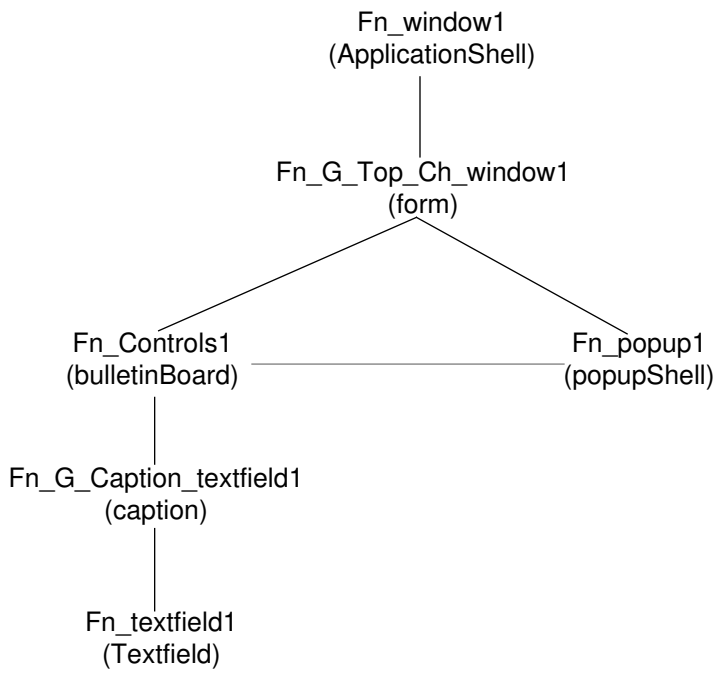

Figure 4-2 Example Widget Tree for a Base Window

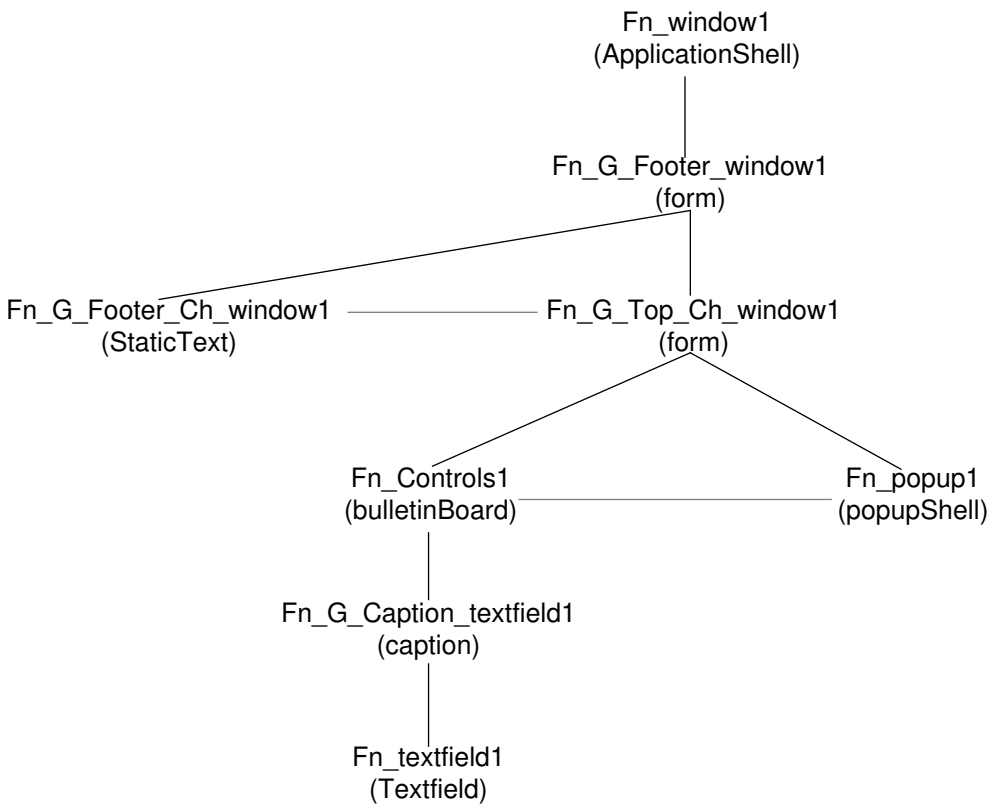

Figure 4-3 Example Widget Tree for a Base Window with a Footer Panel

Table 4-2 and Table 4-3 provide a list of the widgets used to created each Devguide Object.

_ui.h File

This file contains prototypes of all the functions and callbacks. The Golit code generator automatically places an `#include` statement in the `_ui.c` file to include this header file. Do not modify the `_ui.h` file by hand.

Normally, you don't need to look at the `_ui.h` file. However, it can help you quickly ascertain the names of Golit generated widget descriptions available to the application programmer. At the top of every `_ui.h` file is a series of macros that declare all the generated widget description pointers available to the application programmer. These macros take the form `ExternWidgetDesc(wdp)`, where `wdp` is a widget description pointer, such as `Fn_window1` or `Fn_G_Top_Ch_window1`.

_stubs.c

The `_stubs.c` file consists of two principal parts: the `main()` program and the callbacks.

The main() Program in the _stubs.c File

The `main()` program performs initialization for the Xt Intrinsics and the OLIT toolkit and sets up an event handling loop. It uses libgolit functions to instantiate the widget tree: it calls `GolitFetchShellHier()` to instantiate the base window and all of its children and `GolitFetchWidgetHier()` to instantiate each popup (and all of its children) in your interface. These functions traverse the widget tree described in the `_ui.c` file, instantiating a widget for each widget descriptor in the tree.

The libgolit functions give each widget they create an *instance name*. The instance names of the widgets created in the `main()` program are the same as those of the widget description pointers that describe them. For example, if you create a base window and call it `basewindow` and save it in a GIL file called `fn`, the base window's widget description pointer *and* its instance name will be `Fn_basewindow`.

The `main()` program also provides code that applies any fallback resources you have specified.

Callbacks in the _stubs.c File

For each connection you specify in Devguide, Golit generates a callback function template in the `_stubs.c` file. There are two principal varieties of callbacks:

- Predefined action callbacks
  These templates are generated for connections for which you specified a Devguide-defined action (for example, Show or Hide). The template contains all the code necessary to carry out the action and a `printf()` statement that prints the callback name to the console.
- CallFunction callbacks
  These templates are generated for connections for which you specified CallFunction as the action in Devguide. The template uses the function name that you specified in Devguide. It is empty except for the `printf()` statement.

You can add your application code to either type of callback template. Note that predefined action templates already provide a `GolitNameToWidget()` call to retrieve the target widget id.

The following arguments are passed to callbacks:

- widget - the Source widget specified for the connection
- clientData - not implemented
- callData - widget specific data for the callback, for example, slider values

Makefile

This file is a standard C template makefile that builds the executable. When you run Golit, it checks the current directory to see if a file called `Makefile` already exists. If there is no `Makefile`, it generates one. If it detects a `Makefile`, it does not generate one. This feature protects a custom makefile, ensuring that Golit doesn't overwrite it.

You can easily customize `Makefile` to compile your own code files. The beginning of the `Makefile` lists the source files in a section labeled "Parameters." The five parameters are:

- `PROGRAM`, which lists the name of the executable file created during compilation
- `SOURCES.c`, which lists the names of user-supplied C source files for the application
- `SOURCES.h`, which lists the names of user-supplied header files for the application

- SOURCES.G, which lists the names of GIL files used to store interfaces for the application

When Golit first generates the Makefile, the SOURCES.c and SOURCES.h parameters are not set. To add your own source code files to the Parameters list, you must add their filenames by hand. For example, if you have several C source files that provide functions called in the callbacks, you must include their names in the SOURCES.c parameter.

Files Generated for a Project

If you run Golit on a project file, Golit generates _ui.c, _ui.h and _stubs.c files for each GIL file in the project. It also generates the following files:

- <project_name>.c
- <project_name>.h
- <project_name>.make

<project_name>.c File

When you generate code for a project, Golit generates the main() function in the <project_name>.c file.

This file also contains all callback functions that are common to more than one .G file in your project. For example, suppose you create a project which includes two menus saved in separate .G files. Both menus have a Save menu item for which you specify a CallFunction connection to a function called mysave. Golit generates the code for mysave in the .c file. The mysave callback does not appear in the _stubs.c file for either menu.

You can add variable initializations and other application code to the <project_name>.c file.

<project_name>.h File

This file contains the external declarations for the interface objects in the project. It includes the _ui.h files from the GIL files in the project and any objects common to the GIL files. You can add your own declarations to this file.

<project_name>.make File

This file contains lines that are inserted into the makefile to compile the `_stubs.c` files from the different GIL files in the project.

Regenerating Code for a Modified Interface

To regenerate code for an interface after you have modified it, simply re-run Golit on the GIL or project file. If you want to re-run Golit and immediately proceed to compile the generated code in one step, just type `make`.

When you re-run Golit, it overwrites the original `_ui.c` and `_ui.h` files. However, it preserves any application code which you have added to the `_stubs.c`, `<project_name>.c` and `<project_name>.h` files by merging this code in with any new code it generates.

Note – If you change the action for a connection in Devguide, the change will not be reflected in the new `_stubs.c` or `<project_name>.c` file unless you manually delete the original callback for that connection before you re-generate code.

Regenerating Code for an Individual GIL File

If you re-run Golit on a single GIL file, it generates the following files:

- new _stubs.c - Golit creates this file by merging the old _stubs.c with any new code it generates. Any source code you added to the old _stubs.c file is preserved in the new one.
- `_stubs.c.BAK` - back-up of the original `_stubs.c` file
- `_stubs.c.delta` - a list of differences between the original and the new `_stubs.c` file

Regenerating Code for a Project

If you re-run Golit on a project, it merges the `_stubs.c` for each GIL file in the project. It also generates the following files:

- new <project_name>.c - Golit creates this file by merging the old <project_name>.c file with any new code it generates. Any source code you added to the old <project_name>.c is preserved in the new one.
- new <project_name>.h - Golit overwrites the existing <project_name>.h file
- `<project_name>.c.BAK` and `<project_name>.h.BAK` - backups of the original `<project_name>.c` and `<project_name>.h` files
- `<project_name>.c.delta` - a list of differences between the original and the new `<project_name>.c` files

Removing Obsolete Callbacks

When Golit merges the original `_stubs.c` and `<project_name>.c` into the new files, it does not delete or replace any of the original callbacks. This results in unused callbacks accumulating in the following cases:

- When you remove an object or a connection in Devguide
  Golit does not delete callbacks associated with deleted connections or connections for an object that has been deleted. You must remove these callbacks manually.
- When you change the name of an object or connection in Devguide
  Golit adds a new callback for any connection that has a new name, or that connects objects that have new names. It retains the original callback along with any application code you inserted in it. Be sure to transfer that application code to the new callback. You can then delete the original callback.

Integrating Your Application Code with the Interface Code

To add your application code to the Golit-generated interface code, you insert it in the callback templates in the `_stubs.c` file. Simply replace the `printf()` statement in each callback with your own code. Of course, you can include calls to functions that you keep in other source files. Just remember to modify the `Makefile` so that the Make utility will compile these files.

There are essentially three things your application code can do to the interface you designed in Golit:

- change widget resources
- create additional instances of parts of the widget tree

- intercept Golit default widget creation and provide a custom creation procedure (*Create Proc*).

These operations are discussed in the subsections below.

Changing Widget Resources

To change a widget's resources, you need to know the widget's id. Since the `main()` program generated by Golit does not return all the widget ids when it creates the interface, you have to include code to get the widget ids you want. The libgolit library provides functions that enable you to retrieve ids.

Retrieving Widget ids

The `GolitNameToWidget()` function returns a widget id when you pass it a widget instance name, and a location in the tree to start searching for the widget. You use it as follows:

```
Widget w;

w = GolitNametoWidget(root, "widget_name");
```

where:

`widget_name` is the instance name of the widget for which you want to get an id.

`root` is a widget id that specifies where you would like to start searching the widget tree for the widget designated by `widget_name`. Normally, you can just use the widget that is passed to the callback.

The `GolitFetchLastWidgetID()` function returns the id of the last widget created of a given type. You use it as follows:

```
Widget w;

w = GolitFetchLastWidgetID(last_widget_description_ptr);
```

where:

`last_widget_description_ptr` is the widget description pointer of the widget for which you want to get an id. This will be something like `Fn_window1`, or `Fn_button2` (where `fn` is the name of the GIL file).

Getting the Right Widget Instance Name

As discussed in "_ui.c File" on page 54, there isn't a one-to-one correspondence between objects you design in Devguide and widgets in the widget tree. Many Devguide objects consist of a hierarchy of several OLIT widgets.

When you specify a connection for an object in Devguide, the generated callback is only passed the id of one of the widgets that the source object comprises. Normally this design will not pose any problems since the returned id is for the widget you are most likely to want to manipulate.

For example, if you put a Text Field object with a label in your interface in Devguide, Golit creates a textField widget and a caption widget. If you set up a CallFunction connection for the Text Field object, the callback is only passed the textField widget (and not the caption widget). Normally, this is convenient since you are more likely to want to manipulate the textField widget. If you want to manipulate the caption, you must find out its instance name and use `GolitNameToWidget()` to get its id.

It is relatively easy to figure out the instance name. For the widgets that the Golit-generated `main()` instantiates, the instance names are the same as the widget description pointers. Golit bases the widget description pointers on the names of the corresponding Devguide objects. Table 4-2 and Table 4-3 show Devguide objects and the widget hierarchy that Golit uses to create each of them. The Widget Description Pointer column shows the patterns which Golit uses to create each widget description pointer (assuming the GIL file is called `fn`). In each widget hierarchy, the widget that appears in bold type is the one that is passed to callbacks for the object. Note that if you do not specify a label for an object in Devguide, Golit omits the caption widget from the object's hierarchy.

To demonstrate how you use Table 4-2 and Table 4-3, take the example of a Exclusive Setting object. In Devguide, you name the setting `setting1`. You provide two choices: `choice1` and `choice2`. You provide a label for the setting and save it in a GIL filed called `fn`. Golit creates the following widget description pointers (and instance names) for the Setting object:

`Fn_G_Caption_setting1`	(caption widget)
`Fn_setting1`	(Exclusives widget)
`Fn_setting1_choice1`	(Rectbutton widget)
`Fn_setting1_choice2`	(Rectbutton widget)

Note that the first letter of each name is capitalized. If you were to set up a Callfunction callback for the setting object, the callback would be passed the id of `Fn_setting1` (the Exclusives widget).

Table 4-2 Devguide Panes and Windows and their Golit Widget Hierarchies

Devguide Object	Widget Hierarchy	Widget Description Pointer
Control Area	**bulletinBoard**	Fn_<objectname>
Canvas	scrolledWindow **DrawArea**	Fn_G_Scr_Win_<objectname> <objectname>
Text Pane	scrolledWindow **textEdit**	Fn_G_Scr_Win_<objectname> <objectname>
Base Window	**applicationShell** form	Fn_<objectname> Fn_G_Top_Ch_<objectname>
Base Window (with footer)	**applicationShell** form form StaticText	Fn_<objectname> Fn_G_Footer_<objectname> Fn_G_Top_Ch_<objectname> Fn_G_Footer_ch_<objectname>
Popup Window	**popupShell**	Fn_<objectname>
Menu	**menuShell**	Fn_<objectname>

Table 4-3 Devguide Control Objects and their Golit Widget Hierarchies

Devguide Object	Widget Hierarchy	Widget Description Pointer
Scrolling List	caption* **ScrollingList**	Fn_G_Caption_<objectname> Fn_<object_name>
Slider	caption* **Slider**	Fn_G_Caption_<objectname> Fn_<object_name>
Gauge	caption* **Gauge**	Fn_G_Caption_<objectname> Fn_<object_name>
Text Field	caption* **TextField**	Fn_G_Caption_<objectname> Fn_<object_name>
Multiline Text Field	caption* scrolledWindow **TextEdit**	Fn_G_Caption_<objectname> Fn_G_Scr_Win_<objectname> Fn_<object_name>
Exclusive Settings	caption* **Exclusives** RectButton**	Fn_G_Caption_<objectname> Fn_<objectname> Fn_<object_name>_<buttonname>
Nonexclusive Settings	caption* **Nonexclusives** RectButton**	Fn_G_Caption_<objectname> Fn_<objectname> Fn_<object_name>_<buttonname>
Checkbox Settings	caption* **Exclusives** CheckBox**	Fn_G_Caption_<objectname> Fn_<objectname> Fn_<object_name>_<checkboxname>
Settings Stack	caption* AbbrevMenuButton **Exclusives** RectButton**	Fn_G_Caption_<objectname> Fn_G_Abbrev_<objectname> Fn_<objectname> Fn_<objectname>_<buttonname>
Button	**OblongButton**	Fn_<objectname>
Menu Button	**MenuButton**	Fn_<objectname>
Message	**StaticText**	Fn_<objectname>

*If you do not specify a label in Devguide, Golit does not create the caption widget.
**Buttons and boxes in settings can have their own callbacks.

4

Instantiating Parts of the Widget Tree with libgolit

The `main()` program generated by Golit calls libgolit functions to create one instance of each object in your interface. You can also use the libgolit functions within your application code to create additional instances of an object or part of the widget tree. You simply re-use the widget descriptors in the `_ui.c` file and provide different instance names.

If your interface uses a number of objects that are similar, you can save time by instantiating the objects with libgolit library functions. Instead of using Devguide to design each possible object, you just design one object. You then include code to repeatedly instantiate the object and customize it as needed at runtime. This saves you time and makes your code much more efficient.

The following are the libgolit functions you can use to instantiate parts of the widget tree:

- `GolitFetchWidget()` - instantiates the specified widget
- `GolitFetchWidgetUnmanaged()` - instantiates the specified widget, but does not add it to its parent's managed set
- `GolitFetchChildren()` - instantiates the children of the specified widget
- `GolitFetchChildrenUnmanaged()` - instantiates the children of the specified widget, but does not add them to the widget's managed set
- `GolitFetchPopups()` - instantiates the popups of the specified widget
- `GolitFetchPopupsUnmanaged()` - instantiates the popups of the specified widget, but does not add them to the widget's managed set
- `GolitFetchWidgetHier()` - instantiates the specified widget, its popups, and its children
- `GolitFetchWidgetHierUnmanaged()` - instantiates the specified widget, its popups, and its children but does not add the widget to its parent's managed set.
- `GolitFetchShellHier()` - instantiates the shell widget and all its children

Since there isn't always a one-to-one correspondence between objects you design in Devguide and widgets in the widget tree, you should be careful to use a function that creates the portion of the object you want. When in doubt, it is usually safest and easiest to use `GolitFetchWidgetHier()`.

The functions all have similar arguments. The following is the syntax for `GolitFetchWidgetHier()`, the function that you will probably use most frequently:

```
Widget
GolitFetchWidgetHier (String  name,
 GolitWidgetDescPtr wdp,
 Widget parent,
 ArgList args,
 Cardinal num_args,
 XtPointer closure);
```

Where:

`name`	The instance name of the widget whose hierarchy you want to instantiate.
`wdp`	The widget description pointer. This refers to an object that you set up in Devguide, for example, `Fn_button1` or `Fn_popup2`. `GolitFetchWidgetHier()` uses the widget descriptor specified by `wdp` to set the resources of the widget.
`parent`	The id of the parent of the widget you want to fetch.
`args`	A pointer to the Arg structure for the widget you want to fetch.
`num_args`	The number of args in args.
`closure`	A pointer to the application data structure that you want to pass down the widget tree.

Example of Using libgolit to Instantiate Part of the Widget Tree

Let's say you want to create a popup data entry form. You want the user to be able to open several separate copies of the form at once; each one should be numbered so the user can tell which one is which. However, you don't know exactly how many copies the user might want to open.

If you were to create separate forms in Devguide, you would have to create an arbitrarily large number. This would be an awkward process and would create bulky and inefficient interface code. With libgolit routines, there is a much easier way to solve this problem. You just prototype one form in Devguide and add a libgolit call to the `_stubs.c` file to instantiate the form whenever the user presses a button:

1. **Create an interface with a base window and a popup.**
   Put whatever elements the form includes in the popup. Use Devguide's default names for the objects.

2. **Create a button in the base window.**
   Specify a connection for the button that shows the popup - that is, set the Source to button1, the Target to popup1, and the Action to Show.

3. **Save the interface and run Golit on it**
   For this example, call the interface `ex2`. The resulting `_stubs.c` file will contain a callback like the one shown in Figure 4-4.

```
void
button1_connectionCB1(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
      Widget target = GolitNameToWidget(widget,
                                 "Ex2_popup1");

      XtPopup(target, XtGrabNone);
      printf("connection: button1_connectionCB1\n");
}
```

Figure 4-4 Generated Callback that Displays a Popup Window

4. **Modify the `button1_connectionCB1()` callback.**

   a. **Add code that creates a sequential instance name for the popup.**

   b. **Call `GolitFetchWidgetHier()` to instantiate the popup.**
      Use the popup object you designed in Devguide as the prototype (or widget description pointer) and the sequential name as the instance name.

c. Customize the instance of the popup.
Call `GolitNameToWidget()` with the popup widget's instance name to get its id. Then use Intrinsics functions to set the popup's title to its instance name.

The modified `button1_connectionCB1()` in Figure 4-5 demonstrates how you might do this. This modified callback gives each popup a title of the pattern, "popup<n>", where n is the number of the popup. Note that the callback should not attempt to instantiate the popup the first time the user presses the button. This is because the main program has already instantiated the entire widget tree—including the popup—once.

```
void
button1_connectionCB1(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
      char       buf[32];
      Arg        arg[1];
      static int popup_number;
      Widget     target;
      /* Create sequential instance name in buf for each popup */
      sprintf(buf,"popup%1d",++popup_number);

      /* Instantiate a popup for each press after the first */
      if (popup_number > 1)
      {
      target = GolitFetchWidgetHier(buf, Ex2_popup1, widget,
                                NULL, NULL, NULL);
      }
      /* Get the popup's widget id */
      target = GolitNameToWidget(widget,buf);

      /* Set the popup's title to match its instance name */
      XtSetArg(arg[0], XtNtitle, buf);
      XtSetValues(target, arg, 1);

      /* Make the popup appear */
      XtPopup(target, XtGrabNone);
}
```

Figure 4-5 Callback that Displays a New Popup Window for each Button Press

4

Using Create Procs to Provide Your Own Widget Creation

Golit's design allows you to intercept Golit's default widget creation and provide your own creation procedure (Create Proc). This enables you to perform special operations immediately before and after widget creation. For example, prior to creation, you may want to specify some resources. After widget creation you may want to:

- Get a widget id
- Add help for a widget with `OlRegisterHelp()`
- Use `SetValues()` on a widget just after its creation
- Use a specific font for a user interface object

To set up a Create Proc in Devguide, see "Setting up Create Procs to Conduct Your Own Widget Creation" on page 50. Each time an object is instantiated, the Create Proc you have assigned to it will be called. A Create Proc must return a widget id. The Create Proc template that Golit generates includes a call to the appropriate libgolit function (for example, `GolitFetchWidgetHier()`) to return the widget id; this function also instantiates the widget and its children. The arguments for the function are the same as those passed to the Create Proc.

Caution – Treat Create Procs as you would treat any other callback. Do not call them from your application code.

Example of using a Create Proc to Register Help for an Object

Let's say you have designed a Text Field in Devguide and you want to register help for it at its creation time. To do this, you make a Create Proc for the Text Field object:

1. **Set up the Create Proc in Devguide.**
   Set up a connection for the Text Field Object. Make sure that both the Source and the Target are set to the object (`textfield1` in this example). Set the When setting to Create and the Action to CallFunction. Call the function `create_textfield`.

2. **Use Golit to generate the code.**
   Golit generates a template similar to that shown in Figure 4-6.

```
Widget
create_textfield1(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      printf("create proc: create_textfield1\n");
      return(GolitFetchWidgetHier(name, wdp, parent, args,
                                  num_args, closure));
}
```

Figure 4-6 Template for the Create Proc, `create_textfield()`

3. **Modify the** `create_textfield()` **callback.**

   a. **Use** `GolitFetchWidgetHier()` **to instantiate the textField widget and save the widget's id.**

   b. **Use** `OlRegisterHelp()` **to perform the help registration.**
   Figure 4-7 shows how to modify `create_textfield1()` to do this.

Note that the only widget that `GolitFetchWidgetHier()` gets an id for is the textField widget itself. It does not get the id for the caption widget that displays the Text Field caption or for any of the other widgets in the hierarchy. This caption widget is above the textField widget on the hierarchy (see Table 4-3 on page 65) and has already been instantiated. This means that you have only registered help for the textField widget. The help message will not appear if the cursor is on the Text Field label when the user presses the help button.

```
Widget
create_textfield1(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
Widget textfield;

      /* Instantiate the textfield and save the id  */
      textfield = GolitFetchWidgetHier(name, wdp, parent, args,
                                num_args, closure);

      /* Set help text to the specified string  */
      OlRegisterHelp(OL_WIDGET_HELP,
                textfield, "TextField Help",
                OL_STRING_SOURCE, "Enter some text on the
line.");

      return textfield;
}
```

Figure 4-7 Create Proc Modified to Register Help for a TextField Widget

To register help for the Text Field caption:

1. **Figure out the caption widget's instance name.**
   Assuming you called the interface `ex3`, you can do this by adding the prefix, `Ex3_G_Caption_`, to the Text Field object's name. Since the Text Field object's name is `textfield1`, the caption widget's name will be `Ex3_G_Caption_textfield1`.
2. **Use `GolitNametoWidget()` to get the caption's widget id.**
3. **Use `OlRegisterHelp()` to perform the help registration.**

Figure 4-8 shows how to modify the callback to do this.

```
Widget
create_textfield(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
Widget textfield, caption;

      /* Instantiate the textField widget and save the id  */
      textfield = GolitFetchWidgetHier(name, wdp, parent, args,
                                num_args, closure);

      /* Set help text to the specified string  */
      OlRegisterHelp(OL_WIDGET_HELP, textfield, "TextField
Help",
                  OL_STRING_SOURCE, "Enter some text on the
line.");

      /*  Get the widget id for the caption.  */
      /*  Start searching at the textfield widget */
      caption = GolitNameToWidget(textfield,
                               "Ex3_G_Caption_textfield1");

      /* Set help for the caption to the specified string */
      OlRegisterHelp(OL_WIDGET_HELP, caption, "TextField Help",
           OL_STRING_SOURCE, "Your cursor is on the caption.");

      return textfield;
}
```

Figure 4-8 Create Proc Modified to Register Help for both a Text Field and its Caption

Example of Using Create Procs to Set Fonts

Let's say you have designed an interface with a Checkbox Setting and you want to set the font for both the checkboxes and the Setting label at creation time.

To do this, you set up connections that specify Create Procs for each of the objects. Specify the same function name for the individual checkboxes' Create Procs (for example, `create_checkbox`). Specify another function name for the Setting Create Proc (for example, `create_setting`). Golit will generate Create Proc templates similar to those shown in Figure 4-9.

```
Widget
create_checkbox(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      printf("create proc: create_checkbox\n");
      return(GolitFetchWidgetHier(name, wdp, parent, args,
       num_args, closure));
}

Widget
create_setting(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      printf("create proc: create_settings\n");
      return(GolitFetchWidgetHier(name, wdp, parent, args,
       num_args, closure));
}
```

Figure 4-9 Create Proc Templates for a Checkbox and a Setting

Setting the font for the checkboxes is relatively simple. You just modify the `create_checkbox()` Create Proc so that it uses `GolitFetchWidgetHier()` to instantiate the checkbox widget. Then you use XtVaSetValues to set the font (see Figure 4-10). Any checkbox that you specified `create_checkbox()` as the Create Proc for will have a bold label.

Setting the font for the Setting label is not quite as simple. The `create_settting()` Create Proc instantiates the nonExclusives widget, not the caption widget that shows the label. Therefore, you must you must first retrieve the caption widget's id and then set the font for that widget. To get the id, you use `GolitNameToWidget(). GolitNameToWidget()` requires the caption widget's name. To get this name, you add the prefix, `Ex4_G_Caption_` , to the Setting object's name. In this example, the name would be `Ex4_G_Caption_setting1`. Figure 4-10 shows how you might modify the `create_settings()` callback to set the font for a Setting label.

```
Widget
create_choice(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      Widget checkbox;
      /*  Instantiate checkbox and get an id  */
      checkbox = GolitFetchWidgetHier(name, wdp, parent, args,
                                num_args, closure);
      /* Set the checkbox label's font  */
      XtVaSetValues(checkbox, XtVaTypedArg, XtNfont, XtRString,
           "lucidasans-bold", strlen("lucidasans-bold") + 1,
           NULL);
      return(checkbox);
}
Widget
create_setting(name, wdp, parent, args, num_args, closure)
      String name;
      register GolitWidgetDescPtr wdp;
      Widget parent;
      ArgList args;
      Cardinal num_args;
      XtPointer closure;
{
      Widget setting, caption;
      /* Instantiate setting and save id */
      setting = GolitFetchWidgetHier(name, wdp, parent, args,
                           num_args, closure);
      /* Get id for setting's caption  */
      caption = GolitNameToWidget(setting,
                     "Ex4_G_Caption_setting1");
      /* Set the setting caption's font  */
      XtVaSetValues(caption, XtVaTypedArg, XtNfont, XtRString,
           "lucidasans-bold", strlen("lucidasans-bold") + 1,
            NULL);
return(setting);}
```

Figure 4-10 Checkbox and Setting Create Procs Modified to Change Fonts

Compiling

After you've created an interface with Devguide, generated source code files with Golit, and created your own custom source code files, make sure that you are ready to compile. Make sure the environment variables GUIDEHOME and OPENWINHOME are set to point to the Devguide and OpenWindows' home directories respectively. Edit the `Makefile` to include any of your own source code files in the Parameters section. Enter the names of all your source code files in the `SOURCES.c` line and the names of all your header files in the `SOURCES.h` line.

Once your `Makefile` is set to show all of the associated source code files, you compile them by entering the command **`make`**. It runs according to the contents of the `Makefile`: it first checks all files specified in the `SOURCES.G` parameter to see if any have been changed since the last compile. It then uses Golit to generate fresh source code files if you changed the GIL file. It finishes by compiling and linking all specified source code files.

The compiled code is placed in the file named in the `PROGRAM` parameter of the `Makefile`. To run it, simply enter the filename on the command line.

4