Getting Started with Golit 2

This chapter describes how to generate simple user interfaces and integrate them with your applications. It provides a very simple example of an application that uses Golit. This chapter is intended to be a "quick start guide" that provides you with the minimal instructions necessary to start using Golit. The topics discussed here are covered in greater detail in Chapter 3, "Tutorial" and Chapter 4, "Golit Functionality in Detail."

Summary of How to Create an Application with Golit

To create an application with Devguide and Golit, follow these steps:

1. **Design an interface with Devguide and save it in a GIL file.**
   You can save an interface in several GIL files and organize these files into a project (`.P`) file. See the *OpenWindows Developer's Guide: User's Guide* for more information on designing interfaces.

2. **Generate UI code by running Golit on the GIL or project file.**
   Golit generates three C files and a makefile for each GIL file. The C files contain the interface source code. The makefile contains the Make utility commands to build the interface. If you run Golit on a project file, it generates two additional C files and a supplement to the makefile.

3. **Use `make` to build a test copy of the interface.**
   Golit generates all the code necessary to create an executable, so you can compile the code by itself immediately after you generate it. It is recommended that you test the interface before you add your application code.

4. **Insert your application code in the callback function templates generated by Golit in the `_stubs.c` file.**
   Your application code can change the resources of widgets in the interface or it can instantiate additional widgets.

5. **Run `make` again to build the application.**
   The Make utility compiles the interface and your application code into a complete executable.

Each of these steps is discussed in greater detail in the sections below.

Designing an Interface in Devguide

When you design an interface in Devguide, you must keep Golit in mind, since Golit's requirements are different than those of the other Devguide code generators.

Note – Golit does not support some of the user interface features available in Devguide. A list of these features is provided in Appendix B, "Unsupported Devguide Features." Also note that, because of differences between toolkits, some objects and labels won't appear exactly the same in your compiled application and Devguide.

Configuring Devguide to Generate GIL files for Golit

To create an interface that Golit can generate code for, you must make sure that Devguide is set up for Golit. Do the following before you start designing an interface:

1. **In Devguide, choose "Devguide ..." from the Properties menu.**
   The Devguide Defaults popup window appears (see Figure 2-1).

2. **Set the Toolkit setting to OLIT and click on the Apply button.**
   If the toolkit setting is already set to OLIT, don't change anything.

3. **Dismiss the Devguide Defaults popup window.**

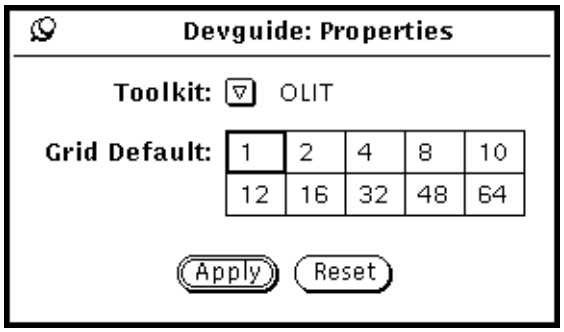

Figure 2-1 The Devguide Defaults Popup Window

Using Connections to Handle Events

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*). For example, you might want to specify that when SELECT is clicked (the Event) on a button (the Source), a popup (the Target) appears (the Action).

To set up a connection, you use the Connections Manager window. See the *OpenWindows Developer's Guide: User's Guide* for instructions on using this window.

Devguide and Golit provide a variety of predefined actions. "Action Menu Items" on page 49 furnishes a complete list of these actions. For each connection you set up in Devguide, Golit generates a callback function in the `_stubs.c` file. The function includes all the code necessary to execute the action you specified for the connection. It also includes a line of code that prints the connection name to the shell tool that you started Devguide from. You can supplement or modify this code with your own application code.

To set up a callback that consists entirely of your own code, create a connection in Devguide and specify CallFunction as the Action. Then enter a function name. Golit will generate a callback under the specified name. This callback contains only a `printf()` statement, which you can replace with your code.

2

If you want to provide some of your own callback code while you are in Devguide, choose ExecuteCode for the action when you create a connection. Devguide will display a popup which you can add your code into. Golit includes this code in the generated callback.

Note – If you are using a project file and more than one of its GIL files invokes the same CallFunction callback, the callback appears in the `<project_name>.c` file.

Generating User Interface Code from GIL Files

There are two ways to generate code from a GIL file: from the command line and with Devguide's code generator tool.

Generating Code from the Command Line

To generate user interface code from the command line, you enter the following command:

```
% golit [<GIL_filename>]
```

If you are generating code for a project, run Golit on the project file by entering the following command:

```
% golit -p [<project_filename>]
```

It is not necessary to type the `.G` or the `.P` filename extension after the GIL or project file filenames, since Golit adds the appropriate extension automatically.

Generating Code with the Code Generator Tool

Devguide provides a code generator tool which enables you to generate, compile, and run your applications within Devguide. The code generator features a menu-driven interface, so you don't have to type anything at the command line.

Setting Up The Code Generator Tool For Golit

To use the code generator tool, you must configure it for Golit. To do this, follow these steps:

1. **Click SELECT on the Properties... button.**
   The Code Generator Properties pop-up window appears (see Figure 2-2).
2. **Use the Code Generator abbreviated menu button to select Golit.**
   The options for Golit appear.

Note – Be sure to use only one code generator on each GIL file. If you use two different code generators, it will corrupt the generated files. For example, if you use GXV to generate code from a GIL file and subsequently use Golit on the same file, the generated files will be corrupted.

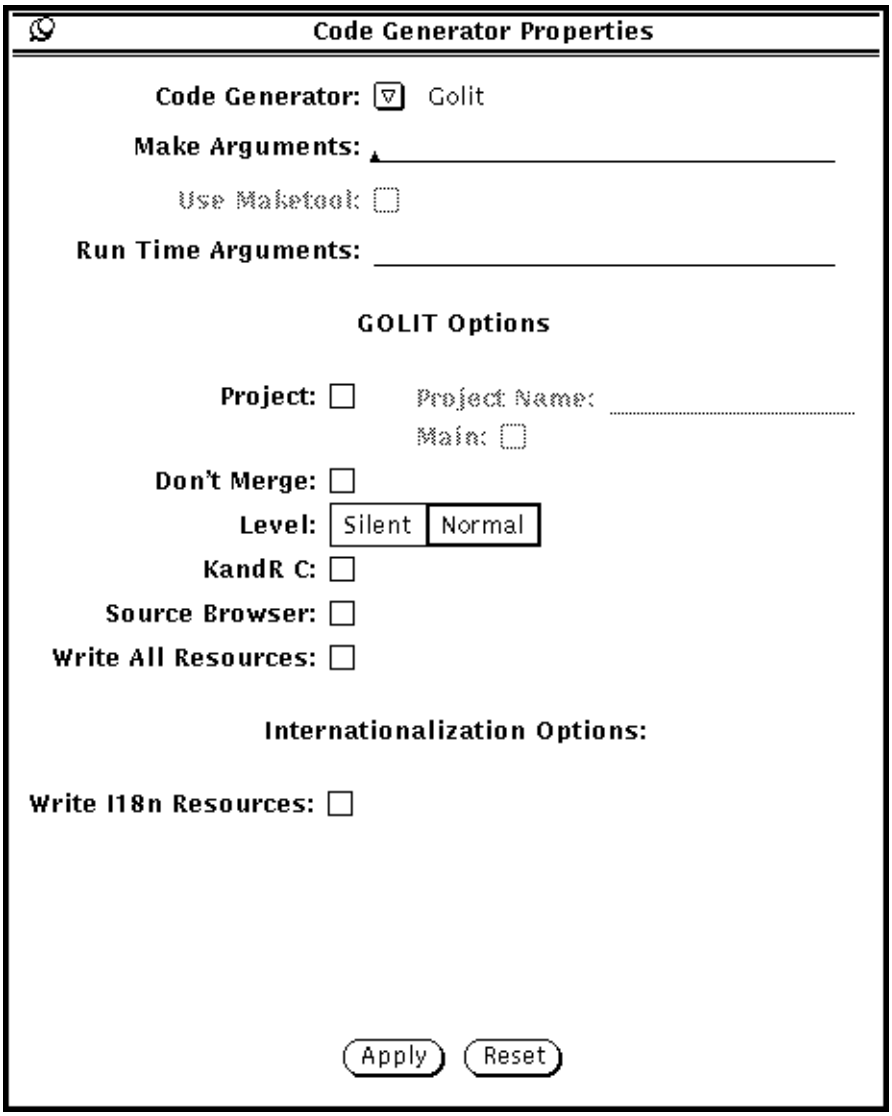

Figure 2-2 Code Generator Properties Window

Choosing Golit Options for Code Generator Tool

The code generator tool provides a variety of options for generating Golit code. In addition to the command line options, which are described on page 51, you can also select the following options in the Code Generator Properties pop-up window.

- **Make Arguments**
  You can enter `make` arguments, such as `clear` in this field. The `Makefile` contains all the property options that you used the last time you generated code. If you load in an application with a `Makefile`, all of the code generator options contained in the `Makefile` are loaded into the tool. When you click SELECT on the Apply button on the property sheet, the `Makefile` is updated.

- **Maketool**
  Check Use Maketool if you want to use the SPARCworks™ or ProWorks Maketool. Type in make or runtime arguments in the appropriate text fields. The Maketool option works only if you have installed the SPARCworks or ProWorks software and updated your path accordingly.

- **Runtime Arguments**
  You can specify any runtime arguments, such as `-scale large` in this field.

For more information on code generator tool, see the *OpenWindows Developer's Guide: User's Guide*.

Golit Generated Files

Golitgenerated files contain all the source code necessary to create an executable that uses the OLIT toolkit and the libgolit runtime library.

When you run Golit on a single GIL file, it generates the following files in the current working directory:

- `<GIL_filename>_stubs.c` - skeleton main program for the executable. It includes the callback templates that you insert your application code into.
- `<GIL_filename>_ui.c` - code that defines the user interface objects. Unless you are writing advanced applications, you don't need to look at this file.

- `<GIL_filename>_ui.h` - header file that declares the user interface objects, external callbacks, and external creation procedures. Normally, you won't need to look at this file.
- `Makefile` - a template makefile to build the executable. `Makefile` contains `make` utility commands to compile your interface and any application code you include in the `_stubs.c` file

The filenames of these files are based on the name of the GIL file. For example, if you use Golit to generate source code for a GIL file named `display.G`, Golit creates the files, `display_ui.c`, `display_ui.h`, and `display_stubs.c`.

When you run Golit on a project file, it generates the files listed above (except for the makefile) for each GIL file in the project. It also generates the following files:

- `<project_name>.c` - includes the main program which is left out of the individual `_stubs.c` files. It also includes templates for callbacks shared by the GIL files in the project.
- `<project_filename>.h` - external declarations for callbacks shared by the GIL files.
- `<project_filename>.make` - addition to the makefile that is automatically included.

The filenames of these files are based on the name of the project file. For example, if you use Golit to generate source code for a project file named `myproject.P`, Golit creates the files `myproject.c`, `myproject.h`, and `myproject.make`.

Compiling and Testing Interface Code

To compile the interface code generated by Golit, use the code generator tool or type the following after you have generated the code:

```
% make
```

The `make` utility compiles and links the Golit-generated code. The resulting executable has the same name as the GIL file, without the `.G` filename extension. If you have generated code for a project, it will have the same name as the project file without the `.P` extension.

Note – Devguide provides a test mode that allows you to test an interface while you are designing it. However, some connections that specify events or actions specific to the Golit code generator do not work in the Devguide test mode. Also, some objects and labels appear differently in the test mode than they do in the compiled applicaton. Always test your interface by compiling the generated interface code once and running it before you add your application code.

The executable you create when you compile the generated code provides a complete working model of the user interface. When you perform an action in the interface (for example, a button press) the program prints the name of any connections associated with that action to the console.

Integrating Application Code with Golit Interface Code

To add your application code to the Golit interface code, you modify the callback templates in the `_stubs.c` file. If you call functions that you keep in separate files, you also need to modify the `Makefile` to compile these files.

Caution – Do not alter the `_ui.c` or `_ui.h` files. If you add any code to these files, you will lose it the next time you run Golit.

Setting Widget Resources

To set a widget's resources in your application code, you must retrieve its id. The libgolit library provides a function, `GolitNameToWidget()`, that does this for you. You use it as follows:

```
Widget w;

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

where:

`Ui_object_name` is the name of the widget for which you want to get an id. Golit constructs widget names by prepending the name of the interface to the name you gave to the

object in Devguide. For example, if you create a button called `button1` in Devguide and save it in a GIL file called `mygilfile`, its widget name will be `Mygilfile_button1`. Note that Golit automatically capitalizes the first character of the name.

`root` is the widget in the widget tree where you would like to begin searching for the widget designated by `Ui_object_name`. Normally, you can just use the widget that is passed to the callback.

Regenerating Code for a Modified Interface

Golitand Devguide make it easy to change an interface, even after you have integrated your application code with the interface code. To change an interface, you load the interface's GIL file in Devguide, alter the interface, and save the GIL file. You then run Golit on the altered GIL file to regenerate the code.

Each time you run it, Golit does the following:

1. Overwrites the existing `_ui.c` and `_ui.h` files.
2. Backs up the current `_stubs.c file` to `_stubs.c.BAK`.
3. Generates a new `_stubs.c` file that contains templates for all the connections specified in the GIL file.
4. Merges any code that you inserted in the original `_stubs.c` into the new `_stubs.c`.
5. Creates a `_stubs.c.delta` file that tells you how it has changed `_stubs.c`. This file lists the added text and the affected line numbers.

If you are regenerating code for a project, Golit performs the steps above for each of the GIL files in the project. It also does the following:

1. Backs up the `<projectname>.c` and `<project_name>.h` files to `.BAK` files.
2. Generates a new `<projectname>.c` file and merges it with the old one, listing the changes in a `.delta` file.
3. Overwrites the original `<project_name>.h` file.

A Simple Example

Let's say that you want to create an application that displays a button and a gauge. Each time the user presses the button, you want the gauge value to increase by one. To create this application, you follow these steps:

1. **Create the interface in Devguide.**
   To do this, drag a Base Window glyph onto the workspace. Then place a control area in it and resize it to fit. Drag a button and a gauge glyph onto the control area. For this example, use the default names and values provided by Devguide for the objects. The interface should appear like the one displayed in Figure 2-3.

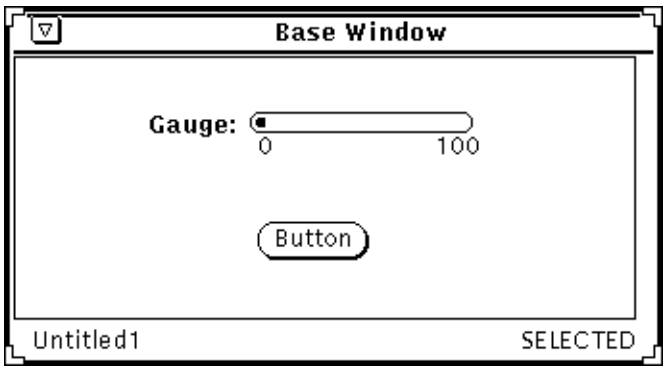

Figure 2-3 Example Interface as it Appears in Devguide

2. **Create a connection for the button in the Connections Manager.**
   Make sure that the button(button1) is the Source. Select CallFunction for the Action and specify `incr_gauge` as the function name. Since you are defining the Action, it does not matter what the Target is. After you have created the connection, the Connections Manager window should look something like the one shown in Figure 2-4.

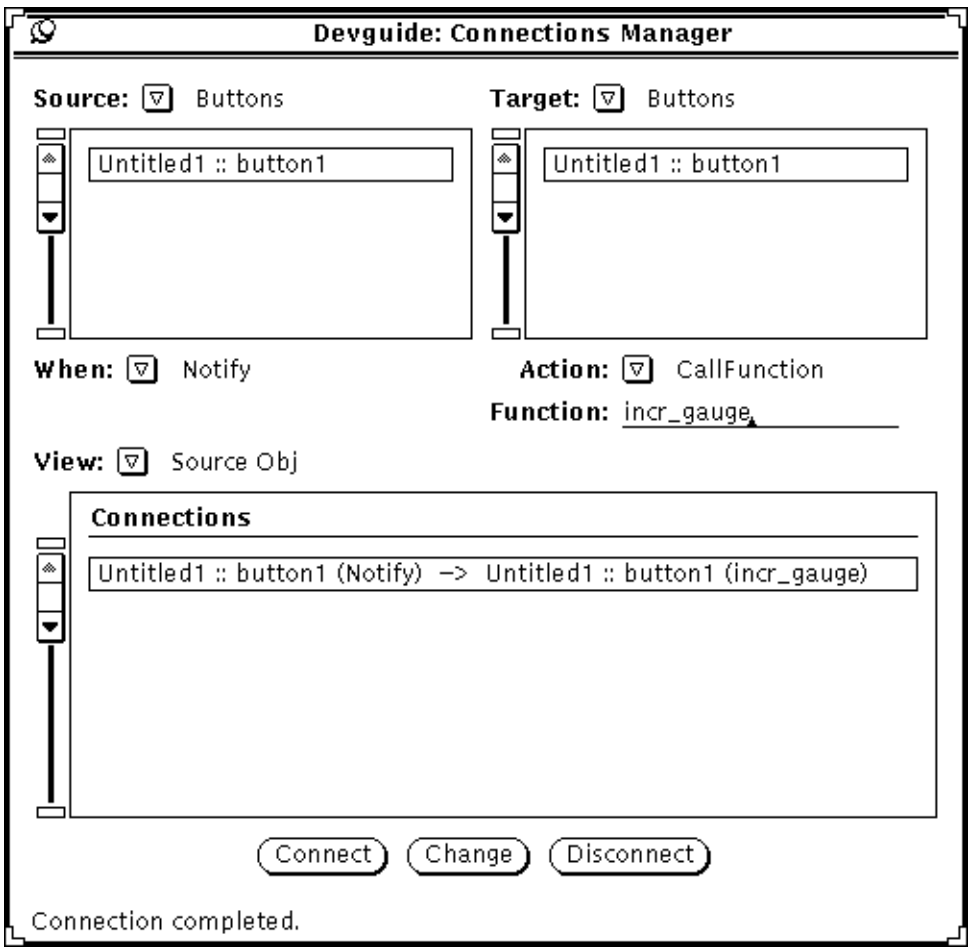

Figure 2-4 Connections Manager Window

3. **Save the interface to a GIL file.**
   For this example, save it to `ex1`.

4. **Type `golit ex1` to generate the code.**
   Golit generates the interface code and the `Makefile`.

5. **Type `make` to build a test copy of the interface.**
   Make builds an executable named `ex1`.

6. **Type** `ex1` **to run the executable.**
   Each time you press the button displayed in the base window, the application prints the connection name to the console.

7. **Modify the** `ex1_stubs.c` **file.**
   The file generated by Golit should look like the one in Figure 2-7. You will only need to change the `incr_gauge()` function template. Add code that uses `GolitNameToWidget()` to retrieve the gauge widget id. Then use Intrinsics functions to increment the gauge. When you are done, the `incr_gauge()` callback should look something like the code in Figure 2-7.

8. **Type** `make` **to build the complete application.**
   Golit builds an executable called `ex1`.

```
/*
 * ex1_stubs.c - Main routine, callbacks, and connections stubs.
 * This file was generated by 'golit' from 'ex1.G'.
 */

#include "ex1_ui.h"


String fallback_resources [] = {
          "*background: gray",
          NULL
};
```

Figure 2-5 Generated stubs.c File for ex1

```
/*
 * main for application ex1
 */
int
main(argc, argv)
      int       argc;
      char      **argv;
{
      Widget widget1;
      Display *dpy;
      XtAppContext app;

      /* Initialize Toolkit */
      OlToolkitInitialize(NULL);
      XtToolkitInitialize();

      / * Create Application Context and set fallback resources
      app = XtCreateApplicationContext();
      XtAppSetFallbackResources(app, fallback_resources);

      /* Open Display */
      dpy = XtOpenDisplay(app, NULL, "ex1", "Ex1",
                (XrmOptionDescList) NULL, 0, &argc, argv);
      widget1 = GolitFetchShellHier("Ex1_window1", "Ex1",
                Ex1_window1, dpy, NULL, NULL, NULL);
      XtRealizeWidget(widget1);

       /* Enter event loop */
      XtAppMainLoop(app);
      exit(0);
}
void
incr_gauge(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
      printf("connection: incr_gauge\n");
}
```

Figure 2-6 Generated _stubs.c File for ex1 (main)

```
void
incr_gauge(widget, clientData, callData)
      Widget widget;
      XtPointer clientData, callData;
{
Arg arg[1];
static int gauge_value;

/* Get the gauge's widget id */
      Widget target = GolitNameToWidget(widget, "Ex1_gauge1");

/* Increment gauge unless limit has been reached */
      if (gauge_value < 100)
      {
      XtSetArg(arg[0], XtNsliderValue, ++gauge_value);
      XtSetValues(target, arg, 1);
      }
}
```

Figure 2-7 Callback Modified to Increment a Gauge Each Time a Button is Pressed

2