GNUstep Renaissance

Nicola Pero n.pero@mi.flashnet.it

December 2002 AD

1 What is GNUstep Renaissance

GNUstep Renaissance is an advanced framework for creating portable user interfaces. It works on top of gnustep-gui (and of Mac OS X Cocoa under Apple), and provides an easy and powerful way to create and manage user interfaces.

In our previous tutorials we have learnt how to create user interfaces by hand, writing code which creates all the objects in the user interface. GNUstep Renaissance simplifies considerably this task. Instead of writing all the code yourself, you write a simple file (in a format called gsmarkup) which describes (very simply) the interface. At run time GNUstep Renaissance can create all the interface from the file, including all size and layout computations, and then connecting the objects created in the user interface with the objects in the application as required.

By using GNUstep Renaissance you can create quickly user interfaces which build and run on both GNUstep and Apple Mac OS X, without changes in source code. This is not possible with traditional nib and gorm files.

In this tutorial you'll learn how to create and edit gsmarkup files, and use GNUstep Renaissance to load them into programs.

GNUstep Renaissance will have an interface builder clone program written for it, which will for the most part obsolete creating and editing the gsmarkup files directly. This program will allow you to edit gsmarkup files by viewing a graphical representation of the gsmarkup contents, and dragging and dropping objects into it. This tutorial will be partially obsolete then; but it will still be of use to anyone wishing to learn something about the GNUstep Renaissance gsmarkup format and files.

2 Prerequisites and target of the tutorial

We assume familiarity with GNUstep (or Apple Mac OS X Cocoa), and with HTML. We will also assume that you have compiled and installed GNUstep Renaissance on your system. Please refer to the GNUstep Renaissance installation instructions for help on compiling and installing GNUstep Renaissance under GNUstep or Apple Mac OS X.

In this tutorial, we want to rewrite the last example of the "First Steps in GNUstep GUI Programming (2): NSWindow, NSButton" tutorial so that it creates the window using GNUstep Renaissance, rather than creating it programmatically. We will go step by step; first, we will just create an empty window; later, we will add the menu, and finally set the window attributes, and add the button inside the window.

We generally focus on writing GNUstep applications; but because GNUstep Renaissance can be used on Apple Mac OS X as well, we will try to show how to build programs and user interfaces which work on both systems. It should be possible to use this tutorial to learn the basics of GNUstep Renaissance on a pure Apple Mac OS X system too.

3 Writing the gsmarkup file

We start by writing a gsmarkup file to create an empty window using GNUstep Renaissance:

```
<gsmarkup>

  <objects>

    <window />
```

```
  </objects>

</gsmarkup>
```

We save this code in a file called Window.gsmarkup. As you can easily see, the code – which is written in the gsmarkup format – is very similar to HTML; as a matter of fact, it is a variant of XML. There are tags (such as `<gsmarkup>`) and each tag is closed after having been opened (for example, `</gsmarkup>` closes `<gsmarkup>`). The syntax `<window />` is equivalent to `<window></window>`, that is, the window tag is opened and immediately closed.

The code starts with `<gsmarkup>`, and ends with `</gsmarkup>`: it's all contained in a `gsmarkup` tag. This is equivalent to an HTML file, which starts with `<html>`, and ends with `</html>`.

Inside the `<gsmarkup>` tag, we find the `<objects>` tag. The `<objects>` tag enclose a list of tags; each of those tags represents an object which is to be created when the file is loaded.

In this case, there is a single tag inside the `<objects>` tag: the `<window />` tag, which tells GNUstep Renaissance to create a single object – a window – when the file is loaded.

4 Loading the gsmarkup file from the program

We now need to load the Window.gsmarkup file (which we have created in the previous section) in our program. GNUstep Renaissance provides facilities to load gsmarkup files; "loading" a gsmarkup file means reading the file and creating all the objects (and connections, as will be clear later) described in the file. In this case, loading the file will create a single, empty, window.

In order to load the `Window.gsmarkup` file, we just need to use the code

```
  [NSBundle loadGSMarkupNamed: @"Window"  owner: self];
```

this will parse the `Window.gsmarkup` file and create the objects (and connections) contained in the file. We can ignore the `owner:` argument for now as we don't need it yet; passing `self` (where `self` is the application's delegate) is OK for now. This code will look for a file `Window.gsmarkup` in the main bundle of the application, and load it. To compile this line, you need to include the Renaissance header file `<Renaissance/Renaissance.h>` at the beginning of your program.

The full program is then:

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>
#include <Renaissance/Renaissance.h>

@interface MyDelegate : NSObject
{}
- (void) applicationDidFinishLaunching: (NSNotification *)not;
@end

@implementation MyDelegate : NSObject

- (void) applicationDidFinishLaunching: (NSNotification *)not;
{
  [NSBundle loadGSMarkupNamed: @"Window"  owner: self];
}
@end

int main (int argc, const char **argv)
{
  [NSApplication sharedApplication];
  [NSApp setDelegate: [MyDelegate new]];

  return NSApplicationMain (argc, argv);
}
```

Save this code in a `main.m` file. Please note that, for simplicity in this first example, we have omitted the creation of the application menu; we'll add it in later sections. The given `#include` directives work on both GNUstep and Apple Mac OS X.

To complete the example, we provide a GNUmakefile for the application:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = Example
Example_OBJC_FILES = main.m
Example_RESOURCE_FILES = Window.gsmarkup

ifeq ($(FOUNDATION_LIB), apple)
  ADDITIONAL_INCLUDE_DIRS += -framework Renaissance
  ADDITIONAL_GUI_LIBS += -framework Renaissance
else
  ADDITIONAL_GUI_LIBS += -lRenaissance
endif

include $(GNUSTEP_MAKEFILES)/application.make
```

The few lines starting with

```
ifeq ($(FOUNDATION\_LIB), apple)
```

add `-framework Renaissance` on Apple Mac OS X, and `-lRenaissance` on GNUstep, which makes sure that your program compiles and runs on both GNUstep and Apple Mac OS X.

The program should now compile (using 'make') and run (using "`openapp Example.app`" on GNUstep, and "`open Example.app`" on Apple Mac OS X). The program won't do much, except displaying a small empty window. To close it, you probably need to kill it (from the command line by typing `Control-C`, or using the window manager).

We are assuming here that you use gnustep-make to compile on Apple Mac OS X; you can do the equivalent with ProjectBuilder if you really want: create a Cocoa application project, then add the Objective-C source file, and the Window.gsmarkup resource file. You also need to specify that you want the program to use the Renaissance framework. Then you should be able to compile and build the program. We will make no more mention of Project Builder; it should be easy to adapt the examples to build using Project Builder if you want, but using gnustep-make and the provided `GNUmakefiles` will give you almost seamless portability, since the same code will compile without changes on GNUstep and Apple Mac OS X.

5 Adding a menu

We now want to add a very simple menu, containing a simple `Quit` item. Clicking on the item should quit the application. Here is the required gsmarkup file:

```
<gsmarkup>

  <objects>

    <menu type="main">
      <menuItem title="Quit" key="q" action="terminate:" />
    </menu>

  </objects>

</gsmarkup>
```

In this file, the `<objects>` tag contains a single `menu` object, which contains a single `menuItem`.

When GNUstep Renaissance loads the file, it will create a menu object (corresponding to the `<menu type="main">` tag inside the `<objects>` tag). This menu will be the application main menu, because `type="main"` has been used as attribute of the menu tag. Inside the menu, GNUstep Renaissance will create a single item, with title `Quit`, key equivalent `q`, and which, when clicked, will execute the action `terminate:`.

Please note that the menuItem object is inside the menu object because the `<menuItem>` tag is inside the `<menu>` tag. Generally, the nesting of tags represents a corresponding nesting of objects.

Because the menus are organized in a completely different way on Apple Mac OS X, this file won't really generate a proper menu on Apple Mac OS X. If you are using Apple Mac OS X, you should rather use the following gsmarkup file for your menu:

```
<gsmarkup>

  <objects>

    <menu type="main">
      <menuItem title="Example">
         <menu title="Example" type="apple">
           <menuItem title="Quit Example" key="q" action="terminate:" />
         </menu>
      </menuItem>
    </menu>

  </objects>

</gsmarkup>
```

The structure in this case is more complex: inside the main menu (displayed horizontally on Apple), there is an item called `Example` which contains a submenu. In the submenu there is a single menu item, with title `Quit Example`, key equivalent `q`, calling the action `terminate:`. Please note that this menu has `type="apple"`. This attribute is ignored under GNUstep, and only used by Apple Mac OS X; you should always have one and only one submenu (the first one) with type `apple` for Apple Mac OS X menus; it is used to identify the first compulsory submenu of an application menu, the one with the title displayed in bold, and containing the `Quit XXX` menu item.

If you need your application to run on both systems, you will need to provide two separate gsmarkup files, each one using its system specific menu layouts: one for GNUstep, the other one for Apple Mac OS X; and then load the specific one depending on the platform you are running on (you can use the `GNUSTEP` preprocessor defined symbol to know on which platform you are; the symbol is defined on GNUstep, but not on Apple Mac OS X). Conventionally, you can call the first one `Menu-GNUstep.gsmarkup`, and the second one `Menu-OSX.gsmarkup`.

The gsmarkup file containing the main menu has to be loaded before calling `NSApplicationMain()`; this is the most portable way of loading it. Here is the new `main.m` file, with the required changes to load our new `Menu-GNUstep.gsmarkup` (or `Menu-OSX.gsmarkup`) file:

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>
#include <Renaissance/Renaissance.h>

@interface MyDelegate : NSObject
{}
- (void) applicationDidFinishLaunching: (NSNotification *)not;
@end

@implementation MyDelegate : NSObject

- (void) applicationDidFinishLaunching: (NSNotification *)not;
{
  [NSBundle loadGSMarkupNamed: @"Window"  owner: self];
}
@end

int main (int argc, const char **argv)
{
  CREATE_AUTORELEASE_POOL (pool);
```

```
  MyDelegate *delegate;
  [NSApplication sharedApplication];

  delegate = [MyDelegate new];
  [NSApp setDelegate: delegate];

#ifdef GNUSTEP
  [NSBundle loadGSMarkupNamed: @"Menu-GNUstep"  owner: delegate];
#else
  [NSBundle loadGSMarkupNamed: @"Menu-OSX"  owner: delegate];
#endif

  RELEASE (pool);
  return NSApplicationMain (argc, argv);
}
```

Here we use `delegate` as the `owner` argument of the `loadGSMarkupNamed:owner:` method. Because we are not using the owner yet, it is not important which object is passed – except for a detail, which is that the owner is used to determine in which bundle to look for the gsmarkup file to load; passing an object of a class defined in your application makes sure that the file is looked for in the main application bundle. In practice, if you are loading files from your application main bundle, you should always pass an instance of an object defined in your application as the owner.

We have also enclosed the loading of the Menu gsmarkup file inside the creation and release of an autorelease pool; this is needed because otherwise no autorelease pool would in place at that point. Generally, whenever you need to do anything non-trivial inside your `main` function, you need to create an autorelease pool at the beginning, and release it at the end.

Do not forget to add `Menu-GNUstep.gsmarkup` and `Menu-OSX.gsmarkup` to the resource files (listed in the `Example_RESOURCE_FILES` variable) in the `GNUmakefile`:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = Example
Example_OBJC_FILES = main.m
Example_RESOURCE_FILES = \
  Menu-GNUstep.gsmarkup \
  Menu-OSX.gsmarkup \
  Window.gsmarkup \


ifeq ($(FOUNDATION_LIB), apple)
  ADDITIONAL_INCLUDE_DIRS += -framework Renaissance
  ADDITIONAL_GUI_LIBS += -framework Renaissance
else
  ADDITIONAL_GUI_LIBS += -lRenaissance
endif

include $(GNUSTEP_MAKEFILES)/application.make
```

The application should now build and run, and display an empty window and a very simple menu with a single “Quit” menu item, which quits the application.

6 Changing the window attributes

In the previous tutorial, we created the window non-closable, and we set `This is a test window` as the window title. To do the same using GNUstep Renaissance, we just need to add the information as attributes of the window object in the gsmarkup file:

```
<gsmarkup>
```

```
  <objects>

    <window title="This is a test window" closable="no" />

  </objects>

</gsmarkup>
```

Every tag can have some attributes set; GNUstep Renaissance will read the attributes and use them when creating the object. The list of valid attributes for each tag, and their meaning, is included in the GNUstep Renaissance manual (currently being written); in this case, we have used the `title` attribute of a window, and the `closable` attribute of a window. Similarly, to make a window non-resizable you would add `resizable="no"`, and to make it non-miniaturizable you would add `miniaturizable="no"`.

Once you have changed your `Window.gsmarkup` file, simply type `make` to have the new file be copied in the application main bundle. After rebuilding, starting the application should use the new title, and the window should be created non-closable.

7 Adding a button in the window

We now want to complete implementing the program of the previous tutorial using GNUstep Renaissance: we want to add a button to the window; clicking the button should print `Hello!` on the console.

Adding the button is just a matter of modifying our `Window.gsmarkup` file, so that it doesn't create the window empty, but with a button inside it:

```
<gsmarkup>

  <objects>

    <window title="This is a test window" closable="no">
      <button title="Print Hello!" action="printHello:" target="#NSOwner" />
    </window>

  </objects>

</gsmarkup>
```

Because the `<button>` tag is inside the `<window>` tag, the button will be created inside the window. The button will be created with title `Print Hello!`, and when you click on it, the method `printHello:` of the object "`#NSOwner`" will be called.

The syntax `#NSOwner` in gsmarkup files is special, and means "the file owner". The file owner is the object which is passed as an argument to the `loadGSMarkupNamed:owner:` method when loading the file, and normally is used to connect together the objects in the interfaces created by a gsmarkup file, to the rest of your application. We have passed the application delegate (an object of a class implemented by us) as the file owner in our examples, which is a reasonably good choice in this situation, since we can add custom methods to it, and call them from the interface.

In this case, by adding `target="#NSOwner"`, we have specified that the target of the button is the file owner object. We can then implement the method `printHello:` of the file owner object to do something, and then clicking on the button will cause that method (and your specific code) to be executed.

To finish our tutorial example, we just need to add this method `printHello:` to the file owner. Here is the code after this final change –

```
#include <Foundation/Foundation.h>
#include <AppKit/AppKit.h>
#include <Renaissance/Renaissance.h>

@interface MyDelegate : NSObject
{}
- (void) printHello: (id)sender;
- (void) applicationDidFinishLaunching: (NSNotification *)not;
```

```
@end

@implementation MyDelegate : NSObject

- (void) printHello: (id)sender
{
  printf ("Hello!\n");
}

- (void) applicationDidFinishLaunching: (NSNotification *)not;
{
  [NSBundle loadGSMarkupNamed: @"Window"  owner: self];
}
@end

int main (int argc, const char **argv)
{
  CREATE_AUTORELEASE_POOL (pool);
  MyDelegate *delegate;
  [NSApplication sharedApplication];

  delegate = [MyDelegate new];
  [NSApp setDelegate: delegate];

#ifdef GNUSTEP
  [NSBundle loadGSMarkupNamed: @"Menu-GNUstep"  owner: delegate];
#else
  [NSBundle loadGSMarkupNamed: @"Menu-OSX"  owner: delegate];
#endif

  RELEASE (pool);
  return NSApplicationMain (argc, argv);
}
```

Finally, we want to underline that the most unpleasant part of the work has been silently done by GNUstep Renaissance for us. If you check the original code in the previous tutorial, you will see that we have had to compute the button size, and to use the button size to build up the window size. This has now all silently been done by GNUstep Renaissance for us. GNUstep Renaissance has made the button of the right size to display its title, then it has sized the window to fit the only object it contains – the button. In more complex windows, the help GNUstep Renaissance gives us by automatically sizing and laying objects and windows can considerably reduce development time.

8 Another small example of using the file owner

Another example of using the `#NSOwner` syntax is when you want to set the delegate of an interface object (for example, of a window) to be your file owner. To do so, you just add the attribute `delegate="#NSOwner"` to the corresponding tag; for example, the following gsmarkup file:

```
<gsmarkup>
  <objects>
    <window title="My Window" delegate="#NSOwner" />
  </objects>
</gsmarkup>
```

creates a window with title `My Window`, and sets the window delegate to be the file owner (that is, the object which is passed as the owner argument to the `loadGSMarkupNamed:owner:` call used to load the file). The delegate is informed of basic events in the window life (such as the window being miniaturized, or closed, or made key), and can modify the window behaviour; please refer to the GNUstep documentation for more information on delegates.

9 Using ids

The file owner syntax explained in the previous sections is just a special case of a more general way of referring to objects by id. An id is just a name internally used to refer to objects; it's never displayed to the user, but it can be used internally in a gsmarkup file to refer to an object: when a gsmarkup file is loaded, objects in the file (or outside the file) can have an id set, and can be referred to by using the special syntax `#id`. The file owner is a special case of this; it is an object (external to the file) which has its id automatically set to `NSOwner`, so that you can refer to it using `#NSOwner`.

To set the id of an object inside that file, you just add an `id` attribute. For example, `<window id="Foo" />` creates a window with an id of `Foo`.

To refer to the window, you can use the syntax `#Foo`; `#Foo` means "the object whose id is `Foo`". For example, `<button target="#Foo" />` creates a button, and sets its target to be the object whose id is `Foo`, that is, the window.

As a more complete example, the following gsmarkup file creates a window, and inside the window a button; clicking on the button will call the `performClose:` method of the window (which closes the window):

```
<gsmarkup>

  <objects>

    <window id="A">
      <button title="Close window" action="performClose:" target="#A" />
    </window>

  </objects>

</gsmarkup>
```

Here the window has an id of `A`, and the button target is set to be the object with id `A`, which is the window. You can try out this example by replacing the Window.gsmarkup file in our example with this one.

10 Translating the application user interface

Finally, we want to show how easy is to translate the application user interface when it's built using GNUstep Renaissance. We will translate the main window of our application; the `Window.gsmarkup` file we used to create it contains the following code:

```
<gsmarkup>

  <objects>

    <window title="This is a test window" closable="no">
      <button title="Print Hello!" action="printHello:" target="#NSOwner" />
    </window>

  </objects>

</gsmarkup>
```

GNUstep Renaissance automatically knows what attributes require translation, and what do not. For example, the `title` attribute of both the window and the button requires translation, while the closable attribute, the action and the target attributes, don't.

When GNUstep Renaissance loads the file `Window.gsmarkup`, it automatically looks for a localized file `Window.strings` containing translations of all attributes which require translation, and uses the translations if found (please note that the name of the strings file is obtained by replacing the `.gsmarkup` extension with the `.strings` extension; in this way, different gsmarkup files automatically use different strings files for translating).

To translate the window, for example, in Italian, all we need to do then is to add a Window.strings file, containing the following code:

```
"This is a test window" = "Finestra di test";
"Print Hello!" = "Stampa Hello!";
```

(it is recommended that you edit these .strings files in UTF-8 if any characters are non ASCII) and put it in a Italian.lproj subdirectory. This file specified that the string `This is a test window` has to be translated as `Finestra di test`, and that `Print Hello!` has to be translated as `Stampa Hello!`.

We then add instructions in the GNUmakefile to install this localized file in the application bundle:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = Example
Example_OBJC_FILES = main.m
Example_RESOURCE_FILES = \
  Menu-GNUstep.gsmarkup \
  Menu-OSX.gsmarkup \
  Window.gsmarkup
Example_LOCALIZED_RESOURCE_FILES = \
  Window.strings
Example_LANGUAGES = Italian


ifeq ($(FOUNDATION_LIB), apple)
  ADDITIONAL_INCLUDE_DIRS += -framework Renaissance
  ADDITIONAL_GUI_LIBS += -framework Renaissance
else
  ADDITIONAL_GUI_LIBS += -lRenaissance
endif

include $(GNUSTEP_MAKEFILES)/application.make
```

Now building the program, and running it with the option

```
openapp Example.app -NSLanguages '(Italian)'
```

should display the window in Italian! Please note that GNUstep Renaissance has automatically translated the strings using the appropriate strings file, and then it has automatically sized the interface objects to fit the translated strings – you don't need to do anything special except translating, everything just works (this is not so with gorm and nib files).

If you are using Apple Mac OS X, to run the program in Italian, try changing the language preferences setting Italian as preferred language in the System Preferences, then running the program.

11 For more information

Please refer to the GNUstep Renaissance manual for more information on Renaissance. It contains a very nice chapter on gsmarkup, which might help you with any point which is still obscure after reading this tutorial.