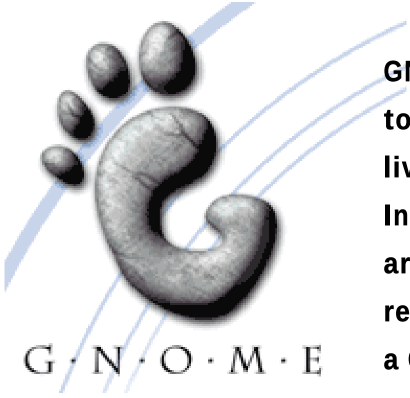
Programming Wizards for GNOME

DRUIDS, WIZARDS AND GURUS



GNOMEs are normally thought of as mystical beings. Related to the leprechaun, they're small hunchbacked creatures who live underground and guard their hoards of precious stones! In the world of Linux there are also small hunchbacked people around (many of them are programmers!) But GNOME itself refers to something entirely different: Thorsten Fischer meets a GUI for X-Windows.

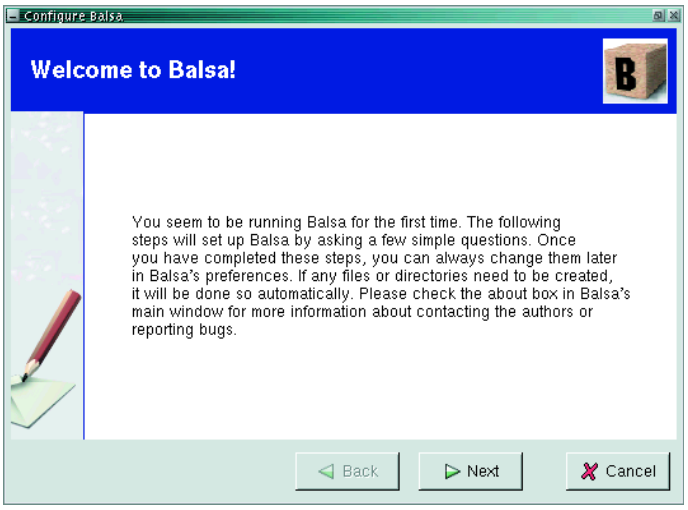
Just as Captain Picard might shout for Number One, his assistant in Star Trek: The Next Generation, so the computer user also expects to have assistants usually called wizards. Under Windows, they're intended to speed up and simplify tasks such as altering settings and configurations.

On the first page of the Wizards used under Windows to install programs, there's also the big, bad licence agreement which takes from users all the freedom that they ought to be able to enjoy with the software!

Under GNOME, Wizards are called Druids. One well known program that uses Druids is Balsa, whose setup dialog box has probably already been seen by everybody involved in Linux – if not, then see Fig. 1. The precursor to Druid was GNOMEGuru. The development of Guru has, however, been suspended in favour of the Druid, owing to the greater flexibility it allows.

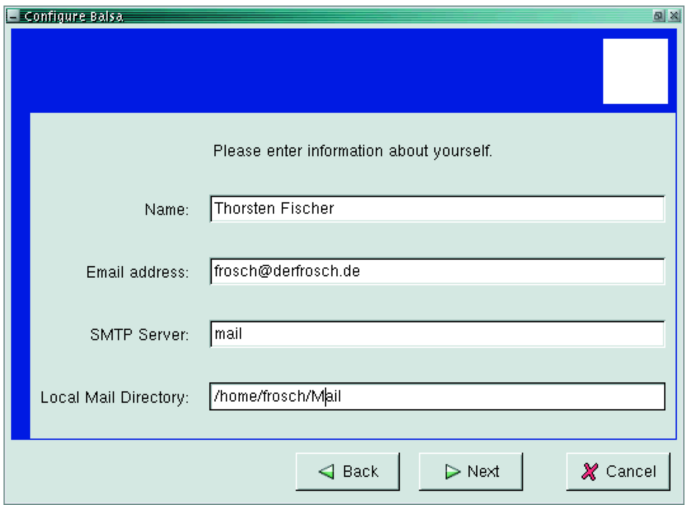
On the following pages we to show how with the aid of the Druid, assistants can be created for simple tasks without having to write a full-blown

Figure 1: The setup Druid of the mail client Balsa.


Configure Balsa

Welcome to Balsa!

You seem to be running Balsa for the first time. The following steps will set up Balsa by asking a few simple questions. Once you have completed these steps, you can always change them later in Balsa's preferences. If any files or directories need to be created, it will be done so automatically. Please check the about box in Balsa's main window for more information about contacting the authors or reporting bugs.


Configure Balsa

Please enter information about yourself.

Name:	Thorsten Fischer
Email address:	frosch@derfrosch.de
SMTP Server:	mail
Local Mail Directory:	/home/frosch/Mail

program. In order to be brief, we will dispense with an *autoconf*-compatible source text tree, and also tackle the problem not in C but in the wonderful script language Python. This means we can also discuss the GNOME API of Python at the same time. The installation of the necessary package *GNOME-Python* is covered in a separate box.

Installation of GNOME-Python

The current version of *GNOME-Python* is 1.0.53 – the version jump to GNOME 1.2 has not yet been made, apparently. The sources can be obtained from a GNOME mirror, for example from ftp://sunsite.icm.edu.pl/pub/Linux/GNOME/. As you might expect, the installation runs like this after unpacking:

```
frog@verlaine:~/gnome-python-1.0.53 # ./conf➥
igure -prefix=/opt/gnome
frog@verlaine:~/gnome-python-1.0.53 # make
frog@verlaine:~/gnome-python-1.0.53 # make i➥
nstall
```

Of course, Python itself must already be installed. However, in a well-configured Linux system this should already be the case – Python is contained in nearly every distribution. The same applies to an already compiled *gnome-python*.

A letter druid

An friend of mine once said that when writing letters using LaTeX the same letter could be used over and over again. He simply created and copied files as required and then inserted different addresses or other things with a text editor.

This is a prime target for automation. A small program with a graphical interface which helps in drawing up letters would save time. It would have to have all the components needed to do the job, however, and bring them together quickly. This is the kind of task GNOME and Python were built for.

The Code

In Listing 1 the code for the executable file *gp-letter* can be seen. The access rights for this – and only this – must be set to execute. That is done, for example, by:

```
chmod 744 ./gp-letter
```

In the first line the code is transferred to the Python interpreter. In the third, the module *gui* is imported, and this will allow us to create our graphical interface. The entry point into the program is line 9 where the function Main is called which creates a new instance of the class and starts its *gtk* main loop.

So far, so good. This is the nice thing about object-based programming: if an empty definition of the class GUI already exists then the program is capable of running. Not that it would actually do anything useful! Next we have to think about a constructor for the GUI class of the program. This can be seen in Listing 2.

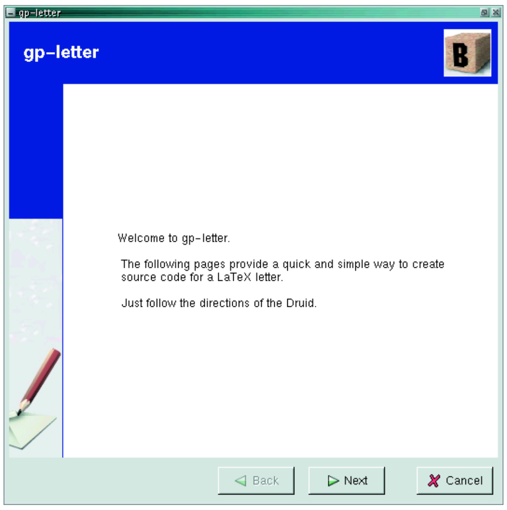
The desired classes are imported at the beginning. There are two ways of importing a module in Python. One is to import the module with its complete name space, as happens here with *GdkImlib*. Thus, the whole content of the module is accessible but with the restriction that every call of a function or class method from the module must be prefixed by *GdkImlib*. In Listing 2 this is seen in lines 15 and 16, in which the two graphs for the logo and the watermark are loaded.

The other possibility is to import using the *from* call, which only imports the names of the classes and functions. Here, trouble can occur if a module defines a name that already exists. However, the widget names from the modules *gnome.ui* and *gtk* are used very frequently.

LinuxMagazine/ gnomedruid

Listing 1: gp-letter
```
001: #!/usr/bin/env python
002:
003: import gui
004:
005: def main ():
006:   gp = gui.GUI ()
007:   gp.mainloop ()
008:
009: if __name__ == '__main__':
010:   main ()
```

Listing 2: Constructor of the GUI class
```
001: from gtk import *
002: from gnome.ui import *
003: import GdkImlib
004:
005: class GUI:
006:
007:   def __init__ (self):
008:     self.lettertype = "dinletter"
009:     self.filename = "./letter.tex"
010:
011:     self.app = GNOMEApp ("gp-letter", "gp-letter")
012:     self.app.connect ("destroy", self.quit)
013:     self.app.connect ("delete_event", self.quit)
014:
015:     self.logo = GdkImlib.Image ("logo.jpg")
016:     self.wmark = GdkImlib.Image ("wmark.jpg")
017:
018:     self.druid = GNOMEDruid ()
019:     self.druid.connect ("cancel", self.quit)
020:
021:     self.dp_start = self.start_page ()
022:     self.dp_letter type = self.letter type_page ()
023:     self.dp_sender = self.sender_page ()
024:     self.dp_content = self.content_page ()
025:     self.dp_finish = self.end_page ()
026:
027:     self.druid.add (self.dp_start)
028:     self.druid.add (self.dp_letter type)
029:     self.druid.add (self.dp_sender)
030:     self.druid.add (self.dp_content)
031:     self.druid.add (self.dp_finish)
032:
033:     self.app.set_contents (self.druid)
034:
035:     self.app.show_all ()
```

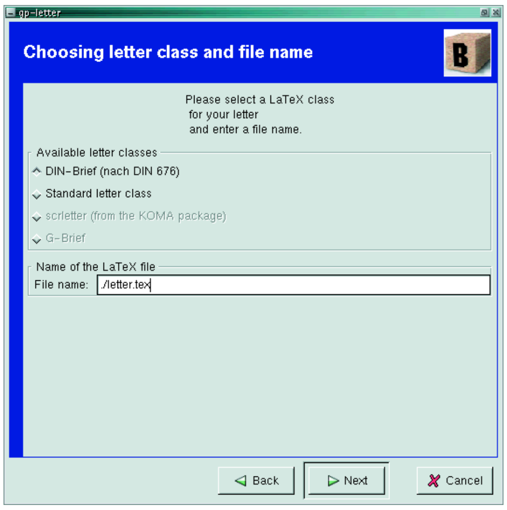


gp-letter

Welcome to gp-letter.

The following pages provide a quick and simple way to create source code for a LaTeX letter.

Just follow the directions of the Druid.



Choosing letter class and file name

Please select a LaTeX class for your letter and enter a file name.

Available letter classes
- DIN-Brief (nach DIN 676)
- Standard letter class
- scrletter (from the KOMA package)
- G-Brief

Name of the LaTeX file
File name: ./letter.tex

Mind the Tab!

The indentation of code blocks in Python must be carefully noted – the structure of the program is defined by them. There are no curly brackets or other symbols to mark the start and end of blocks. Instead, a block is usually marked by a colon on the previous line, say in the case of *for* loops, or in Listing 2 line 5 in the class definition, or in line 7 at the beginning of the constructor function.

The lines 11 to 13 show the typical handling of gtk widgets in Python. Gtk is indeed written in C; but owing to its object-oriented approach wrappers for object orientated (OO) languages can be created quite simply and above all, as we see, easily used.

The design of the Druid begins at line 18. Here we define that clicking on the cancel button is to be followed by the program aborting. For each individual page of the Druid your own text can be entered, which will then form the basis for the Druid.

As a Druid alone is not allowed, it must be contained in a *gtk window* which in this case is provided by a *GNOMEApp*. This happens in line 33. The latter must then also be displayed, which occurs in line 35.

The pages

So far, *gp-letter* has five pages in its basic form. Whatever the case, an initial page for the greeting and a finish page with the close button should be present in every GNOME Druid program. Both page types are implemented using their own widgets *GNOMEDruidPageStart* and *GNOMEDruidPageFinish*. The pages in between are of the type *GNOMEDruidPageStandard*. All three types are derived from the superclass *GNOMEDruidPage*, but only the last one of these possesses a box of the type *GtkVBox* which can be used as a container for user-defined contents.

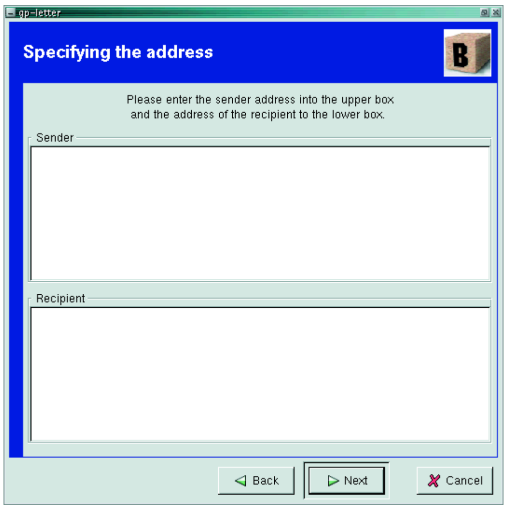
To avoid tedium we won't go through the detailed design of each individual page widget. The method for creating the start and finish pages are clearly demonstrated in Listing 3 along with the method for terminating the program and starting the gtk main loop. The loop is called up in Listing 1 by *gp-letter*. The method *quit* shows the basic struc-

Listing 3: Start and finish page; Program termination; main loop

```
001: def quit (self, event = None, data = None):
002:   mainquit ()
003:
004: def mainloop (self):
005:   mainloop ()
006:
007: def start_page (self):
008:   page = GNOMEDruidPageStart ("gp letter", "Wel➲
come to gp letter.\n\nThis will help you to simply a➲
nd quickly create the source text \nfor a LaTeX lett➲
er.\n\n Simply answer the questions asked by the Dru➲
id.", self.logo, self.wmark)
009:   return page
010:
011: def end_page (self):
012:   page = GNOMEDruidPageFinish ("finish gp lette➲
r", Thank you for using gp letter.\n Click the 'Finis➲
h' button to create your LaTeX file.", self.logo, se➲
lf.wmark)
013:   page.connect ("finish", self.create_letter)
014:   return page
```

Listing 4: A simple GNOMEDruidPageStandard

```
001: def content_page (self):
002:   page = GNOMEDruidPageStandard ("The content of the letter ", self.logo)
003:
004:   box = page.vbox
005:   box.set_border_width (5)
006:   label = GtkLabel ("Please enter into this text box the content of ➲
your letter.")
007:
008:   frame = GtkFrame ("letter text")
009:   framebox = GtkHBox ()
010:
011:   self.contentfeld = GtkText ()
012:   self.contentfeld.set_editable (TRUE)
013:
014:   frame.add (framebox)
015:   framebox.pack_start (self.contentfeld)
016:   box.pack_start (label, FALSE, FALSE, 5)
017:   box.pack_start (frame, TRUE, TRUE, 5)
018:
019:   page.connect ("next", self.get_content)
020:   return page
```


gp-letter
Specifying the address
Please enter the sender address into the upper box
and the address of the recipient to the lower box.
Sender
Recipient
Back
Next
Cancel

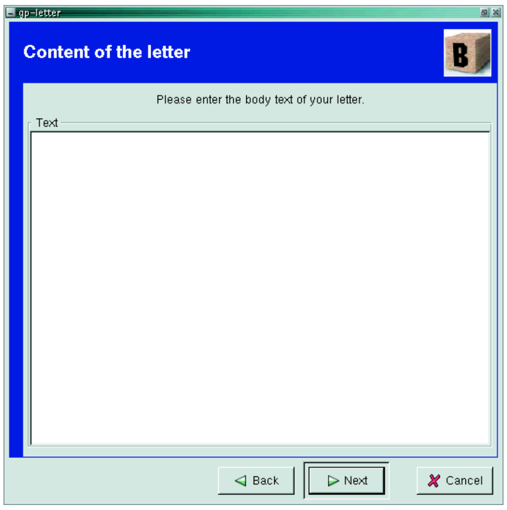


Figure 2: The individual pages of our letter wizard using druid.

ture for a gtk callback function in Python. Note the importance of the two additional parameters which, as in C, describe the event structure by which the callback was called, as well as any transferred data.

A start and a finish page are built up according to the same pattern and require a description as well as a logo – to be seen at the top right on a page – and a "watermark" which is placed on the left edge of the page. For this example we have taken the symbols from Balsa as they are quite suitable. On the finish page the finish button is linked to the function which undertakes the creation of the letter content.

The fourth page is the one in which the user inputs the text of the letter and should be blank. This page is simply constructed. Box is used as a reference for the GtkVBox of the page. The structure – the creation of the widgets and packing them in containers – is similar to the structure of a widget collection in C, except that the widgets are to be accessed again later. They are – like the text box in line 11 – labelled with the prefix Self. Because of this they become properties of the class in which they are used and can be re-used later.

The final function, with which line 19 is linked, fetches the content of the text field and copies it into a variable which is also a property of the class so that during letter creation it can be accessed later.

The start

Now all you need to do is run the program. That occurs quite simply with:

```
./gp-letter &
```

The Druid should now run and appear as in Figure 2. Happy writing!

Non-linear Druids

Druids can be created in the manner described in which the pages follow each other in sequence. But sometimes the structure may need to branch. In a program like gp-letter, for example, it would be nice if, after the dialog which offers the choice of letter class, pages are presented which offer alternative designs although they are still in the same class.

The new pages will still be inserted in the Druid using the *add* method (in Listing 2 in the lines 27 to 31). To make this possible GNOMEDruid defines the following signals:

- next
- back
- cancel
- finish

The first two of these signals can be used for non-linear control flow in conjunction with the *set_page* method of GNOMEDruid. The signal is received and in the respective callback function we simply insert the current page. In this way we can skip to and fro in the linear list of pages in the Druid and the user receives the impression of a flexible program.

This procedure would produce a program with considerably more scope. But whatever the case, the program has several weaknesses: only a few letter classes are supported and, perhaps more importantly, the design of the finished letter is not very pretty. And of course the non-linear control flow is missing because each letter class – and there are a few of these – really needs its own options which would more than justify its own page.

However, these things are enhancements for the future – for after users have got a feel for the program. We used the program and it turned out to be considerably more useful than anticipated. Anyone who is interested in doing so is welcome to make improvements. However, our requirements at the start were for a small and quickly created program. And we've succeeded.

Source code

Of course, no one wants to copy the whole source code. Therefore, it is present on my homepage ready for downloading. Suggestions for improvements and patches for this program which is released under the GPL are of course very welcome! ■

Info

The GNOME.Project: *www.gnome.org/*

Python: *www.python.org/*

gnome-python sources, mirror in Poland: *sunsite.icm.edu.pl/pub/Linux/GNOME/*

GNOMEUI API Reference: *developer.gnome.org/doc/API/libgnomeui/book1.html*

The source code for the example program *www.der-frosch.de/sw/*

■