Rapid Application Development with GNOME and Python

Christian Egli <*christian.egli@stest.ch*>

June 13, 2001

Abstract

RAD is a programming system that enables programmers to build working programs very quickly. How this can be achieved with Free Software tools such as Python, Glade and GNOME will be shown in a hands-on demonstration.

What is RAD?

RAD (Rapid Application Development) is a programming system that enables programmers to build working programs very quickly. In general, RAD systems provide a number of tools to help build graphical user interfaces that would otherwise take a large development effort. Two popular RAD systems for Windows are Visual Basic and Delphi.

RAD and Free Software

With the combination of tools such as GNOME, Python, Glade and LibGlade the world of Free Software also offers an excellent RAD system. Each individual tool is powerful on its own, but it is in combination that new heights of rapid application development are achieved.

GNOME

GNOME [1] is well established as a desktop environment. Lesser known, it also features a development platform which offers a multitude of libraries based on standard and open technologies, such as CORBA and XML. Using this platform allows the developer to write user-friendly and stable software easily and efficiently.

GNOME is part of the GNU Project [2]. More information about GNOME, the source code, binaries and documentation can be downloaded from `www.gnome.org`.

GNOME is built upon GTK+, a free multi-platform toolkit for creating graphical user interfaces.

Python

Python [3] is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java. It includes a rich set of libraries for networking, standard database API and even XML-RPC for example.

More information about Python, binaries and source code can be obtained from `www.python.org`.

Glade

Glade [4] is *the* GUI-Builder for GNOME. It allows straightforward and simple construction of Graphical User Interfaces and gives the developer access to all of the GTK+ and GNOME widgets.

LibGlade

Usually Glade is used to generate C or C++ code directly. However, an XML representation of the GUI that was created can also be generated. This XML file can be loaded at runtime with the help of LibGlade [5] which recreates the GUI as it was designed in Glade.

Hands-on Demo

By writing a small application ”gPizza” for the configuration of the hacker’s favourite food, I will demonstrate live how Python and GNOME can be used to create a running prototype in a very short time.

Highlights of the demo will be:

- creation of the GUI with Glade
- introduction of the layout philosophy of GTK+ and presentation of some of the GTK+ and GNOME GUI Components
- attachment of widgets, i.e. their signals to Python callbacks

- use of the standard Python database API and a few more lines of Python code to connect to a database and fetch some rows that contain all the crucial information to configure pizzas.

Building the GUI

We have in mind a general idea of how we want our GUI to look. Instead of drawing a rough sketch on a piece of paper[1] we launch the GUI-Builder Glade directly.

Glade opens with three windows. The main window contains the menu and a toolbar. The palette window showing in Figure 1 displays the widgets that are available for user interface design and thirdly, the property editor window allows us to configure the currently selected object.

Figure 1: Glade palette

Let's start: First we create a GNOME application window by selecting "Gnome" in the palette and clicking on the "Gnome Application Window" icon. We get a new window with standard menus, a toolbar and a status bar as shown in Figure 2. In the property editor we can see and change all the attributes of this new window.

Now, what are we going to put into this application window? To the left we want to have a list of all the pizzas on the menu. We achieve this by grabbing a horizontal box from the palette (in "GTK+ Basic") and placing it with a mouse click in the empty area in the application window. On clicking a dialog appears where we are asked to specify the number of columns for the box. Two columns are fine for now. Then we insert a columned list on the

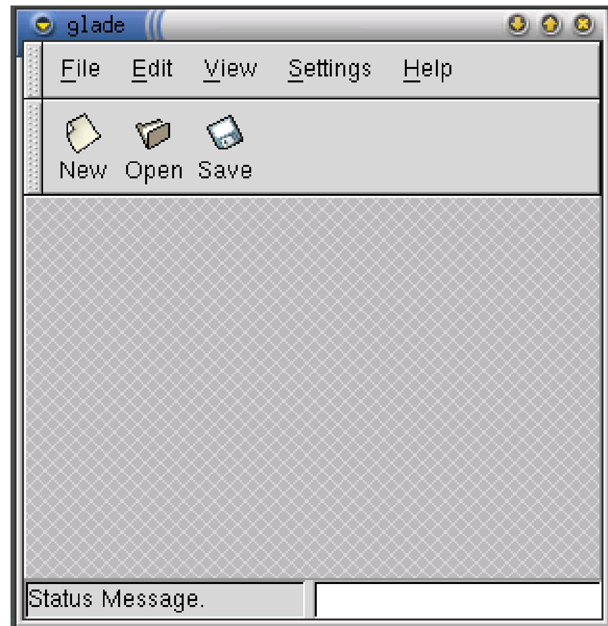

Figure 2: Initial GNOME application window

left using the mouse and palette resulting in another dialog asking for the number of columns. Again we specify two. By clicking on the label on the columned list we can directly modify its name (without having to use the property editor). Let's call the first column "Name" and the second one "Price". The window now looks as shown in Figure 3.

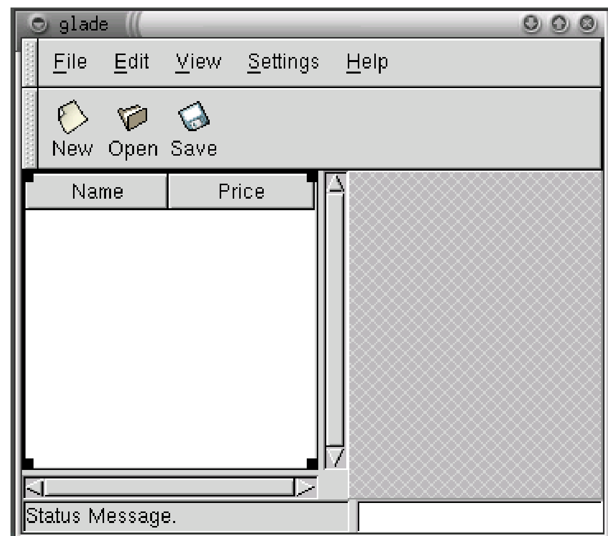

Figure 3: Window with columned list

To the right of the application window we would like to see an image, a short description and a list of additional options for the currently selected pizza. We take a vertical box from the palette and place it with three rows on the right side. With a few more clicks we add a pixmap, a text box and a horizontal box to those three rows. We fill the horizontal box with check buttons that will enable us to select different pizza options. By selecting a check button we can change its label directly. To begin with we

[1] or as all brilliant designs, on a used napkin ;-)

simply assign an image to the pixmap by entering a file name with an image of a pizza in the property editor.

Layout philosophy of GTK+

Our widgets are not yet placed as nicely as we would like. This becomes evident when we try to resize the main window of our new GUI. The horizontal box with the check buttons is enlarged with a lot of empty space. Most likely we only want the text box to be enlarged while the size of the check buttons remains the same.

Luckily, this problem can be taken care of with the help of the property editor. As we do this we will get a glimpse at the layout philosophy of GTK+.

1. We select the horizontal box.
2. In the property editor, we select the tab "Widget". There we toggle the option "Homogeneous" to "Yes". This ensures that all three check buttons are allotted the same amount of screen real estate.
3. In the tab "Packing" we set "Expand" to "No" and "Fill" to "Yes" (see also Figure 4).

While we're at it we also want to make sure that the columned list doesn't expand.

1. We select the columned list or, more accurately, the scrolled window that contains the columned list.
2. In the property editor we go to the tab "Widget" and choose "Never" in the option menu for "H Policy". This results in the columned list not having a horizontal scrollbar anymore.
3. In the tab "Packing" we also set "Expand" to "No".

When we resize the main window, we can see that the behaviour has changed: the horizontal box with the check buttons and the columned list are no longer expanded. Getting immediate feedback from the GUI-Builder Glade enables us not only to understand how the layout mechanism works but also to learn to work with it. More information about the layout philosophy of GTK+ can be found in both the GTK+ Tutorial [6] and Havoc Pennington's excellent book [7].

Flexibility of GTK+

There is a product from an unnamed company located in Redmond that happens to contain a lot of columned lists. This product has a cute feature that allows the user to

Figure 4: Property editor showing "Packing" tab

filter the contents of a column with the help of a combo box in the column header. Let's see how much it takes to add a combo box to the column header "Price".

1. We select the label "Price" in the columned list.
2. Then we cut the above mentioned label and add a vertical box with two rows instead.
3. We paste the label in the top most row. In the lower row we add a combo box.

Voilà, there's the combo box in the column header providing a glimpse into the enormous flexibility of GTK+. Figure 5 shows the resulting GUI.

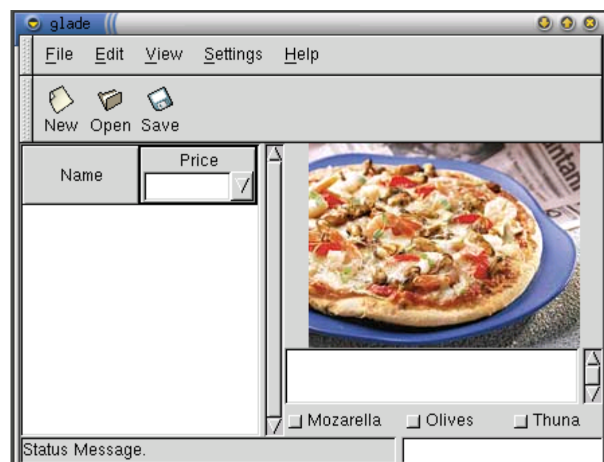

Figure 5: Application window with combo box in column header

Save

So far so good. Before we proceed to save our piece of art we have to set a few things in the project options dialog. Our project is to be saved with the name “gPizza” in the directory `linuxtag2001/src`. Figure 6 illustrates how this is done. Note we enable GNOME support and (although “C” is selected by default) we do not need to specify a programming language since we do not use the code generation features of Glade.

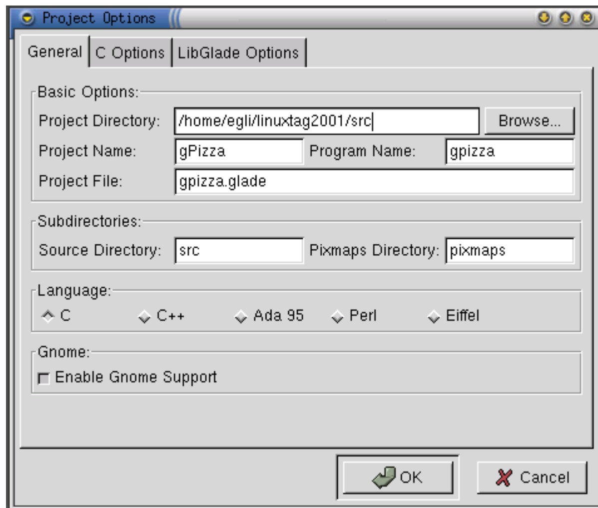

Figure 6: Project options dialog

After all project options are set we can save the project.

XML file

The graphical user interface that we just built and stored is saved as an XML file in `linuxtag2001/src/gpizza.glade`. Let’s have a quick look at it in Listing 1. At the top we see general information for the project such as its name, that GNOME support is enabled, and so forth.

Further down we see the description of the widgets that we added to the project. In Listing 2, for example, we can see the check buttons for olives and mozzarella.

The advantage of saving the entire GUI in a textual form such as XML means not only that it can also be edited with any text editor, but it could also be changed with other tools such as Python or even Perl. It can also easily be kept under revision control with familiar tools such as CVS.

Listing 1: gpizza.glade

```
<?xml version="1.0"?>
<GTK-Interface>

<project>
  <name>gPizza</name>
  <program_name>gpizza</program_name>
  <directory></directory>
  <source_directory>src</source_directory>
  <pixmaps_directory>pixmaps</pixmaps_directory>
  <language>C</language>
  <gnome_support>True</gnome_support>
  <gettext_support>True</gettext_support>
</project>

<widget>
  <class>GnomeApp</class>
  <name>app1</name>
  <title>gPizza</title>
  <type>GTK_WINDOW_TOPLEVEL</type>
  <position>GTK_WIN_POS_NONE</position>
  <modal>False</modal>
  <allow_shrink>False</allow_shrink>
  <allow_grow>True</allow_grow>
  <auto_shrink>False</auto_shrink>
```

Listing 2: gpizza.glade

```
<widget>
  <class>GtkCheckButton</class>
  <name>checkbutton1</name>
  <can_focus>True</can_focus>
  <label>Mozzarella</label>
  <active>False</active>
  <draw_indicator>True</draw_indicator>
  <child>
    <padding>0</padding>
    <expand>False</expand>
    <fill>False</fill>
  </child>
</widget>

<widget>
  <class>GtkCheckButton</class>
  <name>checkbutton2</name>
  <can_focus>True</can_focus>
  <label>Olives</label>
  <active>False</active>
  <draw_indicator>True</draw_indicator>
  <child>
    <padding>0</padding>
    <expand>False</expand>
    <fill>False</fill>
  </child>
</widget>
```

Listing 3: First little Python program

```
#!/usr/bin/env python

from gtk import *
import gnome
import gnome.ui, libglade

def main():
    xml = libglade.GladeXML('gpizza.glade')
    app = xml.get_widget('app1')
    app.connect("destroy", mainquit)
    mainloop()

if __name__ == "__main__":
    main()
```

First flight

Now that the GUI is created and stored, we are able to see how the user interface looks at runtime. For that purpose we write a tiny little Python program that does nothing more than load the XML file with the help of LibGlade, and construct the GUI. After that it simply sits in the mainloop waiting for user input. Listing 3 shows the code.

Let's have a closer look at the program: Line 1 is the standard beginning for a Python script. In lines 3-5 we import the modules `gtk`, `gnome`, `gnome.ui` and `libglade`.

In the function `main` it starts to get interesting: On line 8 the XML file is loaded and the GUI is constructed.

In order for our program to react to user input, i.e. to the signals that are sent by GTK+, we have to connect the `destroy` signal for widget `app1` (which is how Glade automatically named our main window when we created it earlier) with the function `mainquit` in line 9-10. To do this we get a reference (through its name) to the widget and do a `connect` on it. If the main window is closed now the `destroy` signal is sent by GTK+ and, thanks to the connection, the function `mainquit` is invoked and the program quits.

In line 11 we invoke the mainloop, causing the program to wait for user input. Finally, in line 13 an 14, we see a standard Python idiom which starts the function `main`.

Let's run the program. A screenshot of this first invocation is shown in Figure 7. The menu and the buttons all look beautiful but show precious little reaction to user input. This of course is due to the fact that so far we have only connected the signal `destroy` of the main window to a function. Indeed if we close the window the program quits.

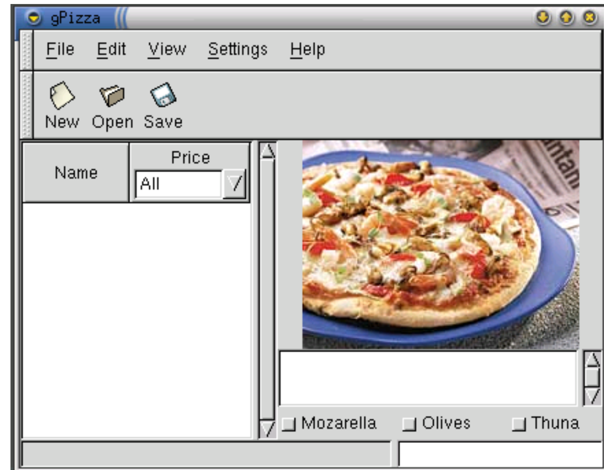

Figure 7: First flight of gPizza

More functionality

Looks like we successfully survived the maiden voyage of "gPizza" without crashing. Now, let's put some more meat on the bone. We shall create another window and add more callbacks to our Python program.

First we go back to the GUI builder Glade, add an About Dialog and extend the toolbar:

1. In the palette we pick a "Gnome AboutDialog" from the "Gnome" tab and fill in the fields "Copyright", "Authors" and "Comments" in the property editor.
2. In the property editor we toggle "Visible" to "No" in the "Basic" tab. This ensures that the About Dialog is not shown automatically when the program starts up.
3. We add an additional button to the toolbar, name it "Quit" in the property editor (in the "Widget" tab) and assign the "Quit" icon to it.

In addition can we connect some menus and buttons with callback functions:

1. We select the menu "Exit". In the "Signals" tab the property editor shows that for this widget the `activate` signal is connected to a handler named `on_exit1_activate`.
2. We change this connection to use the handler `on_exit_activate`.
3. We change the handler for the menu "About..." under the "Help" menu to `on_about_activate`.
4. Finally, we connect the handler `on_exit_activate` for the "Quit" button

in the toolbar with the `clicked` signal (see Figure 8).

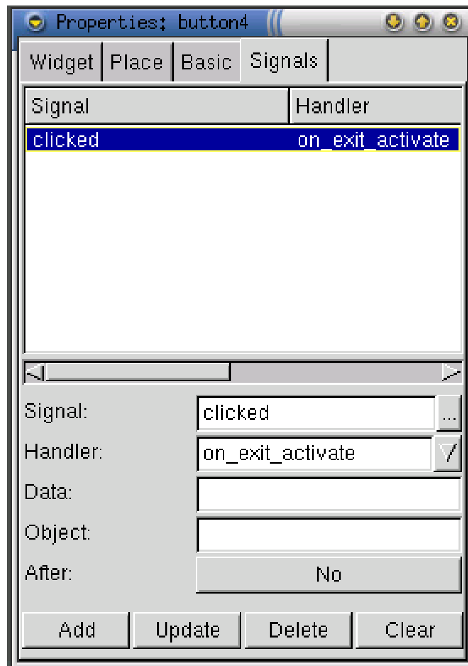

Figure 8: Connecting the `clicked` signal with a handler

Let's have a look at the associated Python script in Listing 4. Very little needs to be changed in comparison to the first program. We add a function `on_about_activate` that takes care of displaying the About Dialog.

There is one interesting thing to note: in our first program we are manually connecting signals to handlers for each widget that we want to react to user input. The second approach (Listing 4) is simpler. We define handlers in Glade directly as outlined above. The code simply defines a mapping between handler name and the actual handler function. Using this mapping the method `signal_autoconnect` in LibGlade takes care of connecting the handlers that we defined in Glade with our associated Python functions.

In Listing 5 the code has been rearranged to benefit from the object-oriented capabilities of Python. We create a class `Application` and move the functions into methods of this class. Instead of invoking the function `main` we simply instantiate an object of class `Application`. The constructor now contains the code that was previously part of the function `main`.

In addition, using the object-oriented and introspection capabilities of Python we are able to use an even simpler approach than in Listing 4. We no longer have to define the mapping between handler name and handler function manually. We leave this task up to the Python interpreter. With the help of a little bit of introspection it finds all

Listing 4: Second Python program

```
#!/usr/bin/env python

from gtk import *
import gnome
gnome.app_version = "0.1"
import gnome.ui, libglade

def on_exit_activate(obj):
    "Quit the application"
    mainquit()

def on_about_activate(obj):
    "Display the about dialog"
    global xml
    about = xml.get_widget("about2")
    about.show ()

def main():
    global xml
    xml = libglade.GladeXML('gpizza.glade')

    nameFunctionMap = {
        "on_exit_activate" : on_exit_activate,
        "on_about_activate" : on_about_activate}

    xml.signal_autoconnect(nameFunctionMap)
    mainloop()

if __name__ == "__main__":
    main()
```

Listing 5: Object-oriented Python program

```
#!/usr/bin/env python

from gtk import *
import gnome
gnome.app_version = "0.2"
import gnome.ui, libglade

class Application:
    def __init__(self):
        self.xml = libglade.GladeXML('gpizza.glade')
        nameFuncMap = {}
        for key in dir(self.__class__):
            nameFuncMap[key] = getattr(self, key)
        self.xml.signal_autoconnect(nameFuncMap)
        mainloop()

    def on_exit_activate(self, obj):
        "Quit the application"
        mainquit()

    def on_about_activate(self, obj):
        "Display the about dialog"
        about = self.xml.get_widget("about2")
        about.show ()

if __name__ == "__main__":
    Application()
```

methods and their names of class `Application` by itself.

Let's start the object-oriented program `gPizza3.py`. With the exception of the additional button "Quit" in the toolbar, the user interface looks exactly as it did in the first flight (Figure 7). However, this time the menus and the buttons for which we connected the signals to handlers do behave just as we expect them to.

Integration with a database

In the final step we want to fetch the pizza information from a database using the Python Database API. The user interface doesn't change. We just add a new handler for the signal `select_row` on the columned list. This handler is invoked whenever a row in the columned list is selected.

1. We select the columned list `clist1`.
2. In the tab "Signals" in the property editor we pick the signal `select_row`.
3. We define `on_clist_select_row` as a handler for above signal. . .
4. . . . and add the connection with "Add"

The program will be extended by several lines. Let's have a look at it in Listing 6:

If we compare it to Listing 5 we notice that there are some additional methods namely `fetchPizzaDescription`, `initCList`, `initComboBox` and `on_clist_select_row`.

In `fetchPizzaDescription` we establish a connection to a PostgreSQL database "mydb" prepared earlier, fetch the data by issuing an SQL Statement (`select * from pizzas`) and store it in an instance variable. The `initCList` and `initComboBox` methods serve to initialize the columned list and the combobox with the data that has been fetched from the database previously. They are invoked in the constructor of class `Application`.

When a row in the columned list is selected at runtime the handler `on_clist_select_row` will be invoked. First it will get a reference to the text box `'text1'`, then delete the old contents of the text box and finally insert the description of the selected pizza.

Let's try this out by starting the program. In the columned list we see the contents of table "pizzas" from the database "mydb". If we select any row the description is displayed to the right in the text box. Figure 9 depicts our application in action.

Listing 6: Python program with database query

```
#!/usr/bin/env python

from gtk import *
import gnome
gnome.app_version = "0.3"
import gnome.ui, libglade

import pgsqldb

class Application:
    def __init__(self):
        self.xml = libglade.GladeXML('gpizza.glade')
        nameFuncMap = {}
        for key in dir(self.__class__):
            nameFuncMap[key] = getattr(self, key)
        self.xml.signal_autoconnect(nameFuncMap)

        self.fetchPizzaDescription()
        self.initCList()
        self.initComboBox()
        mainloop()

    def on_exit_activate(self, obj):
        "Quit␣the␣application"
        mainquit()

    def on_about_activate(self, obj):
        "Display␣the␣about␣dialog"
        about = self.xml.get_widget("about2")
        about.show ()

    def on_clist_select_row(self, obj, row,
                            col, event):
        text = self.xml.get_widget('text1')
        text.delete_text(0, -1)
        description = self.catalog[row][2]
        text.insert_defaults(description)

    def fetchPizzaDescription(self):
        db = pgsqldb.pgsqldb('mydb')
        cursor = db.query("select␣*␣from␣pizzas"
                         + "␣order␣by␣price")
        result = cursor.getresult()
        db.close()

        self.catalog = []

        for name, price, description in result:
            self.catalog.append((name,
                                 str(price),
                                 description))

    def initCList(self):
        clist = self.xml.get_widget('clist1')

        for name, price, descr in self.catalog:
            clist.append([name, price])

    def initComboBox(self):
        comboBox = self.xml.get_widget('combo1')

        priceDict = {}
        for name, price, descr in self.catalog:
            priceDict[price] = price

        comboStrings = priceDict.keys()
        comboStrings.sort()
        comboStrings.insert(0, 'All')
        comboBox.set_popdown_strings(comboStrings)

if __name__ == "__main__":
    Application()
```

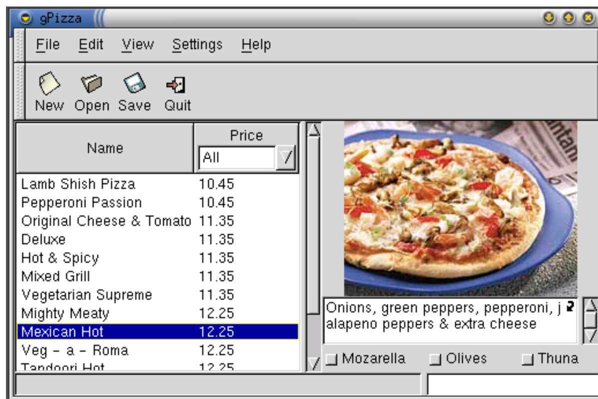

Figure 9: Python program with database integration

Subsequent changes to the GUI

We have now finished a prototype of the “gPizza” application. It’s time to show it to our customers and get some feedback. It doesn’t come as surprise to hear that want to change everything upside down. Thanks to LibGlade we can respond to the customer request with a genuine “No problem”, because we know that working in the changes will be a breeze. We won’t have to change a single line of source code.

We fire up Glade again and start implementing the customer requirements:

1. The customer wants the check buttons where the pixmap is and vice versa. No problem. We cut the horizontal box that contains the check buttons and we cut the pixmap. Using the clipboard window where we choose what to paste we reinsert the check buttons the pixmap back in their new places.
2. The customer also wants to have the columned line to the left as opposed to the right. Again, this is a job for cut and paste using the clipboard window.
3. Finally the customer wants the entire application area, i.e. everything between toolbar and status bar, in a notebook widget. To achieve this we cut the top most horizontal box, insert a notebook instead and paste the horizontal box inside the first page of the notebook.

We start the same Python script *without having changed a single line of source code*. Figure 10 will most certainly make our customer happy!

Conclusion

We have shown that by combining the tools GNOME, Python, Glade and LibGlade a working prototype can be created very rapidly.

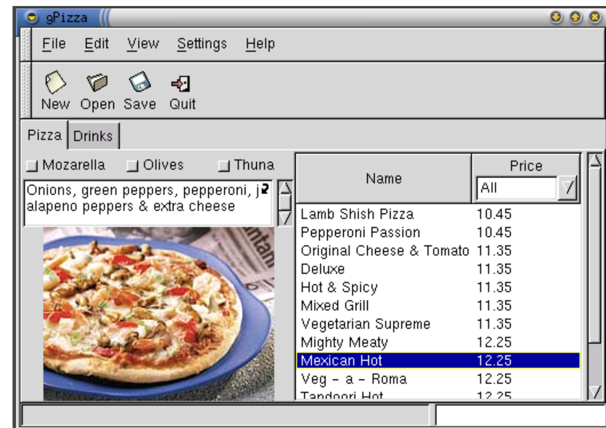

Figure 10: User interface according to customer requirements

With a little extra effort we would be able extend our prototype even further. The possibilities are endless. How about adding a backend to a web server, or fetching the pizza data from an XML file instead of a database? Thanks to the many standard Python modules, features can be developed easily and quickly.

A glimpse into the future

In the world of Free Software development never stands still. Let’s have a look at the general direction in which the tools discussed in this article are expected to go:

GNOME and gnome-python Release 2.0 of GNOME will be based on GTK+ 2.0. There are too many new features too enumerate here. A road map can be found on the GNOME web site [1]. The most prominent feature from a developers point of view is probably the separation of the object system in GTK+. This will allow for a much better integration of the Python bindings. The CVS version currently allows creation of custom GTK+ widgets in Python.

Glade The CVS version of Glade adds more widgets to the palette window. There is now a tab for Bonobo which allows you to insert Bonobo components into your GUI. Bonobo [8, 9] is the GNOME architecture for creating reusable software components and compound documents. Python bindings for Bonobo have been added to GNOME CVS very recently.

There is also a tab for the gnome-db Libraries [10] which contains a series of useful widgets for accessing any database. Unfortunately, at the moment there are no Python bindings for gnome-db.

Python Python has a large user and developer base. There are new releases of the interpreter at regular

intervals. The language itself evolves very conservatively and changes are usually backward compatible.

References

[1] GNOME. http://www.gnome.org/.

[2] GNU Project. http://www.gnu.org/.

[3] Python. http://www.python.org/.

[4] Daemon Chaplin. Glade. http://glade.gnome.org/.

[5] James Henstridge. Libglade Reference Manual. http://developer.gnome.org/doc/API/libglade/libglade.html.

[6] GTK+. http://www.gtk.org/.

[7] Havoc Pennington. *GTK+/Gnome Application Development.* New Riders Publishing, 1999.

[8] Miguel de Icaza. Introduction to Bonobo. http://www.ximian.com/tech/bonobo.php3, 1999.

[9] Bonobo. http://developer.gnome.org/arch/component/bonobo.html.

[10] Rodrigo Moya. GNOME-DB. www.gnome-db.org.

[11] James Henstridge. gnome-python. http://www.daa.com.au/~james/pygtk/.

[12] Hilaire Fernandes. Developing Gnome Application with Python. http://www.linuxfocus.org/English/July2000/article160.shtml.

[13] Deirdre Saoirse. The Ten Minute Total Idiot's Guide to using Gnome/GTK+ & Glade with Python. http://www.baypiggies.org/10mintig.html.

[14] Xianping Ge. Using Python + Glade. http://www.ics.uci.edu/~xge/python-glade/python-glade.html, 2000.

About the author:

Christian Egli holds a master's degree in computer science and works as a software engineer for a company providing telecom test solutions.