Qt Designer and KDevelop

DESIGNER SOFTWARE

JONO BACON

Today, most computer users want graphical applications. But developing graphical applications can be hard work for the programmer. There are, however, tools that can make the job easier. If you want to develop applications using Qt, the library used by the KDE project, one of the best tools to use is Qt Designer. KDE developer Jono Bacon demonstrates how to use it.

***Widget:** An element of a graphical interface, such as a container window, button or field for entering text.*

***Layout management:** This term describes the way in which widgets are arranged in a window. In its simplest form, an element may be placed at a specific position and given a specific height and width. But graphical environments under Linux allow for the system to manage the layout of widgets according to the size and characteristics of the window they are displayed in.*

Software is a funny thing. It is developed by lots of different people with lots of different ideas and ways of working. Some like the simple *vim*+consoles approach in which they manually edit and compile source files. Some prefer a more integrated development system such as KDevelop. Personally I fall into the latter category, preferring to have the class view, automatic Makefile handling and other goodies that KDevelop offers.

Although KDevelop supports the development of GNOME, Qt and console applications it really comes into its own with the development of KDE applications. One feature of KDevelop that has particular appeal is its graphical dialog box designer. With the built-in designer you can visually "draw" your dialog boxes and the code is then generated for you.

Although the designer is good, it isn't great and it has some flaws including:

- limited selection of **widgets** – not all widgets and properties are selectable;
- limited **layout management**;
- limited support for dialog types (QDialog, QWizard, QWidget etc.)

Unfortunately these limitations have quite an impact on the development of KDE applications as you often need widgets that are not supported, or you need layout management. As with any software, there are of course alternatives that provide better support for designing dialogs. But everything changed when Troll Tech, the developers of Qt, released its own dialog creator named *Qt Designer*.

Qt Designer is not just a dialog box creator. It can also design widgets, wizards and more. Qt Designer has advanced layout management support and supports all Qt widgets. Qt Designer also lets you change and edit a great many properties for each widget you use in a dialog and is a very flexible tool in general.

In this article we are going to show you how to harness the power of both KDevelop and Qt Designer to simplify and speed up the development of your applications. Please be aware that this is not a tutorial on KDE/Qt programming or using KDevelop. You should already be familiar with both of these, although the article will still be useful if you are only learning KDE development. Also bear in mind that the example application we will build here is not the most extensive application and is intended simply to illustrate the concepts being described.

Getting going

To get us started doing something useful with Qt Designer we are going to build a simple program that takes a name, email address and a witty comment and generates a signature for an email. To create the application we will need to follow the steps below:

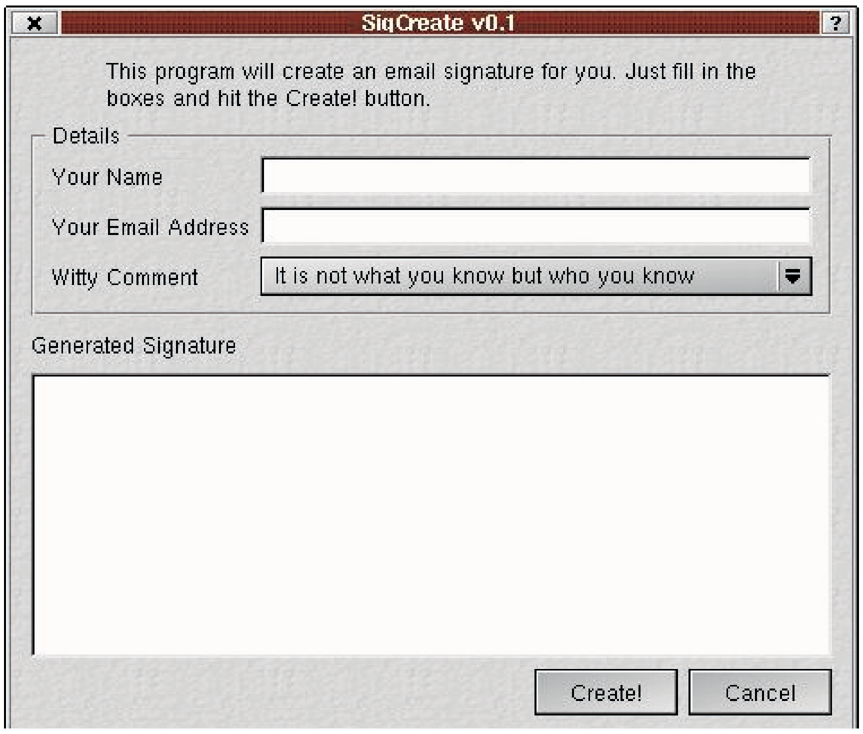
- Design the application in Qt Designer;
- Create the slots and connections in Qt Designer;
- Generate the source code using Qt Designer;
- Embed the generated code in KDevelop and write the functional code.

Designing the program

To design the program we first need a visual idea of the design. To make this easier I have created the design for you: you can see a screenshot of it in Figure 1.

As you can see, we have a window with a number of different items (or widgets) on it, designed so that the user puts the right information in the right boxes. To create this design we need to first fire up Qt Designer. You can do this by either clicking on *Qt Designer* from the Development option in the application starter or by loading the program from the command-line from $QTDIR/bin. ($QTDIR is, of course, the home directory of your Qt files.) When Qt Designer has loaded, you will be presented with a screen similar to that in Figure 2.

Qt Designer's interface is essentially split into three areas. At the top under the menu bar is the widget toolbar section. Here you can see a number of icons representing the different types of widgets



This program will create an email signature for you. Just fill in the boxes and hit the Create! button.

Your Name

Your Email Address

Witty Comment: It is not what you know but who you know

Generated Signature

Fig. 1: The program we will create in the article

How to get Qt Designer

The Qt Designer package itself comes with version 2.2.1 of Qt. It is important that you get at least this version or Qt Designer will not be included. There are several ways to get hold of Qt. The most common method is explained here.

Downloading from Troll Tech

You can download the file from ftp://ftp.troll.no/qt/source. In that directory you will find a variety of versions of Qt available. Make sure you get the right version for your system. Unzip and extract the package when you have downloaded it. You can then compile it.

Compiling Qt

To compile Qt you must first set the QTDIR environment variable. This should point to the directory into which you installed Qt. For example, if you installed it to /build/qt you can set this variable in bash by typing:

```
export QTDIR=/build/qt
```

To make life easier you should edit .bash_profile so this is set automatically when you log in. To compile you should then issue these commands in the following order:

```
./configure -sm -gif -system-jpeg -no-opengl
make
```

Note: make install is not needed.

Qt Designer will be located in the bin directory of your Qt installation directory. You will need to add this directory to your PATH so uic can be found when using Qt Designer.

that can be used in your program. To the left of the screen you can see the Property Editor. This is where your widgets can be fine tuned to behave how you want them to. To the right of this is the Object Hierarchy. This box shows a parent/child relationship view of the various widgets in your program. The space in the middle of the screen is where you will graphically design your program.

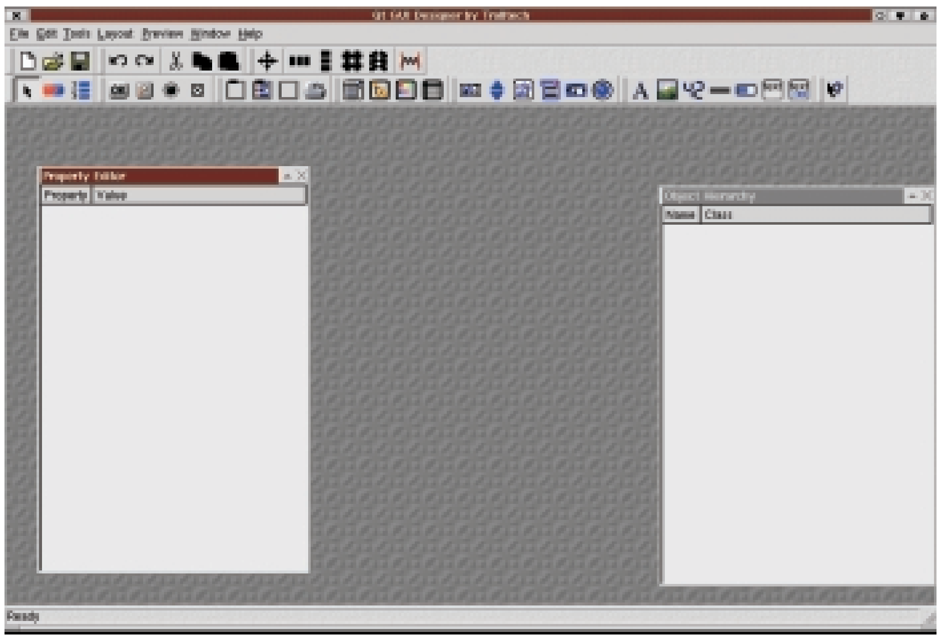
Creating the framework

Let's start by developing the framework that our program will sit in. This can be done quickly and easily in KDevelop by using the Application Wizard to create a KDE2 Mini application. (There was an article about using KDevelop in *Linux Magazine* November 2000.) Call the application SigCreate. Once the Application Wizard has created your application, compile it to ensure that everything is fine. We now have our framework and are ready to start developing our program.

We can now start creating the program interface so that it looks similar to that of Figure 1. First, switch to Qt Designer and select *File->New*. A box will pop up allowing you to select a variety of program templates. The default template is a dialog, and that is fine for our program, so click on the OK button.

A blank window with grid points in it will now pop up. This is your program's window, within which you will design your user interface. You will also see that the Property Editor has been filled with details about the box you have just created. At the top is the name of the box. This will form the class name of the dialog so you should name it something useful. Name it *SigCreateDlg* for now. To do this simply type "SigCreateDlg" into the text box to the right of the *name* property. This is how properties are changed: select the property, then change its' setting on the right.

Figure 2: Qt Designer



Adding widgets

To start we will insert the text at the top of the program window which can be seen in Figure 1. This text tells the user how to operate the program. This type of widget is called a *Label* and you can put one on your program like this:

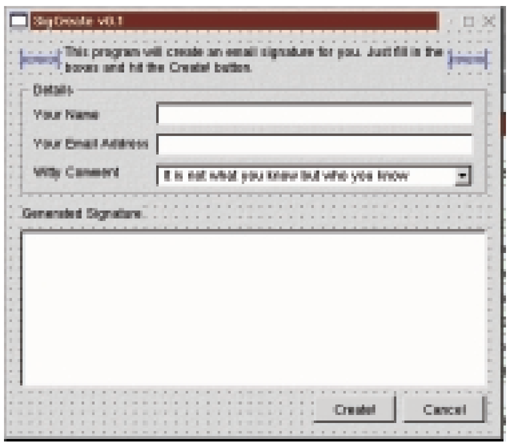
- Select the 'A' icon or *Tools->Display->TextLabel* from the menu;
- The cursor will become a crosshair over your program. Draw a box for the label, just as you would in a paint program, and you will see that the label is created with some dummy text in it;
- To change this text, double click on the label in the box and type in the text;
- Finally, resize the widget using the handles so it is the correct size and at the top of the box. Try to centre the label by moving it with the mouse. This is just a temporary measure. Later on we will look at a more elegant layout management technique.

You follow pretty much the same procedure for embedding any type of widget that is supported by Qt Designer: select it, draw it and finally change its' properties and size.

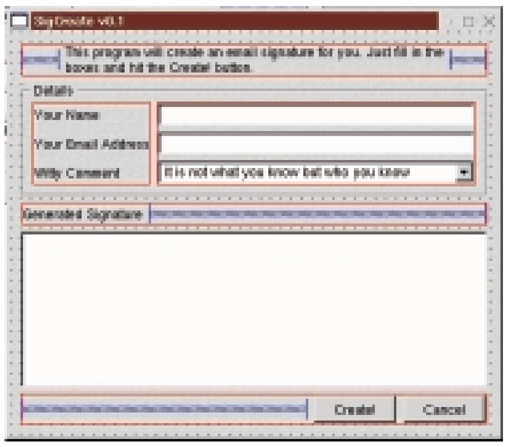
An interesting concept in Qt Designer is that widgets can act as containers for other widgets. This will be demonstrated in our next task, which is to create the input fields inside the frame. You can see that in Figure 1 we have a bunch of labels and text boxes inside a frame. This frame is called a *Group Box* and acts as a container for the labels and text boxes inside it. Let's first create the frame by selecting the Group Box icon or *Tools->Containers->GroupBox* from the menu. You can drag the mouse to create box again. In the Property Editor you can change the *title* property to alter the text in the frame. You may also notice a + symbol in this entry in the Property Editor. This indicates that the property has subproperties that can also be changed.

Once you have created the frame, create three more labels as before but when you draw them, draw them inside the Group Box frame. You can then see in the Object Hierarchy to the right that the labels have become children of the Group Box frame. Once you have done this you can then create the text boxes. The name of this type of widget is a *Line Edit*. (There is also a *Multi Line Edit* that we will use later.) To create a Line Edit select the Line Edit icon or use *Tools->Input->LineEdit* from the menu.

Up to now we have not named any of the widgets that are being placed in our program. We have set the text of the labels and the frame, but we have not set the internal program names for them which are set via the the *name* property at the top. The reason for this is that although it is a good idea to give all widgets a name, it is only really important to set the name of widgets that you are actually going to deal with in your program. In our program we need to manipulate the data from the two Line



This program will create an email signature for you. Just fill in the boxes and hit the Create! button.



This program will create an email signature for you. Just fill in the boxes and hit the Create! button.

[left]
Figure 3: Using spacers in our design

[right]
The completed layout management

Edit boxes so we can generate our signature. As we need to read the text from these boxes we should give them a name using the *name* property. To do this set the name of the top box to ”nameBox” and the bottom box to ”mailBox”. You will see later how these internal names are used.

We can now begin adding the other widgets in the same manner. Most of the widgets simply require you to draw them on the form. One widget that does need a little more explanation is the *Combo Box* widget that will hold the comment for the user to select. Start by creating it like any other widget, then when it is displayed double-click on it. You will then be presented with a box into which you can add the contents of the combo box.

Click on the ‘New Item’ button. A text box will appear. Into it you can type a comment. When you have typed the first comment you can click ‘New Item’ again and start entering the second comment. Repeat this for all the other comments. When you are finished, click the OK button. After you have entered the comments you need to name this widget using the ‘name’ property in the Property Editor. This is because we need to access the comments of the box in our program. Call the Combo Box ”commBox”.

You can now go on and add the other widgets. You will need to add a Label (the text above the large space), a Multi Line Edit (the white space) and two buttons (at the bottom). Name the Multi Line Edit as ”sigBox”. The other widgets do not need to be named, although you can name them if you want.

Getting spaced out

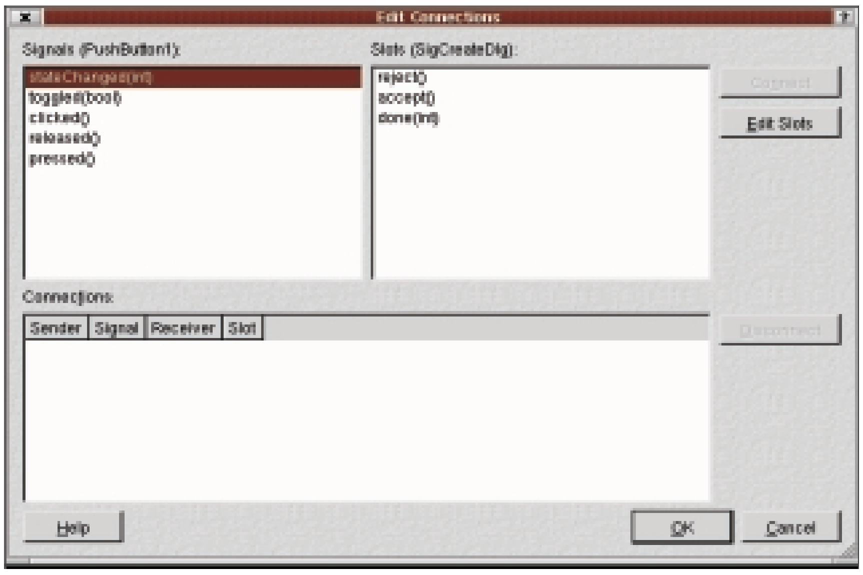
Now all your widgets are in place we can have a quick preview of the form by selecting *Preview->Preview Form* from the menu. Try to resize the window and you will notice that the widgets do not adjust appropriately. To achieve this we need to use a feature of Qt Designer called Spacers. Spacers can be thought of as springs which push the widgets on each side apart. We can use these virtual springs to design our dialogs so that they resize effectively when the user resizes the box.

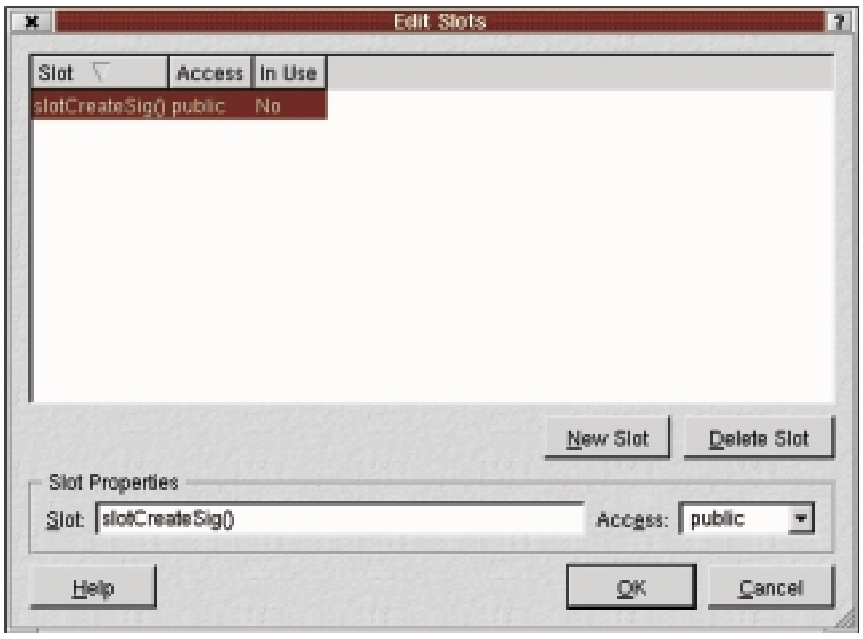
The use of spacers and layout management is a skill that is developed through trial and error. The best idea is just to play with Qt Designer and look at other people’s good and bad efforts. The key rule to remember when dealing with spacers is that you work horizontally first and then vertically. Now you have all your widgets set out, let’s get the spacers created.

The first thing we will do is to make the text at the top of the box centered. To do this we will need a spacer at either side of the text. To create a spacer you can click on the spring icon or select *Layout->Add Spacer* from the menu. Either way you will be presented with a menu from which you can choose either a Horizontal or a Vertical spacer. Choose Horizontal from the menu as we need to center the text horizontally. Next you must click on the form to set the location of the spacer. Click the space to the left of the text and the blue spacer will appear. Repeat this process for the spacer on the right. You can see what this should look like in Figure 3.

Now we have put the spacers in we need to tell Qt Designer how to look after the layout management. To do this we can either use Vertical, Horizontal or Grid management. As we have three objects in a row (the two spacers and our label) we can use Horizontal management. To do this we need to select the left spacer with the mouse, hold down Shift and then select the label and the right spacer so all three are selected. We can then click on the Horizontal Layout icon which is three blocks next to each other, or use *Layout->Lay out Horizontally* from the menu. You will then see a resizable red line around the three objects to indicate that their layout is being managed.

We can now repeat this procedure for the the three labels inside the group box (we don’t need spacers between each label), this time using vertical layout management (not spacers). You can also use vertical management for two text boxes and the combo box. The reason why we are using vertical management for the labels is because we want them to be aligned as they currently are, and spacers can distort objects that need to be aligned in a specific way. Vertical management is also used on the text boxes and combo box as they are equally sized.





[top]
Figure 5: Making the connection

[above]
Figure 6: Making a slot

The text above the multi line edit needs a spacer to the right and horizontal management, and the buttons at the bottom need a spacer to the left and horizontal management. After all this your form will contain a number of red boxes. To finish the layout we need to let the form look after the laid-out boxes. This is a simple matter of right clicking the form and selecting "Lay out in a Grid" from the menu. The final design with layout lines should resemble something similar to Figure 4.

Slotting things into place

Now the widgets are implemented and the layout is arranged the final thing we need to do in the design stage of the form is to create the **signal/slot** connections. To do this manually requires coding a connect() function, but Qt Designer provides a simple yet effective solution. Before we look at how to create the connections in Qt Designer, we must explain how Qt Designer handles slots in your programs.

Signals and slots: *Signals and slots are the mechanism used by a program to communicate with the widgets that form its graphical interface.*

The whole idea of Qt Designer is that you can visually create a dialog box, Qt Designer will generate the code and you will not need to modify that code at all. This is all fine and dandy until you want to create your own slot: to do this you would need to edit the code to create the functionality of your slot. This obviously contradicts the idea of not editing the generated code, so a solution is needed.

Qt Designer solves this problem with a nifty bit of coding. It does it using Virtual Methods. If you are unsure what virtual methods are, I would recommend picking up a good C++ book and reading up on virtuals. What you need to do, very basically, is to create a subclass of the dialog's class and in it create a slot of the same name. When you then create an object of this subclass the right slot will be used. In many ways this is a good thing to do as it keeps the implementation separate from the interface which is at the heart of good C++ coding.

To create the signal/slot connections we need to use the connecting tool. To do this either select the icon (it looks like a red arrow going into a blue square) or select *Tools->Connect Signals/Slots* from the menu. To create the connection click on the widget that is going to be dealing with the slot, drag the line off the form and release the mouse button.

Let's deal first with the Create! button. Click on the button with the crosshair and drag the line off the form completely. When you have released the button you will see the connections tool shown in Figure 5. What we want to do is to create a slot that will create our signature when the user clicks on the button. To do this we first need to create the slot, and then do the connection.

To create the slot we need to click on the "Edit Slots" button. You will see the slot creation boxshown in Figure 6 appear. Now click on the 'New Slot' button and a slot will appear in the box. You can now rename it and set its access specifier. For our project, set the name to *slotCreateSig()* and leave the access specifier as public. When you click on OK you will be returned to the connections box and you will see your new slot in the Slots section of the box.

To make a connection you simply select the appropriate signal (which is *clicked()* in our case) and then select the slot (which is our new slot *slotCreateSig()*). When you have selected both signal and slot you will see the connection made at the bottom of the screen. After you are finished click OK. You can repeat this procedure for the Cancel button by using the clicked() signal and the reject() slot.

Generating the source

Now we have created the box, widgets, layout managers and connections, we can finally generate the source code for your dialog. To do this we will be using a special command line tool called *uic* that is included with Qt Designer. The function of *uic* is to take the saved file that Qt Designer creates when you save your designed dialog (which is a file full of special XML code) and convert it to the C++ code that the compiler understands.

The first thing to do is to copy the Qt Designer file to your projects directory if you haven't done so already. The file needs to be in the same directory as the other project source code (in our example (~/sigcreate/sigcreate/). You can then use *uic* to generate the header. We will assume that your Qt Designer file is called *sigcreatedlg.ui*:

```
uic -o sigcreatedlg.h sigcreatedlg.ui
```

To create the file with the implementation code in it we use the following command:

```
uic -i sigcreatedlg.h -o sigcreatedlg.cpp si➘
gcreatedlg.ui
```

Now that the code for the dialog has been generated we can add it into our KDevelop project. Fire up KDevelop if it is not already open and select *Project/Add existing file(s)* from the menu. You can then import the .h and .cpp files that you just generated into the project. When these have been imported, compile the project to make sure everything worked OK by hitting F9. If no errors are reported, everything worked fine.

The next step is to adjust the class that KDevelop generated for you to inherit the new dialog class. To do this add:

```
#include "sigcreatedlg.h"
```

at the top of the sigcreate.h file, and add ": public SigCreateDlg" to the end of "class SigCreate".

Next we need to add some other include files to the various files so that the new dialog class is loaded correctly. You will need to add:

```
#include <qmultilinedit.h>
#include <qlineedit.h>
#include <qcombobox.h>
```

We can then create the slot in our subclass by right clicking on SigCreate in the class view and selecting "Add member function". In the box create a slot called slotCreateSig() and make sure it is public. In the generated slot we can then actually write the code that generates the signature:

```
this->sigBox->insertLine("\n--", 1);
this->sigBox->insertLine(this->nameBox->tex➘
t(), 2);
this->sigBox->insertLine(this->mailBox->tex➘
t(), 3);
this->sigBox->insertLine(this->commBox->curr➘
entText(), 4);
```

Finally ensure that all references to *QWidget* are removed from the SigCreate class declarations. This is because our dialog uses QDialog and the KDevelop-generated projects use QWidget. The first line of the SigCreate class should be:

```
class SigCreate : public SigCreateDlg
```

and the constructor should be:

```
SigCreate::SigCreate()
```

in the *sigcreate.cpp* file. Yhe class declaration of the constructor should be:

```
SigCreate();
```

Once all of these steps have been completed you can compile and run the program. If you had any trouble understanding the changes that you made to the code, we would recommend reading up on KDE and C++ programming. Unfortunately there isn't the space to fully explain the changes just made, bearing in mind that the focus of the article has been on showing how Qt Designer can help your program development.

This article has provided a simple walkthrough on getting started with Qt Designer. Although we have covered the main elements of Qt Designer, there are many more concepts and techniques that can be learned. It is well worth taking a look at some of the things listed in the Info box to see where you can get more information on getting the best out of Qt Designer and KDevelop.

With KDE becoming increasingly popular with home users, business users and enthusiasts, the scope for KDE development is getting more and more exciting. Taken together with the rapid development and maintenance of KDE itself and the increased productivity that can be achieved using development tools such as Qt Designer and KDevelop and you have a lot of opportunities available. Good luck, and let me know how you get on! If you have an IRC client go to *irc.openprojects.net* and join #kde. My nickname is [vmlinuz]. Come and chat to me if you have any problems. If I am not there, just ask someone in the channel when I will be on and I will try to help as best I can. ■

Getting the example source code

If you would like to download the source code from the example project today you can get it from my web site at: http://www.jonobacon.co.uk/writing/qtdestut/index.html . The file is called sigcreate.tar.gz. Once you have downloaded the code, you can unzip it by typing:

```
gunzip sigcreate.tar.gz
```

You can then unpack the code with:

```
tar xvf sigcreate.tar
```

The code will then be extracted into a directory. In there will be the KDevelop file that you can load to play with the code.

Info

KDE home page
http://www.kde.org/
Troll tech
http://www.troll.no/
KDE Developers' site
http://developer.kde.org/
KDevelop home page
http://www.kdevelop.org/
KDE mailing list info
http://www.kde.org/contact.html
KDE mailing list archives
http://lists.kde.org/
■