

Beginning KDevelop Programming

pseudonym67

Copyright (c) 2006 by pseudonym67. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at <http://www.opencontent.org/openpub/>)

About the Author

Since becoming disabled from work and becoming practically house bound I have had a number of projects [published on CodeProject](#). and have decided to play around with some stuff on Linux. All projects are the results of the few hours a day if any when I am neither asleep or ill and will hopefully continue to prevent me from climbing the walls.

About This Project

This project is the authors attempt to learn how to program using the KDE programming environment. It is by no means a C++ tutorial nor is it a general guide to setting up KDE in fact initially it wont even be a wide ranging look at KDE itself because the idea is to use a familiar environment to standard Windows programming which means that I will initially be concentrating on the Simple Designer Based KDE Application projects which is the KDE version of a rapid application development environment of the type that most windows programmers will be familiar with and will provide a less stressful introduction to Linux programming for people new to the KDE development environment.

The idea is to provide a programming guide along the lines of the many Windows programming guides that can be found in any book shop, although being for Linux this guide will be freely available to all and as I am not being paid by the page it will not detail every single option like some books I could mention but will leave some of the details to the reader to discover.

Requirements

The project is developed using a standard set-up of Suse Linux 10 which comes with KDE 3. There will be no special settings or scripts to run basically if it doesn't work out of the box it wont be covered here. (As with all rules there is an exception. This is the scripts used to set up MySQL) Changes made through the use of the official Suse Updates that are downloaded through the Suse Watcher program are of course accepted.

For people who are not as familiar with C++ as they would like to be or find they are having problems I'd recommend [The C++ Programming Language](#) or you could see what you can find on the web. Or you could just read the C++ Annotations book by Frank B. Brokken that comes as part of the KDevelop help documentation. On the whole we wont be using too much advanced C++ but readers should be familiar with at least function overloading, inheritance and constructors.

All programs will be developed using the KDevelop KDE/C++ version of KDevelop. All other tools used will be part of the KDE setup so as long as KDE is installed they should be available on any Linux distribution.

Release Schedule

When there's something to release.

Beginning KDevelop Programming

KDE 3.x Version

pseudonym67

pseudonym67@hotmail.com

<http://www.beginning-kdevelop-programming.co.uk/>

Table of Contents

| | |
|--|-----|
| Chapter 1 The Basics..... | 9 |
| Version Control | 14 |
| Creating A Repository..... | 16 |
| Viewing The Files..... | 18 |
| Initialising The Repository..... | 19 |
| KDE Project Files..... | 20 |
| The KDE Application Class..... | 31 |
| Summary..... | 31 |
| Chapter 2 The KDE Application..... | 32 |
| The Command Line..... | 33 |
| Code Snippets..... | 41 |
| A Better Demonstration | 42 |
| Coding Styles..... | 43 |
| Adding The Demo Code..... | 43 |
| Debugging Code..... | 45 |
| CVS The Easy Way..... | 52 |
| Documentation..... | 55 |
| Generating The Documentation..... | 60 |
| Doxygen Options..... | 62 |
| Generated Files..... | 65 |
| Summary..... | 71 |
| Chapter 3 Common Widgets..... | 72 |
| Problems Building Forms..... | 82 |
| Buttons..... | 84 |
| Tip Of The Day..... | 87 |
| Boxes..... | 88 |
| Spacers..... | 96 |
| More On Layouts..... | 98 |
| Edits..... | 100 |
| Tab Order..... | 101 |
| Chapter 4 Containers And Views..... | 103 |
| Accelerators..... | 108 |
| Graphical Connections..... | 108 |
| The Views Demonstration | 113 |
| Adding Libraries To A Project | 117 |
| Selecting A Directory..... | 122 |
| Filling The Views..... | 124 |
| Chapter 5 Database Programming With MySQL..... | 127 |
| Order..... | 127 |
| 1. Install My SQL..... | 127 |
| 2. MySQL_Install_DB..... | 128 |
| 3. Start MySQL..... | 129 |
| 4. MySQL_Fix_Privilege_Tables..... | 130 |
| 5. Change Root Password..... | 131 |
| 6. Uninstall MySQL..... | 131 |
| Using a DataTable..... | 146 |
| Adding A Dialog..... | 146 |

| | |
|---|-----|
| Implementing The Dialog..... | 154 |
| Query The Database..... | 154 |
| Tip Of The Day..... | 158 |
| Using the DataBrowser..... | 159 |
| Reusing A Dialog..... | 169 |
| Using The DataView..... | 174 |
| Chapter 6 Input And Display..... | 183 |
| Inputs..... | 183 |
| Tip Of The Day..... | 185 |
| Displays..... | 187 |
| Files And Some Common Dialogs..... | 189 |
| Summary..... | 195 |
| Chapter 7 KDE Display Widgets | 197 |
| KDE Display..... | 197 |
| Tip Of The Day..... | 200 |
| KDE Image Preview..... | 202 |
| KDE Animation..... | 204 |
| Icons..... | 206 |
| KDevelop Icons..... | 207 |
| KDE Colours..... | 209 |
| Implementing Our Own Connections..... | 210 |
| Summary..... | 211 |
| Chapter 8 KDE Buttons And Input..... | 212 |
| Text And Fonts..... | 213 |
| Timers..... | 217 |
| Numbers..... | 218 |
| Tip Of The Day..... | 219 |
| File Viewer..... | 220 |
| Summary | 223 |
| Chapter 9 KDE Containers and Views..... | 224 |
| Views..... | 224 |
| Containers..... | 233 |
| Tip Of The Day..... | 234 |
| Menu Items..... | 241 |
| Summary..... | 246 |
| Chapter 10 Custom Widgets..... | 247 |
| Building Your Own Widgets..... | 247 |
| Testing The Widget..... | 249 |
| Tip Of The Day..... | 251 |
| Adding A Custom Widget..... | 252 |
| Defining The Widget..... | 253 |
| Defining the Signals..... | 253 |
| Defining The Slots..... | 255 |
| Defining The Properties..... | 255 |
| Create A Virtual Keyboard..... | 257 |
| Changes To KSquareButton..... | 258 |
| Creating the KVirtualKeyBoard Widget..... | 259 |
| Summary..... | 262 |
| Chapter 11 Events..... | 264 |
| KeyBoard Basics..... | 264 |
| Tip Of The Day..... | 269 |

| | |
|--|-----|
| Mouse Basics..... | 269 |
| Popup Menus..... | 270 |
| Collection Basics and Drawing..... | 273 |
| Summary..... | 275 |
| Chapter 12 Drawing..... | 276 |
| KDE Colours..... | 276 |
| Tip Of the Day..... | 282 |
| Drawing Example..... | 283 |
| Owner Draw Menus..... | 285 |
| Drawing The Shapes..... | 288 |
| Menus And Double Buffering..... | 293 |
| Menus..... | 293 |
| Actions..... | 295 |
| Saving and Loading QPixmap..... | 298 |
| Double Buffering..... | 299 |
| Adding A Main Window..... | 300 |
| Setting The Menu Text..... | 311 |
| Adding The Status Bar..... | 311 |
| Adding ToolBars..... | 312 |
| Custom ToolBar Buttons..... | 314 |
| The Generated Functions..... | 317 |
| Help | 318 |
| Summary..... | 320 |
| Chapter 13 : Global Information and Configuration Files..... | 321 |
| KDE Information..... | 321 |
| Splitter Windows..... | 321 |
| Tip Of The Day..... | 325 |
| Application Settings With KDE..... | 326 |
| Using KConfig..... | 335 |
| QMaps..... | 336 |
| Opening A Config File..... | 336 |
| Saving A Config File..... | 337 |
| Editing A Config File..... | 338 |
| Finishing Up..... | 340 |
| Summary..... | 341 |
| 14 A Simple Editor Application..... | 342 |
| The KDE Way..... | 342 |
| DocBooks The Simple Guide..... | 346 |
| Chapters..... | 347 |
| Paragraphs..... | 348 |
| Links..... | 348 |
| Screenshots..... | 348 |
| Installing A Docbook..... | 349 |
| Menus..... | 360 |
| Chapterfourteenfiles..... | 362 |
| Loading And Saving..... | 362 |
| Using The Spell Check..... | 363 |
| Tip Of The Day..... | 364 |
| Setting Up The Spell Checker..... | 364 |
| Using The Spell Checker..... | 365 |
| Finishing The Docbook..... | 368 |

Summary.....370

Appendix A Upgrading KDevelop.....371

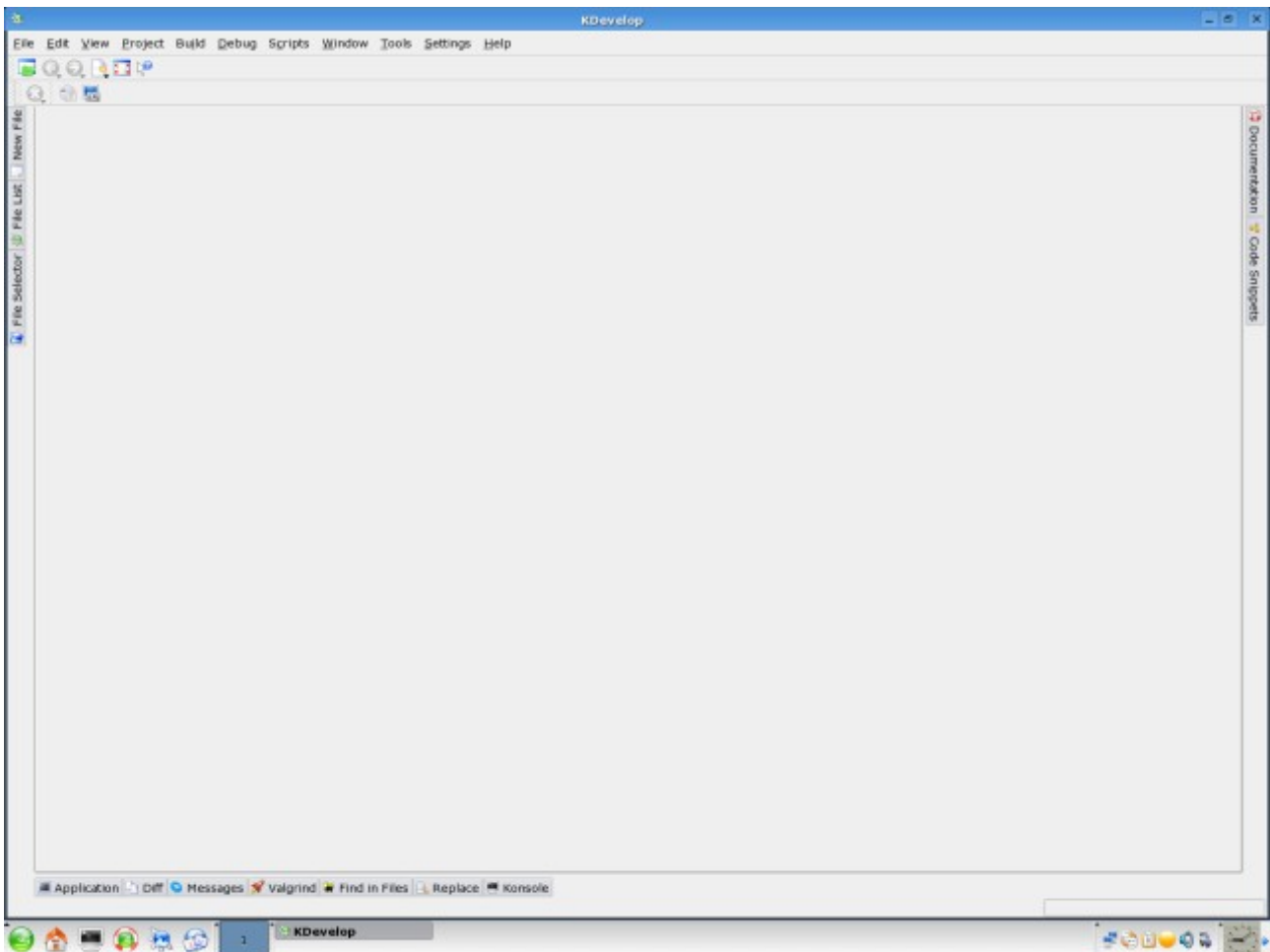
Part One

Introducing KDevelop

Chapter 1 The Basics

In this chapter we will look at the default Hello World application generated by KDevelop and some of the tools that we will be using in future chapters. The aim of this chapter is to give the reader an understanding of how the default Hello World application works to the point where they would be able to start developing on the basic framework. It should be noted that throughout we will be treating different libraries as part of the same development environment. Putting it simply you can't develop KDE programs the way we are in KDevelop without relying on the Qt libraries and at some point probably on the Standard Template Library or on other libraries that come as part of the system. These libraries form the development environment that we are working with and as such will be treated as a whole development environment and not as separate libraries.

Your initial view of the KDE Development Environment will probably look something like this.



"Web

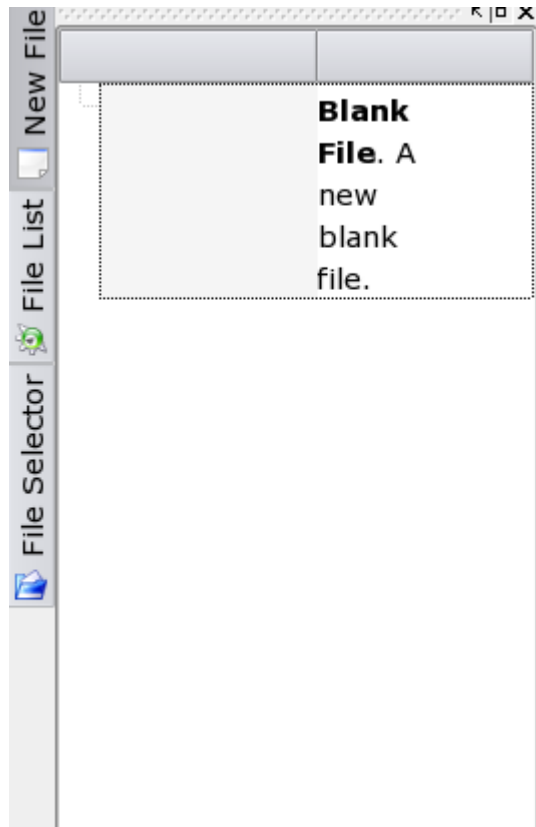
Take a good look as this is probably the last time you will see the KDE environment looking this simple as the KDE environment is configured to automatically open the previous project on opening there will normally be files open in the main window.<http://www.linuxtoday.com/>

The Top Menu

The menu's at the top contain the standard options for a windowed environment with all the file, save, edit etc. options that should be familiar to anyone that has actually used a computer before. The main ones of interest for us at the moment are the Project menu and the Settings menu the Project menu because it will allow us to create our first project and the Settings menu so that we can configure KDevelop to use the tab spaces, colouring, fonts, etc. that we are used to. As these are entirely personal preferences anyway I'll leave you to play around with them.

The Left Side Bar

There are initially three buttons on the left hand side of the application that are, reading down, New File, File List and File Selector. The most important of these is the New File option. If you click on it you will see,



This is initially empty but later on will allow us to create new files to be added to the projects that we are working on.

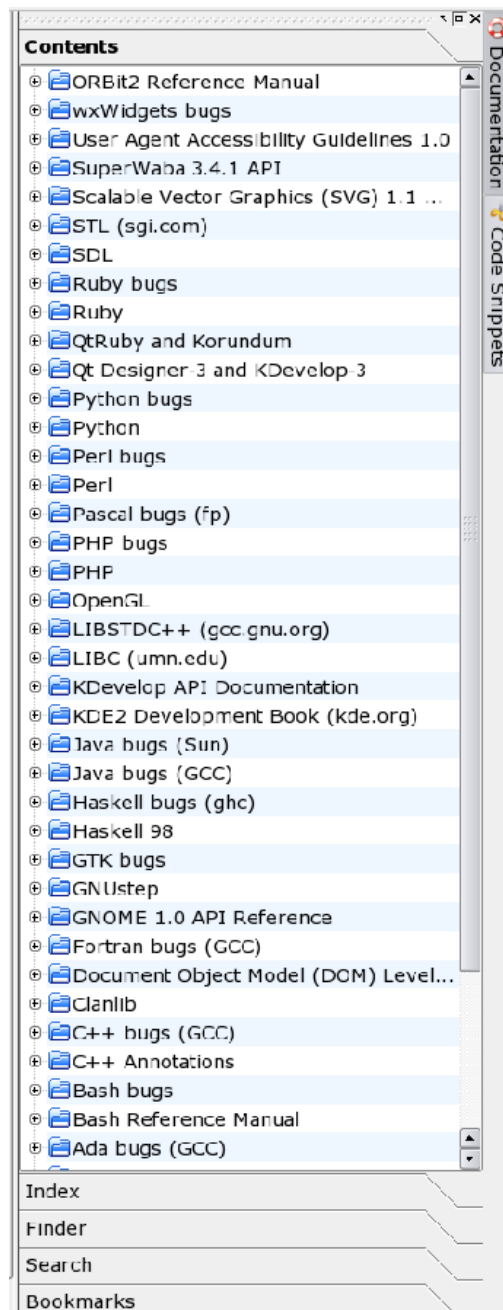
The File List button shows the files that are currently open within the development environment and the File Selector option is a quick access file manager that will allow you to open files in the Development Environment.

Chapter 1 The Basics

The Right Side Bar

There are initially two buttons on the right side bar these are the Documentation and the Code "Web Snippets" buttons.

The Documentation when clicked looks like,



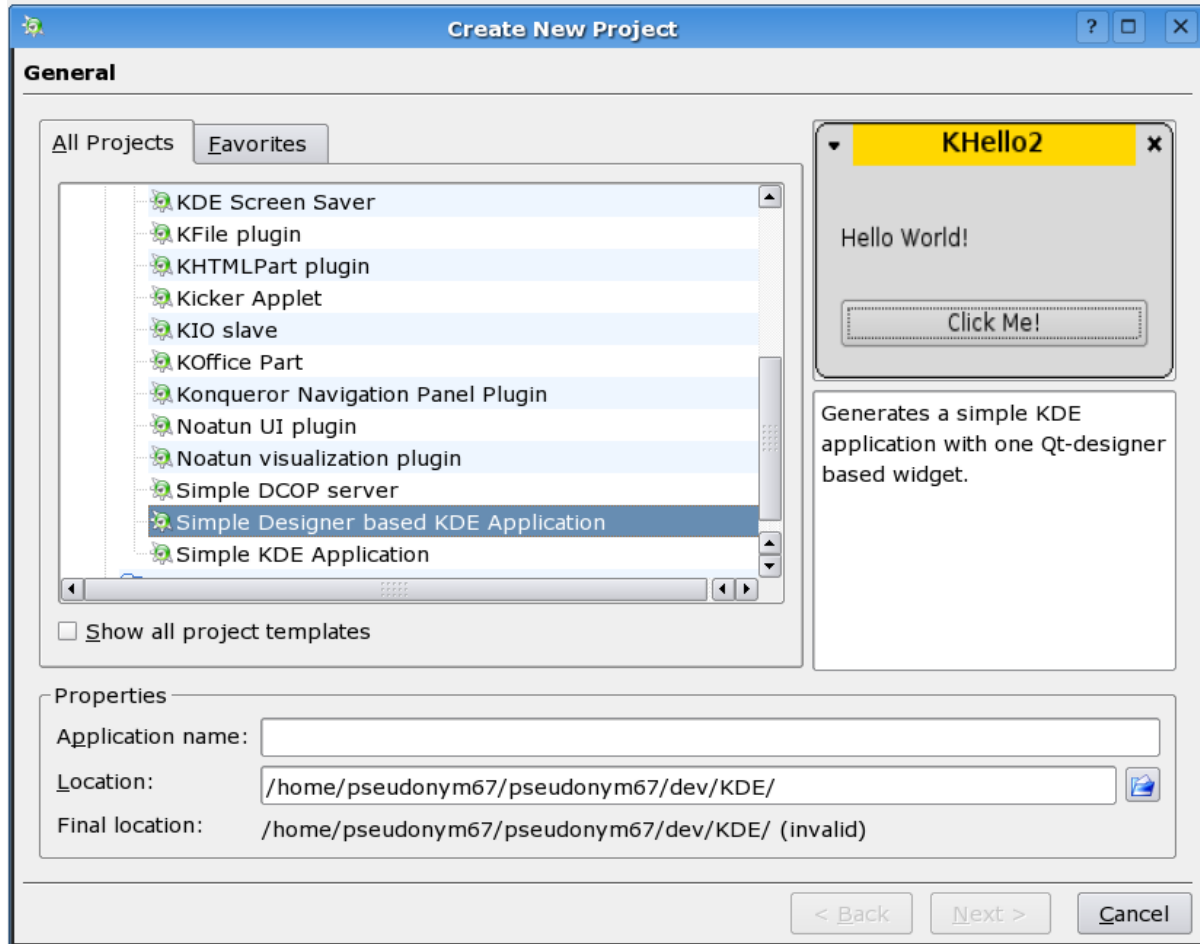
This lists all the programming materials that are installed as part of the set-up. and can be searched whenever you are looking for function references etc.

The Code Snippets button is a place set aside where you can place pieces of reusable code that can

Chapter 1 The Basics

then be inserted into projects at will. This is something we will use as we go along.

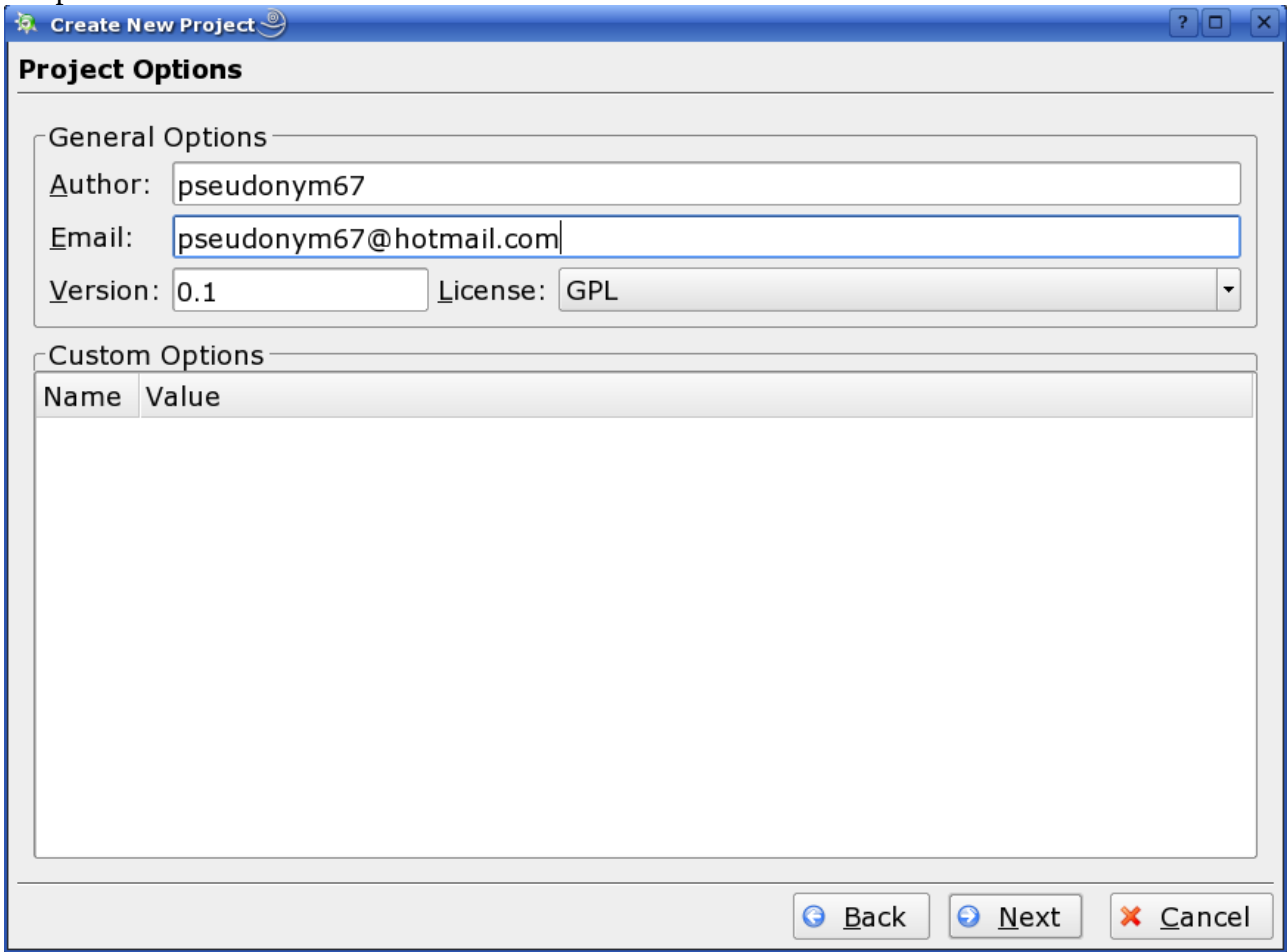
There are also a number of buttons at the bottom of the screen which for now we can ignore as they are mostly output windows and knowing everything about them is not particularly necessary at this point.



The First Project.

The above dialog is how we are going to start all the programs that are developed here, you can get to it by clicking on Project/New Project and then by clicking on C++/KDE and scrolling down until you reach the Simple Designer based KDE Application. All we need to do in this dialog is set our source code directory, in my case it will be off the KDE folder and type in the name of our application, which in this case is ChapterOneHelloWorld. Once this is done click the active Next button.

The next dialog sets up the project options and looks like this,



Create New Project

Project Options

General Options

Author: pseudonym67

Email: pseudonym67@hotmail.com

Version: 0.1 License: GPL

Custom Options

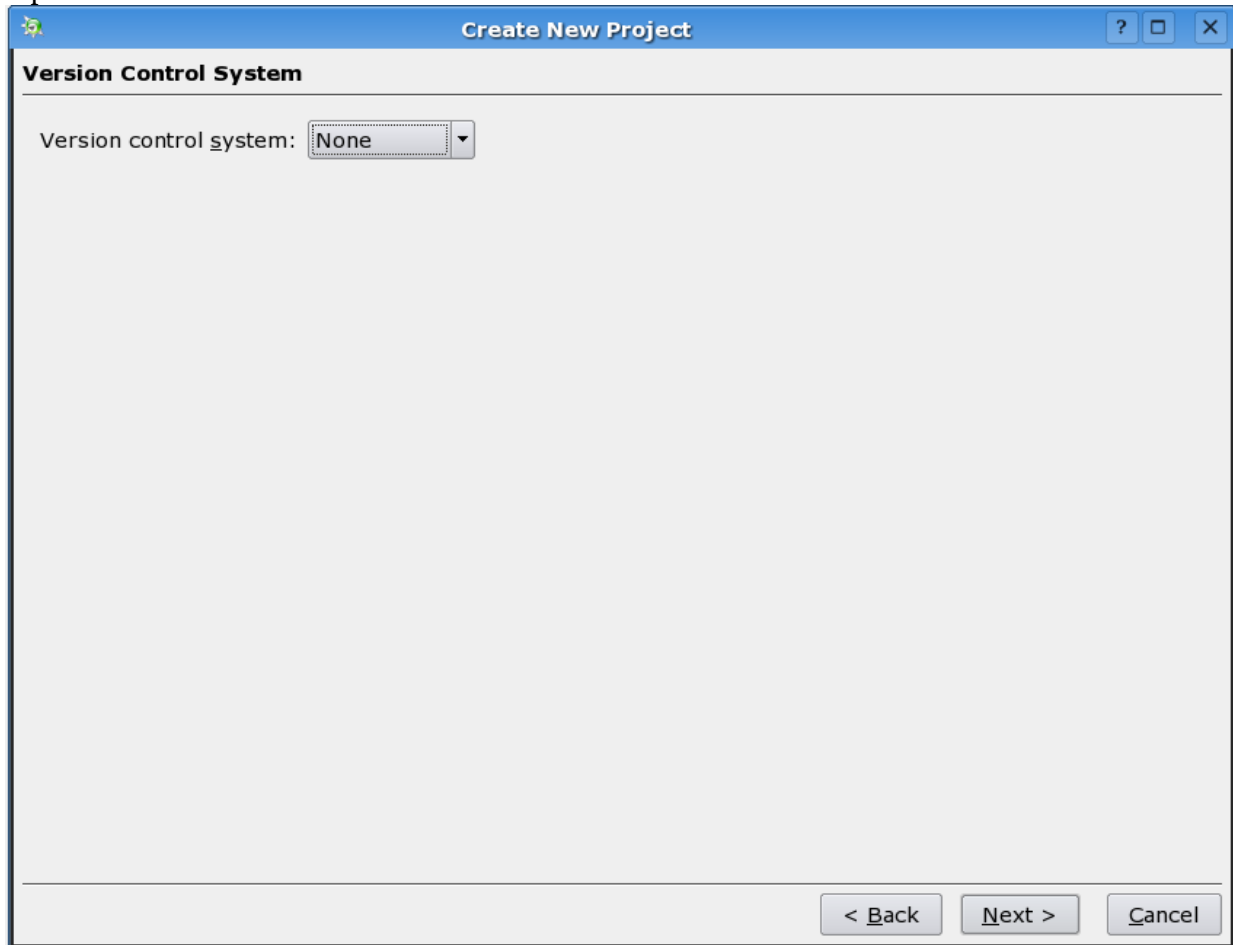
| Name | Value |
|------|-------|
|------|-------|

Back Next Cancel

For this application we set up the Author and the email address. These will be set to defaults when the dialog opens but you have the option to change them here. This license information will be written to the top of all the project files that are created within the project. The licensing options default to the GPL which is fine for our purposes.

Version Control

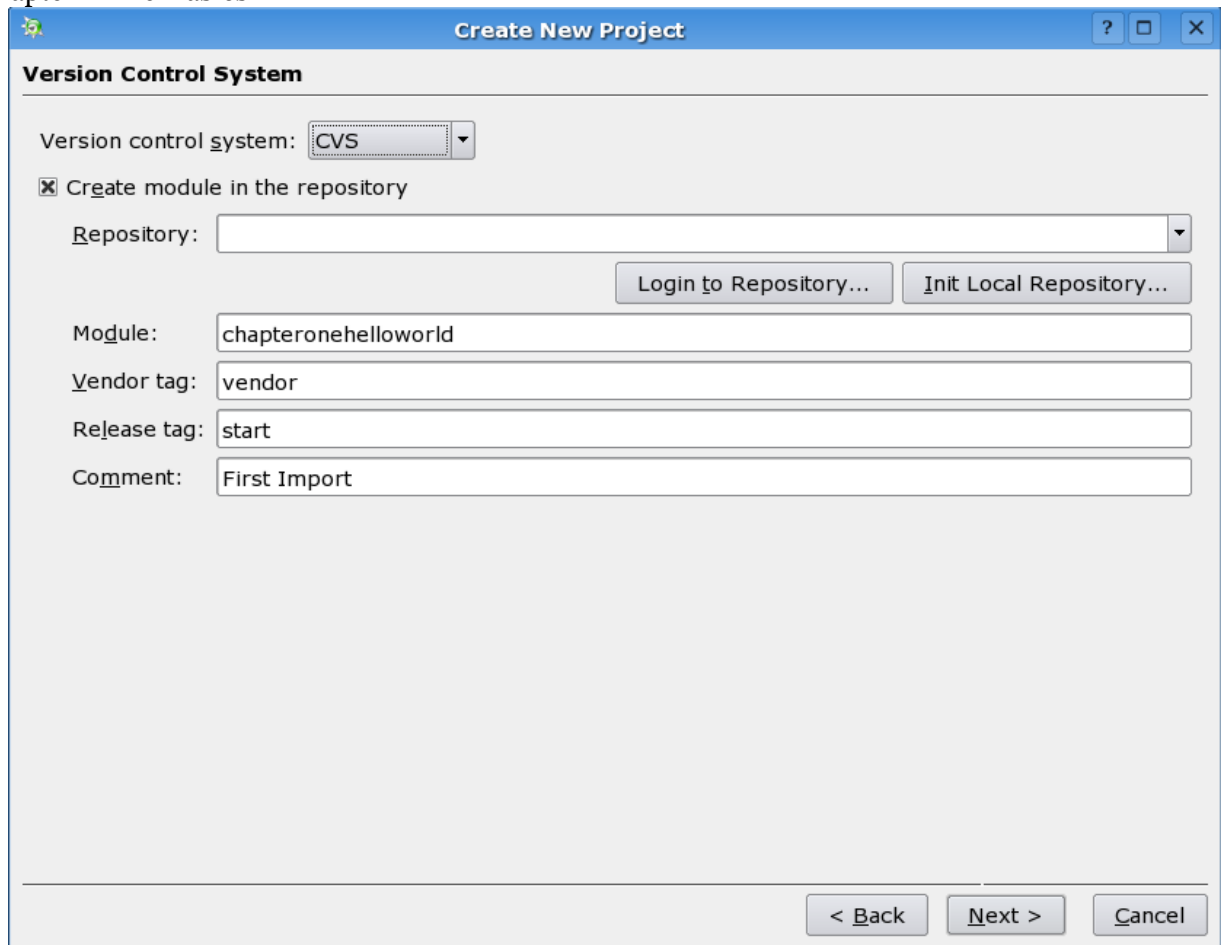
When we are happy with the setting we can move on to the version control options. the original dialog looks like this,



It should probably be noted here that when working on single person projects version control systems are largely unnecessary unless you develop a lot of libraries or do a lot of experimental code and using a version control on a single development computer is kind of defeating the object. We are however, going to use one here just to get used to it, so that development with version control systems will become as it should be second nature when developing projects. Also with this being Linux there is more of a chance that readers will go on to develop parts of the system which will require them to be comfortable with version control systems.

For this application we are going to set up a CVS version control. Why CVS? Simply because we have a gui to make life easy for with Suse under Development/Revision Control/CVS Frontend (Cervisia) from the system menu.

Chapter 1 The Basics



The screenshot shows the 'Create New Project' dialog box with the 'Version Control System' tab selected. The 'Version control system' is set to 'CVS'. The checkbox 'Create module in the repository' is checked. The 'Repository' field is empty, and the 'Module' field contains 'chapteronehelloworld'. The 'Vendor tag' is 'vendor', the 'Release tag' is 'start', and the 'Comment' is 'First Import'. There are buttons for 'Login to Repository...' and 'Init Local Repository...'. At the bottom are '< Back', 'Next >', and 'Cancel' buttons.

Create New Project

Version Control System

Version control system: **CVS**

☒ Create module in the repository

Repository:

Module:

Vendor tag:

Release tag:

Comment:

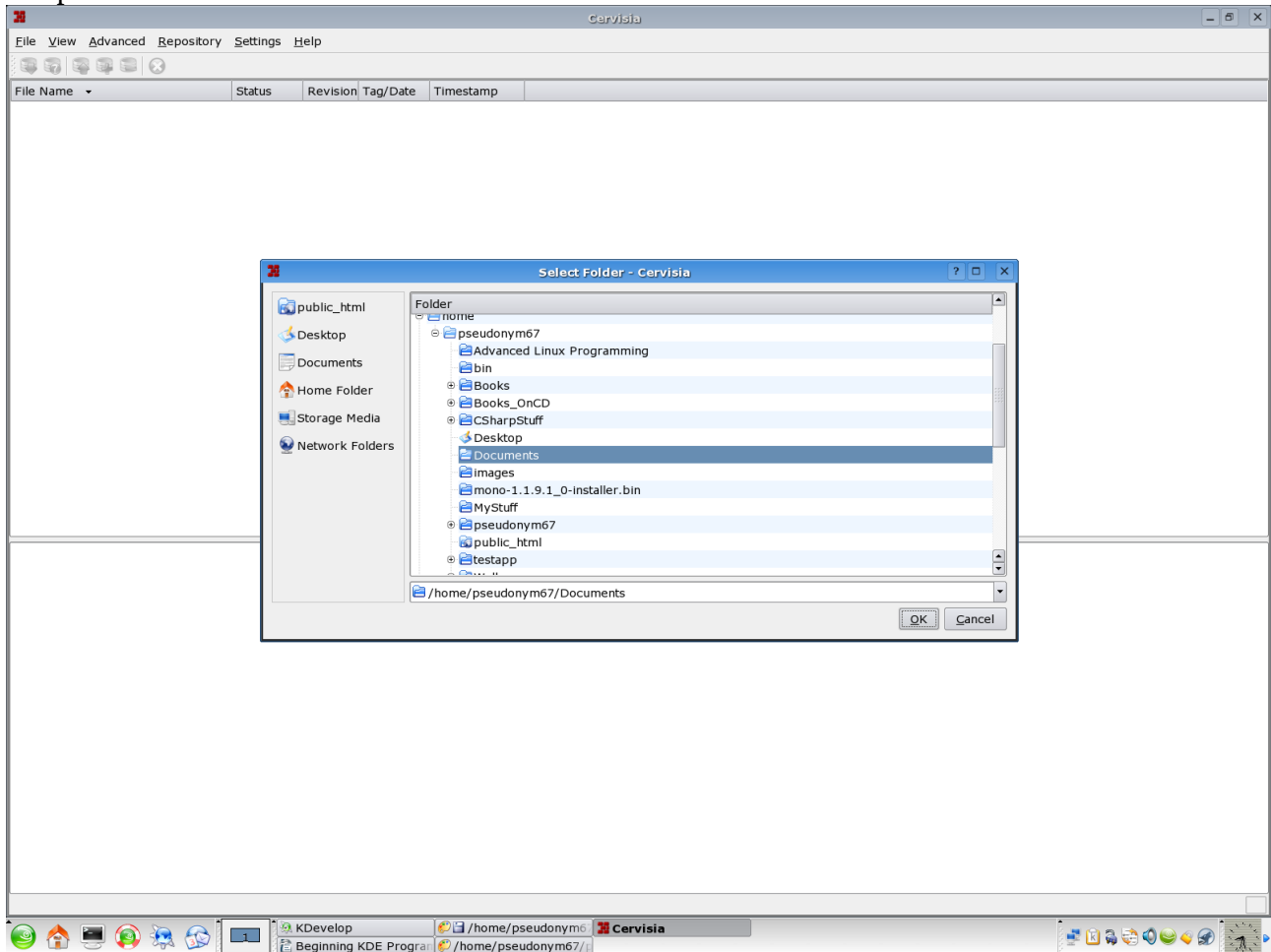
Once we have selected CVS and told KDevelop to create a module in the repository by checking the available box we need a location for our version control. The above image

At this point there are two ways of doing things these being the to initialise the repository or to create one if one doesn't already exist.

Creating A Repository

To Create a repository for your source code open up Cervisia and click Repository/Create

Chapter 1 The Basics

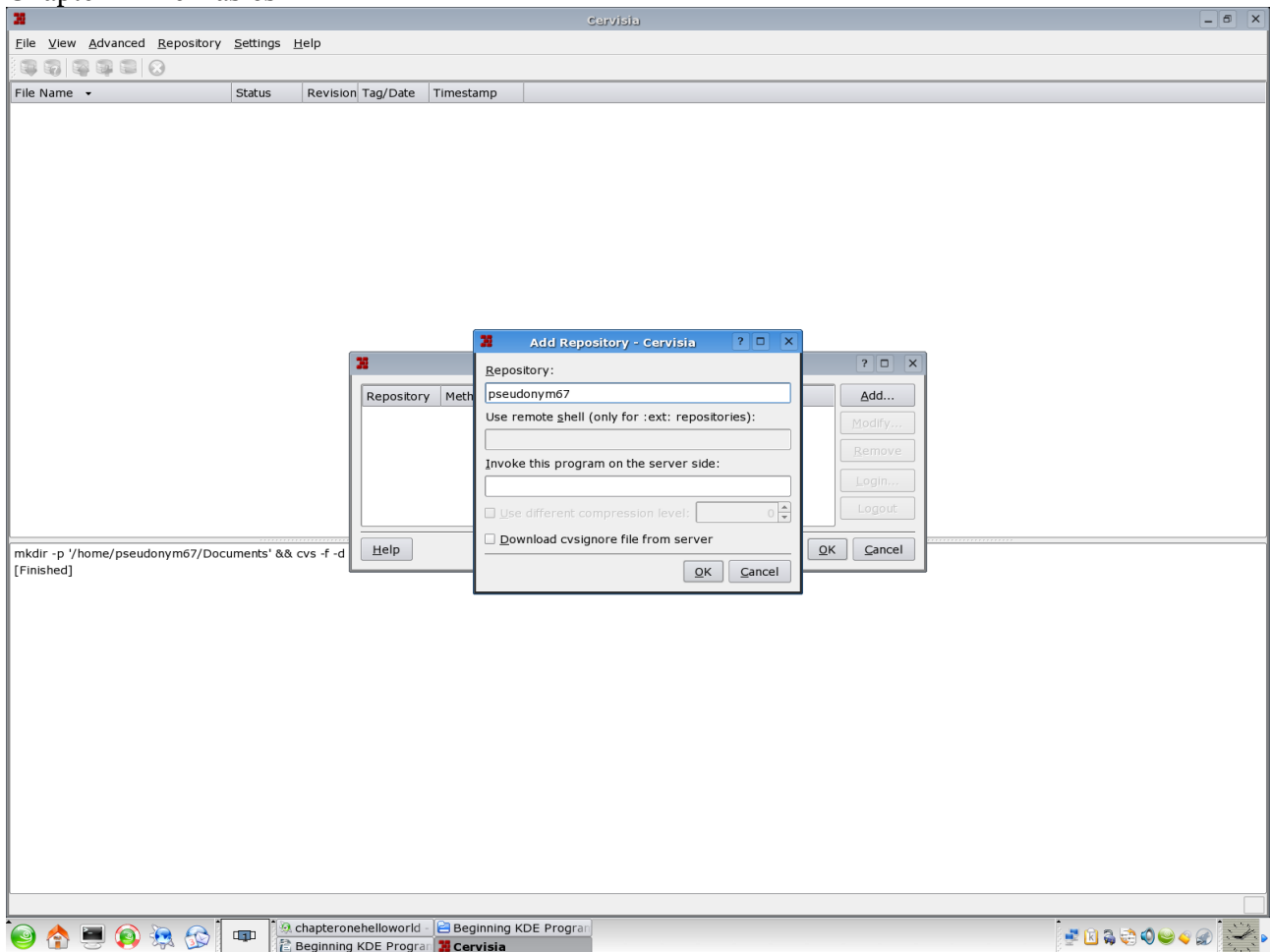


The image above shows Cervisia as it starts up from a clean install when no repositories at all are set on the system. As with KDevelop this is probably the last time you will see it looking like this as it will automatically load the last open project on start up.

When you click on Repository Start you will be asked to select a directory to save your versions to, I've chosen to create the main CVS directory off the Documents folder. You are then given a simple dialog which lists the current available repositories on the system which you can see just behind the dialog that has the focus. This is currently empty so we need to add one. The name you choose is entirely up to you, just type something in and click OK.

Another way to create a repository is to go to Repository/Repositories which will allow you to add new repositories to you system as shown below.

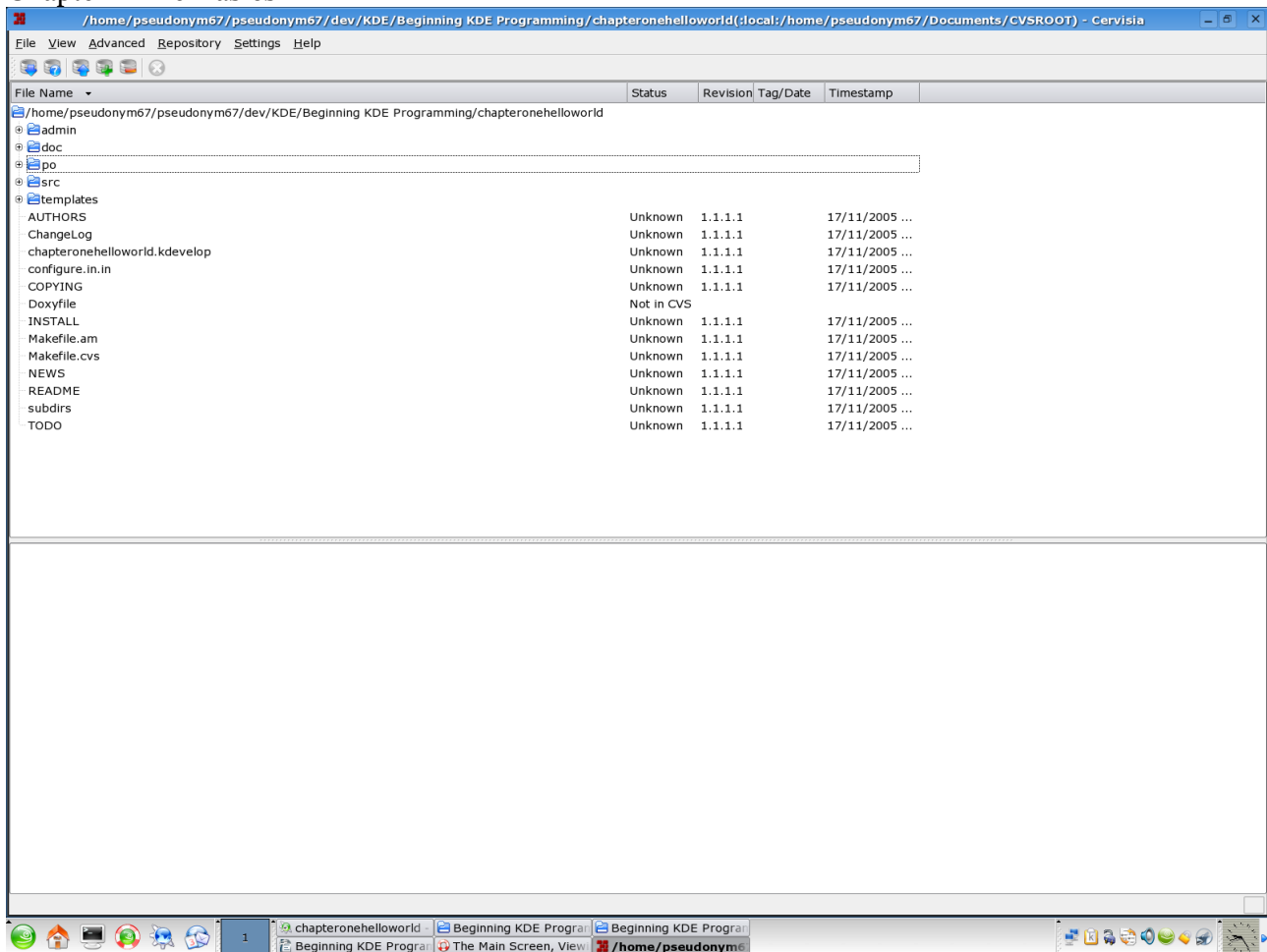
Chapter 1 The Basics



Viewing The Files

By far the quickest and easiest way to view the files in Cervisia is to open Konqueror and move to the folder where you told KDevelop to create the project, then right click on the folder and move down to Open With and Click Cervisia.

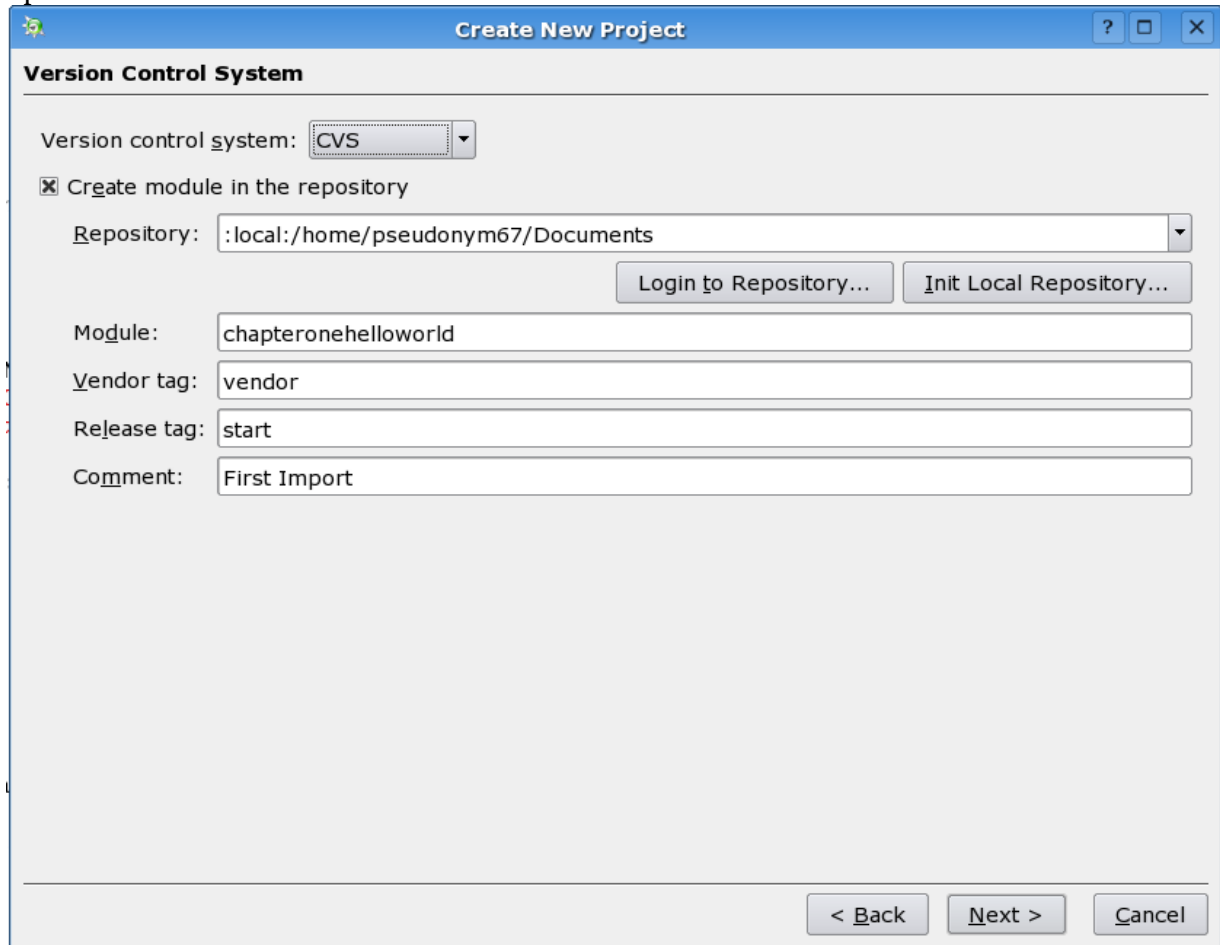
Chapter 1 The Basics



Cervisia comes with its own handbook that can be accessed via the help, we will return to Cervisia to look at some of the more common operations as we go but for now we'll get back to KDevelop and the project files.

Initialising The Repository

If we already have a repository set up and we want to use it for our new project we simply click on the “Init Local Repository” button and we will be asked for the location of our CVS Repository which in this case is `Home/pseudonym67/Documents`



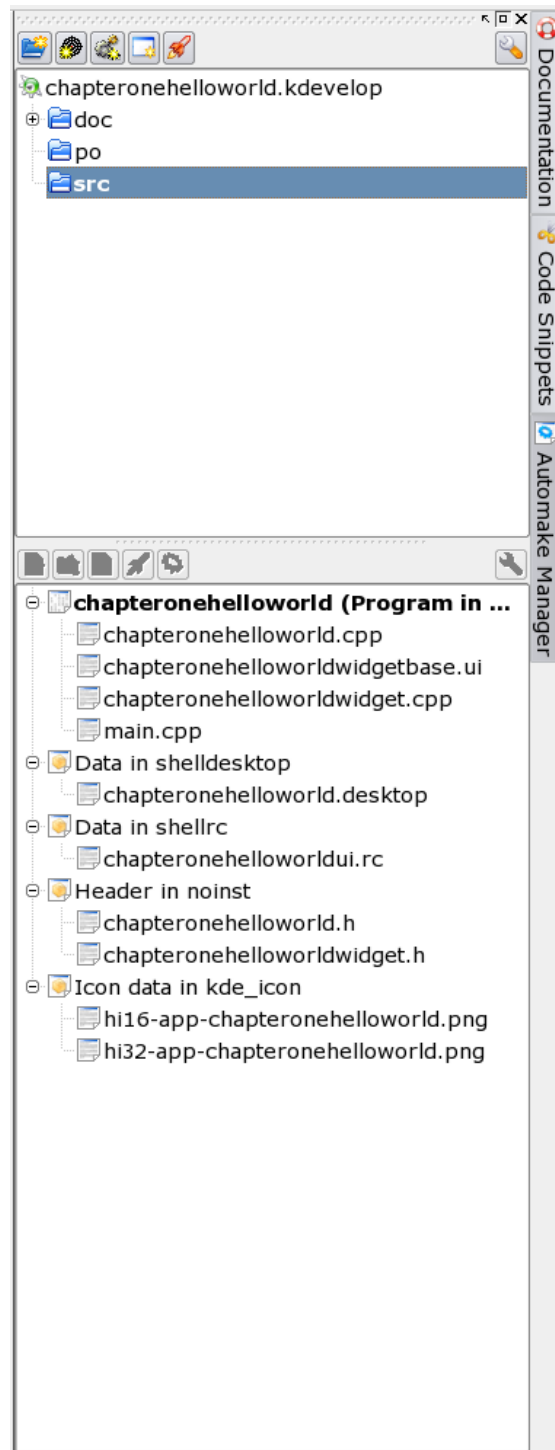
It should be noted that these days KDE developers use the Subversion or SVN version control and that there is a frontend for it on the Suse 10 disc called KDEVSVN. This won't affect the programming in this book in any way as we are concentrating on writing programs on KDE not on developing KDE itself.

KDE Project Files

In keeping with the windows type approach and the idea that we are using KDevelop as a development tool to develop our own programs, we will be concentrating on the files available to us through the Automake Manager to the right of the screen which is where the files important to our project are listed.

Upon creation of our first project we should be able to see this,

Chapter 1 The Basics



We get this view by clicking on the Automake Manager tab to the right of the screen and then on the src folder. Here we can see the files generated by KDevelop for our first project. The main source files for our project are contained in the chapteronehelloworld section which contains four files that we will look at in a moment, First of all we will look at the support files which are contained in the other sections.

The first section is the Data in shelldesktop. This contains the file chapteronehelloworld.desktop

[Desktop Entry]

```
Encoding=UTF-8
Name=ChapterOneHelloWorld
Exec=chapteronehelloworld
Icon=chapteronehelloworld
```

Chapter 1 The Basics

```
Type=Application
Comment=A simple KDE Application
Comment[br]=Ur meztant eeun evit KDE
Comment[ca]=Una aplicació KDE simple
Comment[da]=Et simpelt KDE program
Comment[de]=Eine einfache KDE-Anwendung
Comment[el]=Μία απλή εφαρμογή του KDE
Comment[es]=Una aplicación de KDE sencilla
Comment[et]=Lihtne KDE rakendus
Comment[eu]=KDE aplikazio simple bat
Comment[fr]=Une application simple pour KDE
Comment[ga]=Feidhmchlár Simplí KDE
Comment[hi]=एक सादा केडीई अनुप्रयोग
Comment[hu]=Egyszerű KDE-alkalmazás
Comment[is]=Einfalt KDE forrit
Comment[it]=Una semplice applicazione KDE
Comment[ja]=KDE
Comment[nb]=Et enkelt KDE-program
Comment[nl]=Een eenvoudige KDE-toepassing
Comment[pl]=Prosty program KDE
Comment[pt]=Uma aplicação simples do KDE
Comment[pt_BR]=Um simples Aplicativo do KDE
Comment[ru]=Простое приложение KDE
Comment[sl]=Preprost program za KDE
Comment[sr]=Једноставан KDE програм
Comment[sr@Latn]=Jednostavan KDE program
Comment[sv]=Ett enkelt KDE-program
Comment[ta]=w
Comment[tg]=Гузориши оддиKDE
Comment[tr]=Basit bir KDE Uygulaması
Comment[zh_CN]=KDE
```

As the name suggests and we can see from the file this is the configuration for the file that is used by the desktop when displaying short cuts/a link to the file or when it is running it. This includes information on the text encoding the name given to the application. The name of the executable for the application and the name of the icon for the application And the bulk of the file contains language specific information.

The next section is the Data in shellrc selection which also contains a single file, the chapteronehelloworld.rc file.

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartgui name="chapteronehelloworld" version="1">
<MenuBar>
  <Menu name="custom"><text>C&ustom</text>
    <Action name="custom_action" />
  </Menu>
</MenuBar>
</kpartgui>
```

This file is part of the KGUIClient system and works as a dynamic menu system. Basically the idea is that when you are writing a system where the menu options may vary depending on system state or file types then the menu options are stored here for all states that are catered for. As this is an advanced topic we will not be dealing with it in this project.

The next section, header in noinst, contains the main C++ header files for our project. These are the chapteronehelloworld.h file and the chapteronehelloworldwidget.h file.

The chapteronehelloworld.h file contains the following class declaration.

```
/**
 * @short Application Main Window
 * @author pseudonym67 <pseudonym67@hotmail.com>
 * @version 0.1
```

Chapter 1 The Basics

```
*/  
class ChapterOneHelloWorld : public QMainWindow  
{  
    Q_OBJECT  
public:  
    /**  
     * Default Constructor  
     */  
    ChapterOneHelloWorld();  
  
    /**  
     * Default Destructor  
     */  
    virtual ~ChapterOneHelloWorld();  
}
```

This declaration is for our main form window that we will look at soon. The only thing that this class contains is the constructor and destructor for our C++ class and the `Q_OBJECT` macro. The function of the `Q_OBJECT` macro is to include the code for Qt windowing system that all our gui programming is based on. This macro will automatically be included with any graphical element we create as it is needed to implement the signals and slots mechanism that allows us to make the user interface responsive through the use of button clicks etc. Putting it simply a slot is a normal function in all ways except for the fact that it can be fired by a signal as well as being called normally. And a signal is basically what it sounds like, in Windows terms it would be called a message that fires the function from the Windows message loop.

The `chapteronehelloworldwidget.h` file contains the following class declaration,

```
class ChapterOneHelloWorldWidget : public ChapterOneHelloWorldWidgetBase  
{  
    Q_OBJECT  
  
public:  
    ChapterOneHelloWorldWidget(QWidget* parent = 0, const char* name = 0, WFlags fl = 0 );  
    ~ChapterOneHelloWorldWidget();  
    /*$PUBLIC_FUNCTIONS$*/  
  
public slots:  
    /*$PUBLIC_SLOT$*/  
    virtual void button_clicked();  
  
protected:  
    /*$PROTECTED_FUNCTIONS$*/  
  
protected slots:  
    /*$PROTECTED_SLOT$*/  
  
};
```

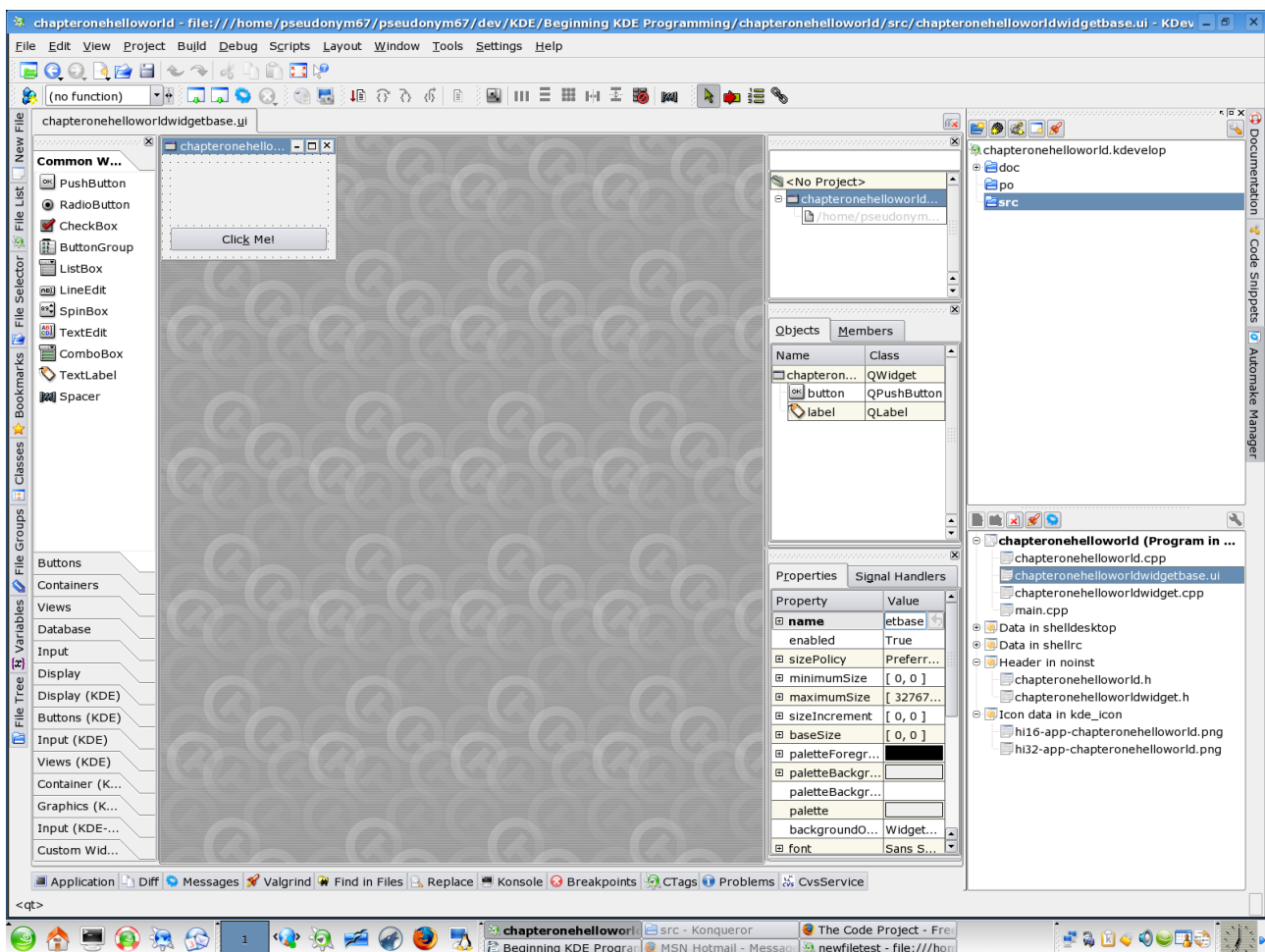
We will look at the relationship between the header files shortly in the next bit but for now we have another class declaration that is pretty standard except for the fact that we have the declaration for the implementation for a slot function so we can safely assume at this point that our hello world application will respond to a button click as long as the function is implemented in the corresponding cpp file. One final point about the header file is the comments that start with `/*$` are place holders for the KDevelop generated declarations so it is probably a good idea to leave them as they are.

The next section is the Icon data in `kde_icon`. This contains the icons for the application with both the 32 and 16 bit icons These should be straight forward but I haven't managed to find any applications that can open them.

Now we come to the `chapteronehelloworld` section which contains the main implementation files for our hello world project. This contains four files, three C++ files and a ui file. If we click on the

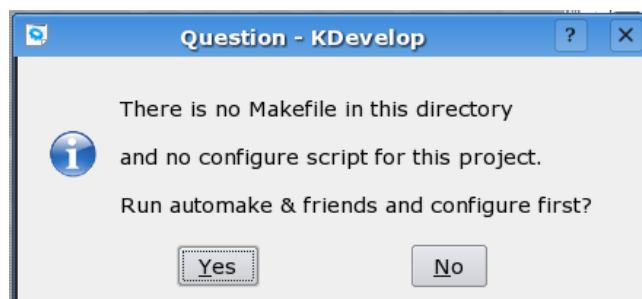
Chapter 1 The Basics

ui file we get this



What we are looking at now is a standard form editing environment, where we can drag and drop user interface components on to our forms and set up responses to the buttons being pressed or default data that will be entered into our components or widgets.

Here we can see the chapteronehelloworld form with a label and a button with click me written on it. If we build the application now, Build/Build Project. We will be told by the environment that there is no make file for the project.

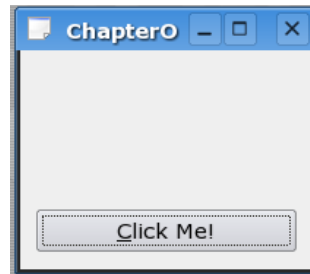


Click yes and the messages window will open. The only thing about the messages we are concerned with here is if the Automake is successful or not and if not why not. Pretty much the only reason why this should fail to work at this point which to be honest is not a valid reason at all but is one that we are going to have to live with is that when first starting this project I named the base folder for it as "Beginning KDE Programming". This is a big no no as it appears that spaces are bad. So if

Chapter 1 The Basics

you want your projects to build get rid of spaces in the names. It's probably best to make sure that there are no spaces in your project or your file names just to be on the safe side.

If we run the program through Debug/Start we get,



If we click the button we get,



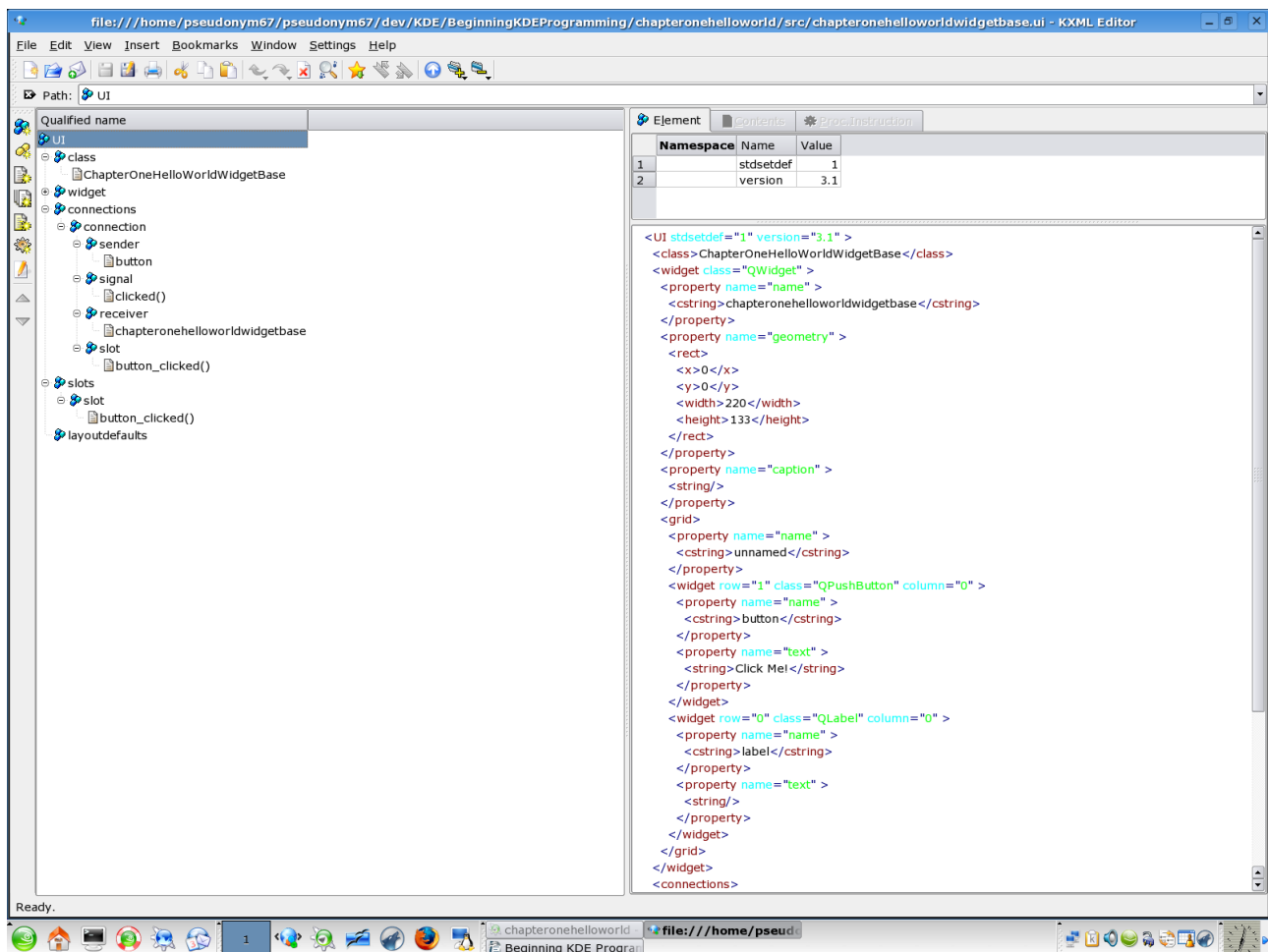
Not very exciting but it's a start. A start that can show us how everything else works so that we now what we are doing when we move on to more interesting programs. So the question is how does it work? If we look at our chapteronehelloworld.ui file as a utf-8 file. By right clicking on the ui file and going to Open With/Open as UTF-8, note the form must be closed or KDevelop wont open the file as a text file.

```
<!DOCTYPE UI><UI version="3.1" stdsetdef="1">
<class>ChapterOneHelloWorldWidgetBase</class>
<widget class="QWidget">
  <property name="name">
    <cstring>chapteronehelloworldwidgetbase</cstring>
  </property>
  <property name="geometry">
    <rect>
      <x>0</x>
      <y>0</y>
      <width>220</width>
      <height>133</height>
    </rect>
  </property>
  <property name="caption">
    <string></string>
  </property>
</grid>
  <property name="name">
    <cstring>unnamed</cstring>
  </property>
  <widget class="QPushButton" row="1" column="0">
    <property name="name">
      <cstring>button</cstring>
    </property>
    <property name="text">
      <string>Click Me!</string>
    </property>
  </widget>
</widget>
</UI>
```

Chapter 1 The Basics

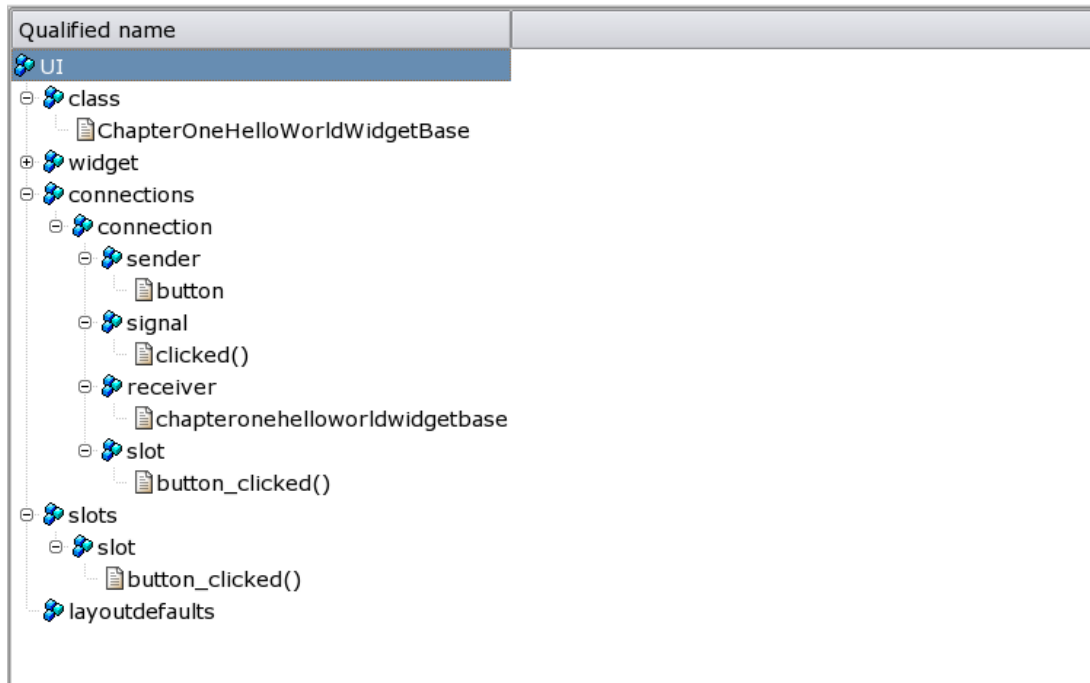
```
</property>
</widget>
<widget class="QLabel" row="0" column="0">
  <property name="name">
    <cstring>label</cstring>
  </property>
  <property name="text">
    <string></string>
  </property>
</widget>
</grid>
</widget>
<connections>
  <connection>
    <sender>button</sender>
    <signal>clicked()</signal>
    <receiver>chapteronehelloworldwidgetbase</receiver>
    <slot>button_clicked()</slot>
  </connection>
</connections>
<slots>
  <slot>button_clicked()</slot>
</slots>
<layoutdefaults spacing="6" margin="11"/>
</UI>
```

What we get is a not very pretty print out of a file that we could read if we were feeling really like making work for ourselves but seeing as we don't we'll just notice that it's an xml file and open it with KXML Editor instead as this can open both .ui and rc.ui files.



Chapter 1 The Basics

Now that we have the ui file in the KXML Editor we can get an easier view of what's going on. If we look at the objects side of the KXML Editor we see



The first thing we see in the file is a class object which refers to the class `ChapterOneHelloWorldWidgetBase` and while we may not have a file called `ChapterOneHelloWorldWidgetBase` we do have one called `ChapterOneHelloWorldWidget` which has both a header and a C++ implementation file. If we look at the header again we notice something new.

```
class ChapterOneHelloWorldWidget : public ChapterOneHelloWorldWidgetBase
{
    Q_OBJECT

public:
    ChapterOneHelloWorldWidget(QWidget* parent = 0, const char* name = 0, WFlags fl = 0 );
    ~ChapterOneHelloWorldWidget();
    /*$PUBLIC_FUNCTIONS$*/

public slots:
    /*$PUBLIC_SLOTS$*/
    virtual void button_clicked();

protected:
    /*$PROTECTED_FUNCTIONS$*/

protected slots:
    /*$PROTECTED_SLOTS$*/

};
```

Our `ChapterOneHelloWorldWidget` class inherits directly from the `ChapterOneHelloWorldWidgetBase` class which if we read the documentation we know inherits from `QWidget`. Although we are given a big clue when we look at the xml text/code in the KXML

Chapter 1 The Basics

Editor,

```
<UI stdsetdef="1" version="3.1" >
  <class>ChapterOneHelloWorldWidgetBase</class>
  <widget class="QWidget" >
    <property name="name" >
      <cstring>chapteronehelloworldwidgetbase</cstring>
    </property>
```

Basically for all intents and purposes the ChapterOneHelloWorldClassWidgetBase exists in name only it is a mechanism for associating the controls/widgets we have placed on the form with the class in our project and providing the inheritance from QWidget. In chapter two we will look at the base classes and where they come from.

You may also have noticed that we have a slot declaration in the header file that says that this class will handle what happens when the button is clicked. If we look at the implementation file we can see how it is implemented.

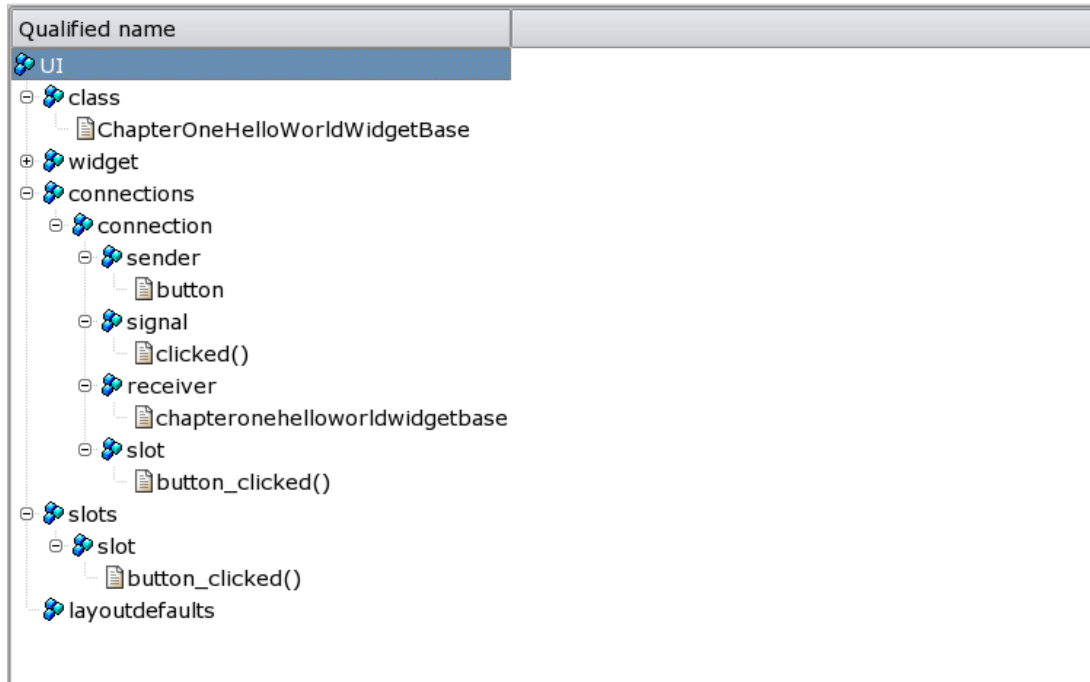
```
ChapterOneHelloWorldWidget::ChapterOneHelloWorldWidget(QWidget* parent, const char* name,
WFlags fl)
    : ChapterOneHelloWorldWidgetBase(parent,name,fl)
{}

ChapterOneHelloWorldWidget::~ChapterOneHelloWorldWidget()
{}

/*$SPECIALIZATION$*/
void ChapterOneHelloWorldWidget::button_clicked()
{
    if ( label->text().isEmpty() )
    {
        label->setText( "Hello World!" );
    }
    else
    {
        label->clear();
    }
}
```

The button_clicked function simply checks to see if the label has any text. If it does it clears the box if it doesn't then the text “Hello World” is added to the label. So if we look at the xml again we can see how this is done.

Chapter 1 The Basics



In the connections section we can see that there is a single connection set up in the ui file. The connection has a sender which is a button and a signal which is clicked. It should be noted here that a sender will send a signal that it has been told to send regardless of if any object has registered to receive the signal. In order for a signal to be received by an object it must first register to receive the signal by declaring a slot to receive the signal. If an object does not register itself as a receiver it will never know anything about the transmitted signal. We can see from the xml that the registered receiver is the chapteronehelloworldwidgetbase class which we are overriding with the ChapterOneHelloWorldWidget class and we are in fact handling the signal when the button is clicked and the signal is sent.

You may be wondering why the xml lists the slots separately in a slots section. The reason for this is that we could have another widget that wants to receive the same signal and this is perfectly acceptable as the only thing tying the receiver to the sender is the fact that it has registered to receive a signal, there is nothing to stop another object registering to receive the same signal.

We are now in a position where we have a user interface or ui file described in an xml document that is then subclassed and implemented by our ChapterOneHelloWorldWidget class that is derived from QWidget. This is effectively what in Windows programming terms would be considered a control as the Widget we have is not a main window and doesn't have a menu or anything that we have come to associate with the main part of a windowed environment. So let's have a look at the remaining files and see how our widget is implemented in the rest of the program.

To start with we have only one remaining header file and this is the ChapterOneHelloWorld.h file which looks like this,

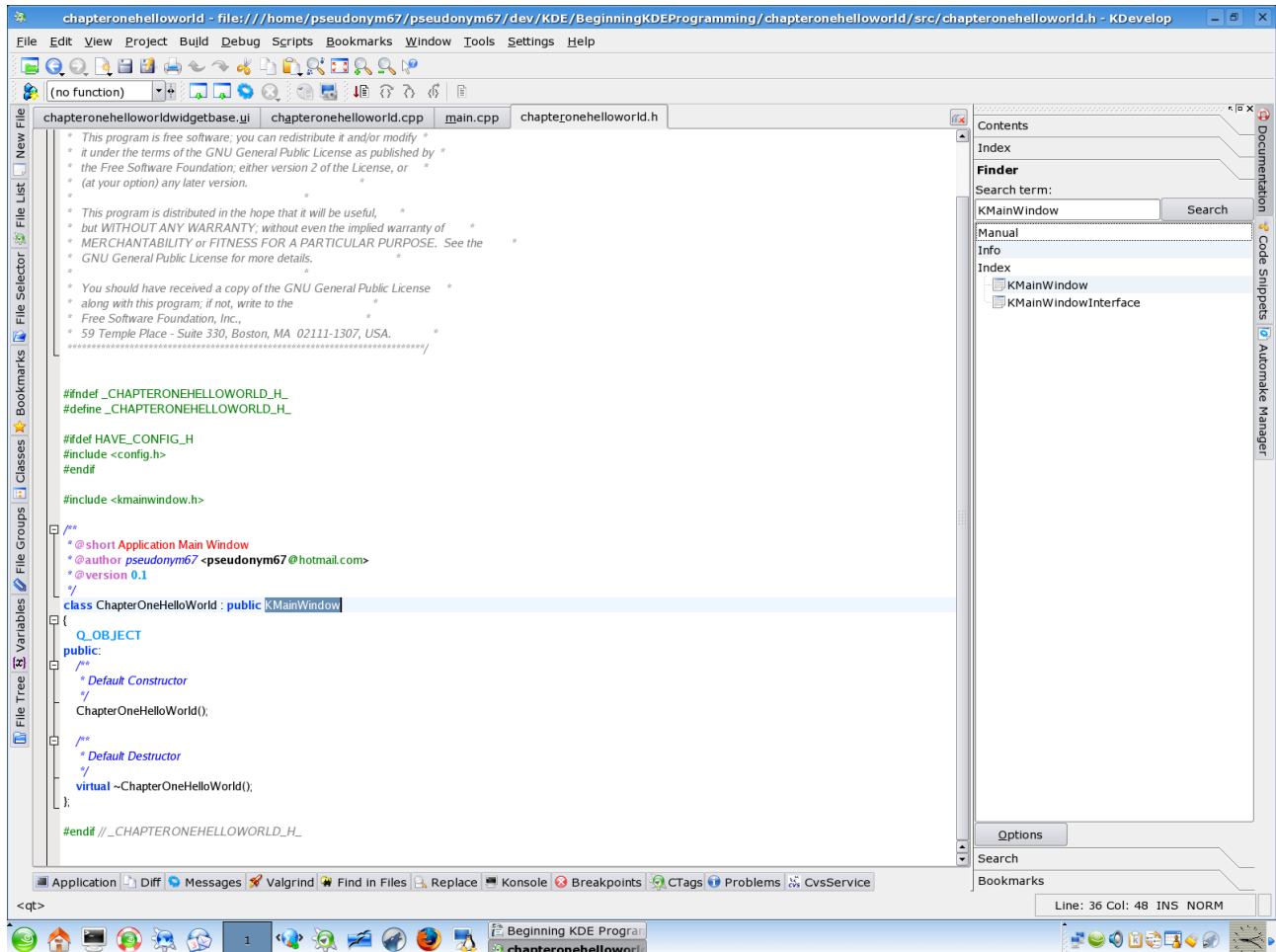
```
class ChapterOneHelloWorld : public QMainWindow
{
    Q_OBJECT
public:
    /**
     * Default Constructor
     */
    ChapterOneHelloWorld();

    /**
     * Default Destructor
     */
};
```

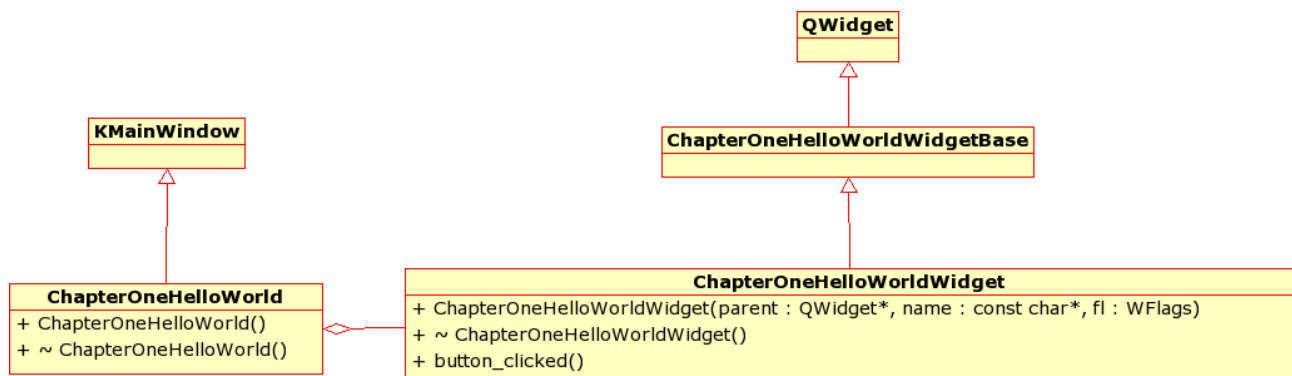
Chapter 1 The Basics

```
virtual ~ChapterOneHelloWorld();  
};
```

As you can see there is literally nothing here beyond the inheritance from KMainWindow. If you highlight KMainWindow and right click on it you will be able to look it up in the help and you will get,



Left clicking KMainWindow in the finder index will open the KMainWindow class reference page in the help. A quick look at the inheritance and the collaboration diagrams at the top of the page shows us what we need to know and that is that KMainWindow inherits from QMainWindow and as such is a top level window. This means that KMainWindow will control the implementation of menu's, status bar etc. and the ChapterOneHelloWorldWidget class will control what happens within the window view. Diagrammatically it looks something like this.



Chapter 1 The Basics

The ChapterOneHelloWorld class inherits from KMainWindow and is the main window for the application and contains the ChapterOneHelloWorldWidget object which controls everything that happens within the main window area of our application. We can see this clearly when we look at the implementation of the ChapterOneHelloWorld class,

```
ChapterOneHelloWorld::ChapterOneHelloWorld()
    : KMainWindow( 0, "ChapterOneHelloWorld" )
{
    setCentralWidget( new ChapterOneHelloWorldWidget( this ) );
}

ChapterOneHelloWorld::~ChapterOneHelloWorld()
{
}
```

The call to setCentralWidget in the constructor of the ChapterOneHelloWorld class sets a new object of ChapterOneHelloWorld application as the main widget for the form using the this pointer to tell the ChapterOneHelloWorldWidget class that the ChapterOneHelloWorld class object is its parent window.

The KDE Application Class

To a large extent we wont be focusing on the KApplication class a great deal but we will be relying on it to do it's job in every program we write and there is on important aspect of KApplication that we need to bear in mind. This is aspect is inherited from QApplication and it is the fact that QApplication works as a sort of garbage collector for widgets.

This means that all widget objects within the program should be defined as pointers and allocated on the the heap i.e.

```
WidgetDerivedClass *object;
object = new WidgetDerivedClass;
```

For the most part we will only be adding new widget objects through the form developer and the code will be automatically added for us, with KApplication (well its parent class QApplication) taking care of the clean up code, But it should be noted in case you find yourself hand coding widgets.

Summary

In this chapter we have walked through all but one of the generated files in order to show in a line by line fashion the way the files interact with each other providing an overview of how the widgets or windows of a KDE application interact and how a KDE application is composed of separate widgets.

Chapter 2 The KDE Application

In the last chapter we looked at a practical demonstration of how a KDevelop generated application works. In this Chapter we are still looking at the same application but whereas previously, in the coming chapters we looked at the parts of the application that are designed entirely by the developer in this chapter we will be looking more at how KDevelop enables us as programmers to do these designs in the first place.

The Application

To start with let's look at the one file in our application that we didn't look at in the last chapter. That is the `main.cpp` file and it looks like this,

```
static const char description[] =
    I18N_NOOP("A KDE KPart Application");

static const char version[] = "0.1";

static KCmdLineOptions options[] =
{
    // { "+[URL]", I18N_NOOP( "Document to open" ), 0 },
    KCmdLineLastOption
};

int main(int argc, char **argv)
{
    KAboutData about("chapteronehelloworld", I18N_NOOP("ChapterOneHelloWorld"), version,
description,
                        KAboutData::License_GPL, "(C) 2005 pseudonym67", 0, 0,
"pseudonym67@hotmail.com");
    about.addAuthor( "pseudonym67", 0, "pseudonym67@hotmail.com" );
    KCmdLineArgs::init(argc, argv, &about);
    KCmdLineArgs::addCmdLineOptions( options );
    KApplication app;
    ChapterOneHelloWorld *mainWin = 0;

    if (app.isRestored())
    {
        RESTORE(ChapterOneHelloWorld);
    }
    else
    {
        // no session.. just start up normally
        KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

        /// @todo do something with the command line args here

        mainWin = new ChapterOneHelloWorld();
        app.setMainWidget( mainWin );
        mainWin->show();

        args->clear();
    }

    // mainWin has WDestructiveClose flag by default, so it will delete itself.
    return app.exec();
}
```

The first thing that we should notice is the `I18N_NOOP` macro which is a location helper for the `KLocale` class. Its task is to mark a string for translation but not to actually translate it. The idea

Chapter 2 The KDE Application

being that when a translator, such as the QTranslator class, comes along then all the strings that need translating are already identified. For more on KDE translation see the [KDE Internationalisation](#) site.

The Command Line

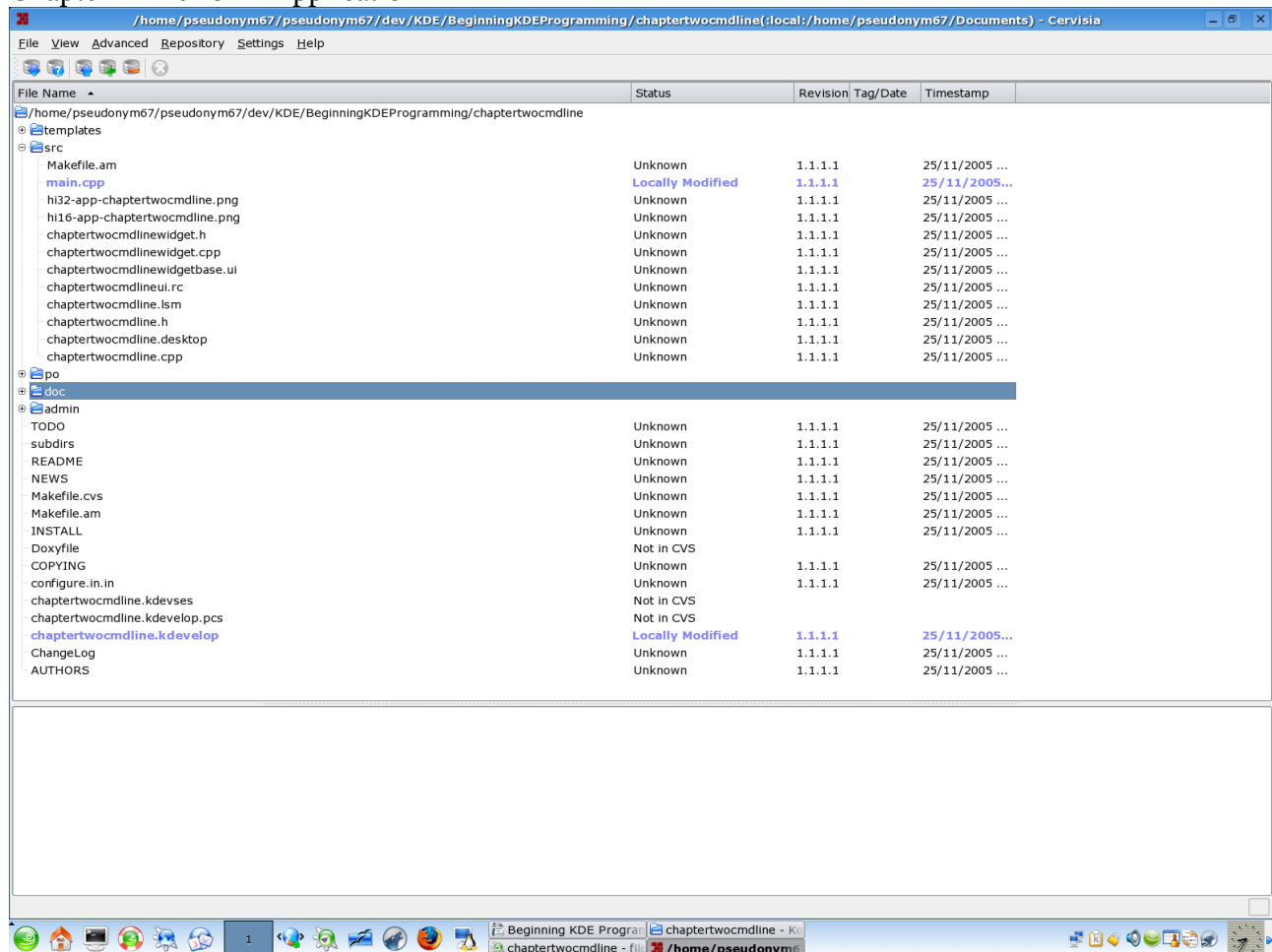
The command line is one of those things in writing computer programs where to a large extent you can completely ignore it. The only problem is that when you do need to use it it hardly ever turns out to be trivial so it is always better to know how to use it than not.

The way the command line is used in KDevelop is that an array is set up of KCmdLineOptions with each part taking three strings, these being the name of the option such as “displayAbout”, The second is a string that describes what the command line option does and the third is the default option. So if we wanted to set up a command line option that showed the about box before the application started then it would read

```
{ “displayAbout”, I18N_NOOP( “Show the about box on Start” ), 0 }
```

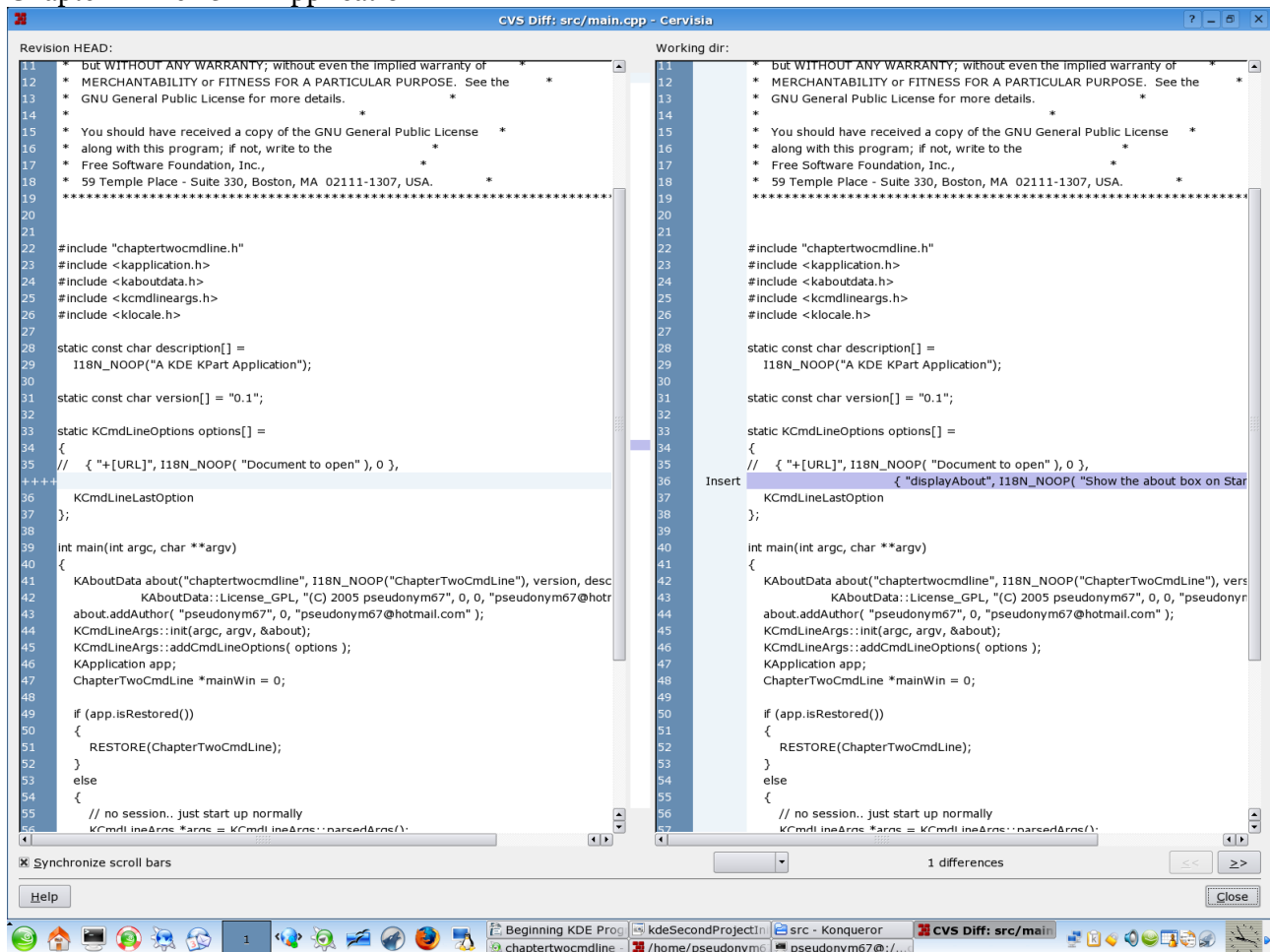
Note that the description string is marked as ready for translation. This should be as it is a string that the user of our application should see. So let's see what this looks like in a code example. Set up the C++/KDE/Simple Designer based KDE Application called ChapterTwoCmdLine as described in chapter one and don't forget that as we have set up the CVS previously we only have to “Init Local Repository” when we get to the CVS screen. Open the main.cpp file by clicking on it in the Automake Manager and add the line above to the KCmdLineOptions array. When this is done build the application for now using Build/Build Project or press F8 (default) and then open Konqueror and go to the chaptertwocmdline directory, right click the directory and select Open With/Cervisia and you'll get,

Chapter 2 The KDE Application



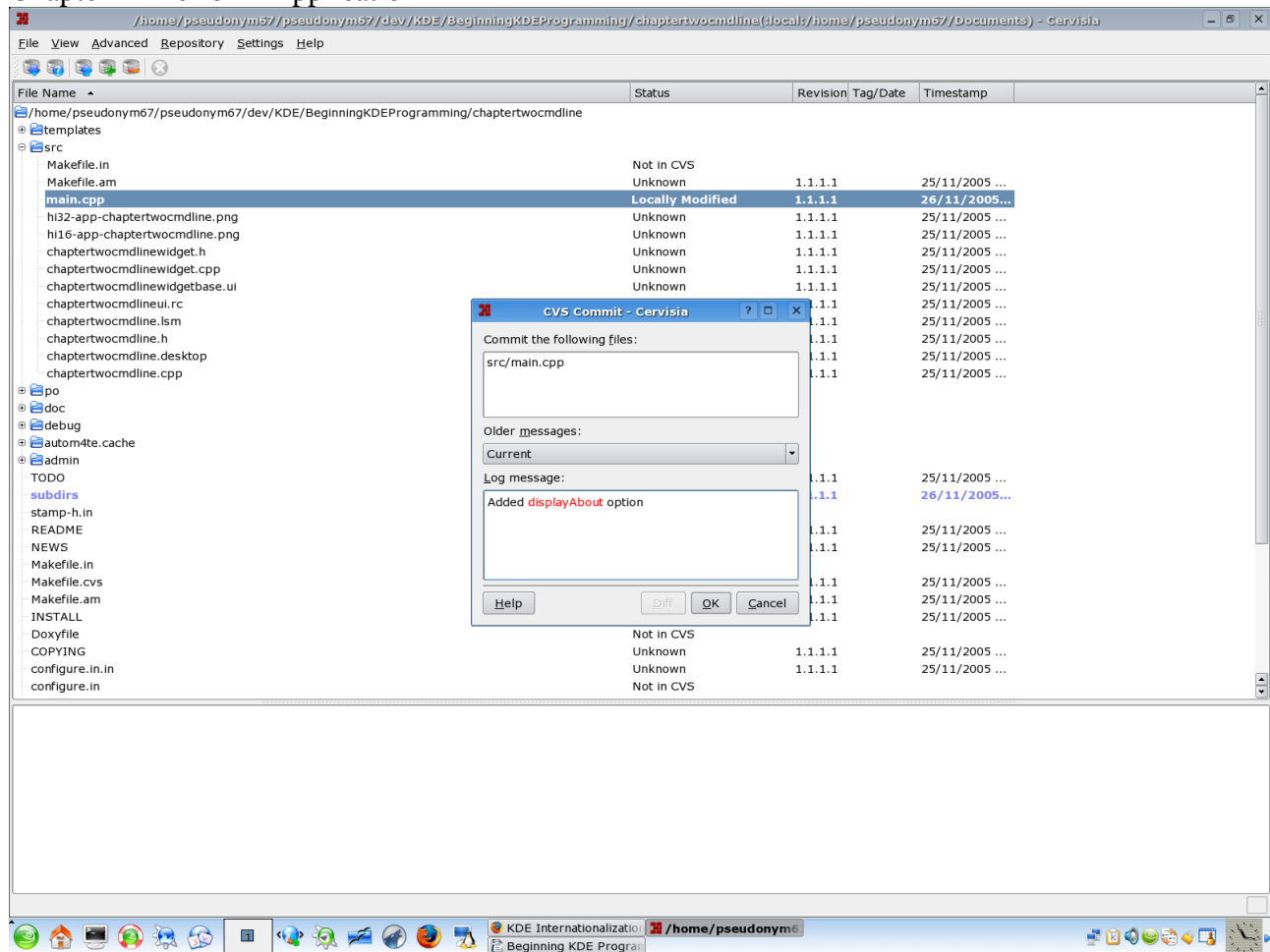
You can clearly see from the image that the main.cpp file has been changed. If we right click on the main.cpp file at this point we can see what the differences are using the difference viewer,

Chapter 2 The KDE Application



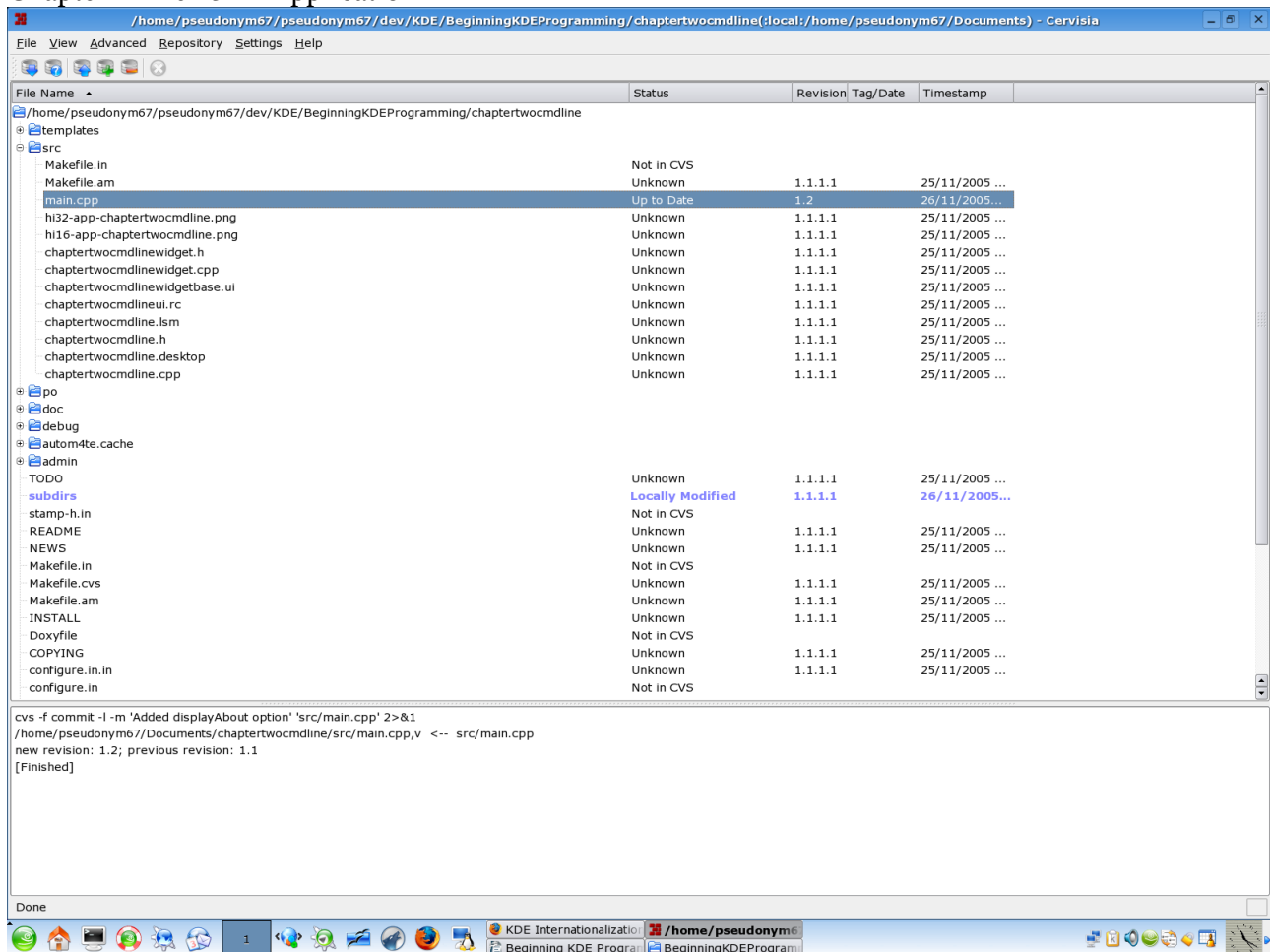
The difference viewer shows the image of the file that is currently in CVS in the left pane and the current version on the hard disk, remember that when doing large project development the copy in CVS is unlikely to be on your hard disk. If you are happy with the version on the you can right click on the file and commit the changes made to the CVS Repository. At this point you will be given to option to log the change.

Chapter 2 The KDE Application



You can add the message explaining what you have done and check use CVS to see what other people have changed in the file. Once committed Cervisia will show the updated file as,

Chapter 2 The KDE Application



You can see from the above that Cervisia has updated the file and incremented the revision number. Of course in a real development we would have tested the file properly first so we'd better have a look and see what we've changed. If you open the development folder for the project in Konqueror and go to `debug/src`, right click on the folder and select `Actions/Open Terminal Here`. This will open a Konsole in the source folder where our project has been built.

Chapter 2 The KDE Application

```
pseudonym67@:/...ocmdline/debug/src - Shell - Konsole
Session Edit View Bookmarks Settings Help
pseudonym67@dhcppc2:~/pseudonym67/dev/KDE/BeginningKDEProgramming/chaptertwocmdline/debug/src> ls -l
total 1785
-rwxr-xr-x 1 pseudonym67 users 851676 2005-11-26 09:10 chaptertwocmdline
-rw-r--r-- 1 pseudonym67 users 2642 2005-11-26 09:09 chaptertwocmdline.moc
-rw-r--r-- 1 pseudonym67 users 239888 2005-11-26 09:10 chaptertwocmdline.o
-rw-r--r-- 1 pseudonym67 users 2270 2005-11-26 09:10 chaptertwocmdlinewidgetbase.cpp
-rw-r--r-- 1 pseudonym67 users 1204 2005-11-26 09:09 chaptertwocmdlinewidgetbase.h
-rw-r--r-- 1 pseudonym67 users 3286 2005-11-26 09:10 chaptertwocmdlinewidgetbase.moc
-rw-r--r-- 1 pseudonym67 users 239324 2005-11-26 09:10 chaptertwocmdlinewidgetbase.o
-rw-r--r-- 1 pseudonym67 users 3153 2005-11-26 09:10 chaptertwocmdlinewidget.moc
-rw-r--r-- 1 pseudonym67 users 229976 2005-11-26 09:10 chaptertwocmdlinewidget.o
-rw-r--r-- 1 pseudonym67 users 203688 2005-11-26 09:09 main.o
-rw-r--r-- 1 pseudonym67 users 29727 2005-11-26 09:09 Makefile
pseudonym67@dhcppc2:~/pseudonym67/dev/KDE/BeginningKDEProgramming/chaptertwocmdline/debug/src> chaptertwocmdline -help
Usage: chaptertwocmdline [Qt-options] [KDE-options] [options]

A KDE KPart Application

Generic options:
--help                Show help about options
--help-qt             Show Qt specific options
--help-kde            Show KDE specific options
--help-all           Show all options
--author              Show author information
--v, --version         Show version information
--license             Show license information
--                    End of options

Options:
--displayAbout        Show the about box on Start
pseudonym67@dhcppc2:~/pseudonym67/dev/KDE/BeginningKDEProgramming/chaptertwocmdline/debug/src> █
```

If we look at the files in the folder with the Konsole using `ls -l` we can see the `chaptertwocmdline`, highlighted above in green along with the rest of the files. Also here we can see the `chaptertwocmdlinewidgetbase` class and header files. You may remember that in chapter one I said that for all intents and purposes that these classes exist in name only as far as we are concerned, well here they are. They are generated by the development environment at build time from the ui file and then our file that ends with the word `widget` inherits from them. We shall take a quick look at them shortly but for now we are concentrating on the output in the Konsole.

About half way down the screen pictured you can see the command “`chaptertwocmdline -help`” entered into the Konsole and the resulting print out. At the top there is a brief usage instruction and then the phrase “A KDE KPart Application” this is from the description mentioned earlier that is tagged with the `I18N_NOOP` macro that marks the string for translation. The reason being that at times the strings marked by `I18N_NOOP` are going to be seen by the users of the program and it's obviously a better user experience if they can understand what it says.

It should be noted that the KDE Developer FAQ at <http://developer.kde.org/documentation/other/developer-faq.html#q2.11.2> states that once the KDE application is initialised you should use the macro `i18n()` to highlight translatable strings within the program code.

As you can see from the above print out the options are split into two parts from a standard “`-help`” command line argument with the general options coming first and then the options entered within the program follow. If you want a complete list of options use the command “`chaptertwocmdline -help-all`”.

I'll leave you to play around with the available options to see what they do and move on to looking at just what we are going to do with our option now that we have one. In a normal program a command line option is something that is used to influence how the program runs and not something that shows a dialog box as I am in this example.

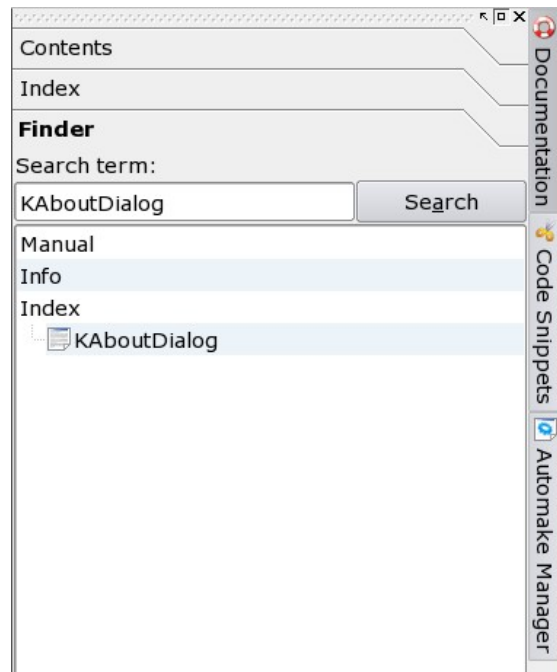
First though here is the code that we are using.

Chapter 2 The KDE Application

```
if( args->isSet( "displayAbout" ) == true )
{
    KAboutDialog *aboutDlg = new KAboutDialog( KAboutDialog::AbtProduct, "Display
        Demo", KDialogBase::Ok, KDialogBase::Ok, 0, "Display About", true );
    aboutDlg->setProduct( "Display About Demo", "1.0", "pseudonym67", "2005" );
    aboutDlg->show();
}
```

The code itself tests to see if the displayAbout option is set, remember the default is set to false or 0 in the third part of the options string. Basically the three lines of code above create a KAboutDialog object and then set the details to be displayed before showing it. Most of the options for the KAboutDialog require a layout type in the first parameter that requires an image to be set but for now we are going to go for a straight text version by using KAboutDialog::AbtProduct

To see the options available for KAboutDialog simply double click on the class name so that it is highlighted then right click on it and select “Find Documentation: KAboutDialog” and you should get this in your right panel,



Clicking on the KAboutDialog will take you to the html help page for the class, although to see the options to be set for the KAboutDialog properly you will have to look at the html version of the C++ header file a link for which is provided in the html help.

We set the LayoutType according to the enum defined in KAboutDialog.h, They can be viewed in the class reference on KAboutDialog under the Public Types section, you can replace the AbtProduct with any of the above to see what effect they have on the about box.

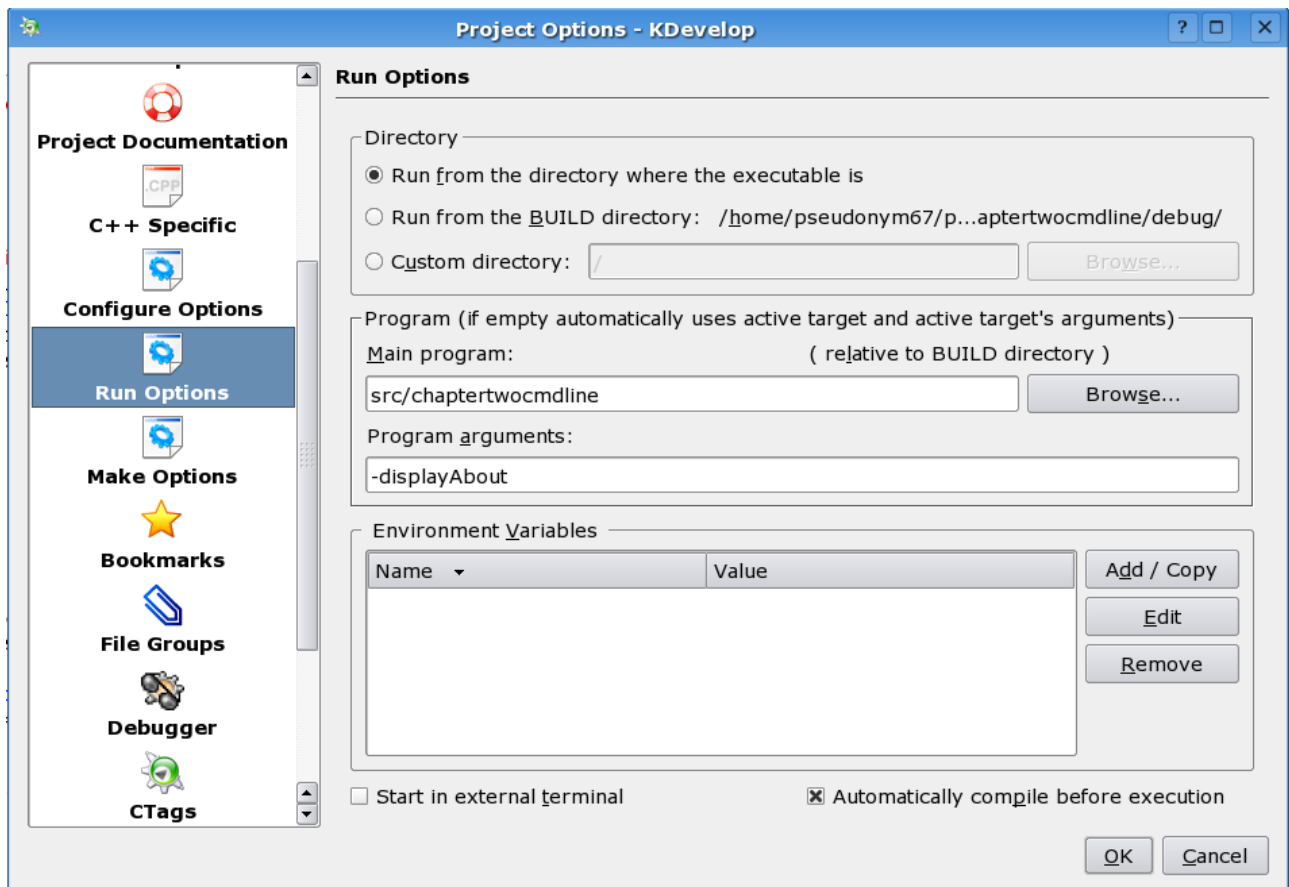
The title is a simple string, the required buttons takes a KDialogBase ButtonCode enumeration as defined in the KDialogBase, and the button to be highlighted takes the same type. The parent window or 0 is next, followed by the name for the dialog and if we require the dialog to be modal or not. We set the product information the we want to display on the dialog and then call show.

If we run the program now though all we will get is the standard default hello world project as KDevelop still hasn't been told that we wish to pass a parameter to the program. There are two things that we need to do to get the program to run properly from the environment. We could go to the command line and type in what we need but that kind of defeats the point of having a

Chapter 2 The KDE Application

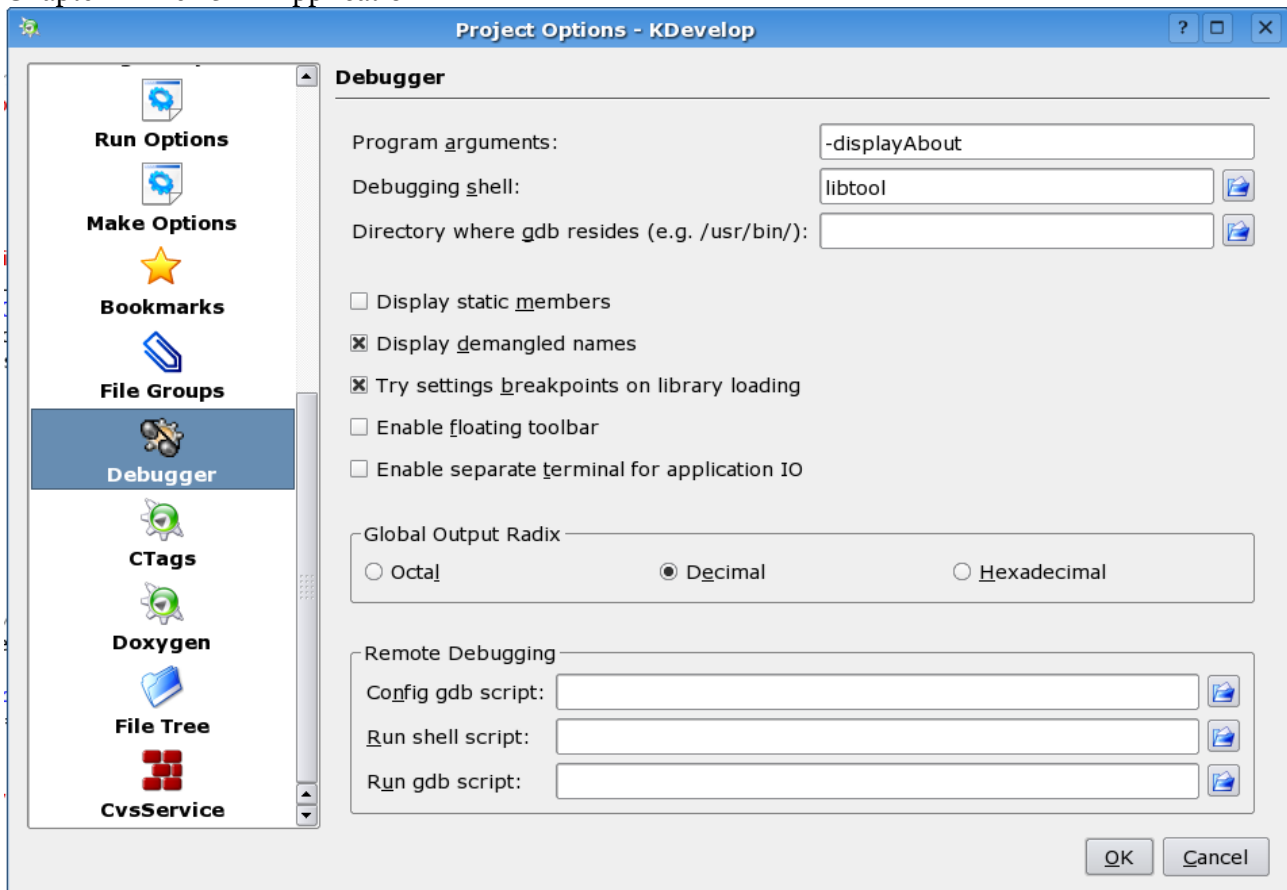
development environment so we'll stick to doing things with KDevelop.

First click on Project/Project Options and then Run Options in the dialog and fill in the Program Arguments like so



Then select the Debugger option and fill in the Program Arguments like so,

Chapter 2 The KDE Application



Now if we run the program from either Debug/Start or Build/Execute Program the program will run with the displayAbout command line argument set.

When the program starts you should now get



displayed with the start of the program and because the dialog has been declared modal you will not be able to use the hello world part of the application until the OK button is clicked and the dialog is removed.

Code Snippets

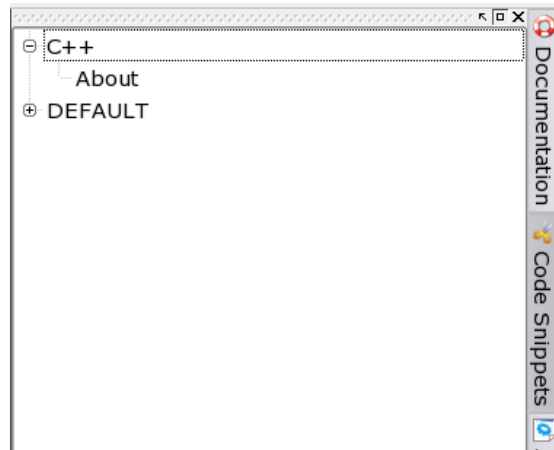
There is one item on the right hand side that we haven't used yet and that is the Code Snippets section. Now that we have the About Box set up the way that we want it we can save the code in the Code Snippets section so that in future projects we can use it as a guide for adding the About Box.

Click on the Code Snippets side bar to open it up and there will be just the default option. To add a new group such as a C++ group for holding C++ code right click in the window and select Add

Chapter 2 The KDE Application

Group. This will give you an option to name the group and then KDevelop will create it.

Select the code you wish to add to the snippet in the code window and copy it with CTRL-C and then right click on the C++ group and select add item. Name the item About and then paste the snippet into available window.



This means that when we want to use the About Box again we have the set up code saved in the snippet box and don't have to go searching through code files trying to find it.

A Better Demonstration

Actually the simplest way to improve the demo is to use a standard KMessageBox such as

```
KMessageBox::information( 0, "Display About Demo\nVersion 1.0\nCopyright  
2005\npseudonym67" );
```

which would give you



But we won't do that because it would be far too easy. So we'll start another project and do it the interesting way instead.

Create a new Simple Designer Based KDE Application called ChapterTwoAboutDemo as we did in Chapter One.

Chapter 2 The KDE Application

Coding Styles

There is a lot said about coding styles and most of it is nonsense. No matter what language you program or what environment you program in there will be people that insist that all classes start with a capital or not and all variables start with a capital or not and all enums should be labelled in capitals and have a variable named for easier access. i.e. enum ENUMTYPE{ zero, one, two }enumType; or EnumType; or even eEnumType; or possibly eenumType; and any other variation you can think of or as in the case with the KDE API not.

The thing is that everyone selects or adjusts to the style they are most used to so the way I always work it is this. If I am programming in a new environment then I use the class labels that are used in that environment and if function variables in the environment start lower case then my function variables start lower case. For anything that is not directly accessible I'll use whatever style I am most comfortable with not because I'm being obnoxious and not even because I'm trying to make code unmaintainable, in fact if anything even I sometimes think that my coding style is overly defensive and overly descriptive where some variable names get so long I get bored with typing them.

The reason I do this is simply because I am usually concentrating on what I want the code to do at the time and as a result of this I usually just type the code in the manner I am most used to, which just happens to be the way I learned to write code in the early nineties. So try to stick within the rules where other people or classes are going to access your code but other than that it's just a religious war that has no bearing on if the code works or not.

Adding The Demo Code

Anyway mini rant over back to the demo. First of all add the following to the

First of all we edit the main.cpp file like so

```
static KCmdLineOptions options[] =
{
    // { "+[URL]", I18N_NOOP( "Document to open" ), 0 },
    { "displayAbout", I18N_NOOP( "Show the about box on Start" ), 0 },
    KCmdLineLastOption
};

int main(int argc, char **argv)
{
    KAboutData about("chaptertwoaboutdemo", I18N_NOOP("ChapterTwoAboutDemo"), version,
description,
                        KAboutData::License_GPL, "(C) 2005 pseudonym67", 0, 0,
"pseudonym67@hotmail.com");
    about.addAuthor( "pseudonym67", 0, "pseudonym67@hotmail.com" );
    KCmdLineArgs::init(argc, argv, &about);
    KCmdLineArgs::addCmdLineOptions( options );
    KApplication app;
    ChapterTwoAboutDemo *mainWin = 0;

    if (app.isRestored())
    {
        RESTORE(ChapterTwoAboutDemo);
    }
    else
    {
        // no session.. just start up normally
        KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

        /// @todo do something with the command line args here
        /*
            One way to do it
            if( args->isSet( "displayAbout" ) == true )
            {
```

Chapter 2 The KDE Application

```
        mainWin = new ChapterTwoAboutDemo( true );
    }
    else
        mainWin = new ChapterTwoAboutDemo( false );

*/

    mainWin = new ChapterTwoAboutDemo();
    mainWin->setShowAboutBox( args->isSet( "displayAbout" ) );
    app.setMainWidget( mainWin );
    mainWin->show();

    args->clear();
}
```

I've added the option `displayAbout` to the command line options and then implemented a couple of the possible ways in which you could use it. Although I have to admit that there aren't many occasions on which I've wanted to start a program with a dialog box there have been one or two so the demo does manage to avoid the being completely useless category this time.

The `KAboutData` and the `KCmdLineArgs` code remains untouched here and there are two options given for how to implement the display of the `aboutBox`. The first is through a custom constructor and the second is done through the setting of a flag variable within the `ChapterTwoAboutBox` class.

The custom constructor is simply the line

```
ChapterTwoAboutDemo( bool showAbout );
```

In the header file while the flag variable is set using,

```
class ChapterTwoAboutDemo : public KMainWindow
{
    Q_OBJECT

private:
    bool bShowAboutBox;

    /// variable access

public:
    bool showAboutBox()
    {
        return bShowAboutBox;
    }

    void setShowAboutBox( bool show )
    {
        bShowAboutBox = show;
    }
}
```

The point here is just to set up a quick and easy flag to check if the command line parameter has been passed from within the code.

The custom constructor is then defined as

```
ChapterTwoAboutDemo::ChapterTwoAboutDemo( bool showAbout ) : KMainWindow( 0,
"ChapterTwoAboutDemo" )
{
    setShowAboutBox( showAbout );
    setCentralWidget( new ChapterTwoAboutDemoWidget( this ) );
}
```


Chapter 2 The KDE Application

in `chaptertwoaboutdemo.cpp`. The next thing to do is add About dialog itself to the `ChapterTwoAboutDemo` header file

```
private:
    bool bShowAboutBox;
    KAboutDialog *aboutDlg;
```

and then initialise it in the constructors.

```
ChapterTwoAboutDemo::ChapterTwoAboutDemo( bool showAbout ) : KMainWindow( 0,
"ChapterTwoAboutDemo" )
{
    setShowAboutBox( showAbout );
    setCentralWidget( new ChapterTwoAboutDemoWidget( this ) );
    aboutDlg = new KAboutDialog( KAboutDialog::AbtProduct, "Display Demo",
KDialogBase::Ok, KDialogBase::Ok, 0, "Display About", true );
}
```

with the above being the custom constructor that we have added taking the boolean `showAbout` variable.

All we need now is a mechanism for showing the dialog box. We could have added the dialog to the `ChapterTwoAboutDemoWidget` class but this would break the design of our application as the class derived from `KMainWindow` should handle code aspects that relate to the main application and the widget that deals with what goes on in the window area should handle only things that pertain to the window area. Ideally we would handle the box through a menu item that allows you to select to view the about box but we haven't got to that part yet, so what we do is override the `show` method from `KMainWindow`.

```
/// override show
virtual void show();
```

If you look this up in the `KMainWindow` header file you will note that it says that `KMainWindow` itself overrides the `show` method from `QMainWindow` and that in KDE 4 this could be removed. It won't make a major difference to the code here though as if the override in `KMainWindow` is removed the code will just need to call `QMainWindow::show()` instead on `KMainWindow show()`

The whole overridden function looks like this,

```
void ChapterTwoAboutDemo::show()
{
    KMainWindow::show();

    /// show the about box if the arg was passed

    if( getShowAboutBox() == true )
    {
        aboutDlg->setProduct( "Display About Demo", "1.0", "pseudonym67", "2005" );
        aboutDlg->show();
    }
}
```

Debugging Code

If we try to run the code now we get the standard hello world application that so we'll debug the application to see what is going on. Firstly add a breakpoint to `main.cpp`

Chapter 2 The KDE Application



We add the breakpoint here so that we can see if the call to `setShowAboutBox` is being passed the correct argument. A breakpoint is set in KDevelop by left clicking on the left side of the screen where you can see the little grey circle, at the start of the red line in the image above. By default left clicking here will leave a bookmark which gives a paper clip image. If you choose to you can right click here and set breakpoint to be the default.

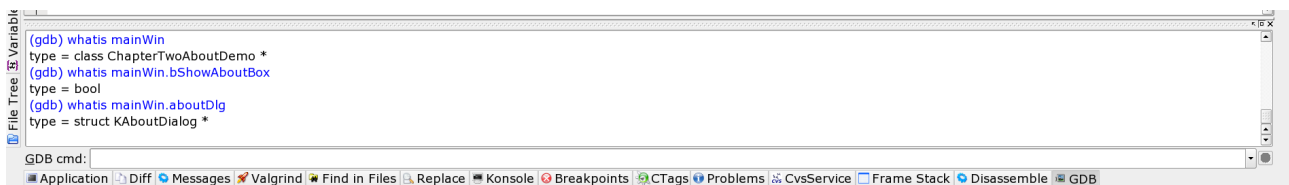
We'll also add a breakpoint to the `ChapterTwoAboutDemo` code,



which shows the point in the `show` function where we test if we want to display the about box or not. If we start the program through the menu `Debug/Start` then the code execution will stop in the `main.cpp` file



The line where we set our breakpoint is highlighted in green and at this point we can examine where the program is up to. There are a number of ways to examine the current program state and the usefulness of these really depends on the level of expertise of the person doing the debugging. The first thing to notice is the menu bar at the bottom of the screen



The last three menu items are the `Frame Stack` which will give a list of the function calls made within the program in order that they were made, so you can trace the way that the code has moved through the program and make sure that it hasn't done anything unexpected. The `Disassemble` window which gives the assembly code for the project and to be honest if you understand the assembly code you don't need to read this. And the `GDB` window. The GNU Project Debugger (GDB) attempts to tell you what is going on in the program at any point in time so if we look at the window we see.

Chapter 2 The KDE Application

```
Breakpoint 3, main (argc=1, argv=0x7ffffff19ae8) at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70:2758: beg:0x4076b2
(gdb) info thread
* 1 Thread 46912550040128 (LWP 7668)  main (argc=1, argv=0x7ffffff19ae8) at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
(gdb) backtrace
#0  main (argc=1, argv=0x7ffffff19ae8) at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
(gdb) info breakpoints
Num Type      Disp Enb Address      What
1 breakpoint keep n   0x0000000000407f23 in ChapterTwoAboutDemo::show()
at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/chaptertwoaboutdemo.cpp:55
2 breakpoint keep n   0x00000000004076b2 in main at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
3 breakpoint keep y   0x00000000004076b2 in main at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
breakpoint already hit 1 time
4 breakpoint keep y   0x0000000000407f23 in ChapterTwoAboutDemo::show()
at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/chaptertwoaboutdemo.cpp:55
(gdb) print *args
$1 = {options = 0x50cbc0, name = 0x0, id = 0x0, parsedOptionList = 0x0, parsedArgList = 0x0, isQt = false, d = 0x0}
(gdb) print *args
$2 = {options = 0x50cbc0, name = 0x0, id = 0x0, parsedOptionList = 0x0, parsedArgList = 0x0, isQt = false, d = 0x0}
(gdb) frame 0
#0  main (argc=1, argv=0x7ffffff19ae8) at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70:2758: beg:0x4076b2
GDB cmd:
Application Diff Messages Valgrind Find in Files Replace Konsole Breakpoints CTags Problems CvsService Frame Stack Disassemble GDB
```

Which is the start of our program, showing the the placing of the breakpoints, listing four of them here as I've been playing around with things to see how they work and if the breakpoints are on or off. Below that there is a print out of the program arguments and below that there is the start of our main function that has an arg count of one which is the call to the program itself. If we double left click on the args variable on the line that the program is stopped at then right click and select “watch args”, The GDB will output a couple more lines

```
(gdb) frame 0
#0  main (argc=1, argv=0x7ffffff19ae8) at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70
/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/main.cpp:70:2758: beg:0x4076b2
(gdb) whatis args
type = KCmdLineArgs *
(gdb) print args
$3 = (KCmdLineArgs *) 0x519850
GDB cmd:
Application Diff Messages Valgrind Find in Files Replace Konsole Breakpoints CTags Problems CvsService Frame Stack Disassemble GDB
```

telling us what it knows about the args variable. If we click on the Variables menu on the left hand side we get,

Chapter 2 The KDE Application

The screenshot displays the KDevelop IDE interface. On the left, the 'Watch' window shows a tree of variables. The 'args' variable is highlighted. The main editor shows the source code for 'main.cpp'. The code includes a conditional check for 'displayAbout' and a call to 'mainWin->setShowAboutBox()'. The bottom status bar shows 'GDB cmd:' and a list of tools like Application, Diff, Messages, Valgrind, Find in Files, Replace, Konsole, and B.

```
if (app.isRestored())
{
    RESTORE (ChapterTwoAboutDemo);
}
else
{
    // no session.. just start up normally
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    /// @todo do something with the command line args here
    /*
    One way to do it
    if( args->isSet( "displayAbout" ) == true )
    {
        mainWin = new ChapterTwoAboutDemo( true );
    }
    else
        mainWin = new ChapterTwoAboutDemo( false );
    */

    mainWin = new ChapterTwoAboutDemo();
    mainWin->setShowAboutBox( args->isSet( "displayAbout" ) );
    app.setMainWidget( mainWin );
    mainWin->show();

    args->clear();
}

// mainWin has WDestructiveClose flag by default, so it will del
return app.exec();
}
```

(gdb) print *args
\$2 = {options = 0x50cbc0, name = 0x0, id = 0x0, parsedOptionList = 0x0, parsedArgL
, d = 0x0}

(gdb) frame 0
#0 main (argc=1, argv=0x7ffff19ae8) at /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/sr

Expression to watch: Add

GDB cmd:

Application Diff Messages Valgrind Find in Files Replace Konsole B

We can see the args variable in the window on the left of the screen so we can monitor any variables within the program whenever we choose. Of course there is the slight technical point at the moment that it is not going to tell us anything because there is only one program argument and that is the name of the program itself. So stop debugging the program (Debug/Stop) and go to Project/Project Options/Debugger/Program arguments and add “-displayAbout” like last time. If you then hit Debug/Start when the program stops we will have some data to see.

Chapter 2 The KDE Application

The screenshot shows a debugger interface with a 'Watch' list on the left and a code editor on the right. The 'Watch' list contains a variable 'args' with a sub-entry 'options' which has a value of '0x409720 "displayAbout"'. The code editor shows the 'main' function of 'chaptertwoaboutdemowidgetbase.ui'. The code includes initialization of 'KAboutData', 'KCmdLineArgs', and 'KApplication', followed by a check for a restored session. If not restored, it parses command-line arguments and checks for the 'displayAbout' flag. If the flag is set, it creates a 'ChapterTwoAboutDemo' window and calls 'setShowAboutBox'. The code ends with 'app.exec()'.

```
int main(int argc, char **argv)
{
    KAboutData about("chaptertwoaboutdemo", I18N_NOOP("ChapterTwoAboutDemo"),
        KAboutData::License_GPL, "(C) 2005 pseudonym67",
        about.addAuthor("pseudonym67", 0, "pseudonym67@hotmail.com"));
    KCmdLineArgs::init(argc, argv, &about);
    KCmdLineArgs::addCmdLineOptions(options);
    KApplication app;
    ChapterTwoAboutDemo *mainWin;

    if (app.isRestored())
    {
        RESTORE(ChapterTwoAboutDemo);
    }
    else
    {
        // no session.. just start up normally
        KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

        /// @todo do something with the command line args here
        /*
         One way to do it
         if( args->isSet("displayAbout") == true )
         {
             mainWin = new ChapterTwoAboutDemo( true );
         }
         else
             mainWin = new ChapterTwoAboutDemo( false );
         */

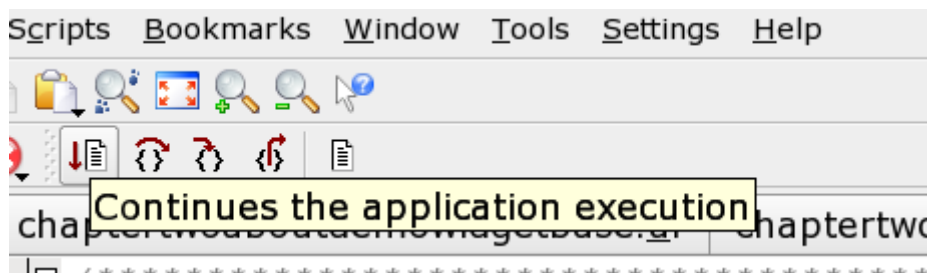
        mainWin = new ChapterTwoAboutDemo();
        mainWin->setShowAboutBox( args->isSet("displayAbout") );
        app.setMainWidget( mainWin );
        mainWin->show();

        args->clear();
    }

    // mainWin has WDestructiveClose flag by default, so it will delete itself
    return app.exec();
}
```

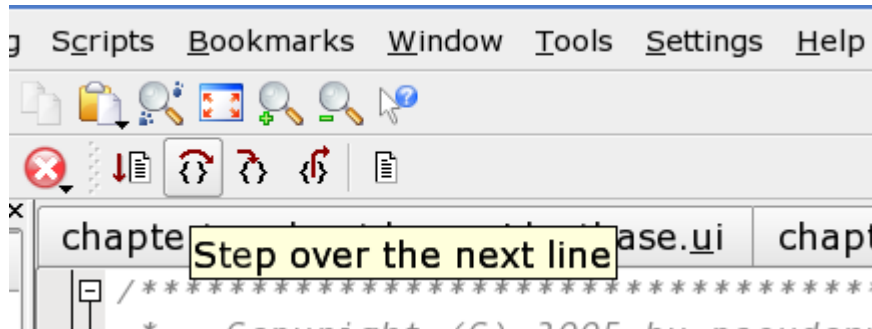
Which shows us the name of one argument is called displayAbout. This means that we already know the outcome of the test for the breakpoint here which is what will the args variable be passing to setShowAboutBox function.

We can now move onto our other breakpoint and we do this by using the debugging toolbar.

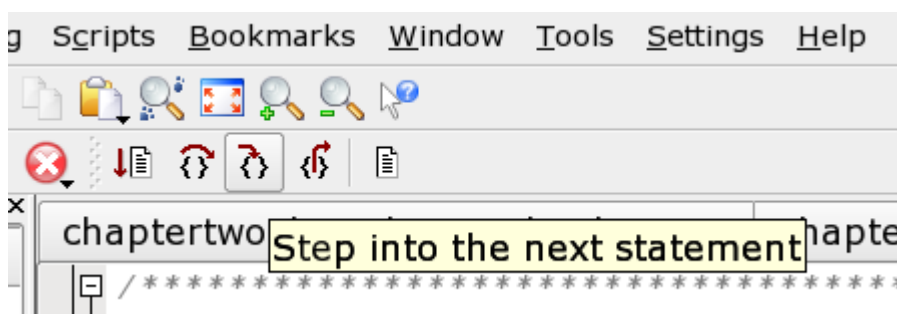


Chapter 2 The KDE Application

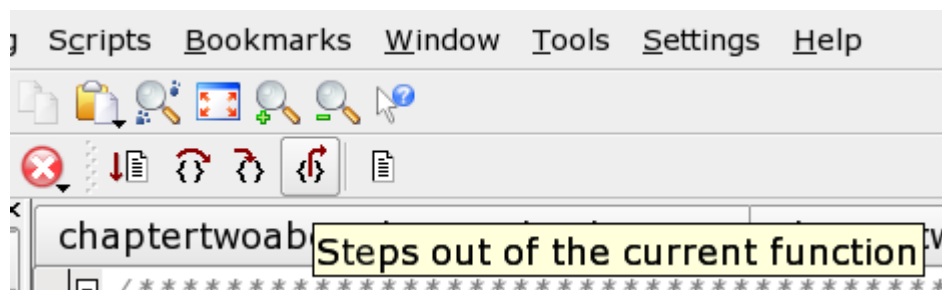
The first button on the debug toolbar is the continue button that will carry on with the program until another breakpoint is reached or the program exits or the program crashes.



The second button on the debug toolbar allows us to step over the next line of code which means that it is executed as normal, so any function calls made on that line will be made as normal and the program execution will stop on the next line down.



The third button on the debug toolbar is the step into button that allows us to follow the code into a function called on the current line. The code execution will stop on the first line of code in the function that is being called.



The fourth button on the debug toolbar allows us to step out of the current function, executing the code in the function as normal and then stopping at the line of code below the line that called the current function.

If we hit the continue button in the debug toolbar the program will continue until we get to the

Chapter 2 The KDE Application

breakpoint that we placed earlier to test the showAboutBox function.

```
/// show the about box if the arg was passed

if( showAboutBox() == true )
{
    aboutDlg->setProduct( "Display About Demo", "1.0", "pseudonym67", "2005" );
    aboutDlg->show();
}
```

If we now use the step into button we can check to see what answer the function will return.

The screenshot shows a debugger interface. On the left, the 'Watch' window displays the variable `bShowAboutBox` with a value of `true`. Below it, the variable `T1#0 ChapterTwoAboutDe` is also shown with a value of `true`. On the right, the source code of `ChapterTwoAboutDemo.cpp` is visible. The code defines a class `ChapterTwoAboutDemo` that inherits from `KMainWindow`. The `showAboutBox()` method is highlighted, showing it returns `bShowAboutBox`.

```
* @short Application Main Window
* @author pseudonym67 <pseudonym67@hotmail.com>
* @version 0.1
*/
class ChapterTwoAboutDemo : public KMainWindow
{
    Q_OBJECT

private:
    bool bShowAboutBox;
    KAboutDialog *aboutDlg;

public:
    /**
     * Get for show about box
     */
    bool showAboutBox()
    {
        return bShowAboutBox;
    }
}
```

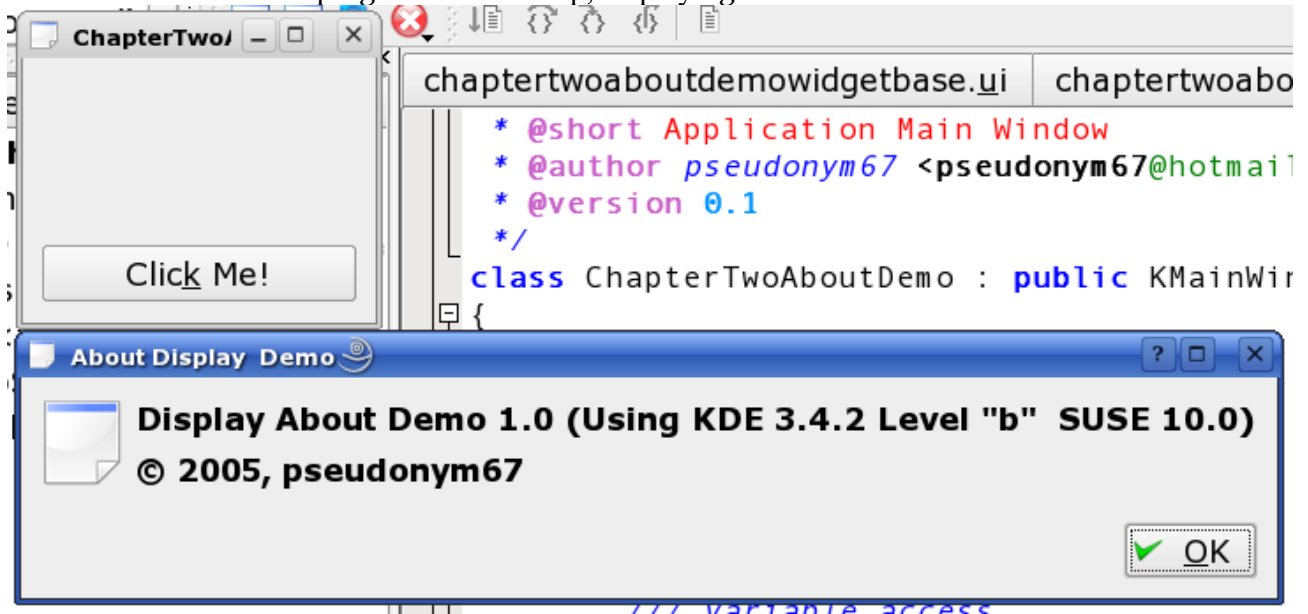
Or of course we could just check the local information in the variable window by clicking on what is already there when we open it and looking at the “this” object

The screenshot shows a debugger interface. The 'Watch' window displays the variable `bShowAboutBox` with a value of `true`. Below it, the variable `T1#0 ChapterTwoAboutDe` is expanded, showing its `this` pointer. The `this` pointer is expanded further, showing the `bShowAboutBox` variable with a value of `true` and the `aboutDlg` variable with a value of `true`.

```
Variable
- Watch
  bShowAboutBox
- T1#0 ChapterTwoAboutDe
  - this
    <>
    bShowAboutBox
    + aboutDlg true
```

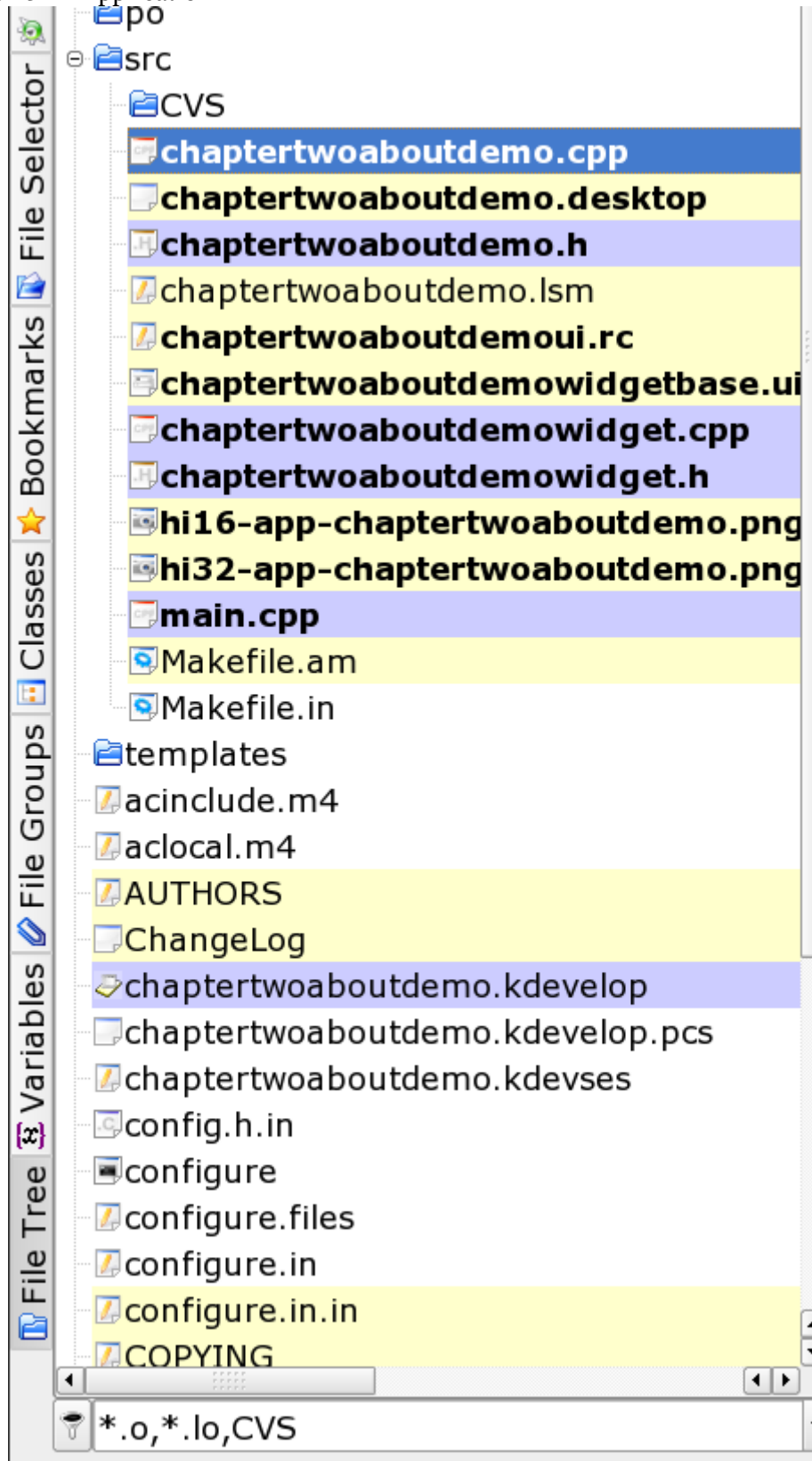
Chapter 2 The KDE Application

Now that we know that the function is going to return the answer that we want we can click on the continue button and the program will start up, displaying the about box first.



CVS The Easy Way

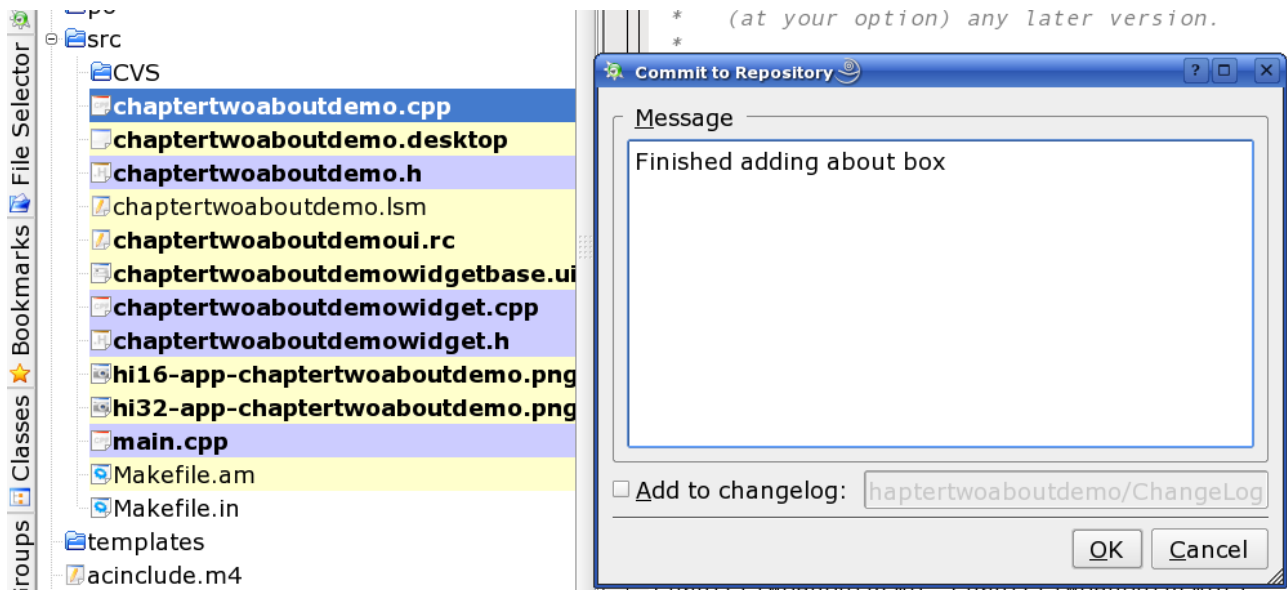
Up until now we have used CVS mostly through the Cervisia gui provided with Suse 10 but a much simpler and easier way exists to use CVS and that is directly through the KDevelop program. If we open up the File Tree on the left hand side menu bar we can see the files that have been modified since we created them.



If we select the `chaptertwoaboutdemo.cpp` file in the view and right click on it, at the bottom of the menu there is an option for `CvsService`. By moving down to the `CvsService` option we get a menu of CVS options with the top two options being the important one here. The second one down allows us to compare the current version of the file in our development directory with a version in CVS which we will usually want to be the most current version of the file stored there. The option at the

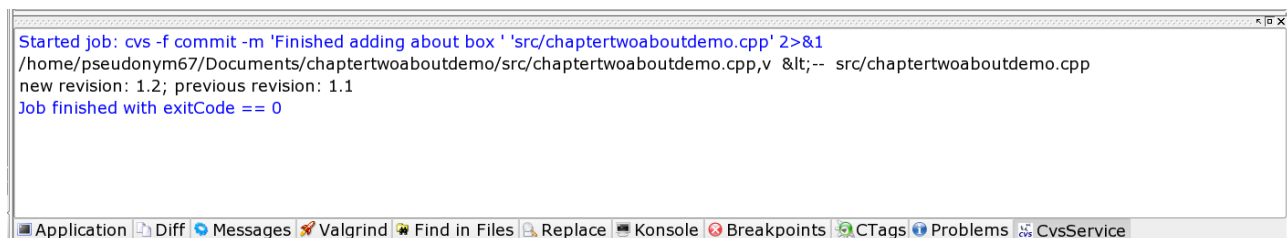
Chapter 2 The KDE Application

top of the file is the Commit To Repository option which works exactly the same as if we were committing the file through Cervisia.



As with a commit through Cervisia you are prompted to add a message usually saying what the reason is for the changes to the file. This is always good practice and if you don't add a message at this point a dialog will appear reminding you that this is good practice.

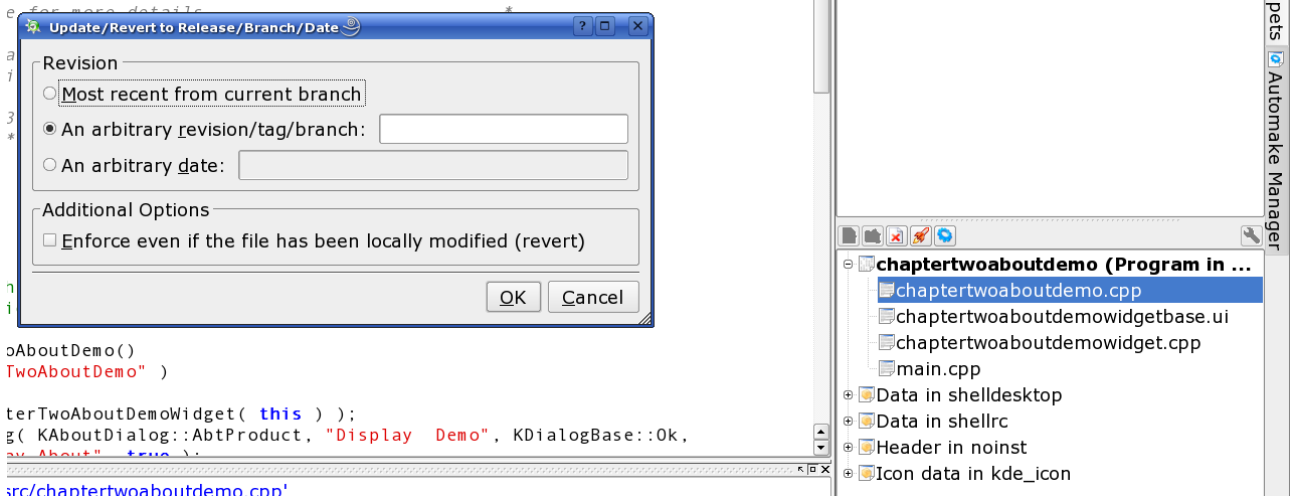
When you click the OK button the CvsService menu bar along the bottom will display the results of the CVS operation.



The same operations can be performed directly on the files through the Automake Manager although the Automake Manager does not give any visual cues as to which files have been changed.

As with any version control system an important aspect of CvsService is the option to change the versions of the files that we are currently using so if you right click on the chaptertwoaboutdemo.cpp file in Automake Manager and select CvsService, then select Update/Revert to Another Release you'll get this dialog.

Chapter 2 The KDE Application



If you have been working through with the code instead of just reading it you can enter 1.1 for the generated version of the `chaptertwoaboutdemo.cpp` file. Clicking on the OK button will mean that the current project file will then be the original file. Note that to see this properly you need to close the file in the editor, if it is open and then reload it. You can update the file back to the currently latest version by selecting CvsService/Update again and typing in 1.2 into the "An arbitrary revision/tag/branch:" box.

One final note about CVS here and one that is approaching another religious topic is just when do you update the code in CVS. I've worked in some places where code has to be put in the repository every night whether it compiles or not but I can see sense in the point that you shouldn't get a version increase in your code until it does something different from the last version so you only update the repository when a change/update/feature is complete. This is one of those things that is largely going to be defined on a project basis which is something that you may or may not have a say in, so for the most part adopt the project practices but for your own projects use the CVS in the way that you are most comfortable with.

Documentation

Documentation has always been a pain when it comes to software, the main reasons for this being that in most business environments the emphasis is on working code and not on explaining how the code works when there's no guarantee that anyone will ever read it. Some people simply aren't interested and others simply aren't very good at it, they can write it in code but can't explain how they did it very well at all. Then there is the point of how do you do it. Formal methods of documentation take us straight into religious wars territory with people shouting the merits of x over y and others saying that neither of them allow them to express what it is they are really doing.

Then there's KDevelop and the built in Doxygen a combination that to be honest makes documenting source code so easy there really isn't any excuse not to do it. Doxygen takes the comments that we should always write in the code but mostly don't and generates a full set of documents that are disgustingly easy to navigate and understand.

To see what I mean, in Automake Manager left click on `chaptertwoaboutdemo.h` so that it shows up in the editor and then go to tools/Preview Doxygen Output and you'll get

Chapter 2 The KDE Application

[Main Page](#) | [Class List](#) | [Directories](#) | [File List](#) | [Class Members](#)

chaptertwoaboutdemo.kdevelop Documentation

0.1

Generated on Wed Dec 7 19:15:33 2005 for chaptertwoaboutdemo.kdevelop by  1.4.4


Which is the main page of the documentation and shows just a header with the class version number as it appears in the header which we can get to if we click on the class list,

[Main Page](#) | [Class List](#) | [Directories](#) | [File List](#) | [Class Members](#)

chaptertwoaboutdemo.kdevelop Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

[ChapterTwoAboutDemo](#) [Application Main Window](#)

Generated on Wed Dec 7 19:15:33 2005 for chaptertwoaboutdemo.kdevelop by  1.4.4

Here we see that the ChapterTwoAboutDemo class is the Main Window for the application which we defined in our header file as

```
/**
 * @short Application Main Window
 * @author pseudonym67 <pseudonym67@hotmail.com>
 * @version 0.1
 */
class ChapterTwoAboutDemo : public KMainWindow
{
```

If we click on the ChapterTwoAboutDemo text we get,

ChapterTwoAboutDemo Class Reference

Application Main Window. [More...](#)

```
#include <chaptertwoaboutdemo.h>
```

[List of all members.](#)

Public Member Functions

| | | |
|--------------|---|------------------------|
| bool | getShowAboutBox () | <i>variable access</i> |
| void | setShowAboutBox (bool show) | |
| virtual void | show () | <i>override show</i> |
| | ChapterTwoAboutDemo () | |
| | ChapterTwoAboutDemo (bool showAbout) | |
| virtual | ~ChapterTwoAboutDemo () | |

Detailed Description

Application Main Window.

Author:

pseudonym67 <pseudonym67@hotmail.com>

Version:

0.1

Constructor & Destructor Documentation

ChapterTwoAboutDemo::ChapterTwoAboutDemo()

Default Constructor

virtual ChapterTwoAboutDemo::~~ChapterTwoAboutDemo() [virtual]

Default Destructor

The documentation for this class was generated from the following file:

- /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/[chaptertwoaboutdemo.h](#)

Generated on Wed Dec 7 19:15:33 2005 for chaptertwoaboutdemo.kdevelop by  1.4.4

which shows us that while there is nothing wrong with the comments that have been placed in the code we could do a little better and make things more complete as the “variable access” comment is just floating, the overridden show function would be better documented as the constructors and destructor's are, and the custom constructor taking the boolean parameter is not documented at all.

If we edit the class definition so that it looks like this,

```
/**
 * @short Application Main Window
 * @author pseudonym67 <pseudonym67@hotmail.com>
 * @version 0.1
 */
class ChapterTwoAboutDemo : public KMainWindow
{
    Q_OBJECT

private:
    bool bShowAboutBox;
```

Chapter 2 The KDE Application

```
KAboutDialog *aboutDlg;

public:
    /**
     * Get for show about box
     */
    bool getShowAboutBox()
    {
        return bShowAboutBox;
    }

    /**
     * set for show about box
     */
    void setShowAboutBox( bool show )
    {
        bShowAboutBox = show;
    }

    /**
     * override show
     */
    virtual void show();

public:
    /**
     * Default Constructor
     */
    ChapterTwoAboutDemo();
    /**
     * Custom Constructor with option to show the about dialog
     */
    ChapterTwoAboutDemo( bool showAbout );
    /**
     * Default Destructor
     */
    virtual ~ChapterTwoAboutDemo();
};
```

The previewed file looks like this,

Public Member Functions

| | |
|--------------|--|
| bool | getShowAboutBox () |
| void | setShowAboutBox (bool show) |
| virtual void | show () |
| | ChapterTwoAboutDemo () |
| | ChapterTwoAboutDemo (bool showAbout) |
| virtual | ~ChapterTwoAboutDemo () |

Detailed Description

Application Main Window.

Author:

pseudonym67 <pseudonym67@hotmail.com>

Version:

0.1

Constructor & Destructor Documentation

ChapterTwoAboutDemo::ChapterTwoAboutDemo()

Default Constructor

ChapterTwoAboutDemo::ChapterTwoAboutDemo(bool *showAbout*)

Custom Constructor with option to show the about dialog

virtual ChapterTwoAboutDemo::~~ChapterTwoAboutDemo() [virtual]

Default Destructor

Member Function Documentation

bool ChapterTwoAboutDemo::getShowAboutBox() [inline]

Get for show about box

void ChapterTwoAboutDemo::setShowAboutBox(bool *show*) [inline]

set for show about box

virtual void ChapterTwoAboutDemo::show() [virtual]

override show

which is a lot better. As long as you remember to save the file first as Doxygen previews from the

Chapter 2 The KDE Application

file saved on the disk which is not necessarily the same as the file currently open in the editor.

You can see from the examples that both “/**” and “///” can be used to specify a comment to be included in Doxygen which is just two of the methods that you can use. With the output depending on the programming language and how they are used. For our purposes we will be sticking with the “/**” from here on out. A full manual for Doxygen can be found at www.Doxygen.org

Generating The Documentation

It may or may not have escaped your notice but for some reason KDevelop only documents the Application Main Window by default. Since the bulk of the work of any program is going to be done by the main widget that is contained within the application we should probably document this as well. If you click on the chaptertwoaboutdemowidget.h file in Automake Manager to open it and edit it to look like,

```
/**
 * @short Application Main Widget
 * @author pseudonym67 <pseudonym67@hotmail.com>
 * @version 0.1
 */
class ChapterTwoAboutDemoWidget : public ChapterTwoAboutDemoWidgetBase
{
    Q_OBJECT

public:
    /**
     * Default constructor
     */
    ChapterTwoAboutDemoWidget(QWidget* parent = 0, const char* name = 0, WFlags fl = 0
);
    /**
     * Default destructor
     */
    ~ChapterTwoAboutDemoWidget();

public slots:
    /**
     * Fired when "Click Me" button is pressed
     */
    virtual void button_clicked();

protected:
protected slots:
};
```

If we preview the class with tools/Preview Doxygen Output we get,

Chapter 2 The KDE Application

Public Slots

```
virtual void button_clicked ()
```

Public Member Functions

```
ChapterTwoAboutDemoWidget (QWidget *parent=0, const char *name=0, WFlags fl=0)  
~ChapterTwoAboutDemoWidget ()
```

Detailed Description

Application Main Widget.

Author:

pseudonym67 <pseudonym67@hotmail.com>

Version:

0.1

Constructor & Destructor Documentation

```
ChapterTwoAboutDemoWidget::ChapterTwoAboutDemoWidget( QWidget * parent = 0,  
                                                       const char * name = 0,  
                                                       WFlags fl = 0  
                                                       )
```

Default constructor

```
ChapterTwoAboutDemoWidget::~ChapterTwoAboutDemoWidget( )
```

Default destructor

Member Function Documentation

```
virtual void ChapterTwoAboutDemoWidget::button_clicked( ) [virtual, slot]
```

Fired when "Click Me" button is pressed

The documentation for this class was generated from the following file:

- /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/[chaptertwoaboutdemowidget.h](#)

Generated on Fri Dec 9 11:01:59 2005 for chaptertwoaboutdemo.kdevelop by  1.4.4

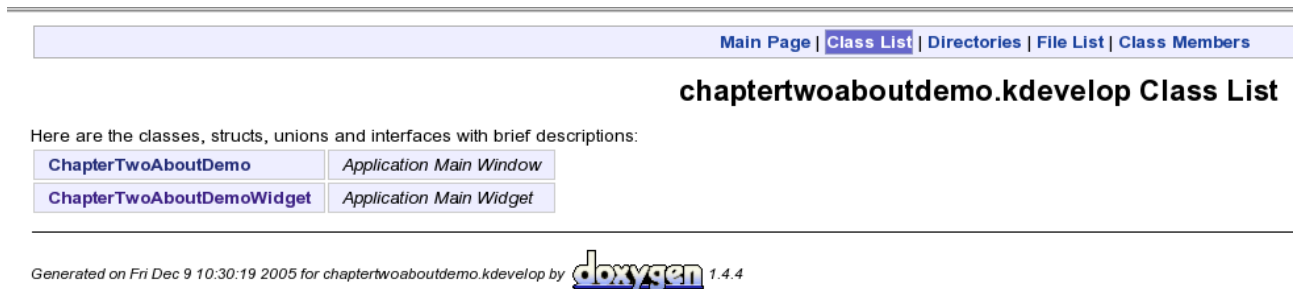
Now we can build the documentation for our project go to Build/Build API Documentation. The KDevelop Messages window will be focused on and the Doxygen output will appear there. The window will give warnings of anything that isn't documented that Doxygen thinks should be. For this reason there will be a lot of warnings at the start referring to the `chaptertwoaboutdemowidgetbase.h` and mostly the `chaptertwoaboutdemo.moc` file. Both of these files are generated from the `chaptertwoaboutdemowidgetbase.ui` file by the compiler, we shall look at these generated files next but you can't do anything about warning messages generated here and it's not a good idea to try as the files will be regenerated from scratch the next time you do a build of the project.

By default there are three document types output by Doxygen. The one we are focusing on here is the html documentation which is located in the `html` folder off the project directory folder so in this case it's at `chaptertwoaboutdemo/html`. A `Latex` folder is also created with the build files for the

Chapter 2 The KDE Application

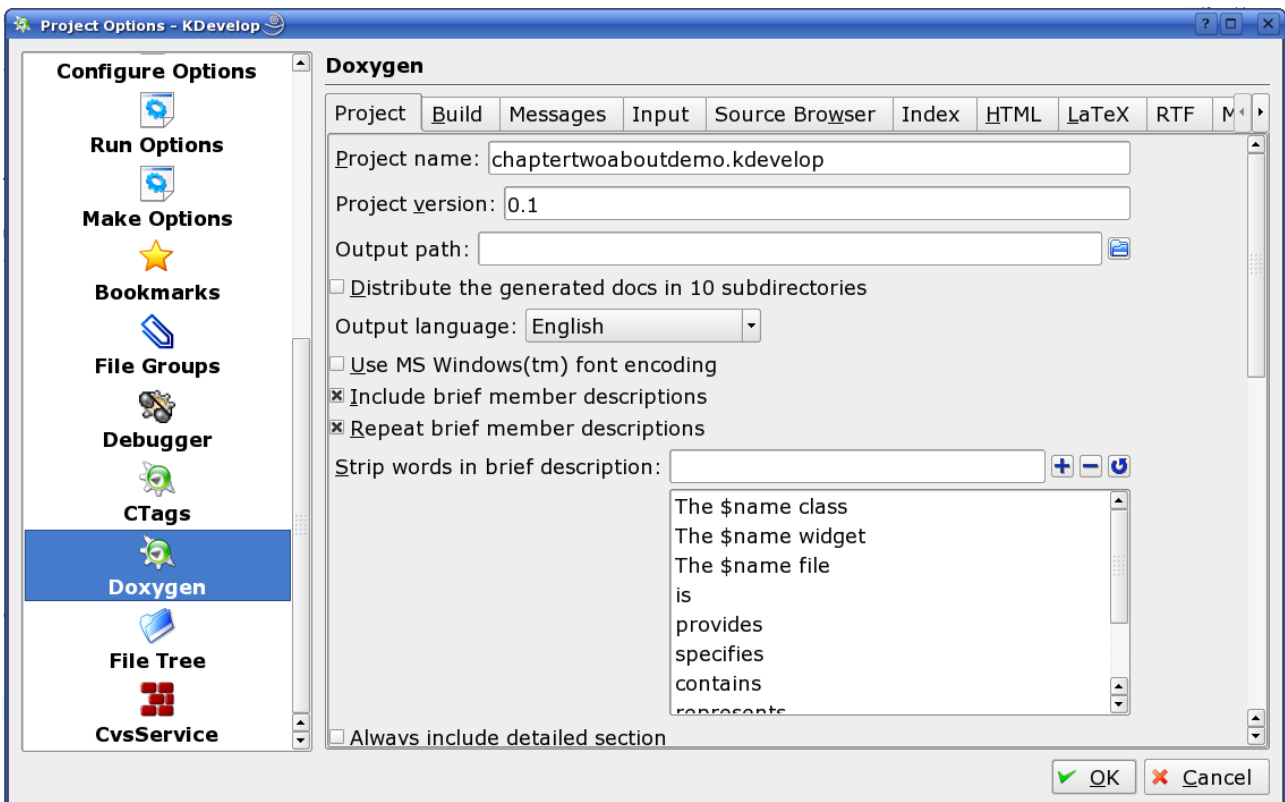
latex documentation. To view this open a Konsole in the Latex directory and type make, alternatively if you want pdf documentation type “make pdf” here. The third directory created by Doxygen is the xml directory which the Doxygen manual says is still under development. You can view the files in an xml viewer but for now it is probably better to leave them alone.

The Doxygen output is exactly as we have seen in the previews when we look through the html folder with the exception that the Class List now looks like



Doxygen Options

You can set the options for Doxygen in Projects/Project Options/Doxygen.



If you scroll along until you get to the Dot tab and uncheck the “Hide undocumented relations” box Doxygen will include a class diagram for the hello world classes in your html page.

Chapter 2 The KDE Application

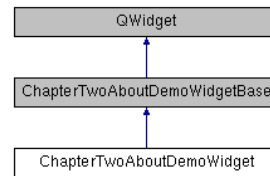
[Main Page](#) | [Class Hierarchy](#) | [Class List](#) | [Directories](#) | [File List](#) | [Class Members](#)

ChapterTwoAboutDemoWidget Class Reference

Application Main Widget. [More...](#)

```
#include <chaptertwoaboutdemowidget.h>
```

Inheritance diagram for ChapterTwoAboutDemoWidget:



[List of all members.](#)

Public Slots

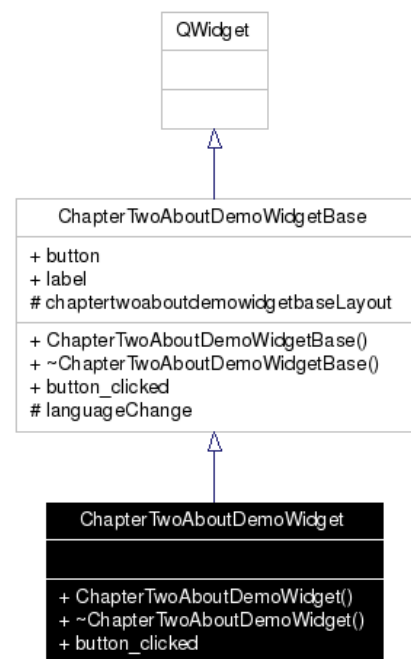
```
virtual void button_clicked ()
```

If you then check the “Use Dot” option on the same page you get more detailed diagramming.

Application Main Widget. [More...](#)

```
#include <chaptertwoaboutdemowidget.h>
```

Inheritance diagram for ChapterTwoAboutDemoWidget:

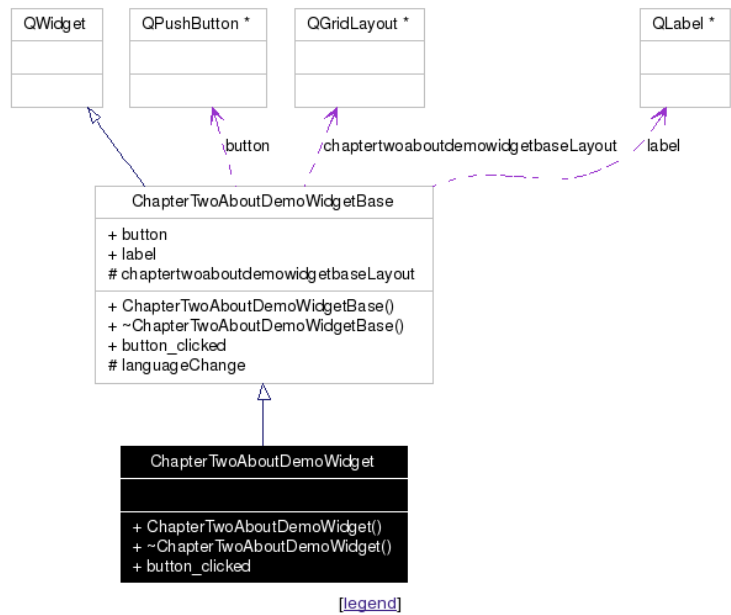


[\[legend\]](#)

Dot is part of Graphviz and information about it can be found at www.Graphviz.org By default it will also include a collaboration diagram for you class,

Chapter 2 The KDE Application

Collaboration diagram for ChapterTwoAboutDemoWidget



[List of all members.](#)

[\[legend\]](#)

which isn't half bad for a file that still looks like this,

Chapter 2 The KDE Application

```
00001 /*****
00002 * Copyright (C) 2005 by pseudonym67 *
00003 * pseudonym67@hotmail.com *
00004 *
00005 * This program is free software; you can redistribute it and/or modify *
00006 * it under the terms of the GNU General Public License as published by *
00007 * the Free Software Foundation; either version 2 of the License, or *
00008 * (at your option) any later version. *
00009 *
00010 * This program is distributed in the hope that it will be useful, *
00011 * but WITHOUT ANY WARRANTY; without even the implied warranty of *
00012 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the *
00013 * GNU General Public License for more details. *
00014 *
00015 * You should have received a copy of the GNU General Public License *
00016 * along with this program; if not, write to the *
00017 * Free Software Foundation, Inc., *
00018 * 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. *
00019 *****/
00020
00021
00022 #ifndef _CHAPTERTWOABOUTDEMOWIDGET_H_
00023 #define _CHAPTERTWOABOUTDEMOWIDGET_H_
00024 00025 #include "chaptertwoaboutdemowidgetbase.h"
00026
00032 class ChapterTwoAboutDemoWidget : public ChapterTwoAboutDemoWidgetBase
00033 {
00034     Q_OBJECT
00035
00036 public:
00040     ChapterTwoAboutDemoWidget(QWidget* parent = 0, const char* name = 0, WFlags fl = 0 );
00044     ~ChapterTwoAboutDemoWidget();
00045
00046 public slots:
00050     virtual void button_clicked();
00051
00052 protected:
00053
00054 protected slots:
00055
00056 };
00057
00058 #endif
00059
```

Generated on Fri Dec 9 19:00:05 2005 for chaptertwoaboutdemo.kdevelop by  1.4.4

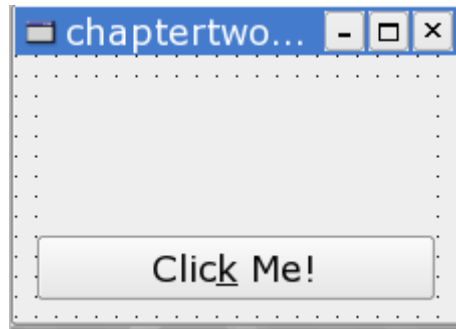
There are plenty of options we haven't looked at here and you're encouraged to play around with them to see what they do. For instance you could include a search engine or have it build your documentation into a Windows style .chm help file which you can show to Windows programmers and say “Yeah it took me ages to get the diagrams right”.

But we can't avoid it any more we're going to have to take a look at what is going on in the background with the ChapterTwoAboutDemoWidget class.

Generated Files

There are so far three files that we have come across that are generated from the ui file, which if you remember is the xml file generated from the form,

Chapter 2 The KDE Application



These files are the `chaptertwoaboutdemowidgetbase.h`, `chaptertwoaboutdemowidgetbase.cpp` and `chaptertwoaboutdemowidgetbase.moc`. The `chaptertwoaboutdemowidgetbase.h` and the `chaptertwoaboutdemowidgetbase.cpp` are generated directly from the gui and resemble the sort of file that you would expect to see in any Windows programming environment as they directly control whatever it is that is on the form.

```

/*****
** Form interface generated from reading ui file
** '/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/chaptertwoaboutdemowidgetbase.ui'
**
** Created: Mon Dec 5 19:22:09 2005
** by: The User Interface Compiler ($Id: qt/main.cpp 3.3.4 edited Nov 24 2003 $)
**
** WARNING! All changes made in this file will be lost!
*****/

#include <qvariant.h>
#include <qwidget.h>

class QVBoxLayout;
class QHBoxLayout;
class QGridLayout;
class QSpacerItem;
class QPushButton;
class QLabel;

class ChapterTwoAboutDemoWidgetBase : public QWidget
{
    Q_OBJECT

public:
    ChapterTwoAboutDemoWidgetBase( QWidget* parent = 0, const char* name = 0, WFlags fl = 0 );
    ~ChapterTwoAboutDemoWidgetBase();

    QPushButton* button;
    QLabel* label;

public slots:
    virtual void button_clicked();

protected:
    QGridLayout* chaptertwoaboutdemowidgetbaseLayout;

protected slots:
    virtual void languageChange();

};
```

Chapter 2 The KDE Application

```
#endif // CHAPTERTWOABOUTDEMOWIDGETBASE_H
```

In the generated ChapterTwoAboutDemoWidgetBase class we can see pointers to the button and the label on the form and a slot declared for when the button is clicked. We can also see a pointer to QGridLayout has been included which controls the layout of the items on the form. We will look at the layouts of forms in the next chapter. and there is finally a languageChange function declared.

These are all created and implemented in the chaptertwoaboutdemowidgetbase.cpp file

```
#include <kdialog.h>
#include <klocale.h>
/*****
** Form implementation generated from reading ui file
** "/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chaptertwoaboutdemo/src/chaptertwoaboutdemowidgetbase.ui"
**
** Created: Mon Dec 5 19:22:12 2005
**      by: The User Interface Compiler ($Id: qt/main.cpp 3.3.4  edited Nov 24 2003 $)
**
** WARNING! All changes made in this file will be lost!
*****/

#include "chaptertwoaboutdemowidgetbase.h"

#include <qvariant.h>
#include <qpushbutton.h>
#include <qlabel.h>
#include <qlayout.h>
#include <qtooltip.h>
#include <qwhatsthis.h>

/*
 * Constructs a ChapterTwoAboutDemoWidgetBase as a child of 'parent', with the
 * name 'name' and widget flags set to 'f'.
 */
ChapterTwoAboutDemoWidgetBase::ChapterTwoAboutDemoWidgetBase( QWidget* parent, const
char* name, WFlags fl )
: QWidget( parent, name, fl )
{
    if ( !name )
        setName( "chaptertwoaboutdemowidgetbase" );
    chaptertwoaboutdemowidgetbaseLayout = new QGridLayout( this, 1, 1, 11, 6,
"chaptertwoaboutdemowidgetbaseLayout");

    button = new QPushButton( this, "button" );

    chaptertwoaboutdemowidgetbaseLayout->addWidget( button, 1, 0 );

    label = new QLabel( this, "label" );

    chaptertwoaboutdemowidgetbaseLayout->addWidget( label, 0, 0 );
    languageChange();
    resize( QSize(220, 133).expandedTo(minimumSizeHint()) );
    clearWState( WState_Polished );

    // signals and slots connections
    connect( button, SIGNAL( clicked() ), this, SLOT( button_clicked() ) );
}

/*
 * Destroys the object and frees any allocated resources
 */
ChapterTwoAboutDemoWidgetBase::~ChapterTwoAboutDemoWidgetBase()
```

Chapter 2 The KDE Application

```
{
    // no need to delete child widgets, Qt does it all for us
}

/*
 * Sets the strings of the subwidgets using the current
 * language.
 */
void ChapterTwoAboutDemoWidgetBase::languageChange()
{
    setCaption( QString::null );
    button->setText( tr2i18n( "Click Me!" ) );
    label->setText( QString::null );
}

void ChapterTwoAboutDemoWidgetBase::button_clicked()
{
    qWarning( "ChapterTwoAboutDemoWidgetBase::button_clicked(): Not implemented yet" );
}

#include "chaptertwoaboutdemowidgetbase.moc"
```

The constructor of `chaptertwoaboutdemowidgetbase.cpp` is where all the fun happens, it starts by checking to see if the “name” value is not 0. Which if you remember is called from the default constructor from our inheriting class,

```
ChapterTwoAboutDemoWidget::ChapterTwoAboutDemoWidget(QWidget* parent, const char* name,
WFlags fl)
    : ChapterTwoAboutDemoWidgetBase(parent,name,fl)
{}
```

which is instantiated as,

```
setCentralWidget( new ChapterTwoAboutDemoWidget( this ) );
```

and means that as the constructor is called with only one parameter then the other two parameters take their default values in the constructor declaration,

```
/**
 * Default constructor
 */
ChapterTwoAboutDemoWidget(QWidget* parent = 0, const char* name = 0, WFlags fl = 0 );
```

This means that the line

```
setName( "chaptertwoaboutdemowidgetbase" );
```

is executed and the name for the widget is `chaptertwoaboutdemowidgetbase`. The next line

```
chaptertwoaboutdemowidgetbaseLayout = new QGridLayout( this, 1, 1, 11, 6,
"chaptertwoaboutdemowidgetbaseLayout");
```

we're going to skip for now as we'll be looking at grid layouts later and the following four lines should be simple enough,

```
button = new QPushButton( this, "button" );

chaptertwoaboutdemowidgetbaseLayout->addWidget( button, 1, 0 );

label = new QLabel( this, "label" );

chaptertwoaboutdemowidgetbaseLayout->addWidget( label, 0, 0 );
```

The two objects the `QPushButton` and the `QLabel` are created as children of the `chaptertwoaboutdemowidgetbase` class and then added to the grid layout passing the row and the

Chapter 2 The KDE Application

column number so that the layout class can determine where they go, remember though that we won't be calling this code often if at all ourselves as this code is generated from what we do in the form designer.

The next section in the file sets up the signals and slots that have been defined in the form designer

```
// signals and slots connections
connect( button, SIGNAL( clicked() ), this, SLOT( button_clicked() ) );
```

This is done by calling the QObject function connect that takes the object that is emitting the signal as its first parameter, the signal that is being emitted as its second parameter, the object that is receiving the signal as its third parameter and the slot or function that is called when the signal is emitted. Which as far as we as users of the Qt Library are concerned when we click on the button our button_clicked function is called.

Or at least mostly anyway what we have to remember from the libraries point of view is that we are talking about generated files here so the function that originally deals with the button_clicked signal is defined at the end of the cpp file as

```
void ChapterTwoAboutDemoWidgetBase::button_clicked()
{
    qWarning( "ChapterTwoAboutDemoWidgetBase::button_clicked(): Not implemented yet" );
}
```

But seeing as KDevelop gives us a class that inherits from ChapterTwoAboutDemoWidgetBase and automatically gives us an override on this function we should never see the warning unless we delete the slot function

```
public slots:
    /**
     * Fired when "Click Me" button is pressed
     */
    virtual void button_clicked();
```

in the ChapterTwoAboutDemoWidget class.

The only things that we have missed so far are the languageChange function which sets the I18N specified text strings to the current language selection and the final line

```
#include "chaptertwoaboutdemowidgetbase.moc"
```

Which appends the chaptertwoaboutdemowidgetbase.moc file at the end,

```
/* *****
** ChapterTwoAboutDemoWidgetBase meta object code from reading C++ file
'chaptertwoaboutdemowidgetbase.h'
**
** Created: Mon Dec 5 19:22:12 2005
** by: The Qt MOC ($Id: qt/moc_yacc.cpp 3.3.4 edited Jan 21 18:14 $)
**
** WARNING! All changes made in this file will be lost!
***** */

#undef QT_NO_COMPAT
#include "chaptertwoaboutdemowidgetbase.h"
#include <qmetaobject.h>
#include <qapplication.h>

#include <private/qucomextra_p.h>
#if !defined(Q_MOC_OUTPUT_REVISION) || (Q_MOC_OUTPUT_REVISION != 26)
#error "This file was generated using the moc from 3.3.4. It"
#error "cannot be used with the include files from this version of Qt."
#error "(The moc has changed too much.)"
```

Chapter 2 The KDE Application

```
#endif

const char *ChapterTwoAboutDemoWidgetBase::className() const
{
    return "ChapterTwoAboutDemoWidgetBase";
}

QMetaObject *ChapterTwoAboutDemoWidgetBase::metaObj = 0;
static QMetaObjectCleanUp cleanUp_ChapterTwoAboutDemoWidgetBase(
    "ChapterTwoAboutDemoWidgetBase", &ChapterTwoAboutDemoWidgetBase::staticMetaObject );

#ifdef QT_NO_TRANSLATION
QString ChapterTwoAboutDemoWidgetBase::tr( const char *s, const char *c )
{
    if ( qApp )
        return qApp->translate( "ChapterTwoAboutDemoWidgetBase", s, c,
QApplication::DefaultCodec );
    else
        return QString::fromLatin1( s );
}
#endif
#ifdef QT_NO_TRANSLATION_UTF8
QString ChapterTwoAboutDemoWidgetBase::trUtf8( const char *s, const char *c )
{
    if ( qApp )
        return qApp->translate( "ChapterTwoAboutDemoWidgetBase", s, c,
QApplication::UnicodeUTF8 );
    else
        return QString::fromUtf8( s );
}
#endif // QT_NO_TRANSLATION_UTF8

#endif // QT_NO_TRANSLATION

QMetaObject* ChapterTwoAboutDemoWidgetBase::staticMetaObject()
{
    if ( metaObj )
        return metaObj;
    QMetaObject* parentObject = QWidget::staticMetaObject();
    static const QUMethod slot_0 = { "button_clicked", 0, 0 };
    static const QUMethod slot_1 = { "languageChange", 0, 0 };
    static const QMetaData slot_tbl[] = {
        { "button_clicked()", &slot_0, QMetaData::Public },
        { "languageChange()", &slot_1, QMetaData::Protected }
    };
    metaObj = QMetaObject::new_metaobject(
        "ChapterTwoAboutDemoWidgetBase", parentObject,
        slot_tbl, 2,
0, 0,
#ifdef QT_NO_PROPERTIES
0, 0,
0, 0,
#endif // QT_NO_PROPERTIES
0, 0 );
    cleanUp_ChapterTwoAboutDemoWidgetBase.setMetaObject( metaObj );
    return metaObj;
}

void* ChapterTwoAboutDemoWidgetBase::qt_cast( const char* cname )
{
    if ( !strcmp( cname, "ChapterTwoAboutDemoWidgetBase" ) )
        return this;
    return QWidget::qt_cast( cname );
}

bool ChapterTwoAboutDemoWidgetBase::qt_invoke( int _id, QUObject* _o )
{
    switch ( _id - staticMetaObject()->slotOffset() ) {
    case 0: button_clicked(); break;
    case 1: languageChange(); break;
    }
```

Chapter 2 The KDE Application

```
default:
    return QWidget::qt_invoke( _id, _o );
}
return TRUE;
}

bool ChapterTwoAboutDemoWidgetBase::qt_emit( int _id, QObject* _o )
{
    return QWidget::qt_emit(_id,_o);
}
#ifdef QT_NO_PROPERTIES

bool ChapterTwoAboutDemoWidgetBase::qt_property( int id, int f, QVariant* v)
{
    return QWidget::qt_property( id, f, v);
}

bool ChapterTwoAboutDemoWidgetBase::qt_static_property( QObject* , int , int ,
QVariant* ){ return FALSE; }
#endif // QT_NO_PROPERTIES
```

This is the Meta Object Compiled file and we specified that we wanted it by adding

`Q_OBJECT`

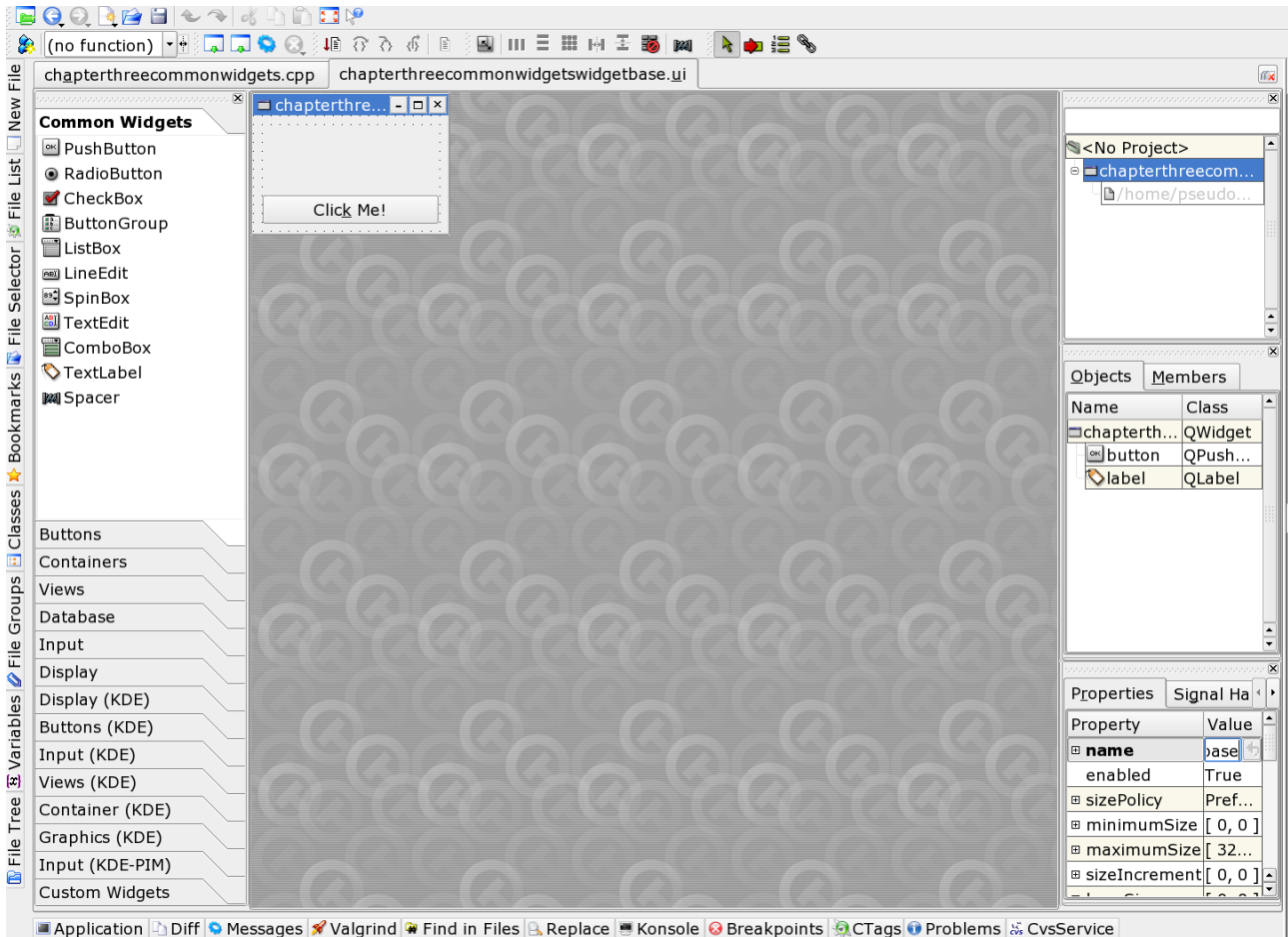
to our header file, or at least by not deleting it when KDevelop added to our class header. I'm not going to go into this file in detail as it's getting into how the entire Qt library works territory which is beyond the scope of not only this chapter but this book. Suffice to say that even a cursory glance at the file shows that this is the glue that makes the language and function calls when buttons etc are pressed work with it's definition of the emit and invoke functions.

Summary

In this chapter we have looked at the make up of KDE Application including the basic tools that we will be using on a day to day basis such as CVS, documentation and a basic look at the requirements of using the debugger. We ended with a look at the files that are generated in the background for us, giving us an idea of how the program works so we can hopefully approach the programming with a little understanding.

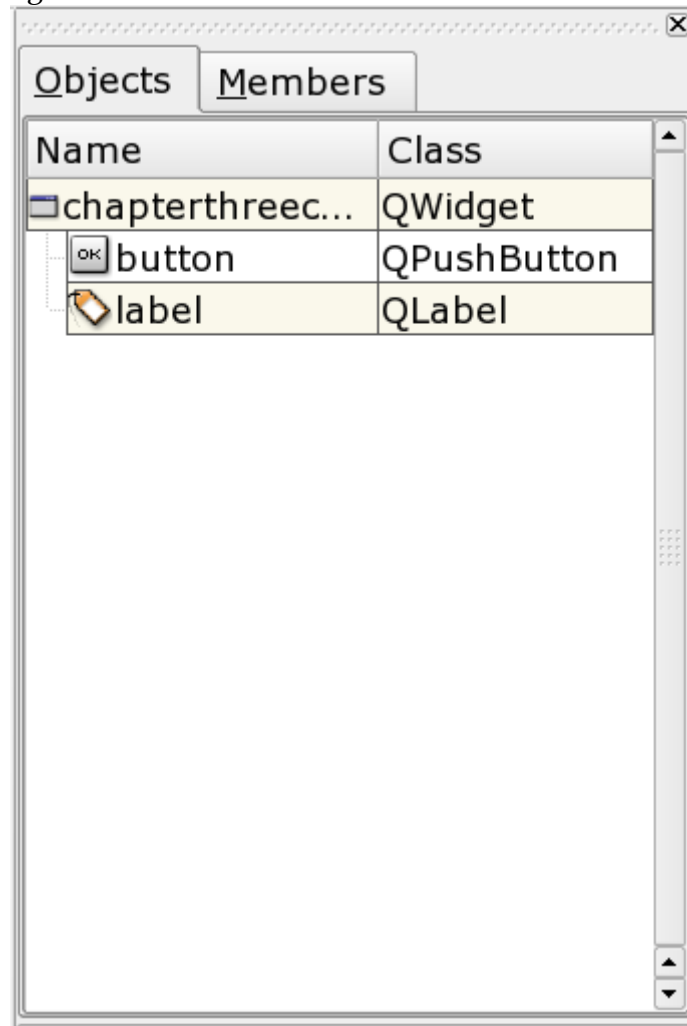
Chapter 3 Common Widgets

KDevelop comes with a built in interface designer which would be better known to Windows programmer as a form designer. This is used for developing dialog boxes and small applications and is where we will be spending most of our time for the next few chapters. If you start the project chapterthreecommonwidgets in the same way as described in earlier chapters or open the sample project provided then open up the Automake Manager and click on the ui file, then the interface designer will open automatically and will look like,

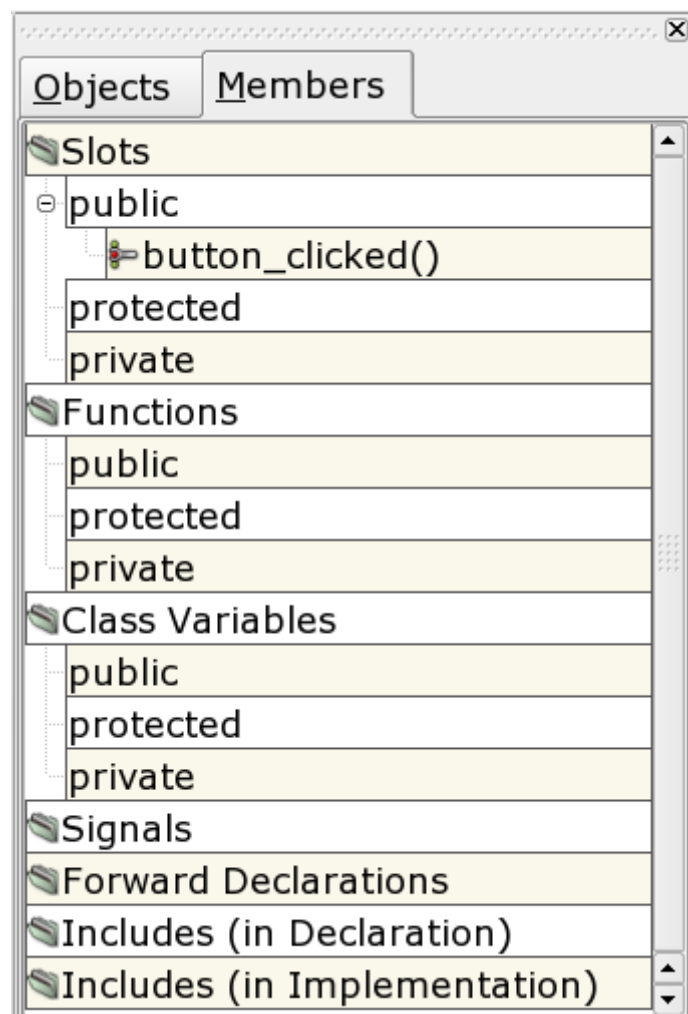


In the centre of the interface designer you will see the form or dialog that we are working with. This at the start of the development will be the main form for your application, to the left you will see the Toolbox widget that lists all the available widgets that you can use in the application and to the right you will see three boxes containing information. The first box contains project information and for our purposes can just be closed to make room for the things that we are going to use. The second box contains information about the widgets or objects that are placed on the form, this has two tabs the first listing all the objects or widgets that are placed on the form

Chapter 3 Common Widgets

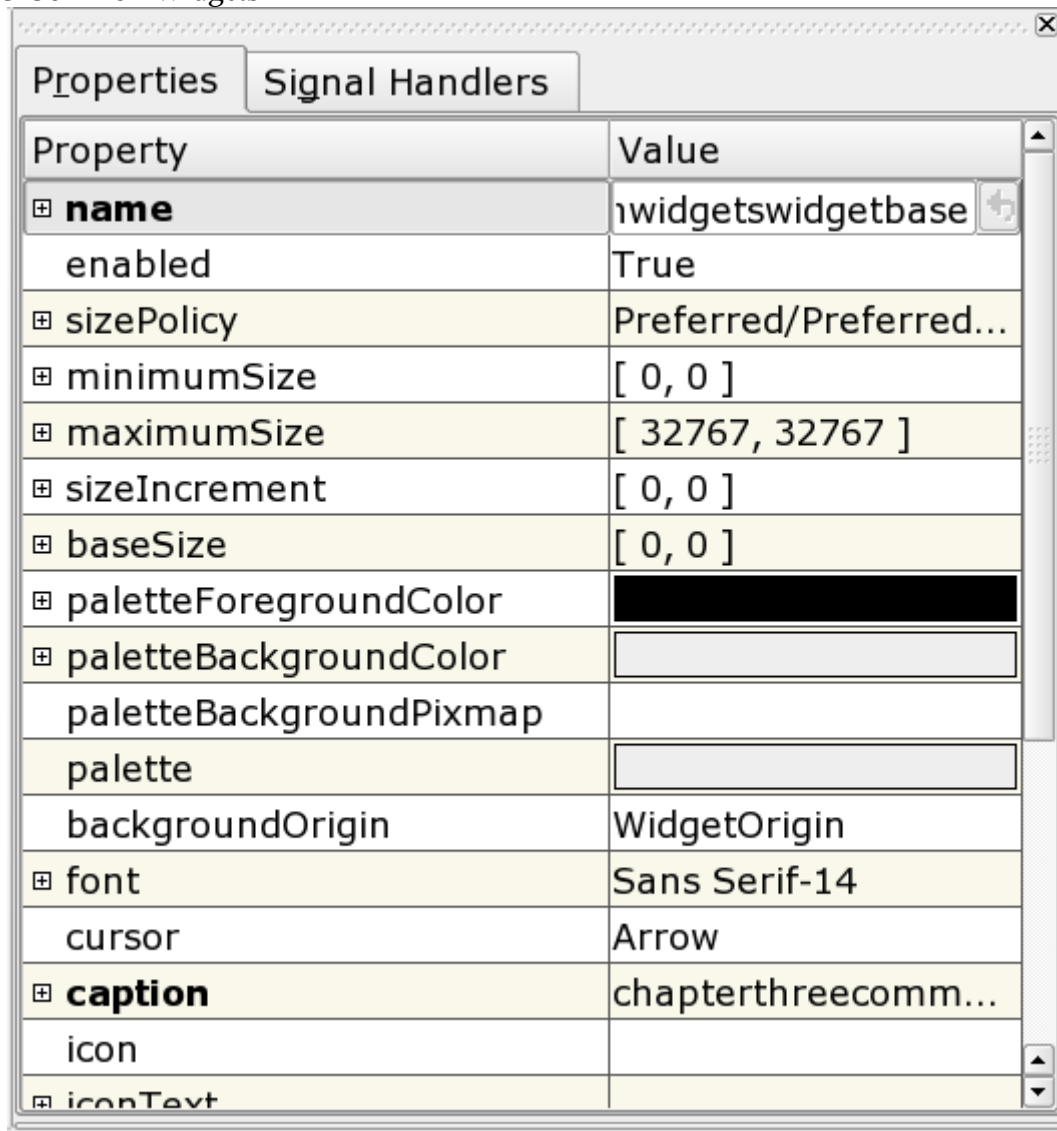


which for the standard default project is just a label and a button and the tab for members,

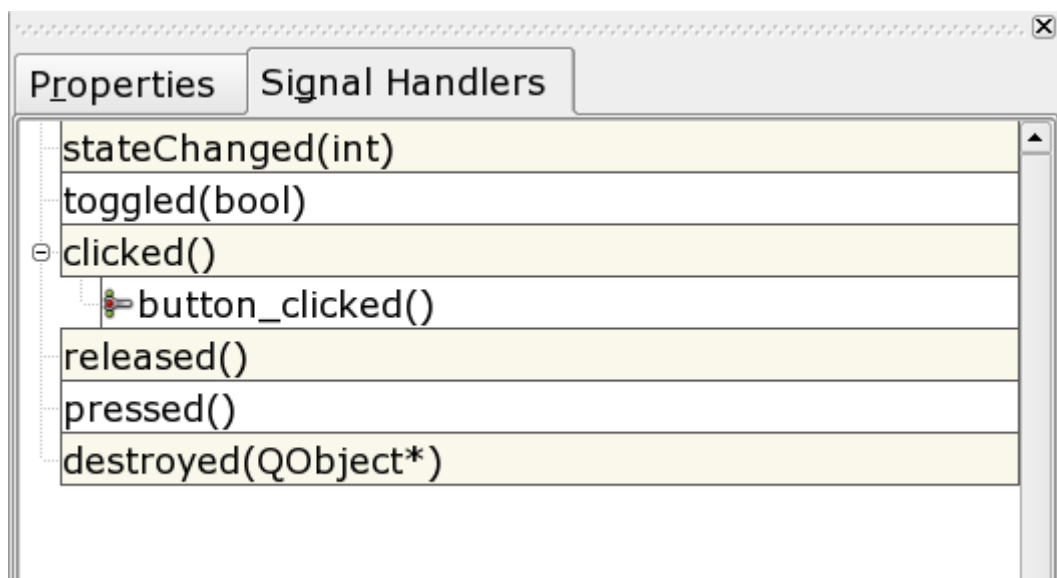


which lists all the members associated with the form. We can see from the image that there is one slot which responds to the button being clicked but you may notice that there is no corresponding signal. This is because the signal is sent by the button and not the form, the form merely receives the signal it doesn't send it.

The third box is the properties box which should be familiar to anyone who has used a form designer in any environment before. The main box lists all the publicly available options that the designer of the project can adjust on the form,



These properties are available for all widgets that appear on the form and the form itself. The Signal Handlers tab shows all the signals that the currently selected widget can output and what function if any we are using to handle the signal.



Chapter 3 Common Widgets

Once again I've selected the button on the form and we can see that we are handling the clicked() signal with the button_clicked() slot.

Layouts

The Qt library has a built in concept of Layouts, by this they mean the way that the widgets contained in a form are displayed when the size of the form is changed. If you build and run the default project created by KDevelop and then expand the window either by clicking on the resize button on the top border or by selecting an edge and expanding it manually then you will see that the widgets in the form maintain their relative positions automatically. This is because they are maintained in what is called a GridLayout. In a previous section I put off talking about layouts until they were necessary and now they are necessary. If you look in the generated header file for this ui file. (Note. It won't be there if you are just looking at the demo code and haven't built the project.) For the current project it is the chapterthreecommonwidgetswidgetsbase.h file in the debug/src folder off the project folder.

The header file contains the following forward class declarations,

```
class QVBoxLayout;  
class QHBoxLayout;  
class QGridLayout;  
class QSpacerItem;  
class QPushButton;  
class QLabel;
```

We already know that the QPushButton class and the QLabel class are used in the project. The other forward declarations are for the layout of the form. QVBoxLayout is the class that controls the vertical layout, QHBoxLayout controls the horizontal layout while QGridLayout imposes a table style layout that controls both vertical and horizontal positions. With the QSpacerItem being included to allow the designer of the form to add spacings between widgets as they see fit.

The type of Layout class that we are using is declared in as,

```
protected:  
    QGridLayout* chapterthreecommonwidgetswidgetbaseLayout;
```

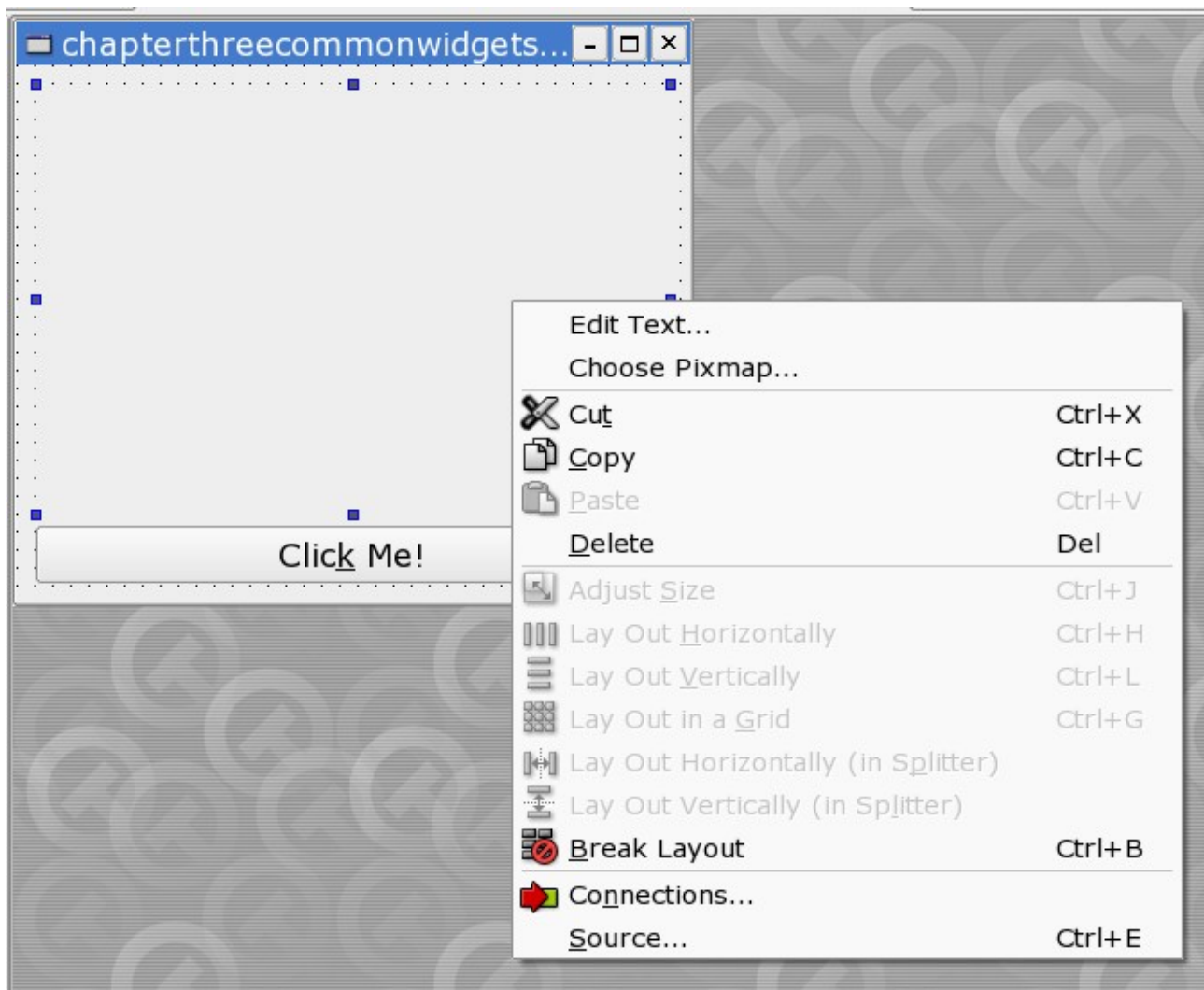
and in the constructor we get

```
chapterthreecommonwidgetswidgetbaseLayout = new QGridLayout( this, 1, 1, 11, 6,  
    "chapterthreecommonwidgetswidgetbaseLayout");  
  
button = new QPushButton( this, "button" );  
  
chapterthreecommonwidgetswidgetbaseLayout->addWidget( button, 1, 0 );  
  
label = new QLabel( this, "label" );  
  
chapterthreecommonwidgetswidgetbaseLayout->addWidget( label, 0, 0 );
```

The first line creates the GridLayout object by passing it the required parameters. The first parameter being **this** which is the parent widget for the layout. You can also pass a QLayout pointer as the first parameter if you are using layers of layouts. The second and third parameters are the rows and columns, in that order, as the GridLayout class divides the form into the specified number of rows and columns. The fourth parameter is the margin between the widget and the sides of the parent or form and the fifth parameter is the spacing between the widgets.

The rest of the code in the example creates and places the widgets with the label being in the top left row 0 column 0 and the button being in the bottom left row 1 column 0. Of course to see how something works properly we are going to have to break it. Right click on the form,

Chapter 3 Common Widgets



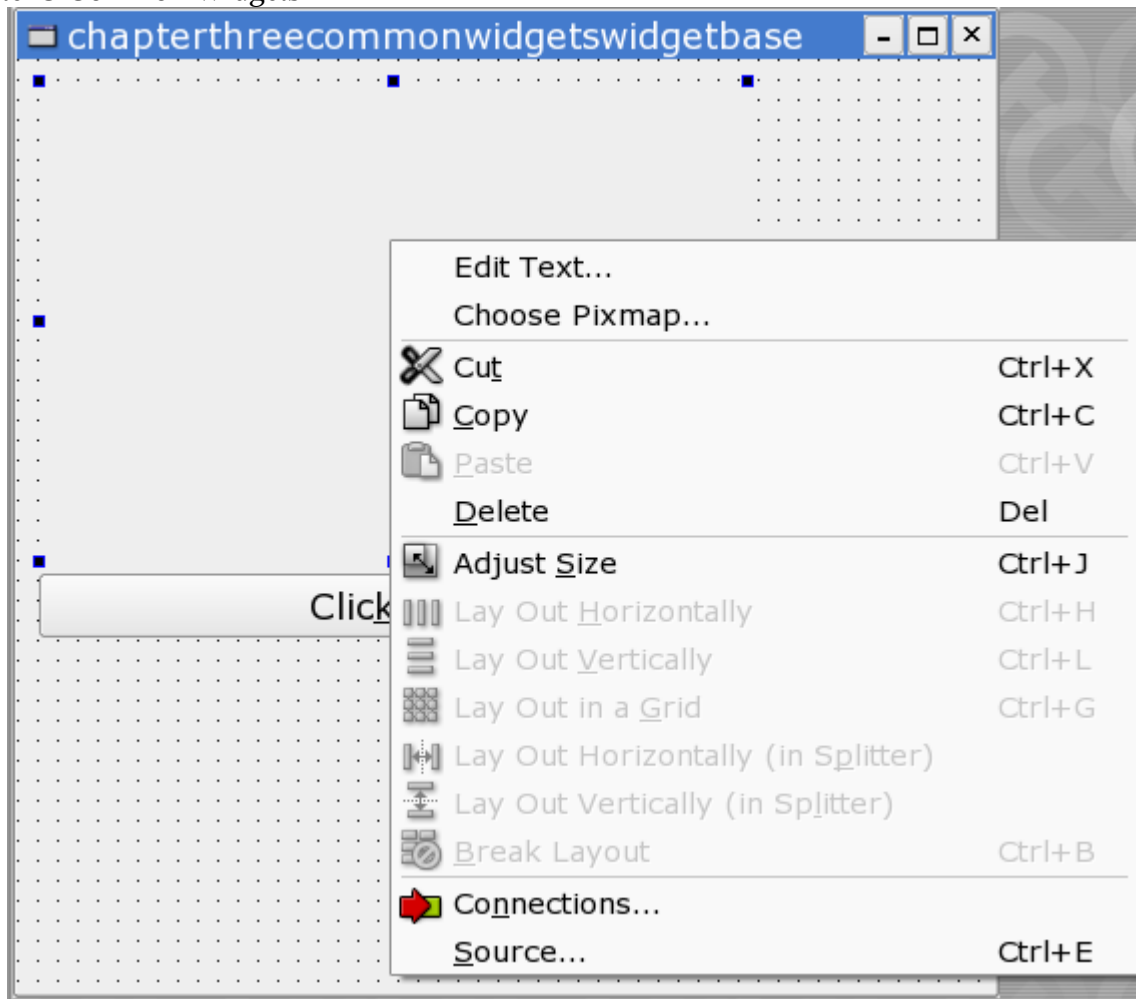
and select Break Layout. You will now find that if you select the edge of the form and resize it then the controls do not automatically resize. If you decide to try the demo in this state you will find that the controls on the form don't respond to any size changes made to the form. In fact you are well on your way to creating your first completely unusable project. So we'd better have a look at fixing it. to start with you can give yourself all the space you need to develop your forms, so please design them with usability in mind and don't try to cram everything into an impossibly small area. You can have a form that looks like,

Chapter 3 Common Widgets

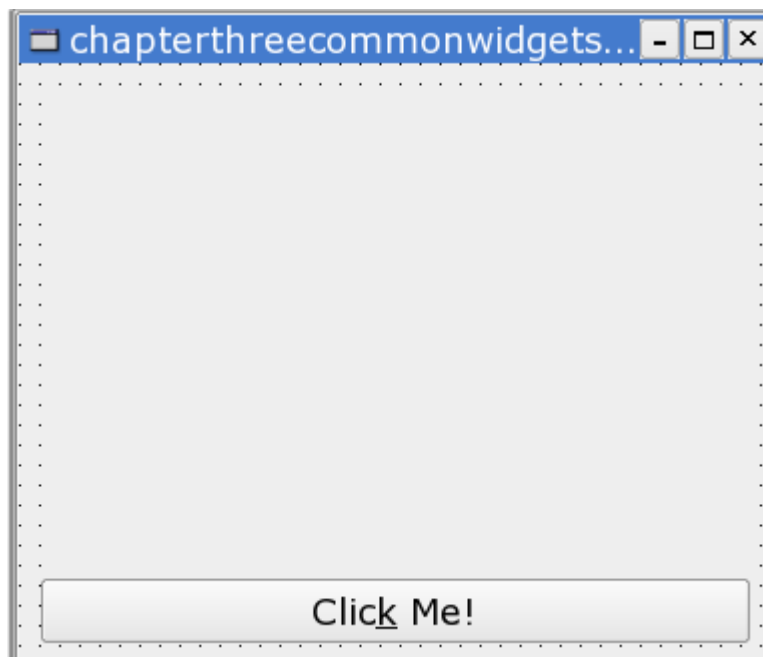


To correct this,

Chapter 3 Common Widgets



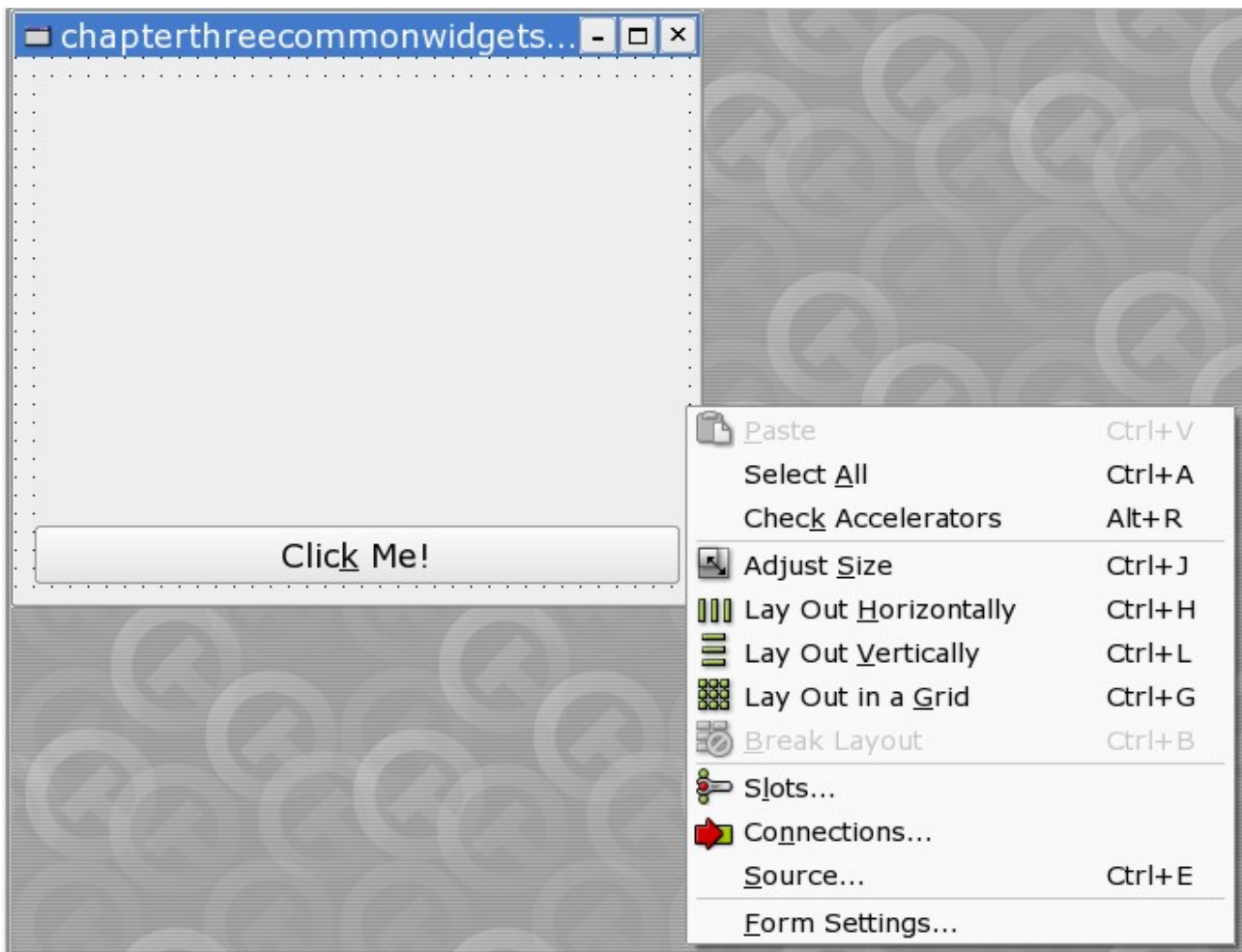
simply choose Adjust Size



and the form is automatically resized around the controls. the problem still remains however that

Chapter 3 Common Widgets

there is no layout class controlling the form so things are still wrong. To add a layout we need to right click on a section of form and not on a control,

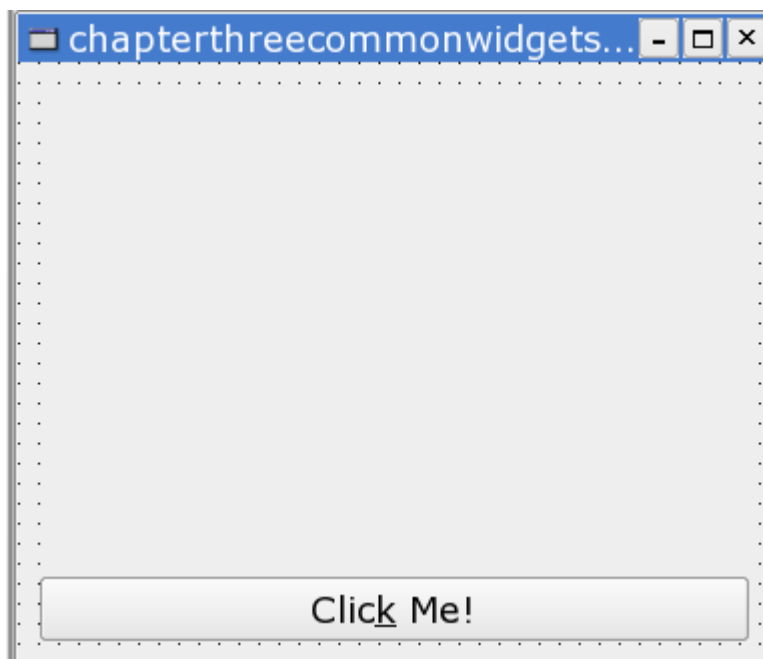


If we select Lay Out Horizontally we get,

Chapter 3 Common Widgets



which is just plain wrong, so we'll try vertical,



This does actually work although only because we are using such a simple dialog. If we wanted to half the size of the label and add another widget then things would fail in the same way they did with the horizontal layout although it is worth bearing in mind that if all your controls are all lined up either horizontally or vertically you can use the horizontal or vertical layout options safely but if things do start to break then go for the grid layout.

Chapter 3 Common Widgets

Problems Building Forms

When I moved on from showing how the layouts worked and started to look at how actual common widgets that this chapter is named for, I deleted the label and added two group boxes one for check boxes and one for radio buttons like so.



on saving the file and trying to build the project the compiler complained that it couldn't find the declarations for the new widgets, a little investigation revealed that the chapterthreecommonwidgetswidgetbase.h file contained.

```
public:
    chapterthreecommonwidgetsWidgetBase( QWidget* parent = 0, const char* name = 0, WFlags
fl = 0 );
    ~chapterthreecommonwidgetsWidgetBase();

    QLabel* label;
    QPushButton* button;
```

while the chapterthreecommonwidgetswidgetbase.h~ file contains,

```
public:
    chapterthreecommonwidgetsWidgetBase( QWidget* parent = 0, const char* name = 0, WFlags
fl = 0 );
    ~chapterthreecommonwidgetsWidgetBase();

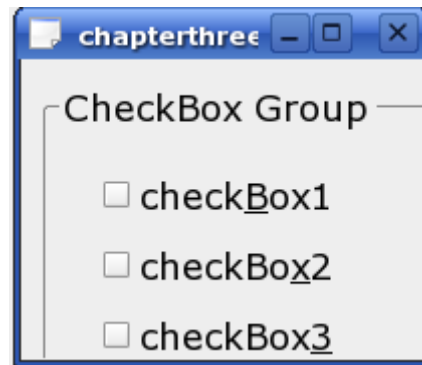
    QPushButton* button;
    QButtonGroup* buttonGroup1;
    QCheckBox* checkBox1;
    QCheckBox* checkBox3;
    QCheckBox* checkBox2;
    QButtonGroup* buttonGroup2;
    QRadioButton* radioButton1;
    QRadioButton* radioButton2;
    QRadioButton* radioButton3;
```

There is obviously some confusion going on here. Fortunately there is an easy way to solve this go to the debug/src folder and delete all the files there. Then return to KDevelop and go to Build/Run automake & friends. When that has finished go to Build/Run Configure. If the program refuses to work open the widget.cpp file in your project in this case it's the chapterthreecommonwidgetswidget.cpp and remove the references to the label in the button_clicked

Chapter 3 Common Widgets function.

Also now is probably a good time for a reminder that building a project does not automatically save the ui file which means that if you have made any changes to the form they will not be reflected in the build until you save the form file.

Another problem that can arise is that you occasionally lose the sizing of the form so that it displays like this,



The way to fix this is to make sure that you have a layout on the form, if this alone doesn't clear the problem up then select Adjust Size from the right click menu and you should get this,



It should be noted that this doesn't work with frames. For some reason when you click on adjust size with frames it ignores the size of the frame so the best bet is to use a Group Box which works the same as a Button Group and just delete the title, same effect, easier life. But more on that when we get to containers.

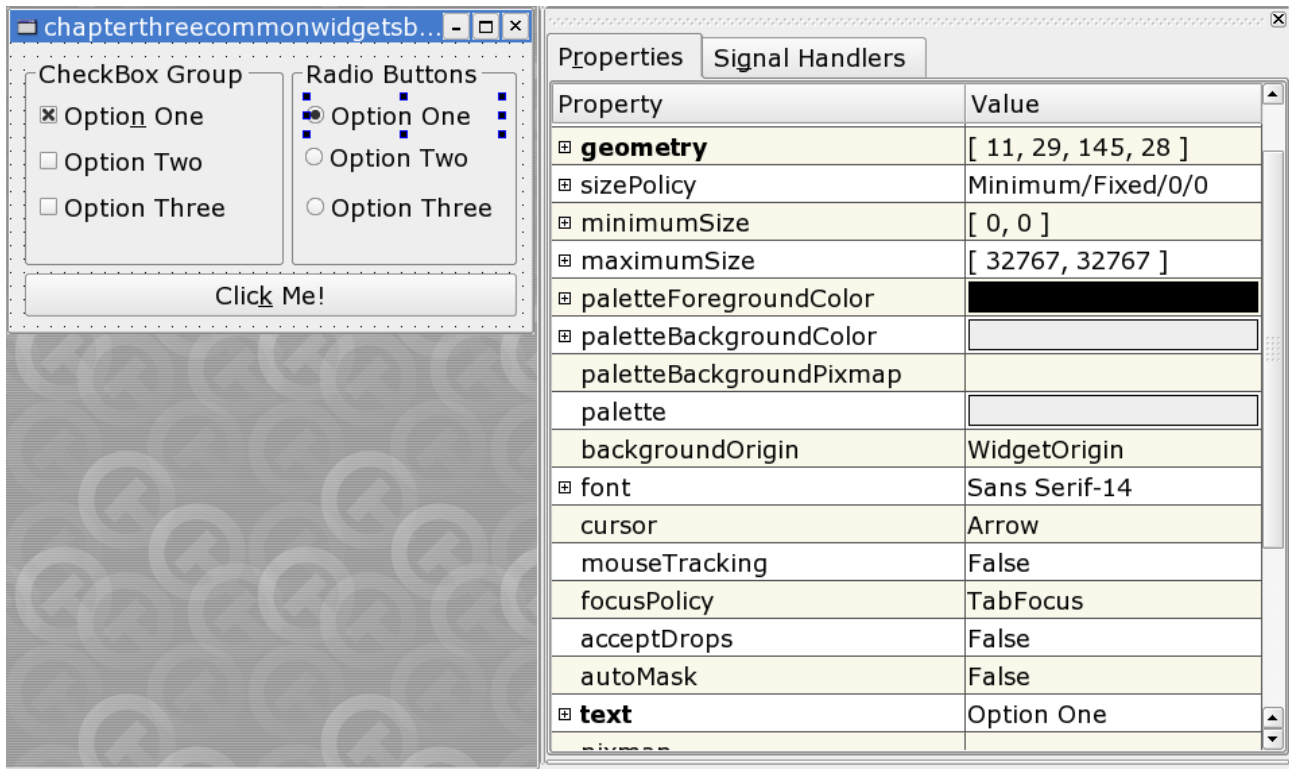
Common Widgets

Now we'll take a look at the common widgets that are provided by the form designer, rather than try to fit everything on a single form I have arbitrarily grouped them into individual projects. The first project looks at the buttons and the button groups.

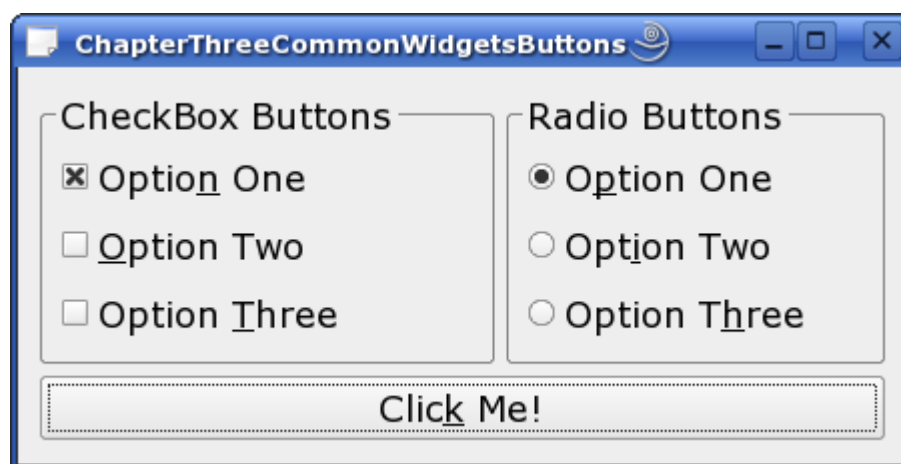
Chapter 3 Common Widgets

Buttons

The `chapterthreecommonwidgetsbuttons` project demonstrates the use of buttons and button groups, it is set up exactly the same as all the previous projects described earlier, being a Simple Designer Based KDE Application.



The above image is the creation phase of the form in the designer, the `QButtonGroups` are placed first and the `QCheckBoxes` and `QRadioButtons` are placed on top of them, with the details being edited in the properties box. Even if you have never seen a properties editor before the names should be pretty straight forward and if there is any doubt about them then you can do a search in the KDevelop documentation, though a quick hint is that most of them can be found in the `QWidget` documentation.



Chapter 3 Common Widgets

The application shows two `QButtonGroup` objects each containing three objects that represent options within the program. At the bottom of the form is the standard generated button that we have seen before. The Form is set to a grid layout with the `QButtonGroups` being set with a vertical layout. In the generated code this looks like,

```
chapterthreecommonwidgetsbuttonswidgetbaseLayout = new QGridLayout( this, 1, 1, 11, 6,
"chapterthreecommonwidgetsbuttonswidgetbaseLayout")

...

buttonGroup4Layout = new QVBoxLayout( buttonGroup4->layout() );
buttonGroup4Layout->setAlignment( Qt::AlignTop );

...

chapterthreecommonwidgetsbuttonswidgetbaseLayout->addWidget( buttonGroup4, 0, 0 );

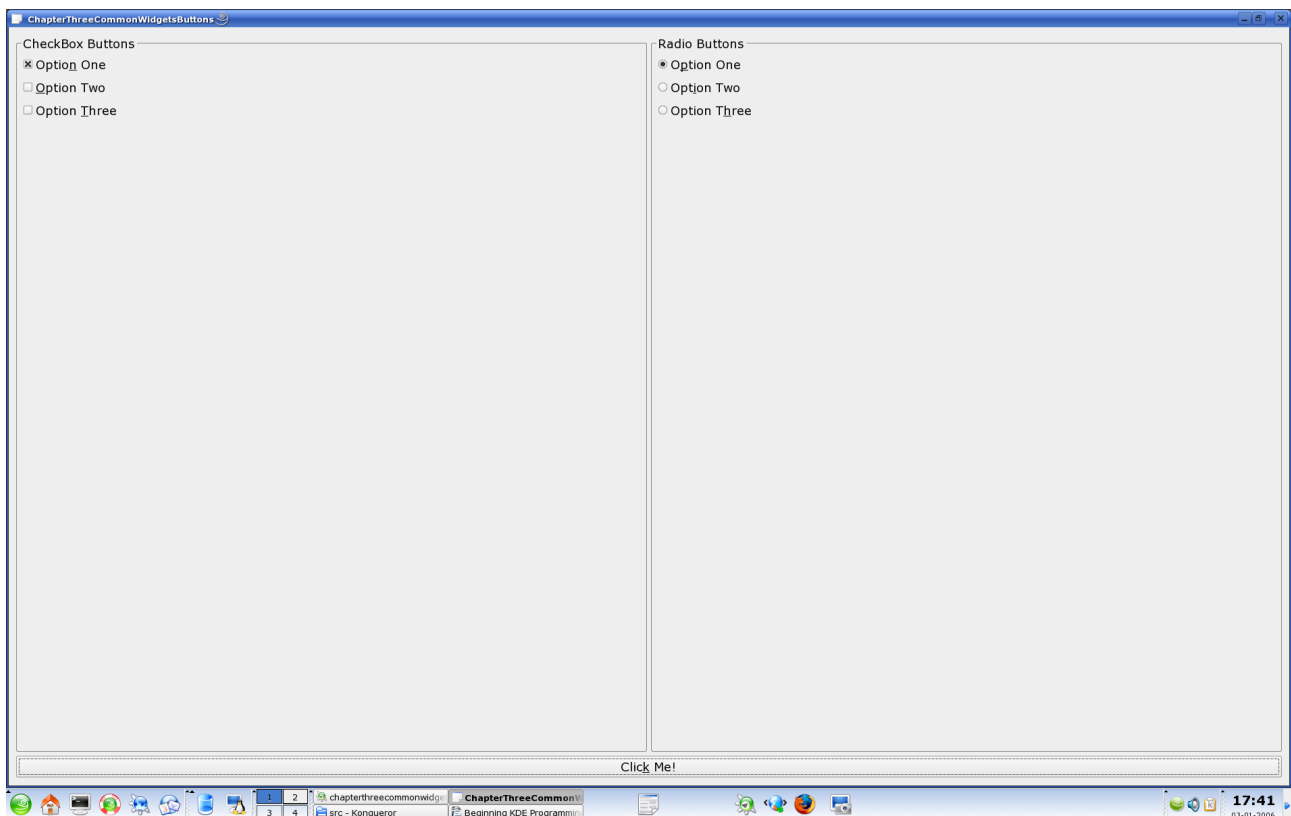
...

buttonGroup3Layout = new QVBoxLayout( buttonGroup3->layout() );
buttonGroup3Layout->setAlignment( Qt::AlignTop );

...

chapterthreecommonwidgetsbuttonswidgetbaseLayout->addWidget( buttonGroup3, 0, 1 );
```

with each vertical layout being created to hold either the `QCheckBoxes` or the `QRadioButton`s and then the `QButtonGroup` containing the vertical layout is added to the grid layout. The effect of this is if you create or just run the `chapterthreecommonwidgetsbuttons` application and then expand it to fill the screen the `QCheckBoxes` and the `QRadioButton`s maintain there relationship to each other rather than spacing out evenly across the larger surface area of the form.



This of course is just a demonstration to show that you can have multiple layers of layouts within

Chapter 3 Common Widgets

the form with each controlling a separate given area. In this case if you didn't add the extra layouts the code would respond in the same way. As the original code without the extra layers uses the `setGeometry` function to position the `QCheckBoxes` and `QRadioButtons` and these are not changed when the form is maximised.

```
buttonGroup3 = new QButtonGroup( this, "buttonGroup3" );

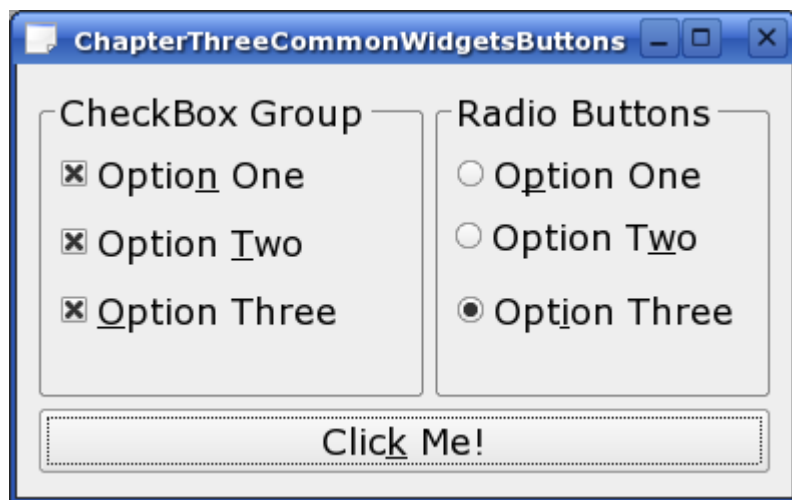
radioButton4 = new QRadioButton( buttonGroup3, "radioButton4" );
radioButton4->setGeometry( QRect( 11, 29, 145, 28 ) );
radioButton4->setChecked( TRUE );

radioButton5 = new QRadioButton( buttonGroup3, "radioButton5" );
radioButton5->setGeometry( QRect( 11, 63, 145, 28 ) );

radioButton6 = new QRadioButton( buttonGroup3, "radioButton6" );
radioButton6->setGeometry( QRect( 11, 97, 145, 28 ) );

chapterthreecommonwidgetsbuttonswidgetbaseLayout->addWidget( buttonGroup3, 0, 1 );
```

As you can see from the code above the `QButtonGroup` is created and then the `QRadioButtons` are created as members of the button group with their positioning being set by the `setGeometry` function. The `QButtonGroup` controls the layout of the buttons that are part of its group. Setting the `QButtonGroup` to a vertical layout helps the designer of the form but they also have another advantage and that is if you run the program and then Option Two followed by Option Three in the `QCheckBox` Group and then Left Click Option Two and Option Three in the Radio Button Group, you'll get a form that looks like this.



`QCheckBoxes` work as either on or off options so when you click down the row all the boxes become checked whereas with the `Radio Buttons` only one button within a `QButtonGroup` will be checked at any one time so the selected option within the `QRadioButton` group moves down as you do with the mouse clicking them.

The usual way to check if either a Check Box or Radio Button option is selected is to check before the beginning of an operation by calling the `isChecked` function for both `QCheckBox` and `QRadioButton` as I have done in `button_clicked` function when you left click on the "Click Me!" button,

...

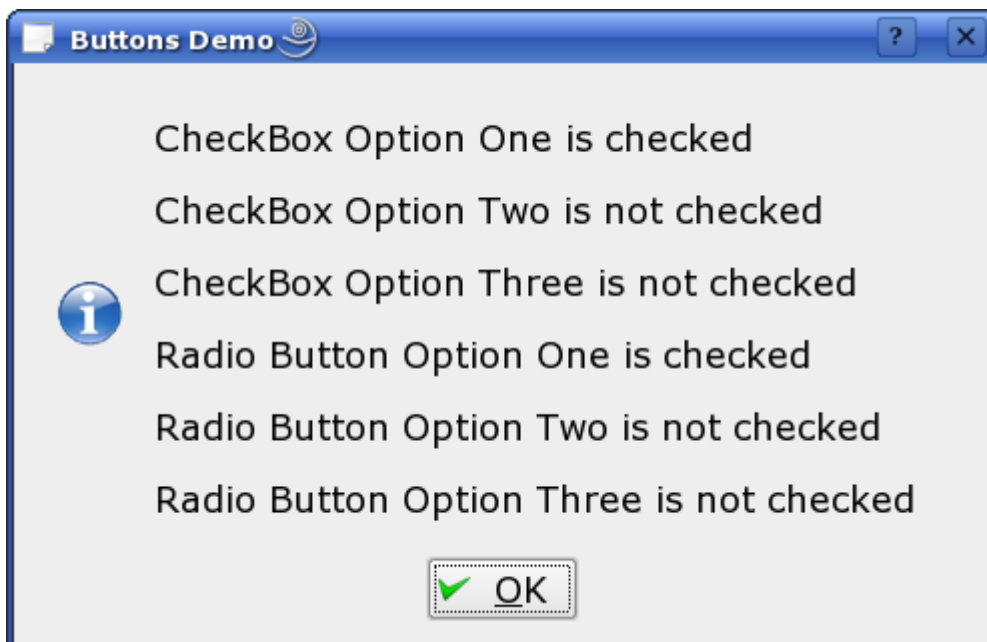
```
QString strRadioButtonTwo = i18n( "Radio Button Option Two is " );
```

Chapter 3 Common Widgets

```
QString strRadioButtonThree = i18n( "Radio Button Option Three is " );
QString strTitle = i18n( "Buttons Demo" );
QString strChecked = i18n( "checked\n" );
QString strNotChecked = i18n( "not checked\n" );
...
strDisplay.append( strRadioButtonThree );
if( radioButtonOptionThree->isChecked() == true )
{
    strDisplay.append( strChecked );
}
else
    strDisplay.append( strNotChecked );

QMessageBox::information( this, strTitle, strDisplay );
```

The code simply goes through all the Check Boxes and Radio Buttons calling the `isChecked()` function and then builds a string to report the results.



Of course if this was a real program the code would execute depending on what options were selected rather than just displaying a message box.

Tip Of The Day

When you are building the files and KDevelop generates the “projectname”widgetbase.h and the “projectname”widgetbase.cpp file along with the “projectname”widgetbase.moc files it includes all the header files that you need in your class that is derived from “projectname”widgetbase usually called “projectname”widget but it doesn't add the include files for the widgets in the “projectname”widget.cpp file so open the “projectname”widgetbase.cpp file and copy and paste the headers to the “projectname”widget.cpp file.

For example in the current project the generated file in debug/src called chapterthreecommonwidgetsbuttonswidgetbase.cpp has the includes,

```
#include <qvariant.h>
#include <qpushbutton.h>
#include <qbuttongroup.h>
```

Chapter 3 Common Widgets

```
#include <qcheckbox.h>
#include <qradiobutton.h>
#include <qlayout.h>
#include <qtooltip.h>
#include <qwhatsthis.h>
```

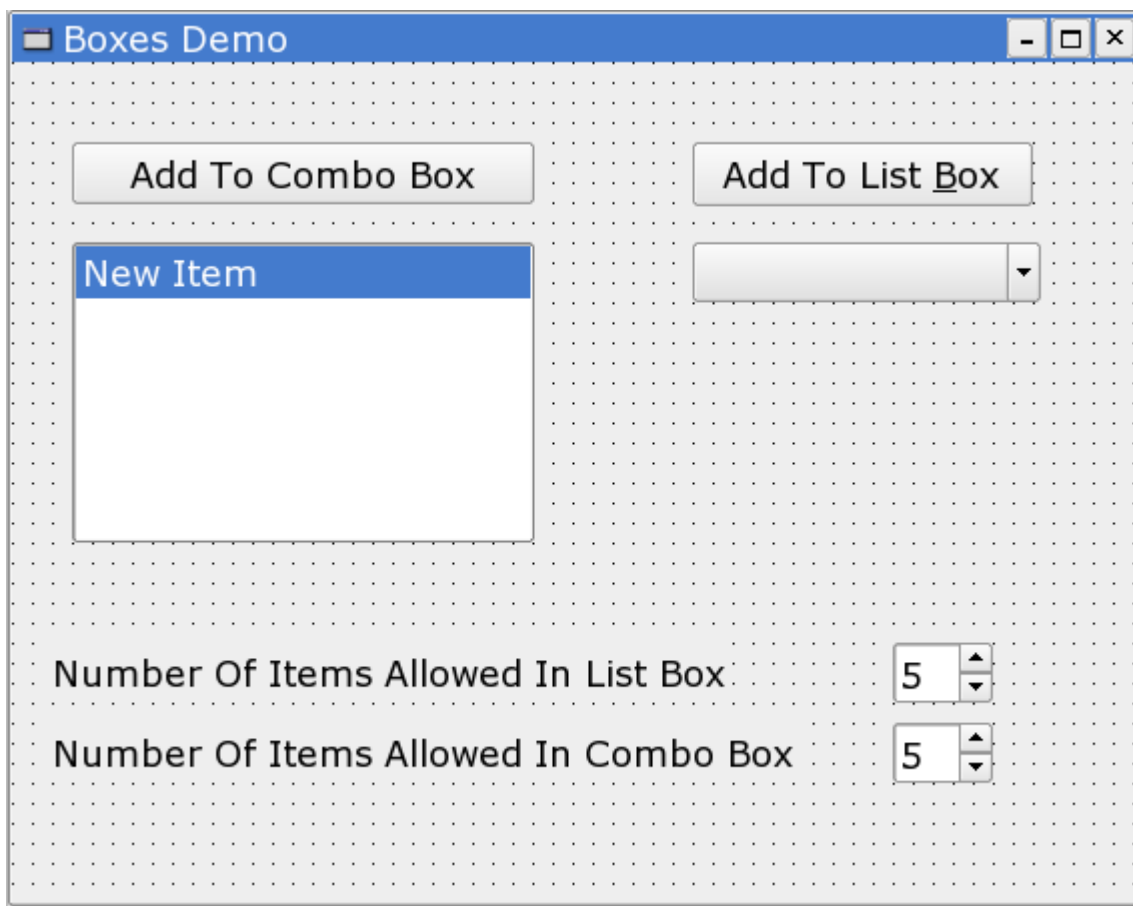
while the chapterthreecommonwidgetsbuttonswidget.cpp file has the include files

```
#include <qlabel.h>
```

So the includes from the previous file should be copied to this file and everything should compile as expected.

Boxes

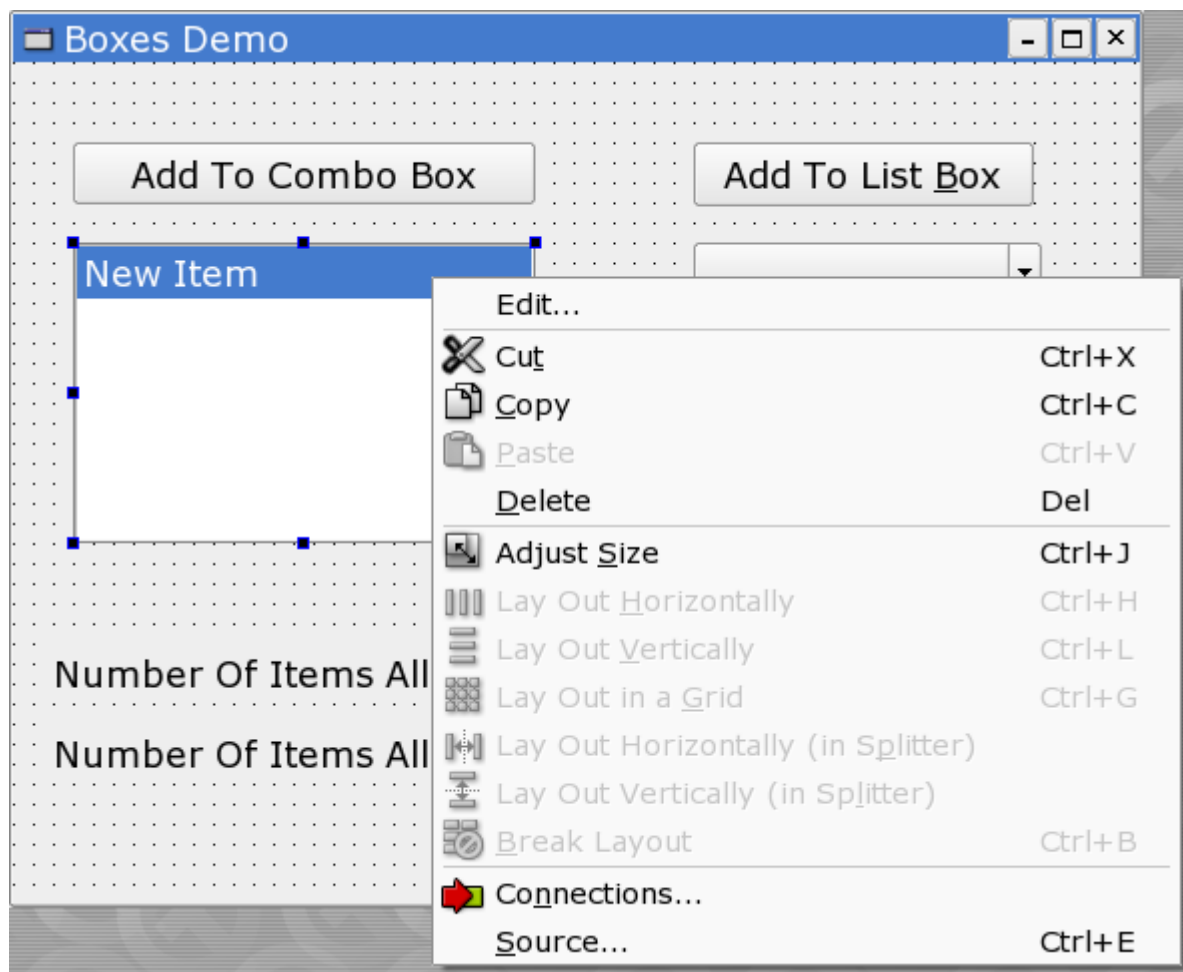
The ChapterThreeCommonWidgetsBoxes program looks at using the standard ComboBox and the standard ListBox provided by the Qt library as well as looking at the SpinBox.



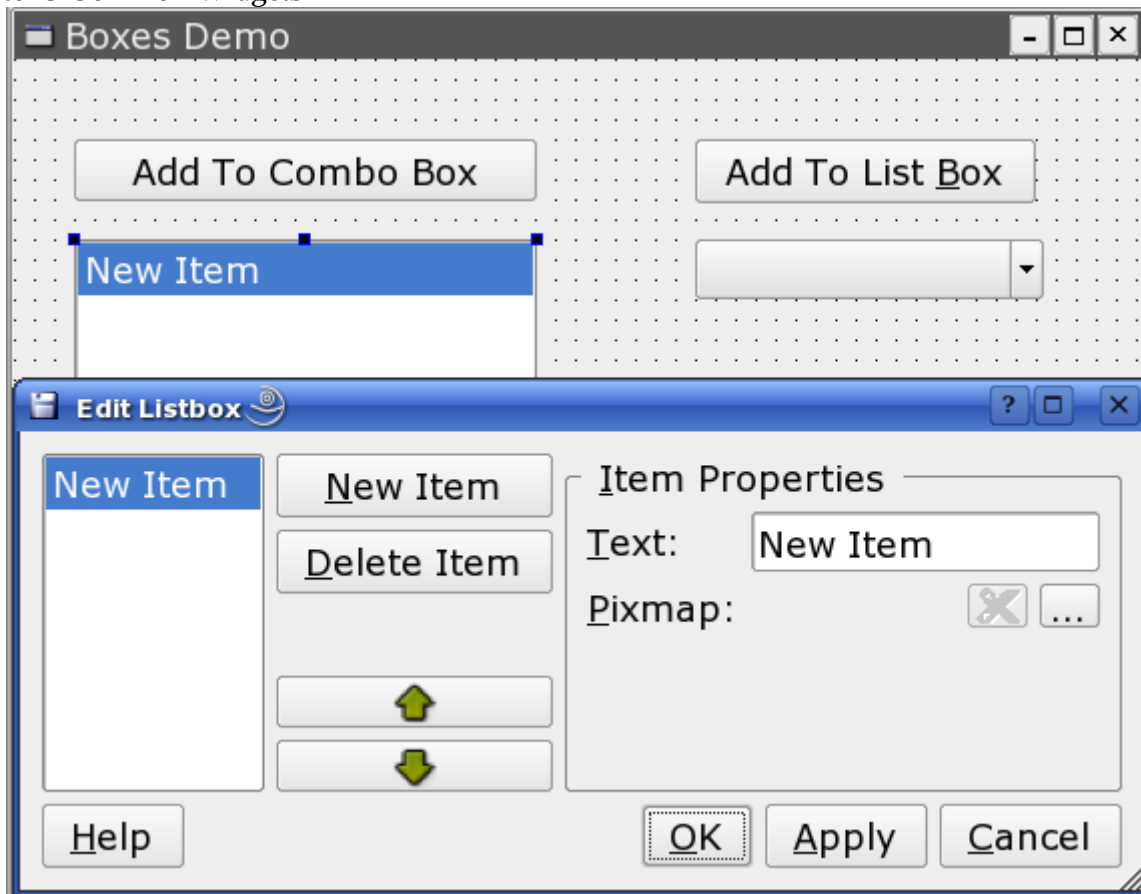
As you can see from the designer shot above the idea is that we have a simple design that we have a QComboBox and a QListBox on the same form and we copy items between them using the buttons provided while the QSpinBox's provide limits to the number of items allowed in both the Listbox and the QComboBox. The idea being to show examples of how things actually work rather than talking about how they should work.

First of all we need to add some items to the QListBox and the QComboBox we do this in the designer by right clicking on the widget and selecting edit.

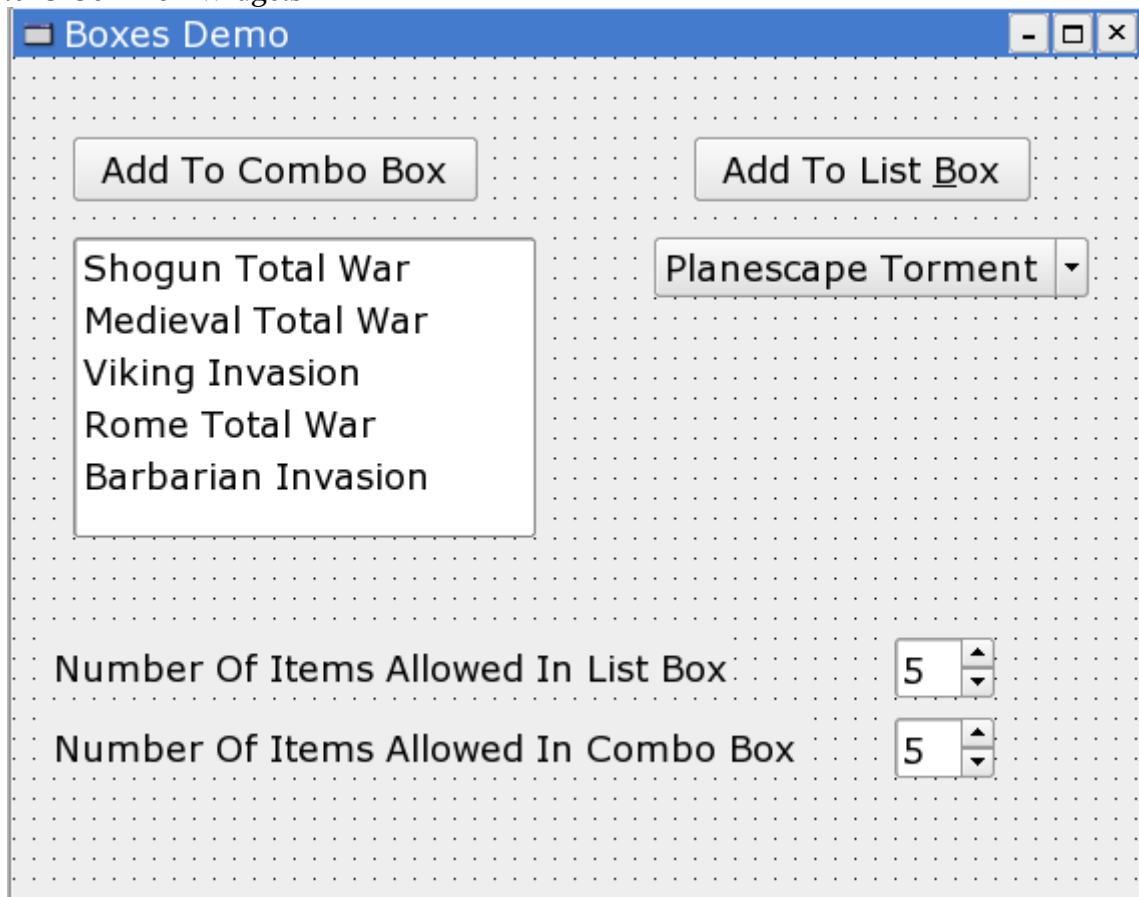
Chapter 3 Common Widgets



which brings up a dialog which we can enter data into.

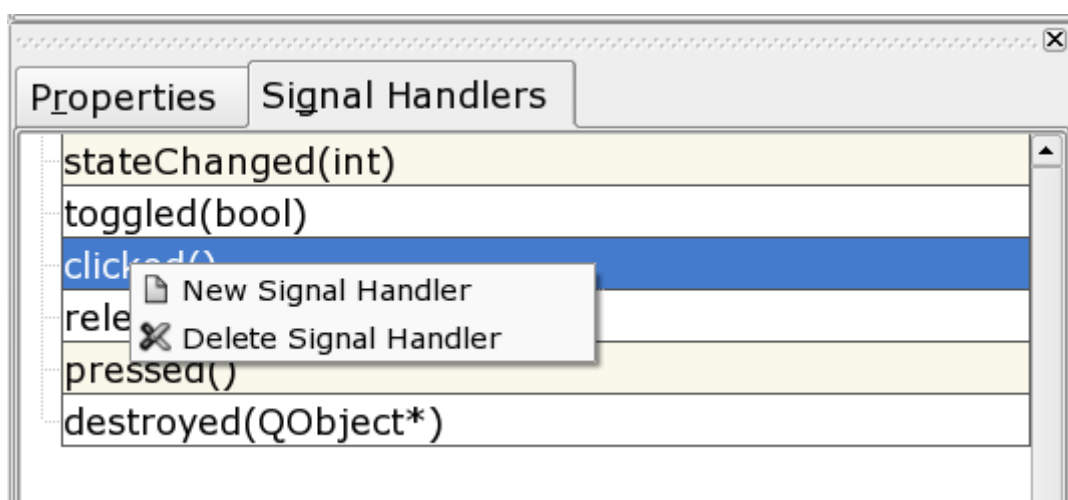


As you can see we can add either QPixmap's which means any image type that the QPixmap class can open or simple text items. For this demonstration we will be sticking to simple text items, simply because it is 1, easier and 2, I have no artistic talent whatever. Adding items to the QComboBox is done through exactly the same dialog. So let's add a few items.



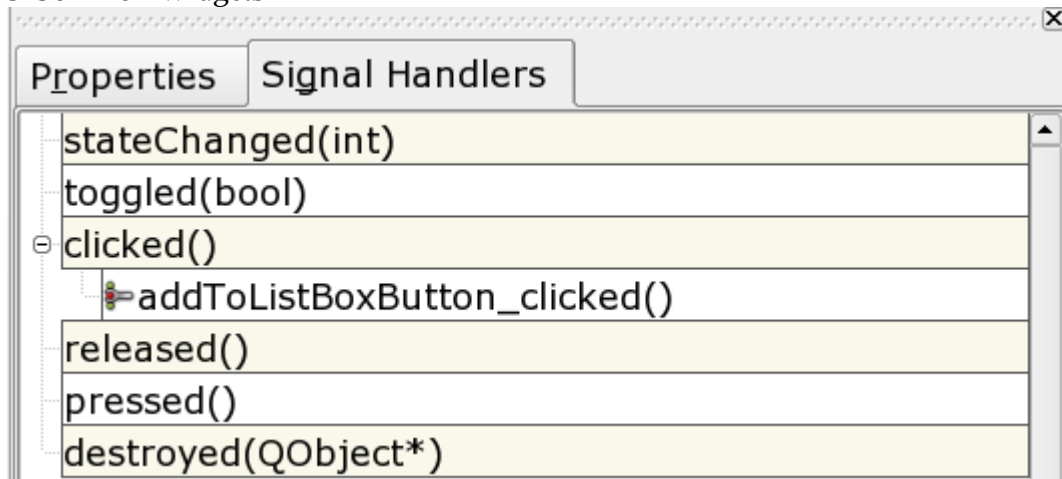
The items added are just some games that are usually lying around on my desk but for the purposes of this demo you could use any old text strings as we are only going to be moving them about it doesn't really matter what they say.

To get the program to work we are going to need to add a couple of signal handlers to the buttons.

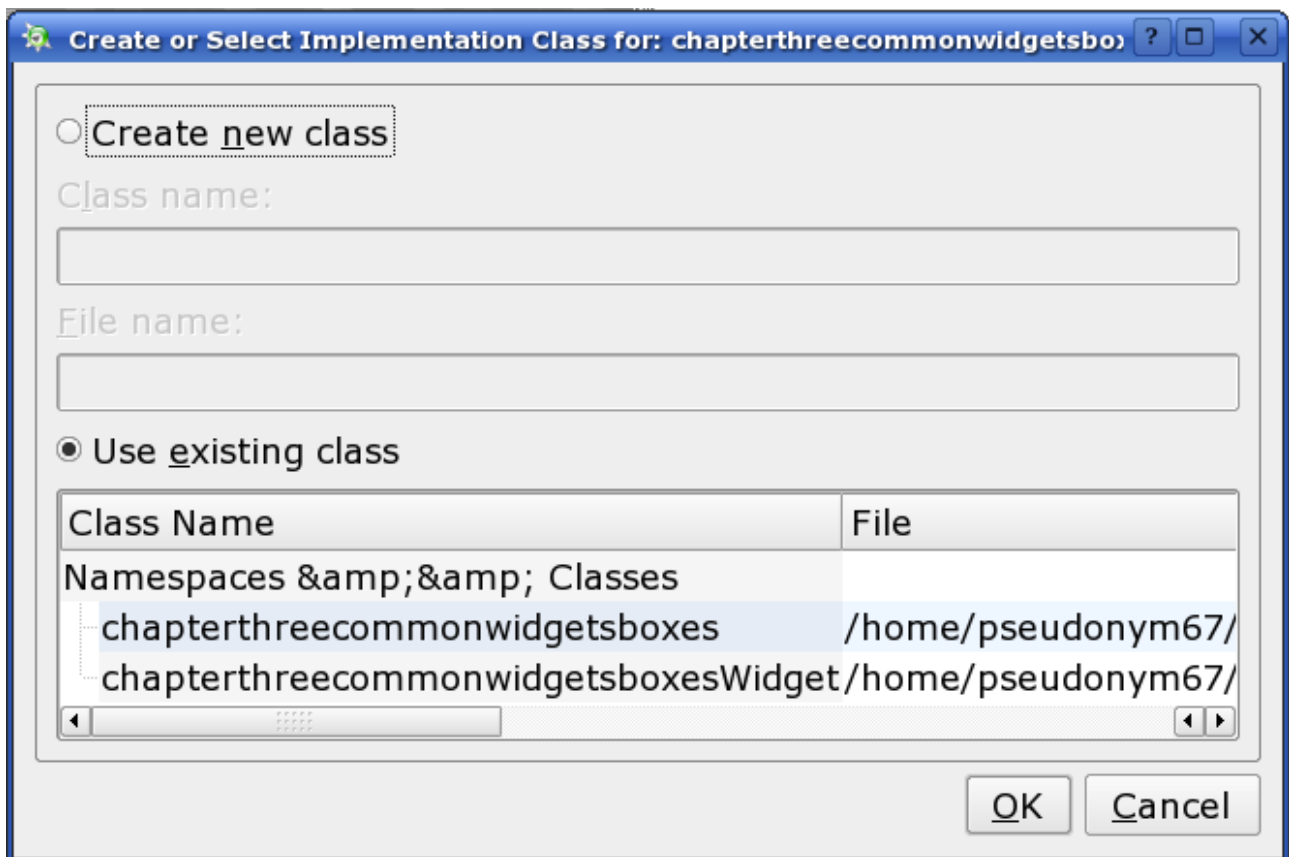


This is done by opening up the Signal Handlers tab on the Property Editor and right clicking on the clicked() signal. When you select the New Signal Handler the Editor will automatically add a signal called "buttonname"_clicked()

Chapter 3 Common Widgets



If you hit return when you add the signal handler you should be given a class dialog



when you get this you have the option to create a new class to handle the implementation of the slot. You are free to do this if you can think of a good reason why you need to do it but for the most part you should use the class that ends with Widget as this is already derived from the generated class in your debug/src directory. Once you have set this up KDevelop will automatically add any new Signal Handlers to the class chosen. So with this project you will end up with a chapterthreecommonwidgetsboxeswidget.h that contains

```
public slots:
    /*$PUBLIC_SLOT$*/
    virtual void addToComboBoxButton_clicked();
    virtual void addToListBoxButton_clicked();
```


Chapter 3 Common Widgets

```
virtual void numberOfItemsInListBox_valueChanged(int);  
virtual void numberOfItemsInComboBox_valueChanged(int);
```

with the corresponding function outlines in the .cpp file

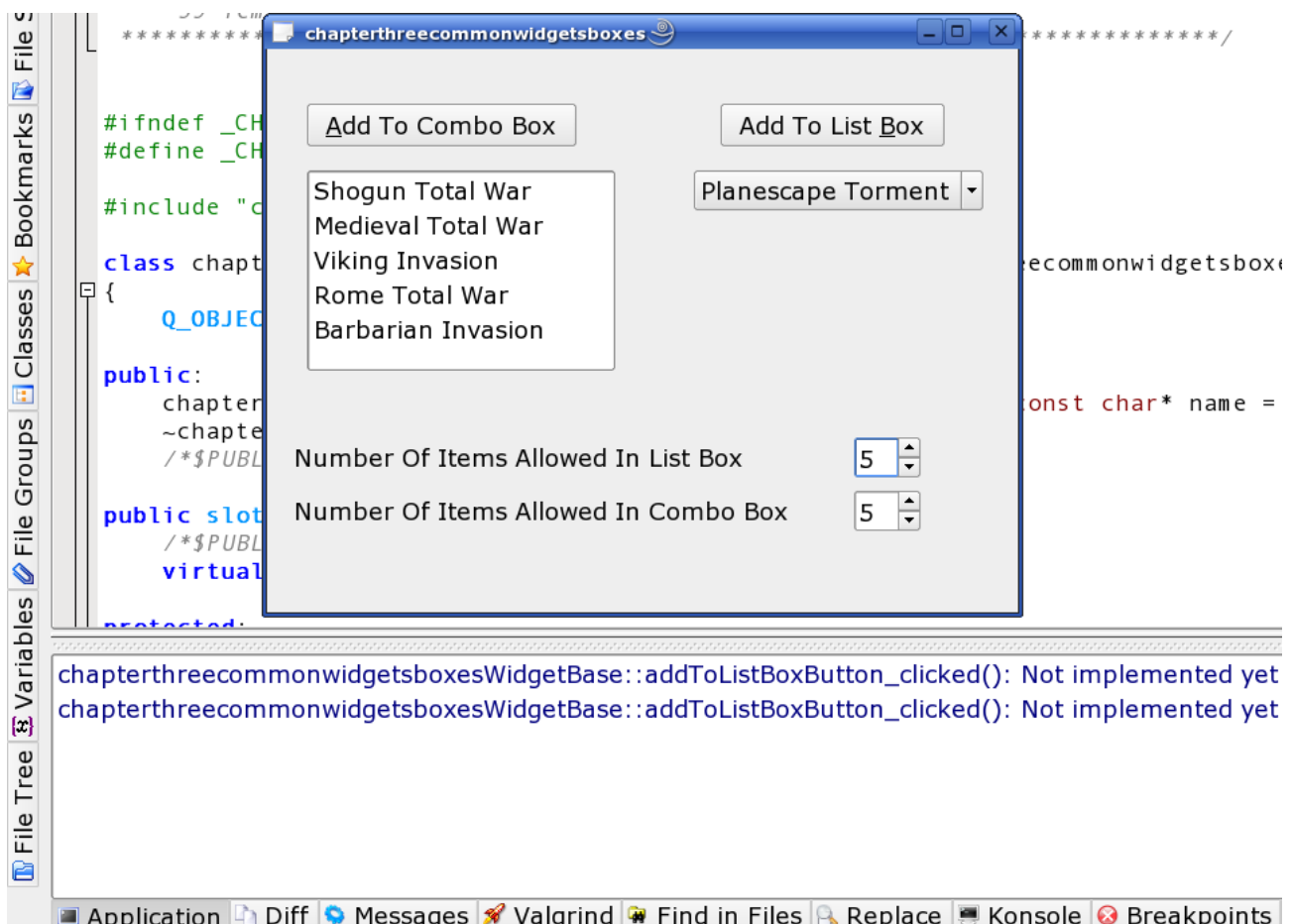
```
void chapterthreecommonwidgetsboxesWidget::numberOfItemsInListBox_valueChanged(int)  
{  
}
```

Of course if while this code will compile perfectly, if you want to access the variables passed in the valueChanged functions you will need to add a variable name for the int to both the .h and the .cpp file.

```
virtual void numberOfItemsInComboBox_valueChanged(int value);
```

```
void chapterthreecommonwidgetsboxesWidget::numberOfItemsInListBox_valueChanged(int value)
```

It is however very easy to miss or dismiss this dialog and you could find yourself having to implement the connections by hand. If dismiss the dialog and try to run the program what you get when you press the “Add To List Box” button is,



which is fair enough as we haven't done anything more than add the signal handlers at this moment in time. If you looked at the source code expecting to find that the error message was generated by chapterthreecommonwidgetsboxeswidget class but it isn't because by missing the dialog we haven't set up the connection between the two. If we go to the debug/src folder for the project and look in

Chapter 3 Common Widgets

the chapterthreecommonwidgetsboxeswidgetbase.h file we see,

```
public slots:
    virtual void button_clicked();
    virtual void addToComboBoxButton_clicked();
    virtual void addToListBoxButton_clicked();
```

but if we look at our chapterthreecommonwidgetsboxeswidget.h file,

```
public slots:
    /*$PUBLIC_SLOT$*/
    virtual void button_clicked();
```

which means that if we want to do anything with the slots that we have just added then we need to override them from the base class by hand.

```
public slots:
    /*$PUBLIC_SLOT$*/
    virtual void addToComboBoxButton_clicked();
    virtual void addToListBoxButton_clicked();
```

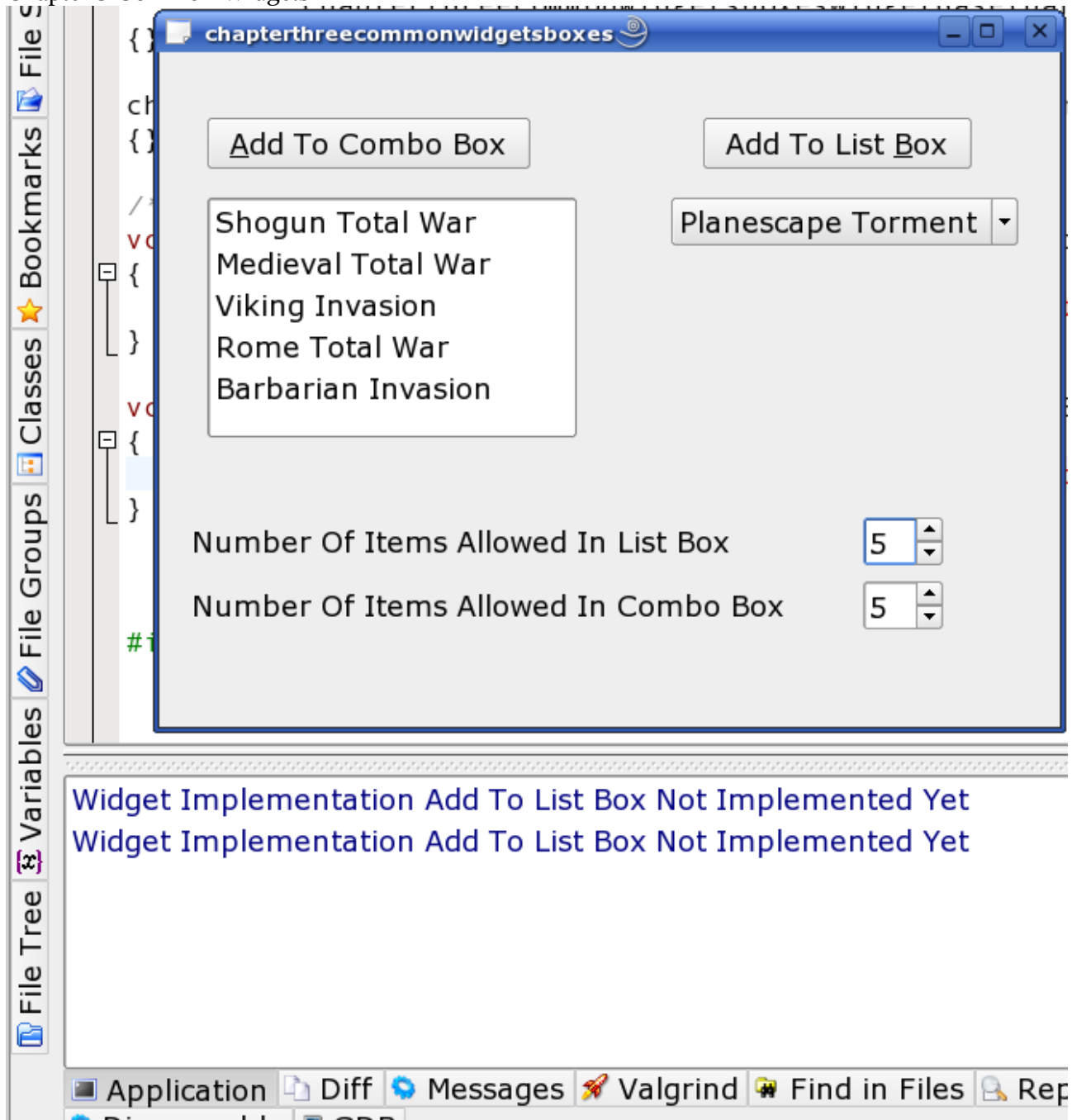
While we are there we might as well remove the now useless button_clicked() slot. And in the cpp file, add our own qDebug messages to make sure that we are getting the results we expect.

```
void chapterthreecommonwidgetsboxesWidget::addToListBoxButton_clicked()
{
    qDebug( "Widget Implementation Add To List Box Not Implemented Yet" );
}

void chapterthreecommonwidgetsboxesWidget::addToComboBoxButton_clicked()
{
    qDebug( "Widget Implementation Add To Combo Box Not Implemented Yet" );
}
```

which gives us,

Chapter 3 Common Widgets



Now we can add the code to move items between the two boxes

```
if( demoListBox->count() < numberOfItemsInListBox->value() )
{
    demoListBox->insertItem( demoComboBox->currentText() );
}
else
    QMessageBox::information( this, "Boxes Demo", "Maximum Items Allowed in ListBox" );
```

As you can see the code is very simple when the slot for `addToListBox` is called the code checks the number of items in the `QListBox` against the number of items in the `QSpinBox` that sets how many items can be in the list box. If there are less items in the `QListBox` than the value in the `SpinBox` then the currently selected item text from the `QComboBox` is added to the list box. If not a dialog is displayed.

The code for the `QSpinBox` is equally simple,

Chapter 3 Common Widgets

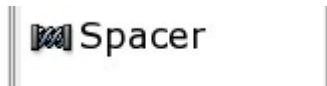
```
if( demoListBox->count() > numberOfItemsInListBox->value() )
{
    demoListBox->removeItem( demoListBox->count()-1 );
}
```

as long as you take into account the 0 based reference for C++ arrays even though the count value starts at 1 the final item in the QListBox or QComboBox is count -1 Although if you add the name of the integer variable passed to your function you don't need this,

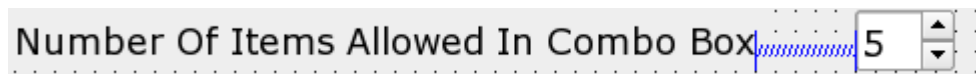
```
if( demoListBox->count() > value )
{
    demoListBox->removeItem( value );
}
```

Spacers

You can manually layout the form exactly the way that you want it through the use of spacers. which are the



Common Widgets tab. You insert the spacer between the two widgets that you want it to control.



You then select the group of items by drawing an outline around them with the mouse,

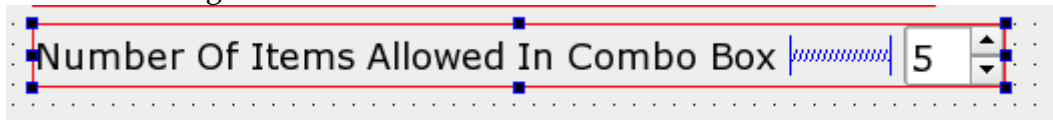


Once you let go of the mouse the entire group of widgets will be selected.

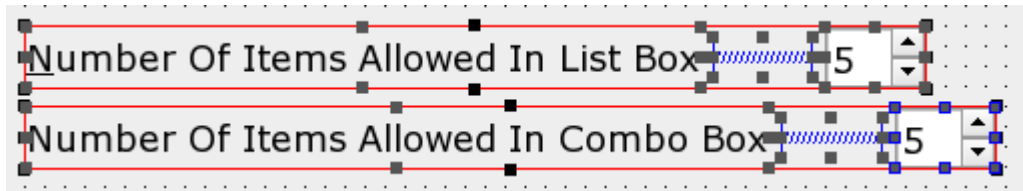


Now right click on the group and select the layout from the popup menu, in this case the horizontal layout and the group will be enclosed in a red square,

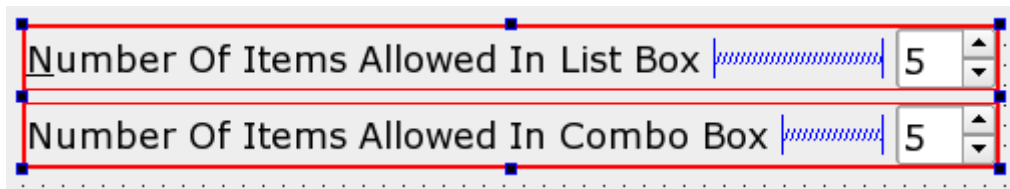
Chapter 3 Common Widgets



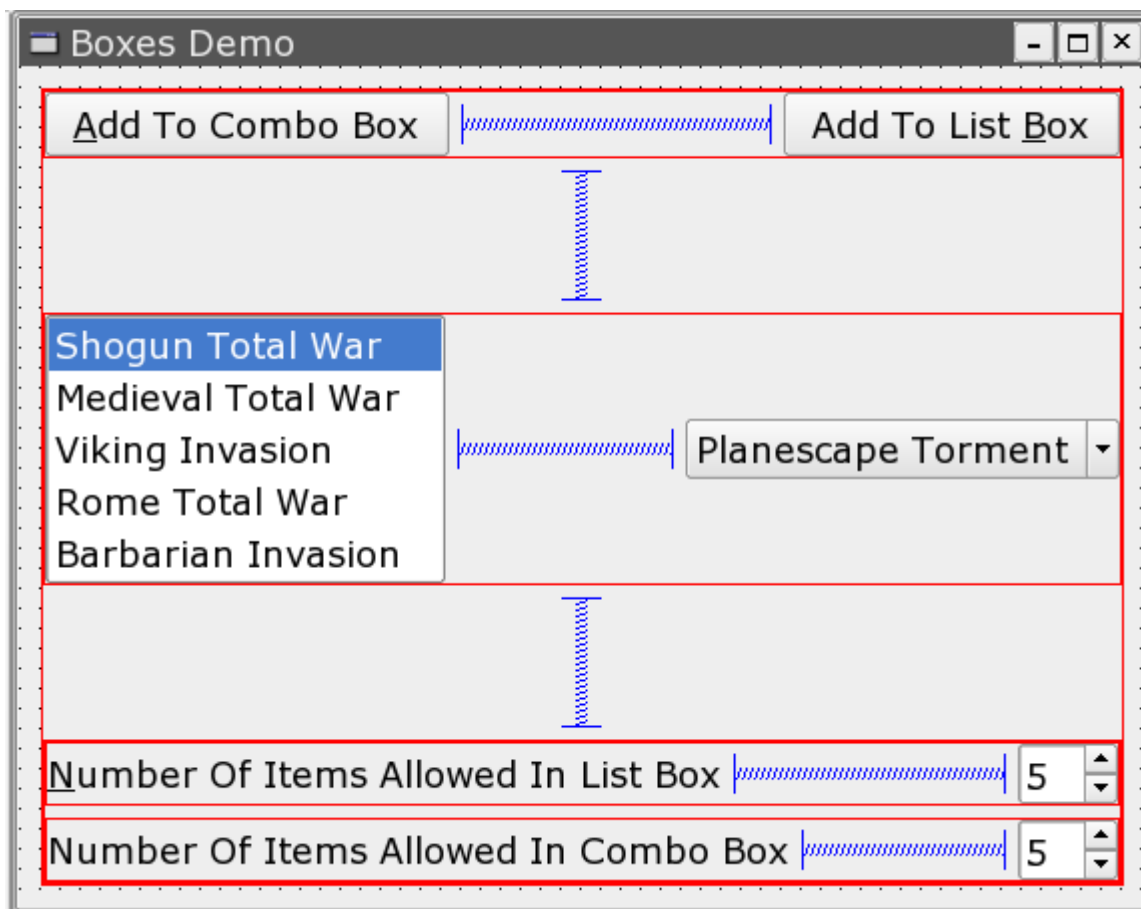
like so. You can also use then select groups and add layouts to them,



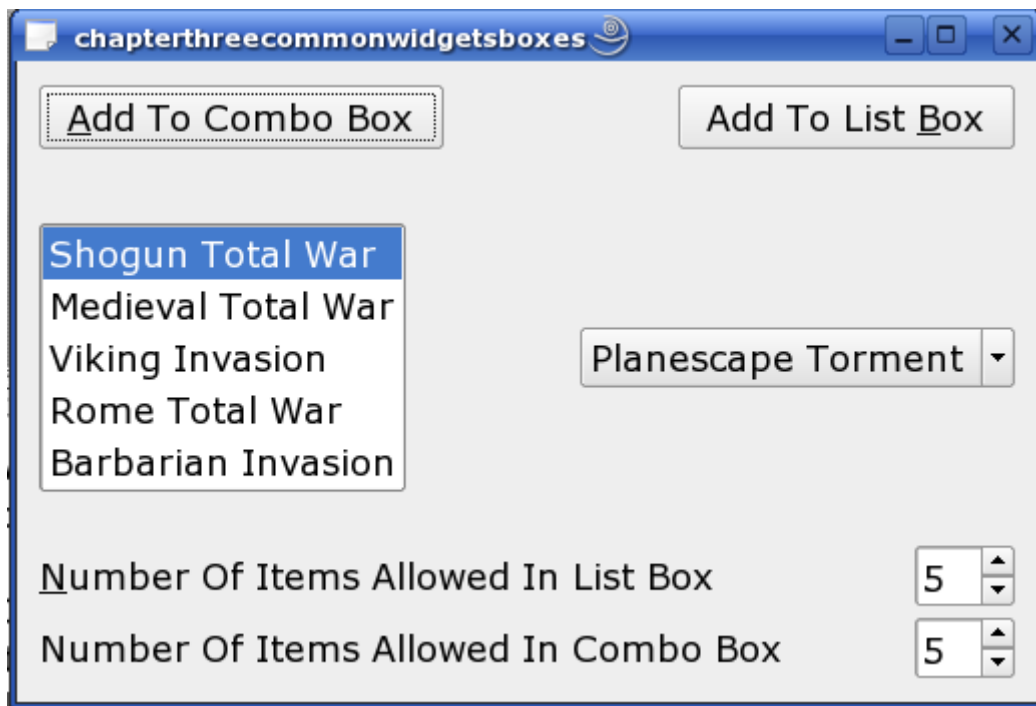
Right click on these and select a vertical layout.



If we go through the whole dialog we get something like this,



Chapter 3 Common Widgets
which will give us a running dialog of,



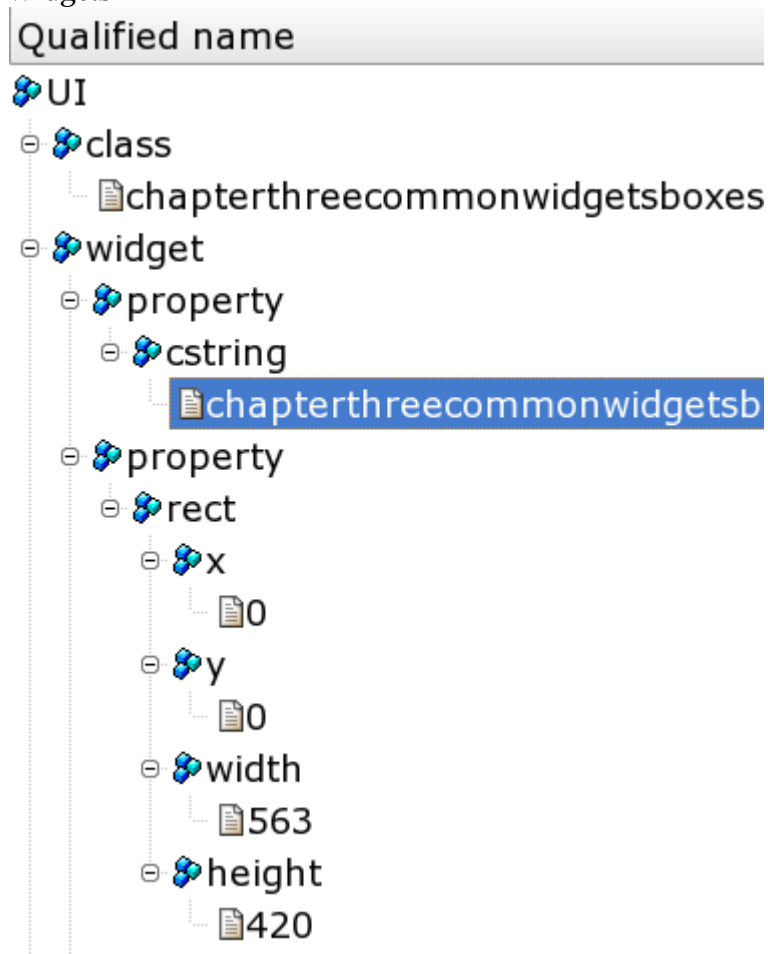
More On Layouts

It should be obvious from the layout of the chapterthreecommonwidgetsboxes program that it does not conveniently fall into any of the available layout patterns, which means that if you try imposing any of the layouts onto the form then it messes things up quite nastily, because the controls are not a uniform size. Spacers can be one solution to this but they can also be finicky and have a tendency to not always do what you think they are going to the first time round, which means when using them you always keep one eye on the undo button.

There are a couple of ways to deal with this the first is with Containers which we haven't covered yet and the second is to manually take control of the dialog sizing.

You can do this by going to src and opening the .ui file for the project with KXML Editor,

Chapter 3 Common Widgets



At the very top of the file you can see the layout properties for the form you are working on. The rect property shows the x and y screen coordinates for where the form will be displayed when the program is started and the width and height of the form. If you copy these values to the form properties, minimum size,

| Properties | | Signal Handlers | |
|----------------------|--|------------------------|--|
| Property | | Value | |
| + name | | etsboxeswidgetbase | |
| enabled | | True | |
| + sizePolicy | | Preferred/Preferred... | |
| + minimumSize | | [563, 420] | |

then the form will not be resized so that you can't see anything when you run the program in fact if right click the form and select Adjust Size it will not resize the form below the minimum size that you have specified.

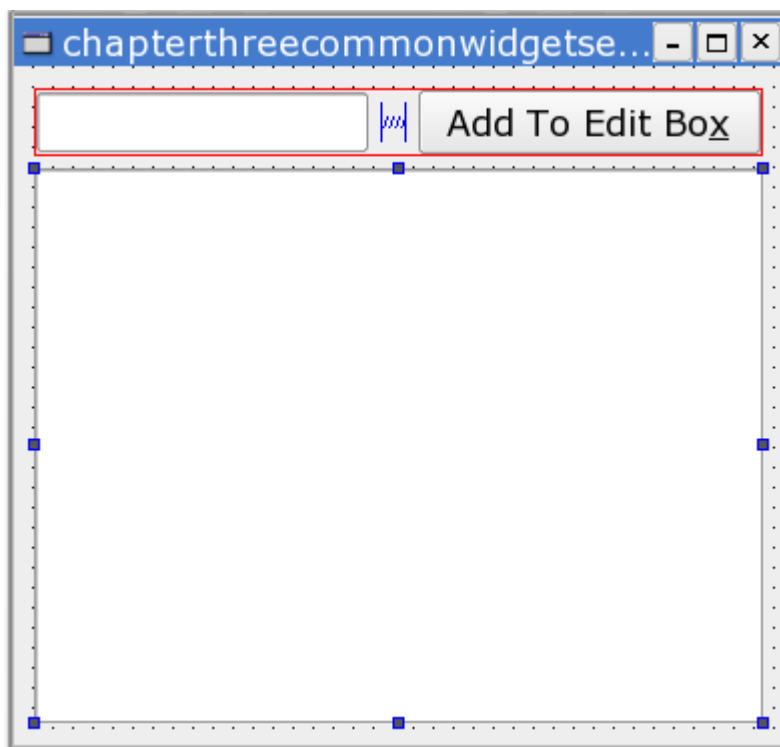
There is also the option to change the sizePolicy of the dialog in the properties which allows you to set a number of policies for the resizing of the dialog including a fixed size. This brings us to the

Chapter 3 Common Widgets

universal problem with layouts on any system and that is simply put that by saying do it in any one particular way I would expect to be inundated with emails saying “I used the dialog layout method you said to use and my dialog looks rubbish.” Basically, on any system it is always going to be trial and error to see which technique suits both the user and their project.

Edits

Create a new Simple KDE Designer application as described in earlier chapters and then delete the label and the button and save the ui file. Open the chapterthreecommonwidgetsedsits .h and .cpp files and remove the button_clicked slot. Save the .h and .cpp files, it might be a good idea to build the project here just to make sure you have gotten rid of all the references, as we don't want anything unexpected popping up later. Then add a QLineEdit, a QPushButton and a QTextEdit and arrange them something like this,



This project demonstrates the QLineEdit and the QTextEdit in a rather simple manner, as they behave in exactly the way you would expect them to in that the QLineEdit allows you to type in a single line of text and the QEditBox allows you to enter multiple lines of text.

The signal for the button is added through the properties tab Signal Handlers tab as described previously by selecting the button, then right clicking the clicked option and hitting return to bring up the the dialog that allows us to select the class to implement the slot. As recommended we are handling the slot in the projects widget class that is derived from the class generated by the Form Designer.

```
void chapterthreecommonwidgetsedsitsWidget::addToEditBox_clicked()
{
    QString strText = lineEdit->text();

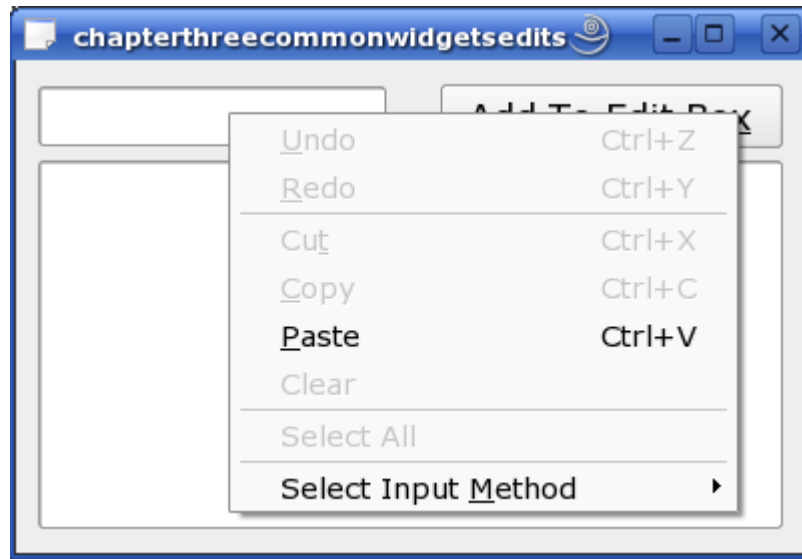
    if( strText.isNull() == false && strText.isEmpty() == false )
    {
        editBox->append( strText );
    }
}
```


Chapter 3 Common Widgets

```
}
```

The code isn't doing anything fancy at all. When the button is clicked we get the current text from the QLineEdit and test to see if it is valid as text. You could add a QValidator at this point if you required a specific text format, for our purposes though as long as the QLineEdit contains any text we append it to the editBox.

The Q*Edit classes contain all the functionality that you would require from a standard edit box in a windowed environment by default,



As you can see all the copy, cut and past functionality you would expect is already built into the widget.

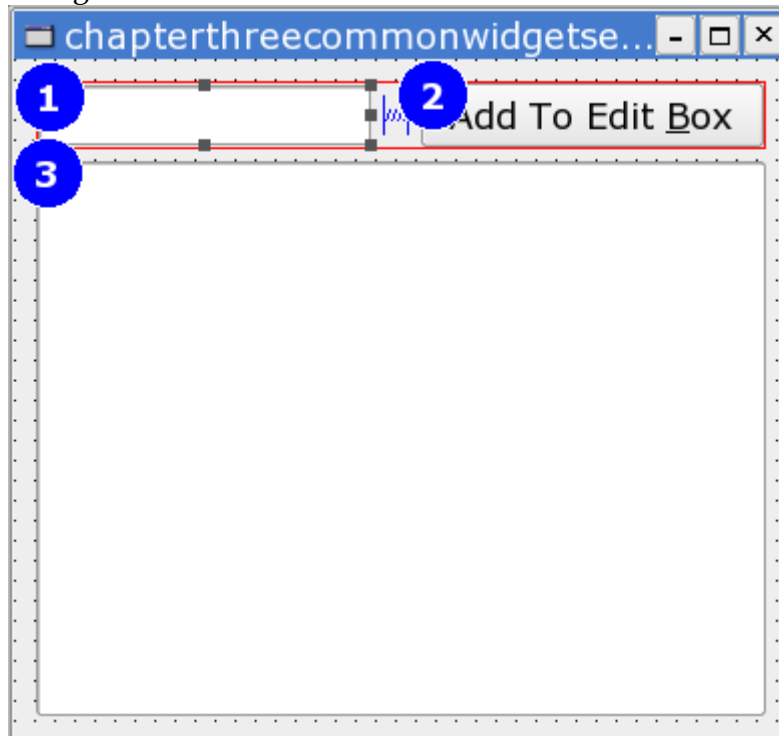
Tab Order

When using applications, especially ones where they are filling out forms people like to be able to use the Tab key to move around the form. You can specify the tab order of you application by clicking on the,



button, which should be on the KDevelop tool bar, Alternatively it is shown on the Tools menu and is accessible by pressing F4.

Chapter 3 Common Widgets



The widgets that you can tab to will be given a blue circled number as shown above and if you click on the circled one and then on the circled two the numbers and the tab order will be changed.

Summary

In this chapter we have looked at the common widgets provided by the KDevelop environment most of which will form the basic makeup of any application that runs in a windowed environment. We can now put together a basic application and lay it out the way we want it using a number of different layout methods and we have seen how to make the widgets on the form interact with each other through the use of the signals and slots mechanism.

Chapter 4 Containers And Views

In this chapter we will look at the next three tabs in our Gui toolbox, these being the Buttons which contains one button that we haven't seen yet and the Containers and Views tabs.

The Tool Button

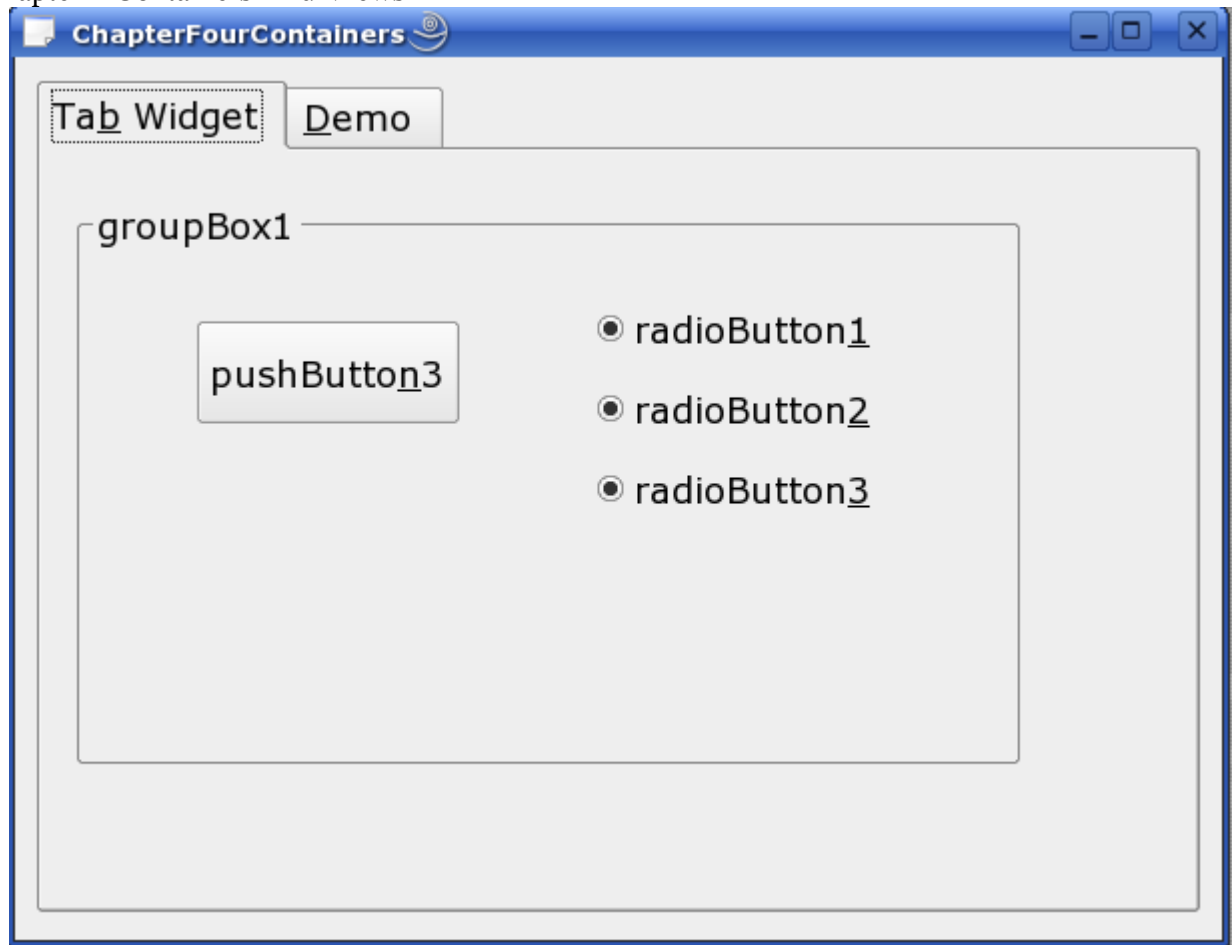
The QToolButton widget is more of a design guideline than an actual new widget implementation in that it is designed with the idea that it is used for tools and configuration options, hence the name. It is shown in the workspace as a button with the picture of a spanner head and when you place it on the form it has a text of ... The idea here is that you add an icon to the QToolButton and then display it without text.



The button on the left above is the QToolButton while the button on the right is the standard QPushButton. The QToolButton is capable of displaying either an icon or text whereas the QPushButton can display both icons and text at the same time. The QToolButton is used mostly on toolbars and dialogs where it is used to setup some specific options.

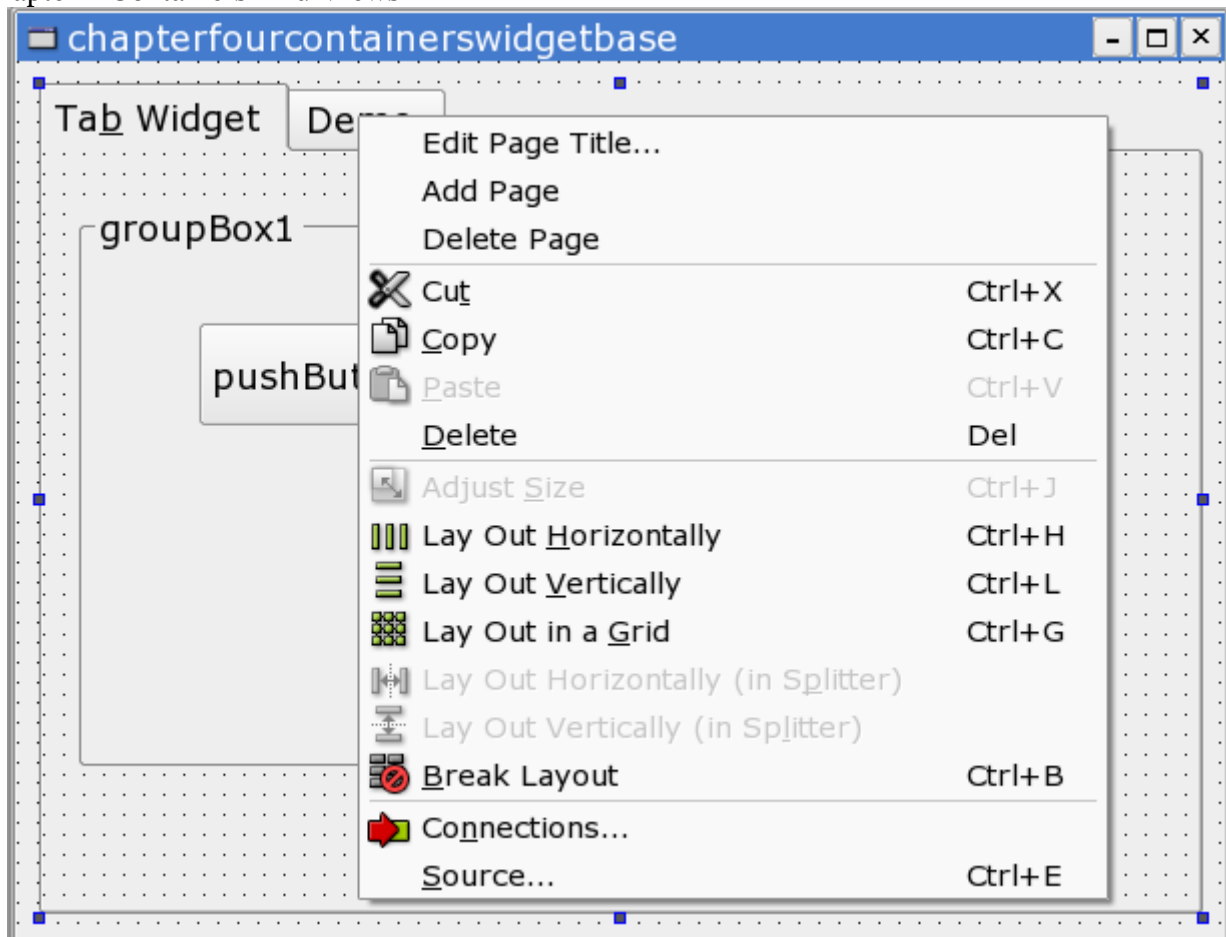
The Tab Widget

The QTabWidget class displays pages on a form in an organised way through the use of tabs at the top of each page. Customarily each page will have a name and they will be used for configuring a program before assigning it a specific task with each page being assigned a certain common area of operability ie one tab page may contain page options for a printer while another may contain control options for the printer hardware.

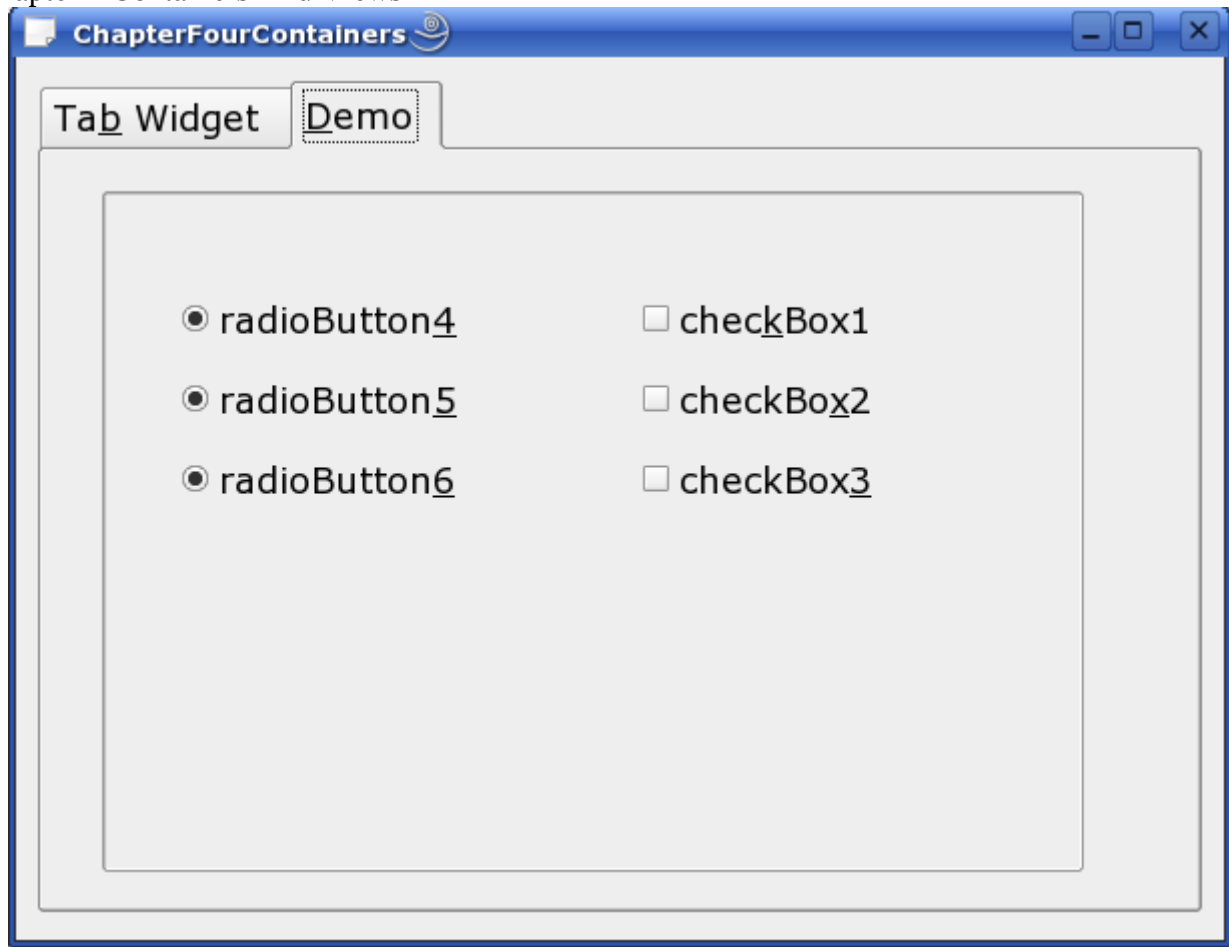


In the example we can see a `QTabWidget` with two tabs and the first tab is on display. The first tab contains a `QGroupBox` widget which has a button and a number of `QRadioButtons`. You can see here that unlike the `QButtonGroup` class the `QGroupBox` class doesn't control the `QRadioButtons` for you. This gives the `QGroupBox` a purely presentational role in that you can have related items under a collective header within the form but it adds nothing else.

Using properties in KDevelop you can change the appearance of the tabs slightly and have them appear at the bottom or the top of the page, although you can't have both and the most commonly used position is the top of the page. To add or edit the tab right click on the widget for the menu.



The top three items of the menu apply to the QTabPage widget. The page title is the text that is seen on the top of the tab page and you can add or delete pages as required.

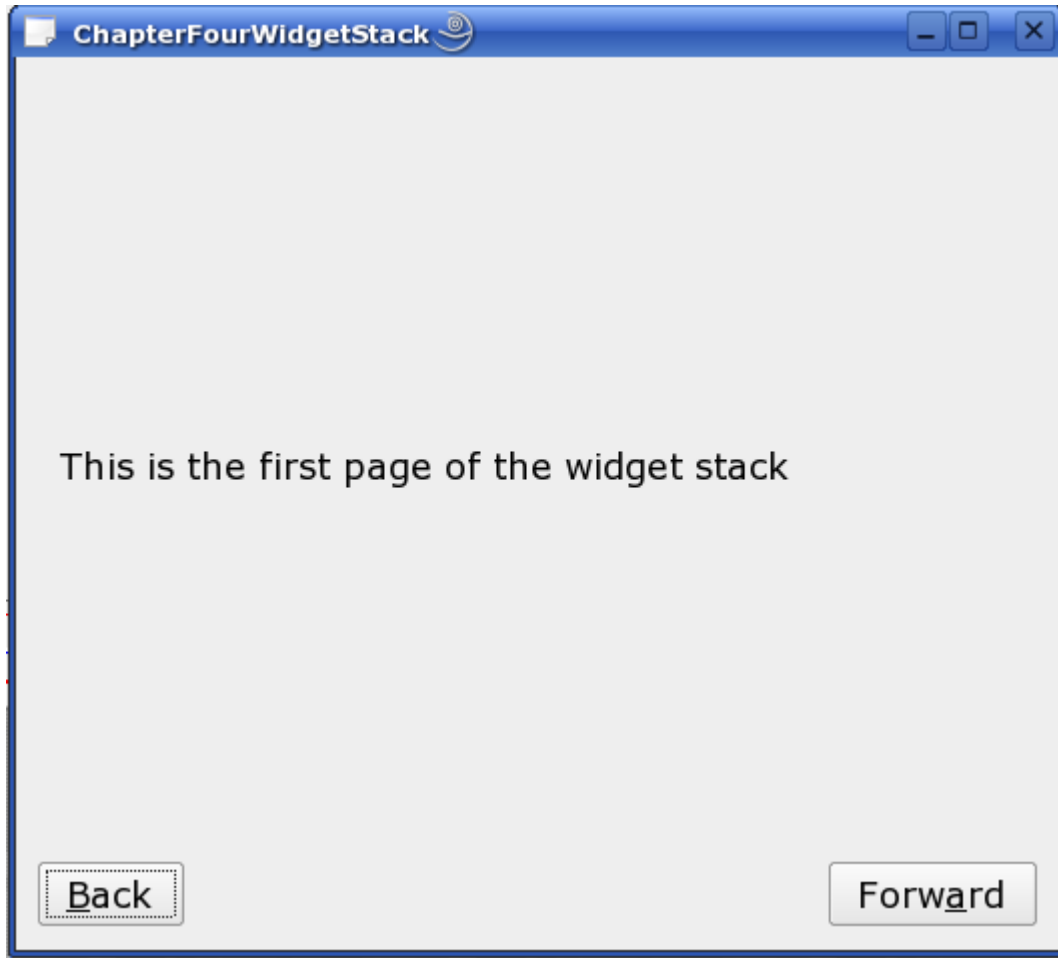


With the second tab we can see a frame and once again a few random buttons. The `QFrame` class is the base class of all the container classes and can be used to organise the layout of widgets within the form but does not provide any text options as a group header nor does it control the selection of radio buttons as the `QButtonGroup` class does.

The Widget Stack Widget

The `QWidgetStack` displays a user configured number of pages with no discernable page selection and it is usually implemented with forwards and back (next and prev) buttons. It is basically a wizard widget where the user can select options and progress steadily through the pages rather than being able to jump from page to page as they can with the tab widget or the tool box widget.

Chapter 4 Containers And Views



The application has three pages each containing a QLabel with some text which are moved through with the following code,

```
void ChapterFourWidgetStackWidget::forwardButton_clicked()
{
    if( widgetStack->widget( ChapterFourWidgetStackWidget::widgetPageOne ) ==
widgetStack->visibleWidget() )
    {
        widgetStack->raiseWidget( ChapterFourWidgetStackWidget::widgetPageTwo );
        return;
    }
    if( widgetStack->widget( ChapterFourWidgetStackWidget::widgetPageTwo ) ==
widgetStack->visibleWidget() )
    {
        widgetStack->raiseWidget( ChapterFourWidgetStackWidget::widgetPageThree );
    }
}

void ChapterFourWidgetStackWidget::backButton_clicked()
{
    if( widgetStack->widget( ChapterFourWidgetStackWidget::widgetPageTwo ) ==
widgetStack->visibleWidget() )
    {
        widgetStack->raiseWidget( ChapterFourWidgetStackWidget::widgetPageOne );
    }
    if( widgetStack->widget( ChapterFourWidgetStackWidget::widgetPageThree ) ==
widgetStack->visibleWidget() )
    {
        widgetStack->raiseWidget( ChapterFourWidgetStackWidget::widgetPageTwo );
    }
}
```

Chapter 4 Containers And Views

The widget pages are added to the QWidgetStack with the code,

```
widgetStack->addWidget( WStackPage, 0 );
```

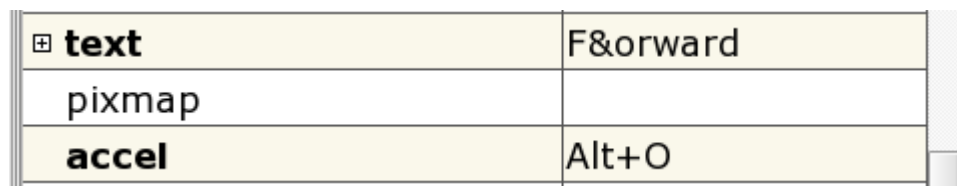
in the generated “projectname”widgetbase.cpp file and rather than use magic numbers I added an enum to the ChapterFourWidgetStackWidget class,

```
enum WidgetPages{ widgetPageOne, widgetPageTwo, widgetPageThree };
```

so if moving forward the code checks if the currently visible page is widget page one using the visibleWidget function and the widget(int) function which takes the id of the widget to retrieve. If the two match then raiseWidget(int) is called with the required id to give the widget we want.

Accelerators

In the picture and if you run the application you can see that the Back and the Forward buttons have an underlined letter this is the accelerator for the button. An accelerator is the keyboard shortcut that the user can use so that they don't have to use the mouse to operate the application. Accelerators are set by putting an & in front of the letter that you want to use as the keyboard shortcut when typing the text into the properties tab.



| | |
|---------------|----------|
| ⊕ text | F&orward |
| pixmap | |
| accel | Alt+O |

you can also set it explicitly in the accel section. Accelerators are part of the guidelines for what makes a good application so if you are going to publicly release your application they are required.

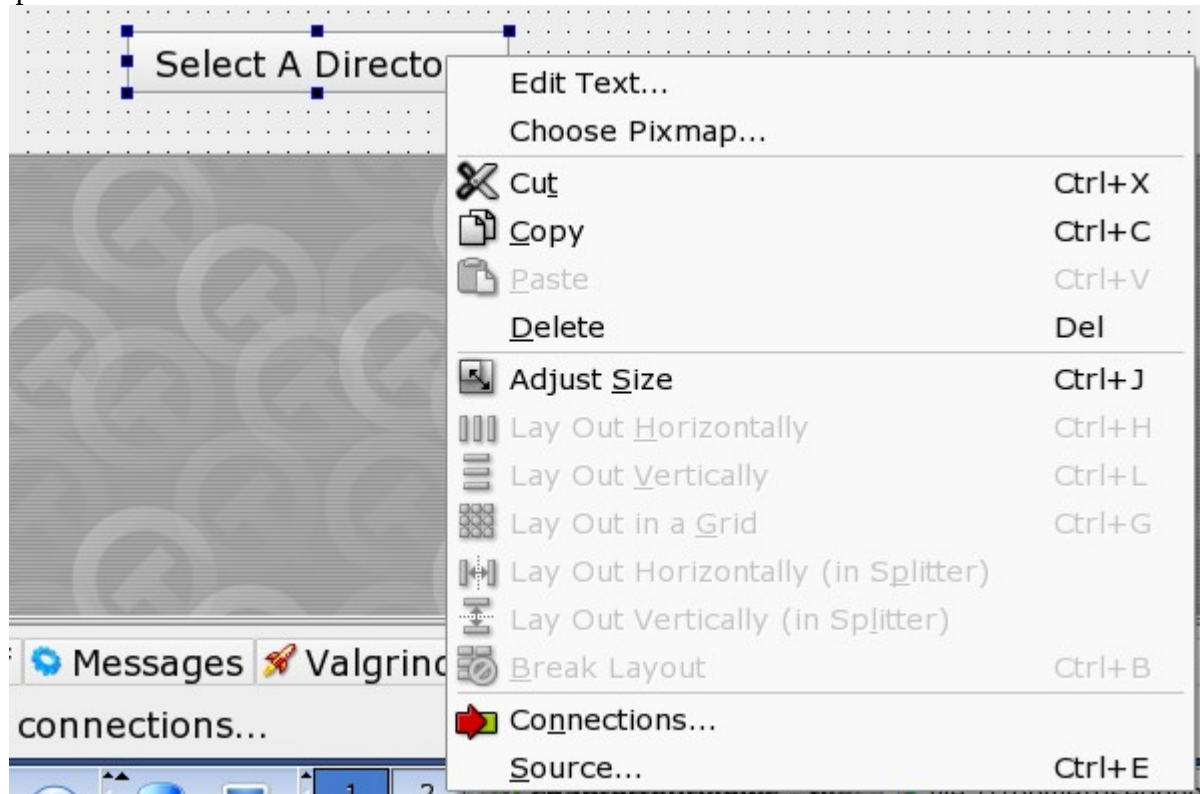
Views

In the next example we look at the remaining containers and the views tab of the toolbox and see how to fill each one with data, which we do by setting up a Simple Designer based KDE Application as described earlier and draw the widgets on the form, but first,

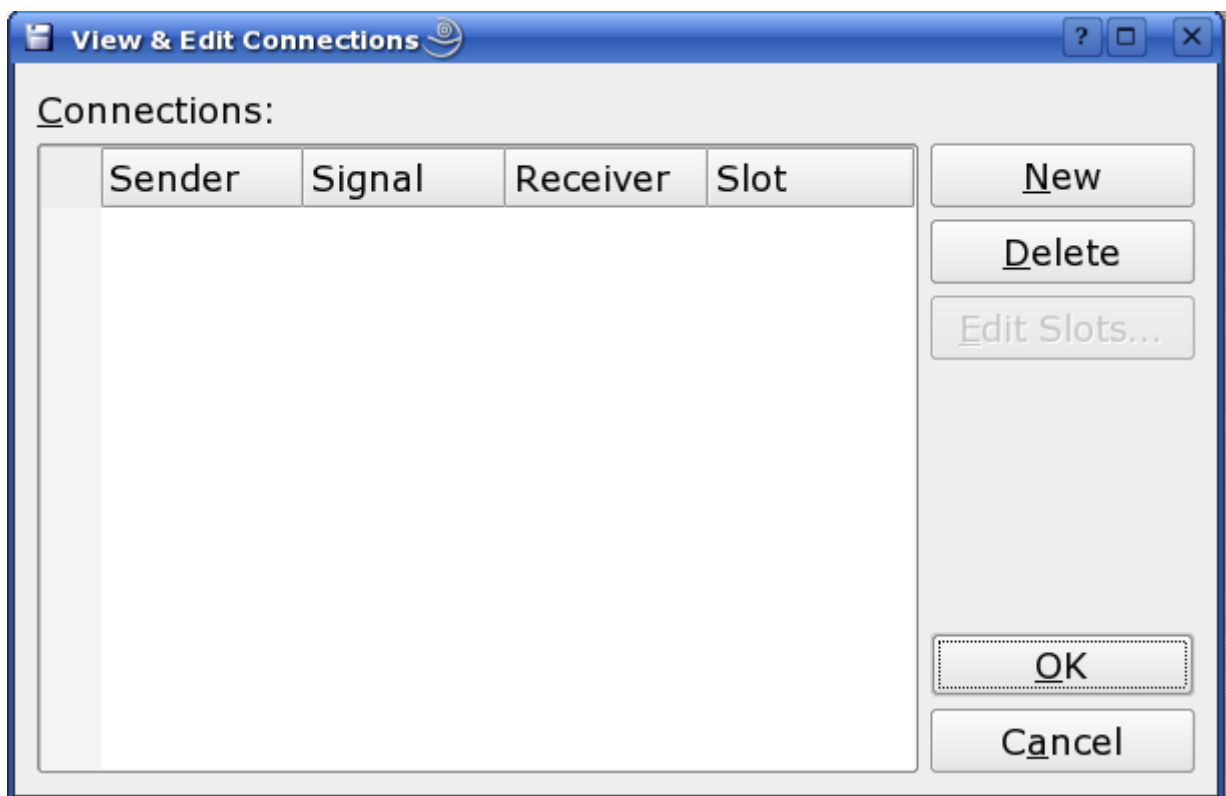
Graphical Connections

It is also possible to use KDevelop to add Signals and Slots to the program by using the Connections menu if we right click on the form,

Chapter 4 Containers And Views



This will bring up the dialog,

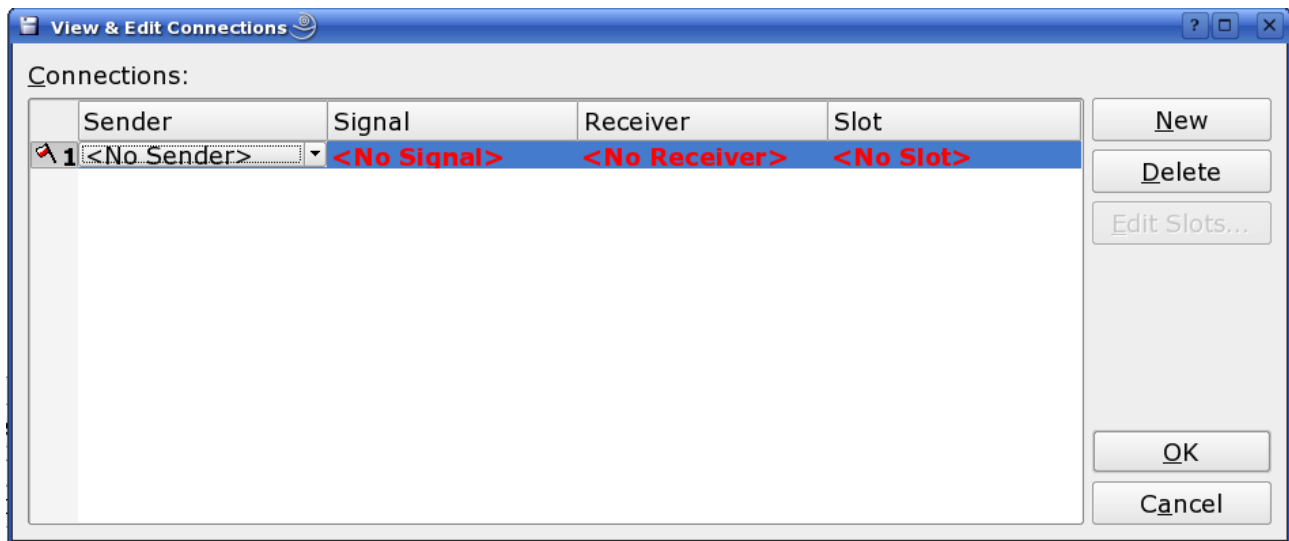


This gives us a way to add the four parameters that we need for a call to connect which if you remember,

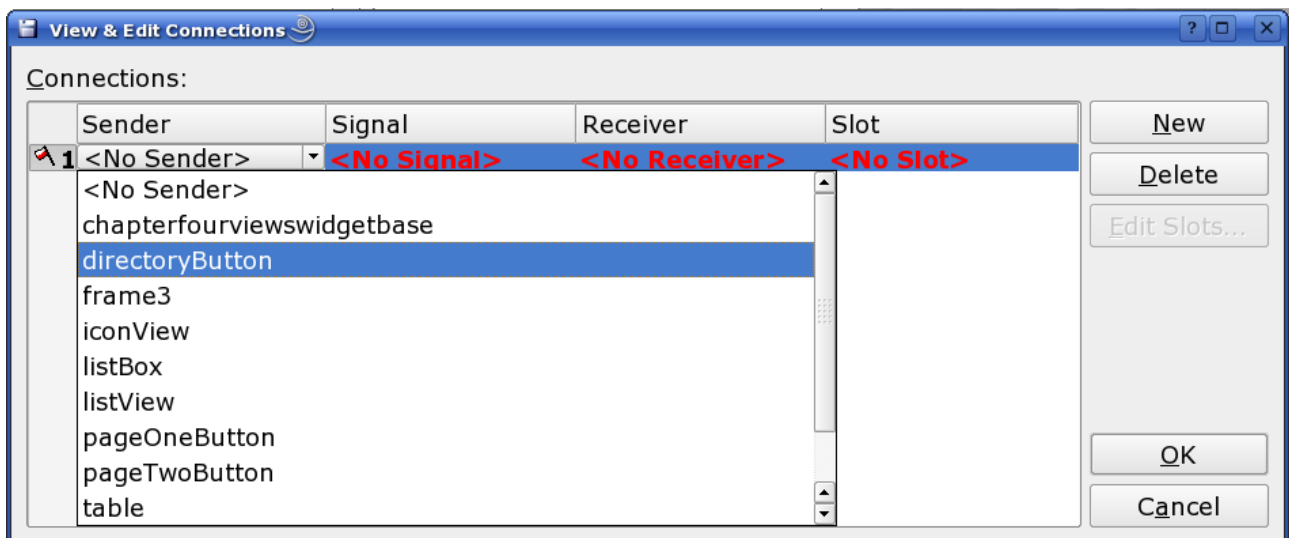
Chapter 4 Containers And Views

```
connect( pageOneButton, SIGNAL( clicked() ), this, SLOT( pageOneButton_clicked() ) );
```

has the format of parameter one is the object emitting the signal, parameter two is the signal to be emitted, parameter three is the object receiving the signal and parameter four is the slot or function that will respond to the signal.

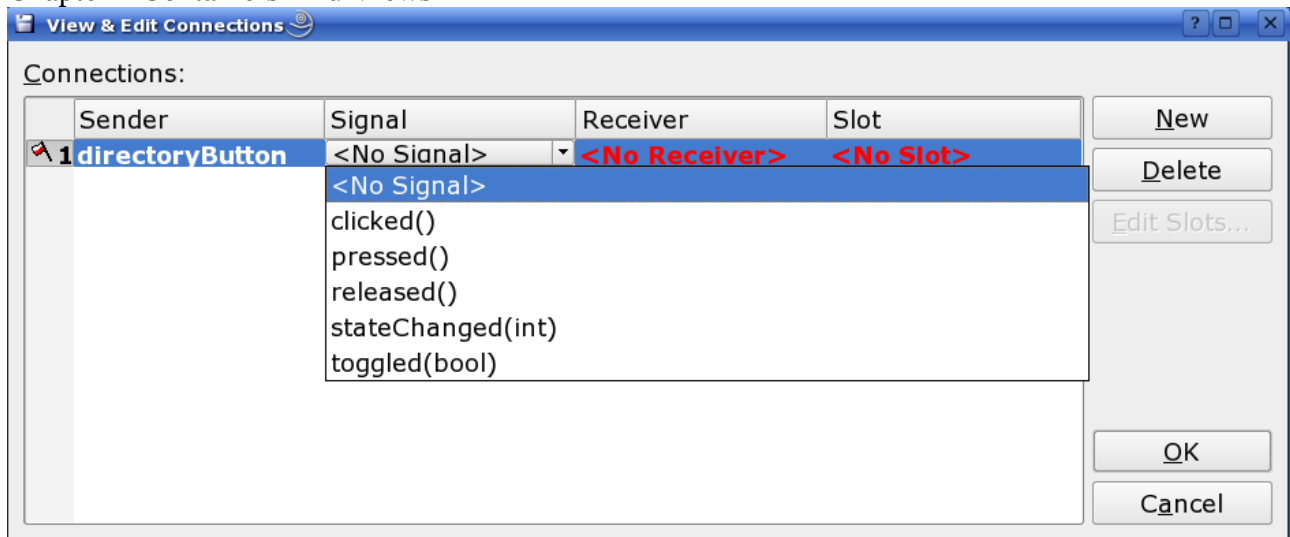


If you click on the New button then the dialog inserts a line of QComboBoxes that allow you to select the widgets and functions. The Sender is,

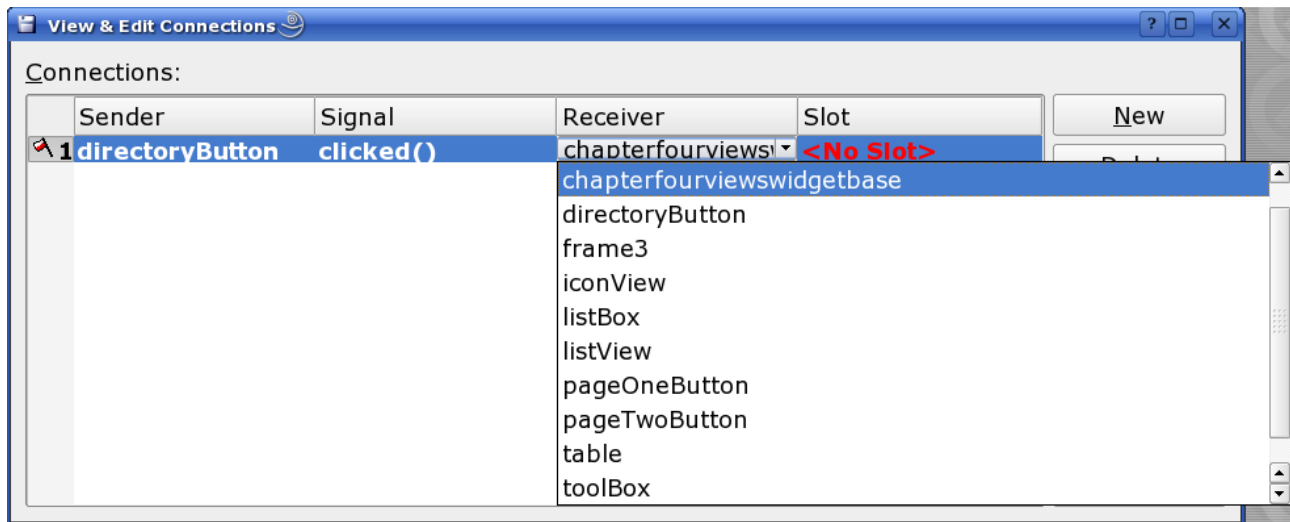


the object emitting the signal and in this case it is the directoryButton object, The Signal is,

Chapter 4 Containers And Views

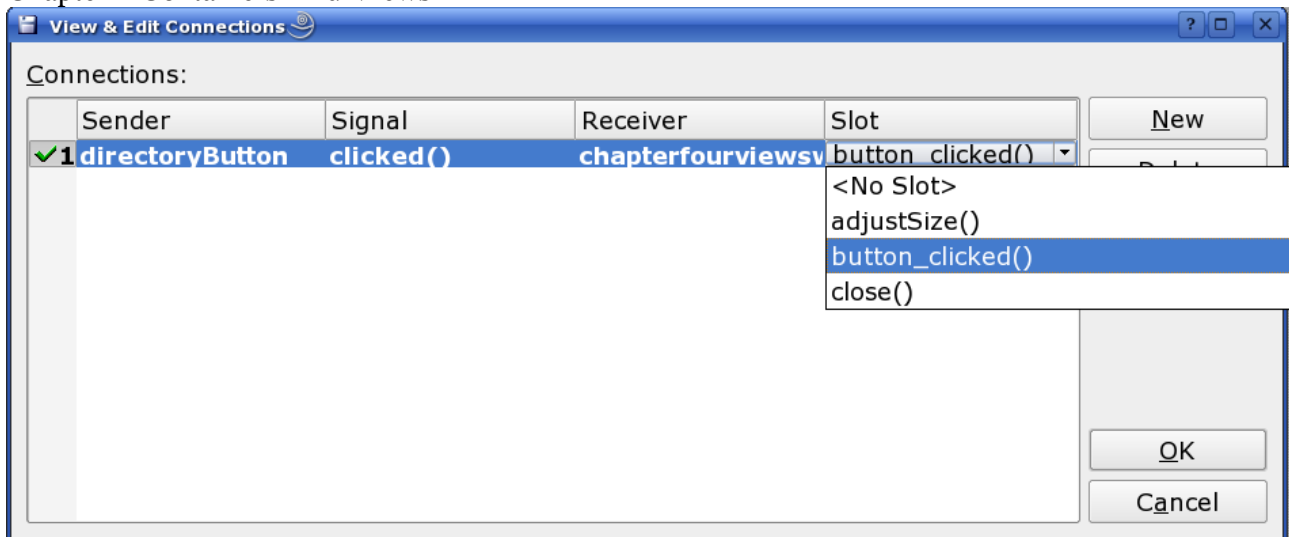


one of the standard signals emitted by the QPushButton class. The receiver,

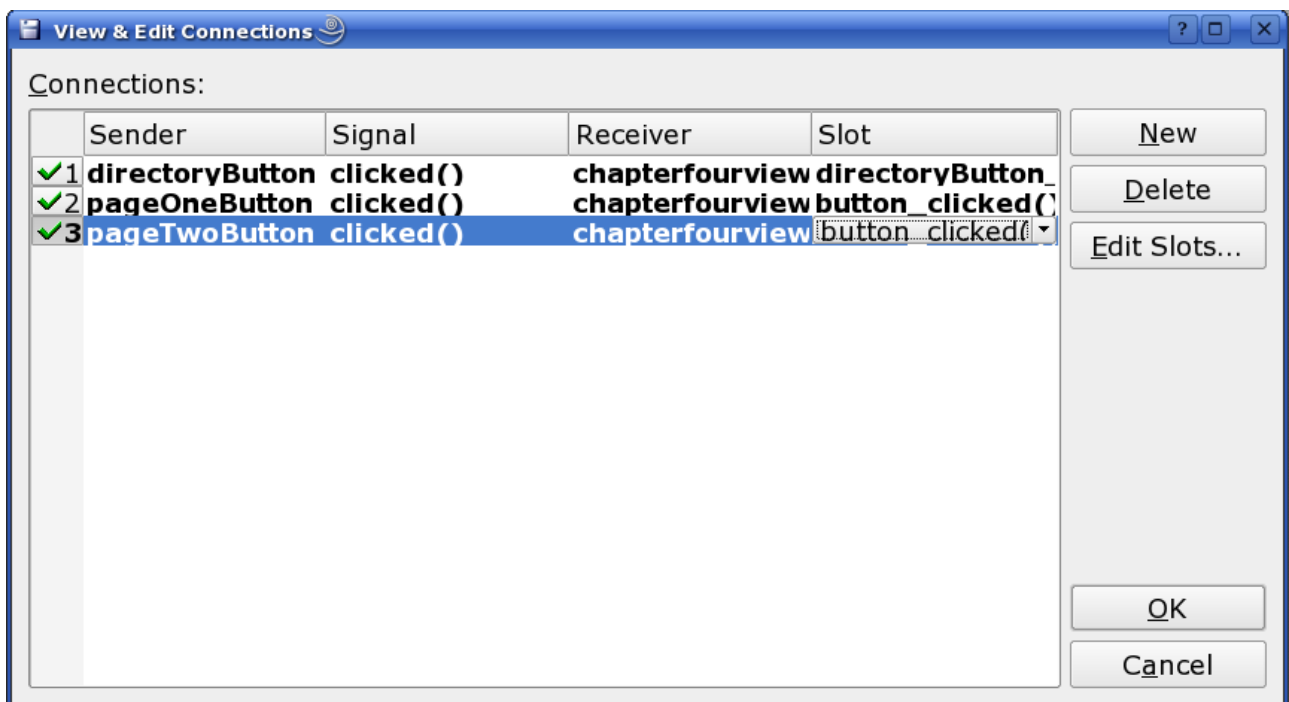


can be anyone of the widgets within the program, which is one of the reasons you should be cautious when using this method as it only takes a slight lack of concentration to mess up your program here, if you start trying to send signals to classes that you don't want receiving them. The slot to handle the signal is,

Chapter 4 Containers And Views



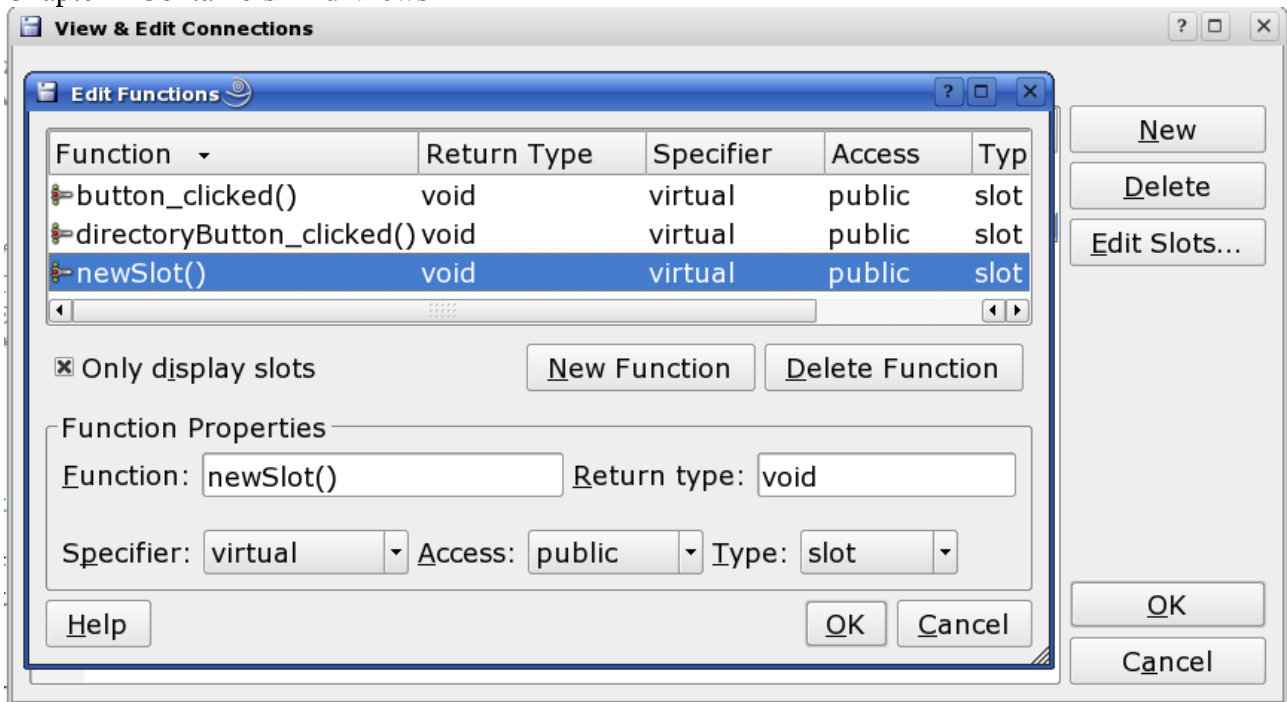
and this is where we can get ourselves into trouble. If you think that this is a wizard that is going to create a slot for you that responds to the directoryButton widget being pressed then you are just about as wrong as you could be and are going to end up with a dialog that looks something like this,



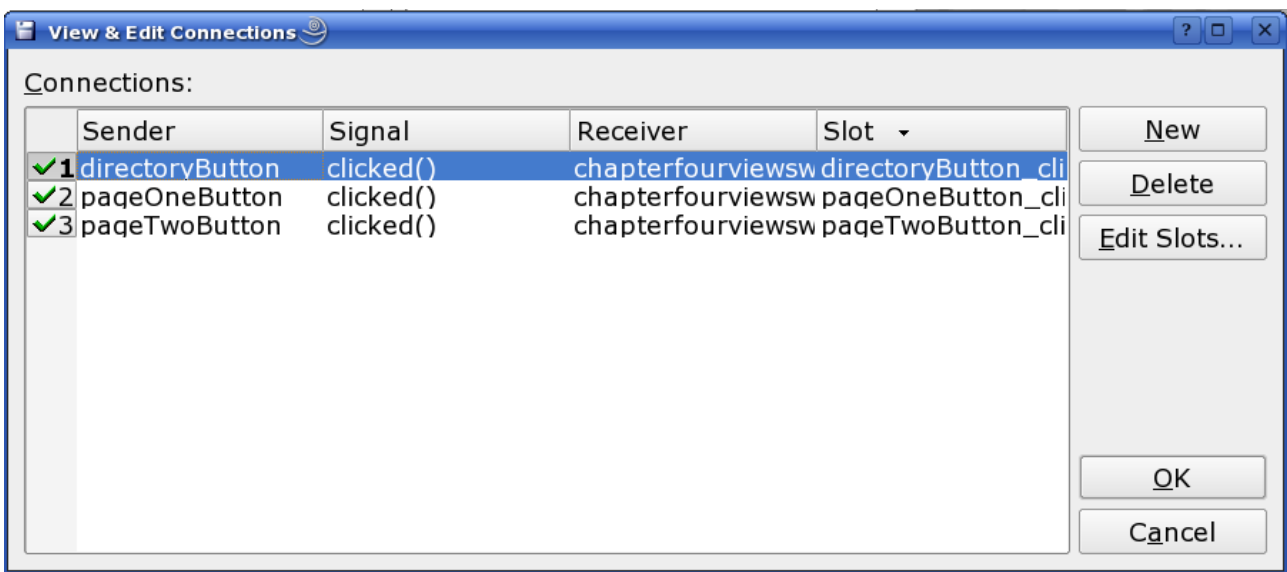
what you actually have here is a pair of QButtons that are emitting signals when clicked but both of them are being directed to the same button_clicked slot which in this case is the button_clicked slot that was left behind from the automatically generated code when the project was created.

What the Slot is showing when you look at it is the slots that are already defined, unlike when you create a connection by right clicking on the Signal Handlers tab in the Properties box earlier, here you are just selecting one of the available slots. To add a new slot you need to click on the Edit Slots button,

Chapter 4 Containers And Views



here you can add the slots and functions that you want to implement in your code. It is a matter of personal preference if you use this method of connecting signals and slots but when you open the View and Edit Connections dialog you want to see something like this,



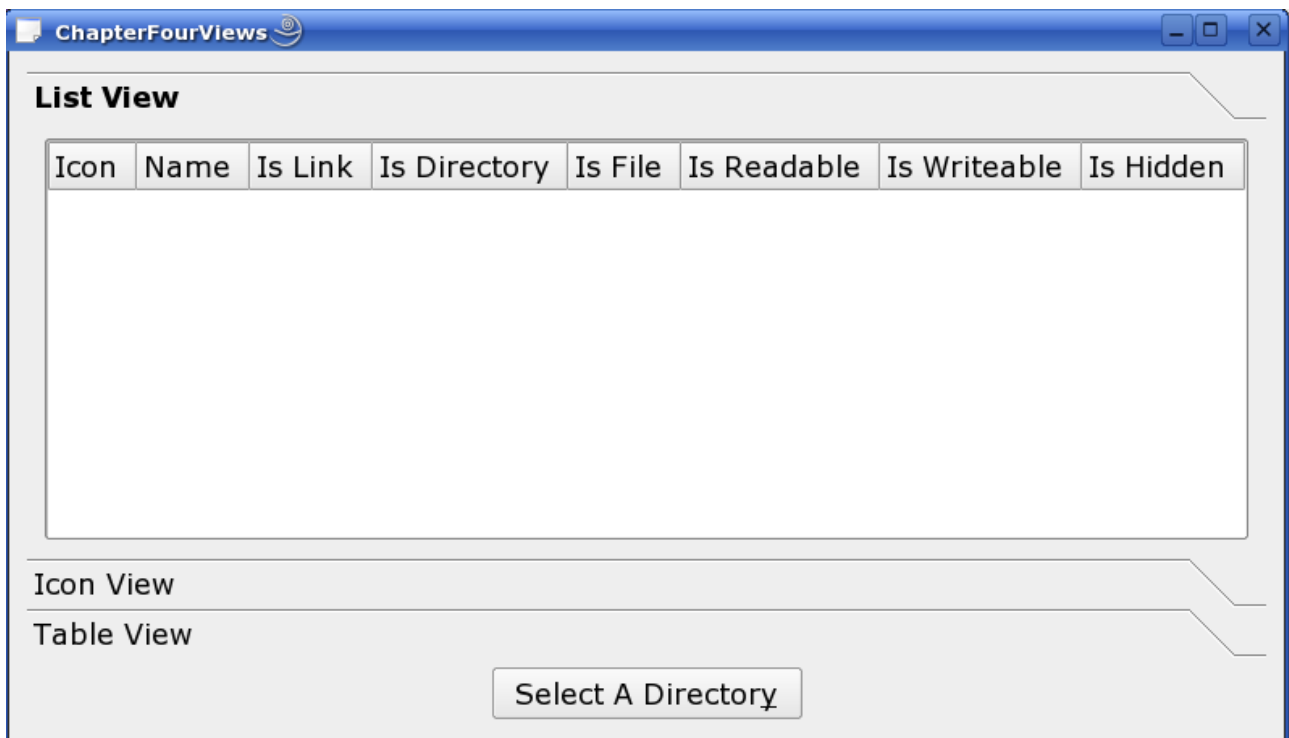
with each signal being clearly directed to it's own slot. There will of course be occasions when you want to send signals from different widgets to the same functions but these will probably be few and far between.

The Views Demonstration

The idea behind the views demonstration is that as all the widgets are use for displaying data we add all the widgets to a form open up a directory and then display the files in the directory in the different views. Note though that this is just for display purposes only and we wont be adding any

Chapter 4 Containers And Views

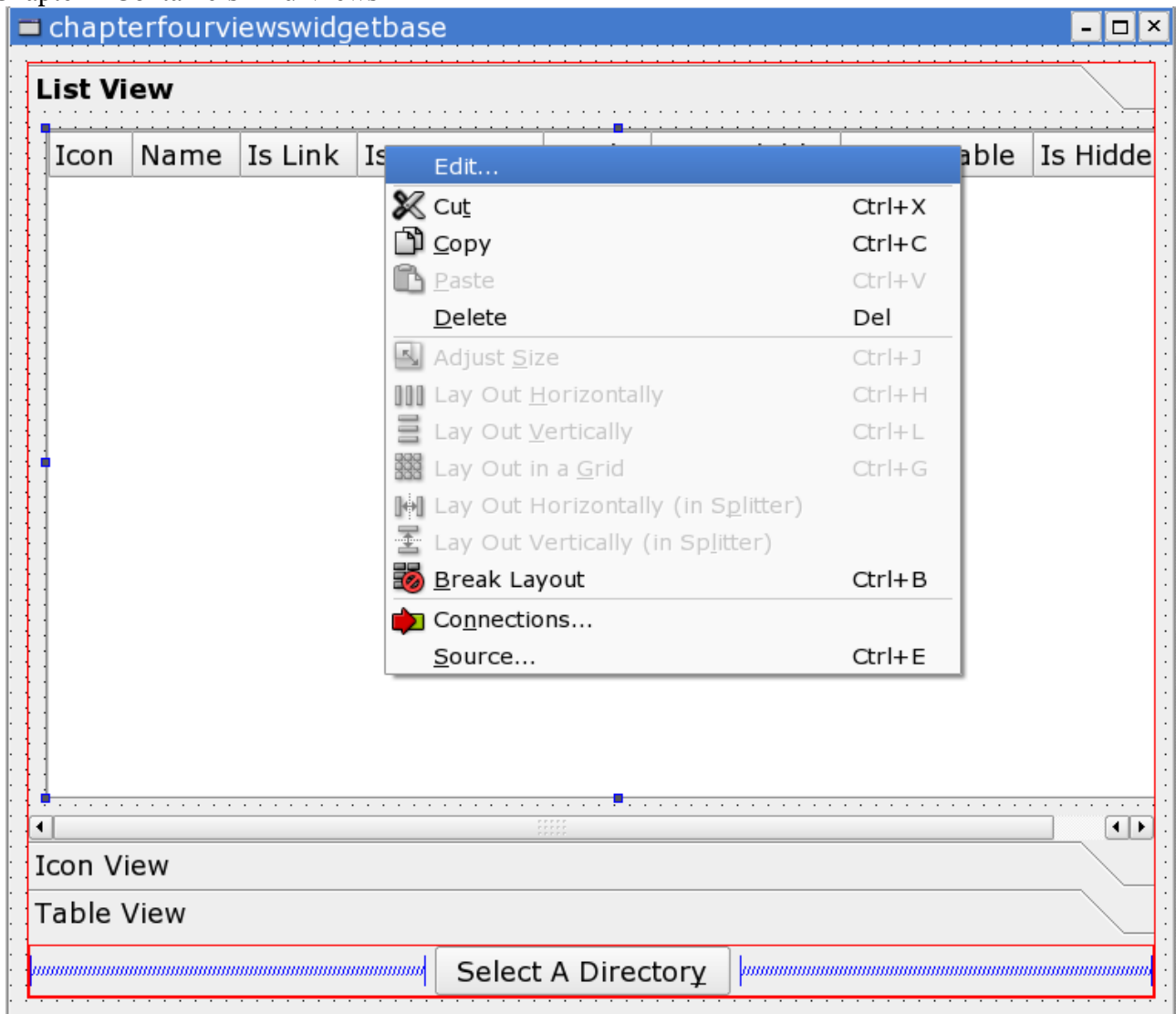
functionality to program other than what we need for display purposes.



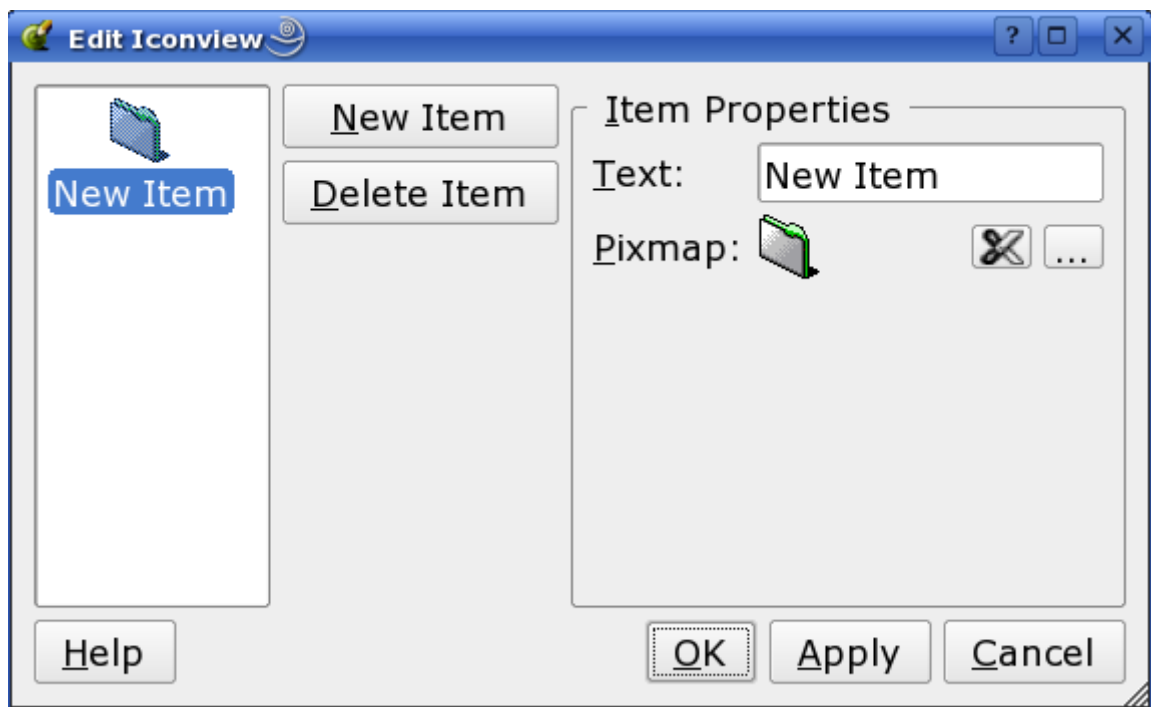
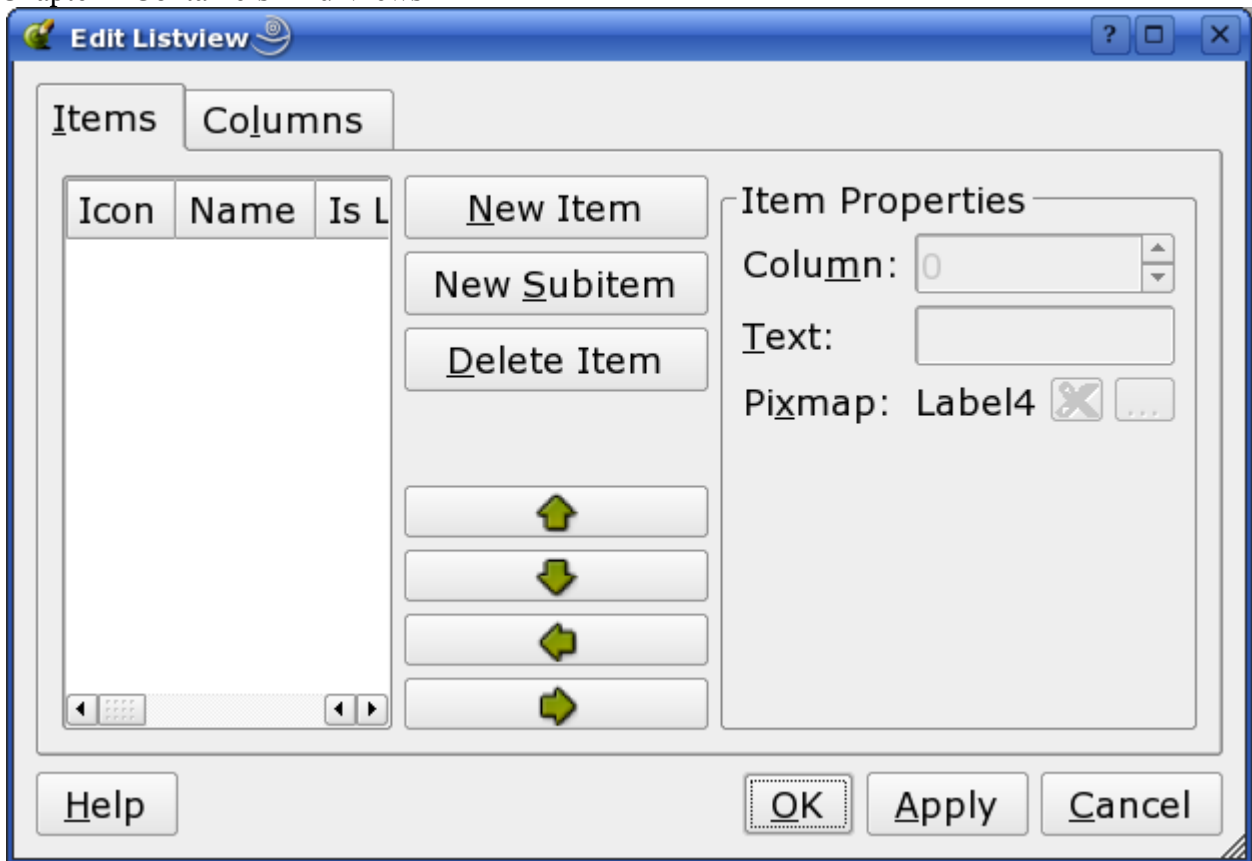
The application consists of a `QToolBox` containing a `QListView`, a `QIconView` and a `QTableView` on different pages. The user selects a directory through the "Select A Directory" button and each view displays the details of the file or directory in a slightly different way.

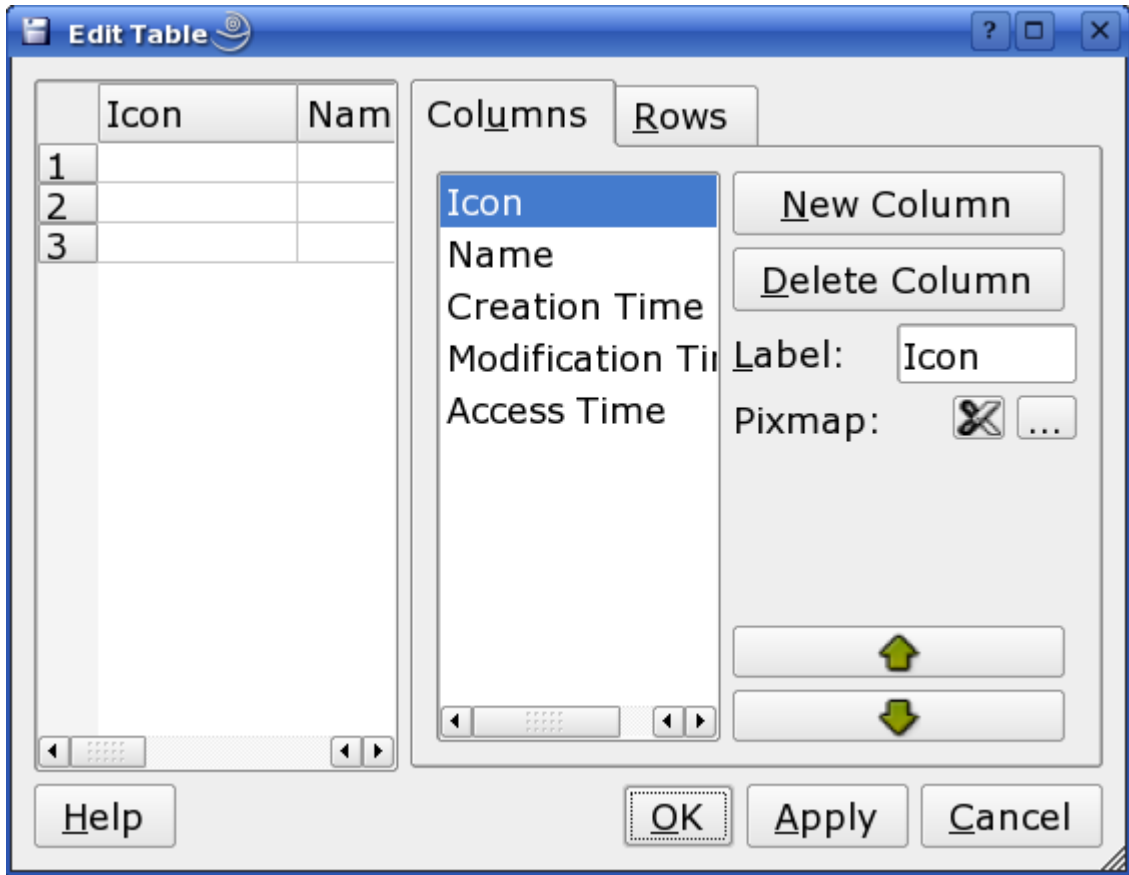
The options for each widget are setup through the right clicking on the widget and selecting edit

Chapter 4 Containers And Views



The options when you click on edit are all pretty similar and straight forward,

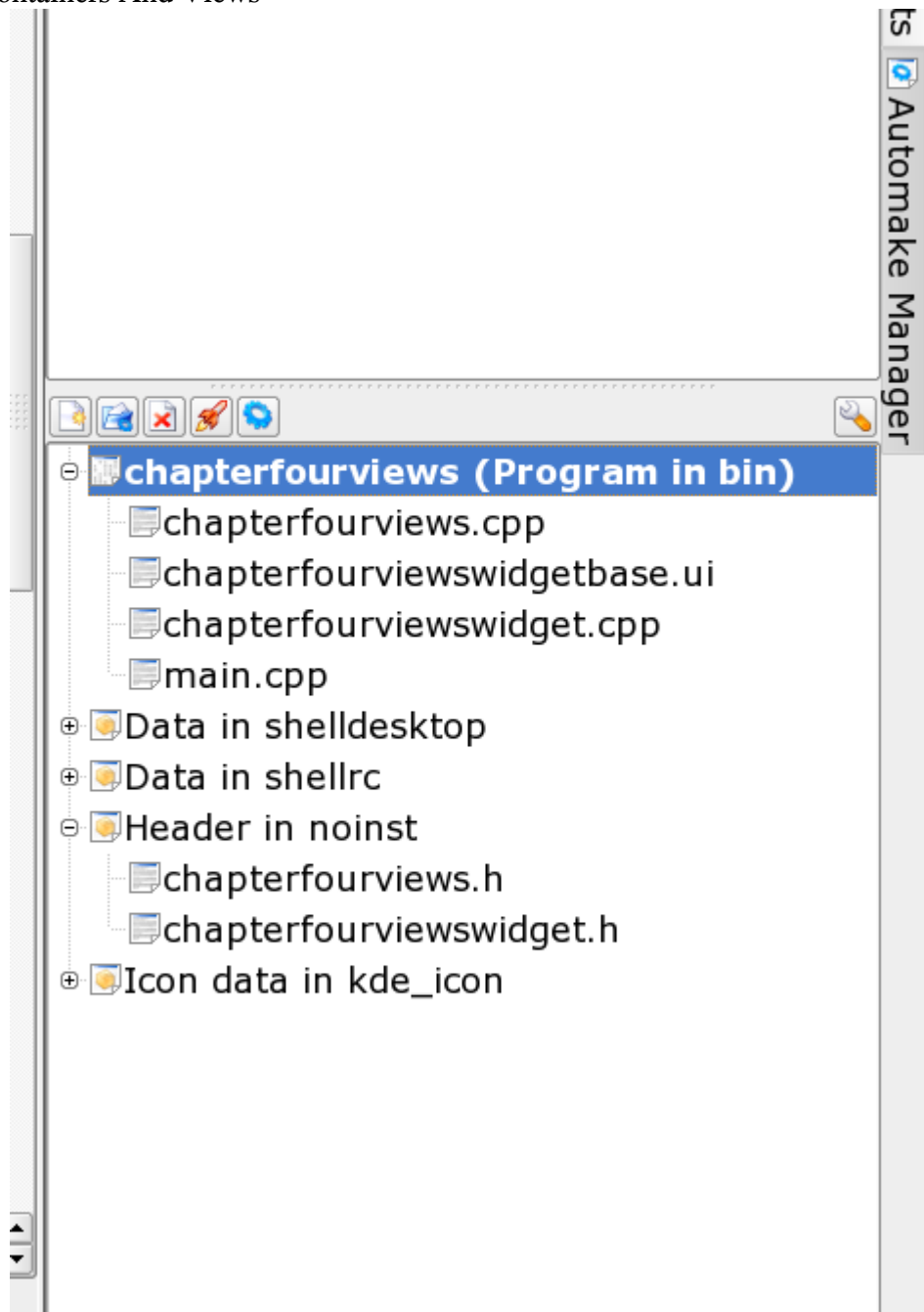




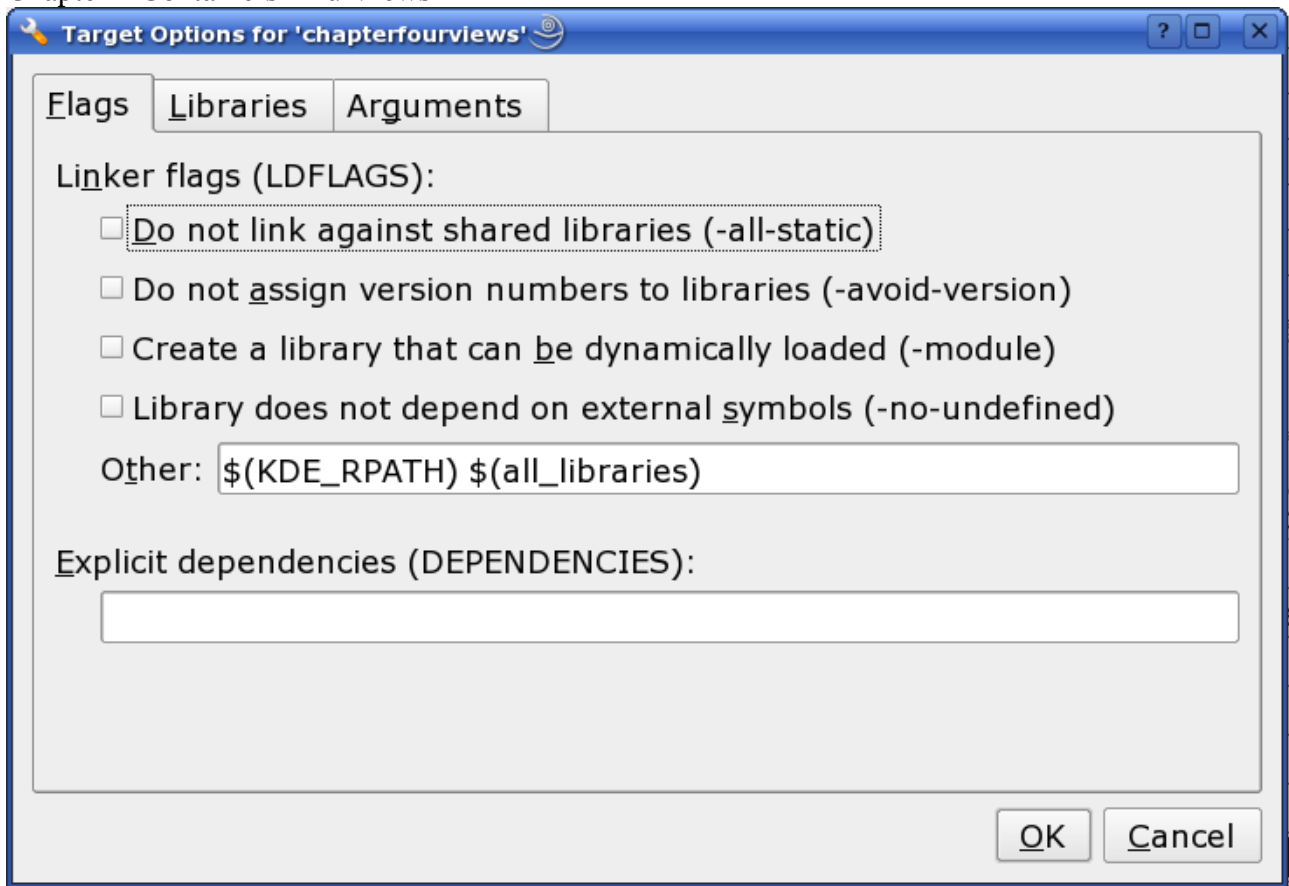
The choices about what each view should show were made by looking at the `KFileItem` class in the KDevelop help files and splitting the information into manageable chunks across the three views.

Adding Libraries To A Project

In the following code we use classes from both KDE and Qt, the rather surprising thing is that while the Qt Libraries appear to be included as a whole the KDE libraries aren't and this is the first place we come across this as we are trying to use classes from the KIO library. In order to add a library to a project open the automake manager and select the project line,

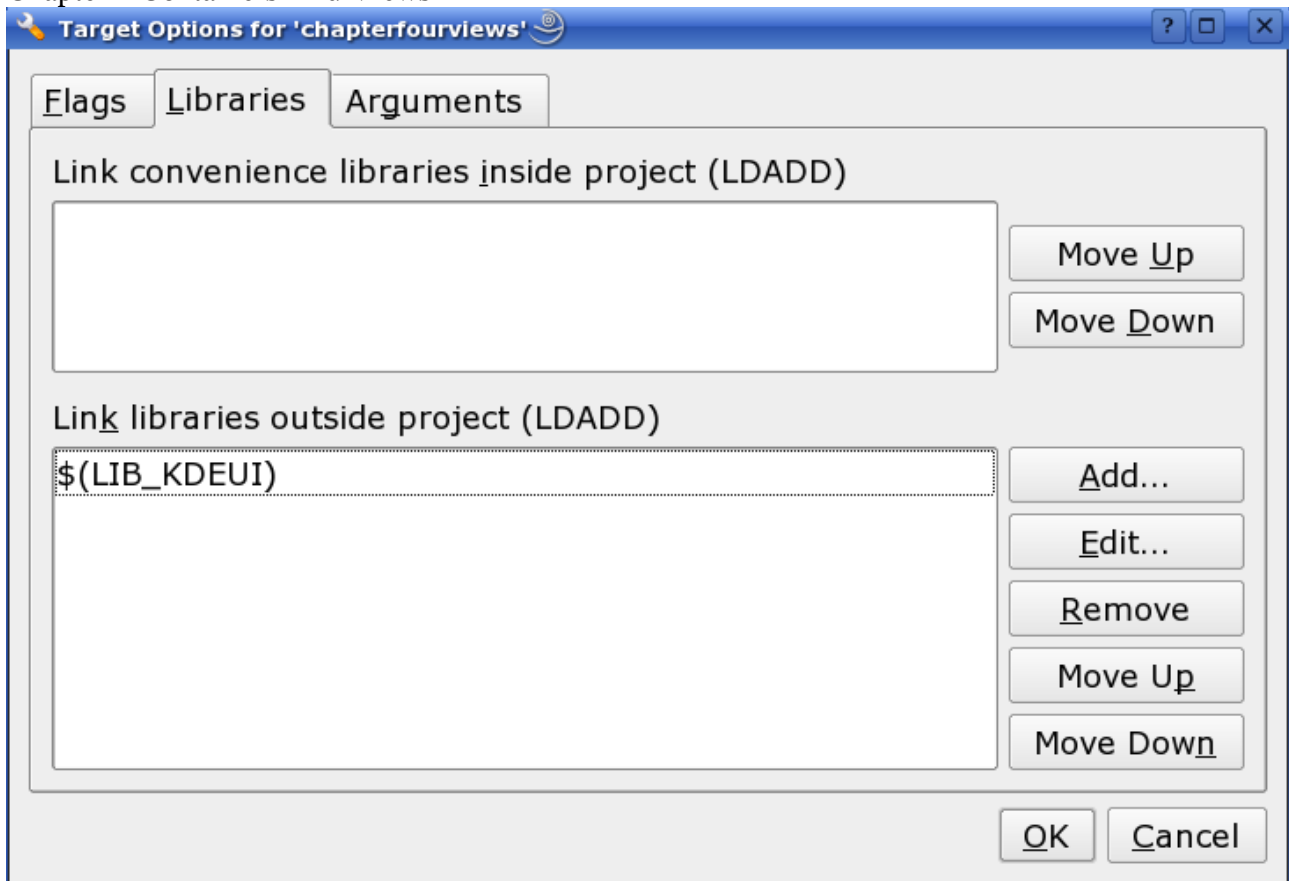


You will notice that the tool button on the right has ceased to be greyed out, if you click on it or alternatively right click on the highlighted line and select options from the drop down menu, you get this dialog.

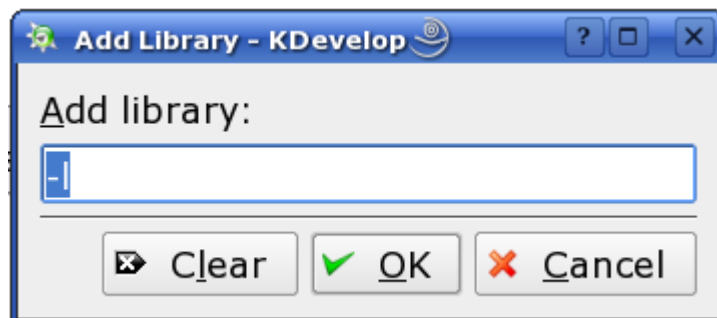


The flags option gives you some specific options that you can use when linking your project. We are happy to leave these at the defaults for the moment, what we are interested in is the Libraries tab.

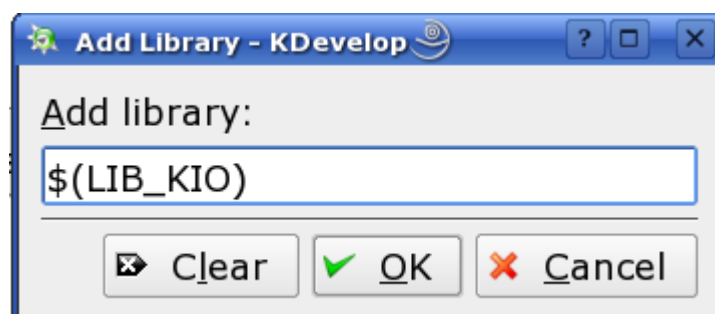
Chapter 4 Containers And Views



Here we can see the libraries that are already included in the project and we want to add the KIO library so click on add,

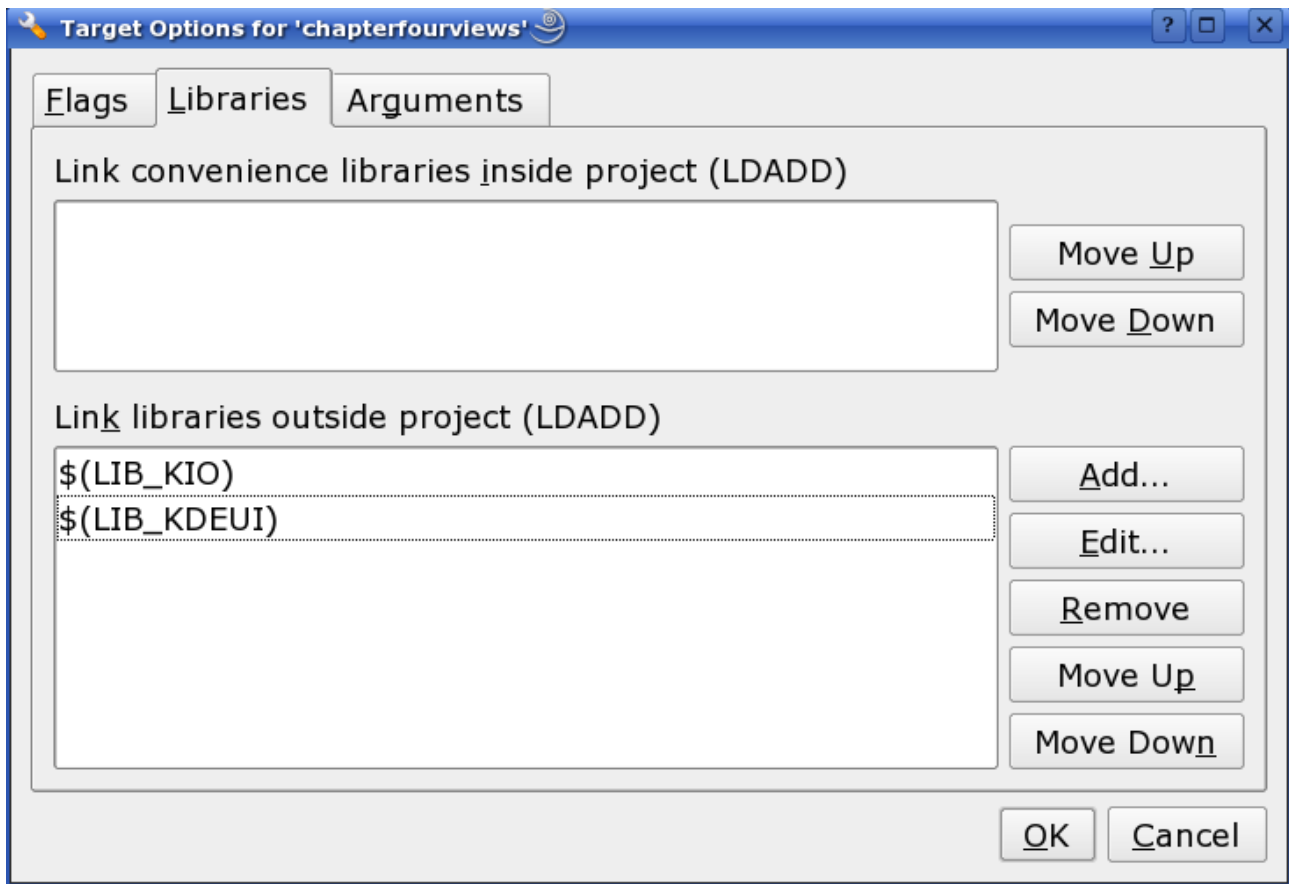


The dialog pops up with the include option already to pass to the linker. Because we are using one of the KDE libraries delete this and add,



Chapter 4 Containers And Views

Click on OK,



Now we can add any class from the KIO library, as long as we include the header files. So what does it mean.

The `$(LIB_KIO)` tells the makefile to include the KIO library from the library directory. For the purposes of KDE on a 64 bit system this will be something like `/opt/kde3/lib64`. The library itself will be called `libkio`. with the `.o` and `.so` files being the files that are loaded at run time or linked to the project and the `.la` file being the file that the linker reads. The `.la` file looks like this,

```
# libkio.la - a libtool library file
# Generated by ltmain.sh - GNU libtool 1.5a (1.1240 2003/06/26 06:55:19)
#
# Please DO NOT delete this file!
# It is necessary for linking the library.

# The name that we can dlopen(3).
dlname='libkio.so.4'

# Names of this library.
library_names='libkio.so.4.2.0 libkio.so.4 libkio.so'

# The name of the static archive.
old_library=''
```

Chapter 4 Containers And Views

Libraries that this one depends upon.

```
dependency_libs=' -L/opt/kde3/lib64 -L/usr/lib/qt3/lib64
-L/usr/X11R6/lib64 /opt/kde3/lib64/libkdeui.la -L/usr/lib64 -L/usr/lib64/
-L/usr/X11R6/lib64/ /opt/kde3/lib64/libkdesu.la /opt/kde3/lib64/libkwalletclient.la /opt/
kde3/lib64/libkdecore.la /usr/lib64/libstdc++.la /opt/kde3/lib64/libDCOP.la -lresolv
-lutil /usr/lib64/libart_lgpl_2.la /usr/lib64/libidn.la /opt/kde3/lib64/libkdefx.la /usr/
lib/qt3/lib64/libqt-mt.la -lfreetype -lfontconfig -lXi -lXrandr -lXcursor -lXinerama
-lXft /usr/lib64/libfontconfig.la /usr/lib64/libfreetype.la /usr/lib64/libexpat.la -ldl -
lpng -lXext -lX11 -lSM -lICE -lpthread -lXrender
-lz /usr/lib64/libfam.la /usr/lib64/libstdc++.la'
```

Version information for libkio.

current=6

age=2

revision=0

Is this an already installed library?

installed=yes

Should we warn about portability when linking against -modules?

shouldnotlink=no

Files to dlopen/dlpreopen

dlopen=''

dlpreopen=''

Directory that this library needs to be installed in:

libdir='/opt/kde3/lib64'

As you can see this is one of those files that is only complicated if you have to try and write it yourself. It is simply a list of what the names for the library are, what files it needs to be present to work properly and some version and location details.

Selecting A Directory.

Now that we have all the necessities out of the way we can get on with getting the program to do something. The first task is to get allow the user to add select a directory which they do by clicking on the Select A Directory button.

The code for the slot `directoryButton_clicked()` looks like,

```
bool bTest = true;
KURL kurlDir;

/// The Qt Way
///
/*
kurlDir = QFileDialog::getExistingDirectory();
setDirectory( kurlDir.pathOrURL() );
if( bTest == true )
{
    KMessageBox::information( this, directory(), "Qt version" );
}
```

Chapter 4 Containers And Views

```
}
*/

/// The KDE KFileDialog Way
///

KFileDialog *ptrFileDlg = new KFileDialog( QString::null, QString::null, this, "KDE
KFileDialog Version", true );
if( ptrFileDlg == 0 )
{
    KMessageBox::error( this, "Unable to allocate the file dialog", "KDE KFileDialog
Version" );
    return;
}

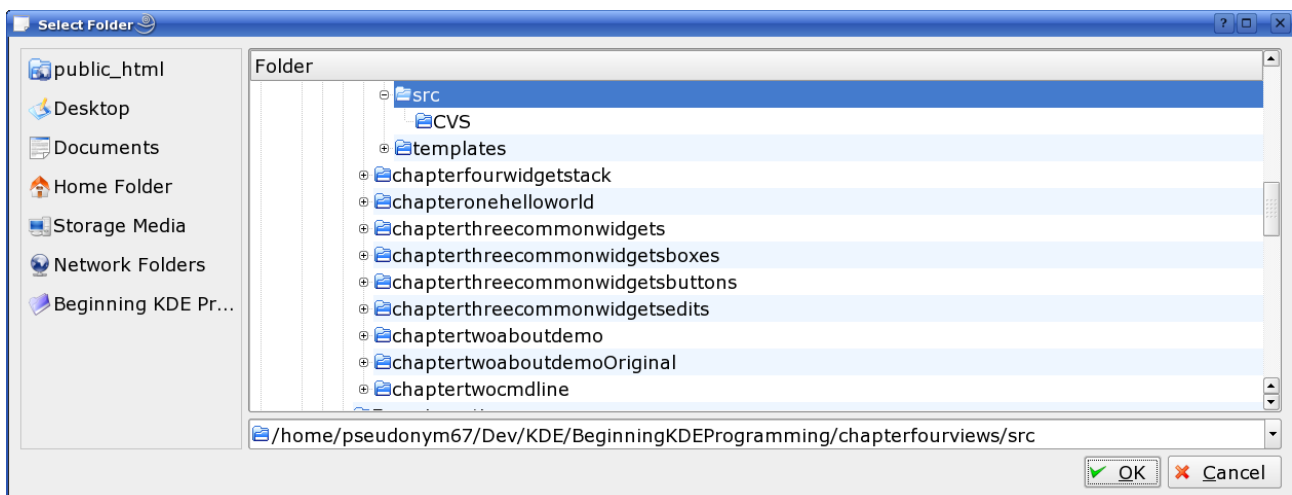
ptrFileDlg->setMode( KFile::Directory );
if( ptrFileDlg->exec() == KDialog::Accepted )
{
    kurlDir = ptrFileDlg->selectedURL();
    setDirectory( kurlDir.pathOrURL() );
    if( bTest == true )
    {
        KMessageBox::information( this, directory(), "KDE KFileDialog Version" );
    }
}

delete ptrFileDlg;

fillViews();
```

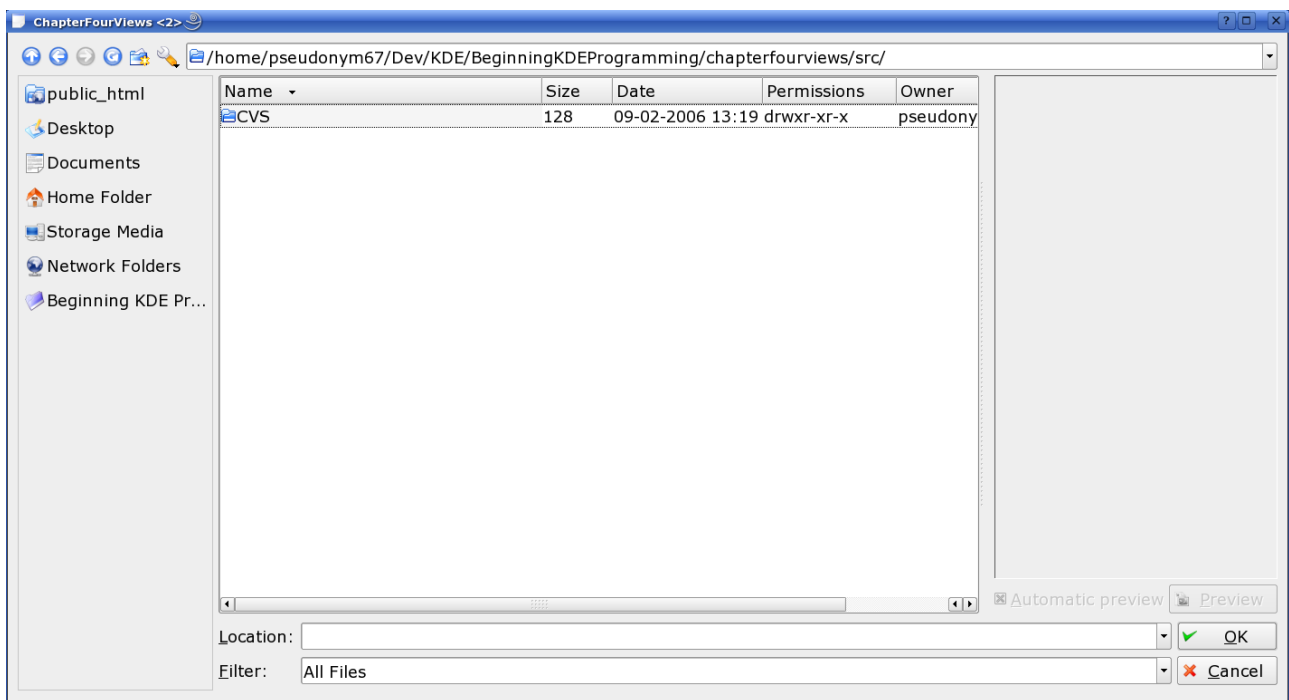
As you can see there are different ways to select a directory implemented in the function. Each of them display a standard dialog for selecting the directory and store the result in `QString strDirectory` which is a private member of the class through the `setDirectory` function. The first way demonstrated is the Qt way as it should be noted that although we are using KDevelop and programming for KDE at the moment we are using very little of the KDE libraries and it wouldn't be too difficult to alter all the examples so far so that they were stand alone Qt applications.

If you try running the separate versions the dialogs look different,



is the Qt version and

Chapter 4 Containers And Views



is the KDE version, personally I prefer the Qt version when selecting directories and the KDE version when selecting files. As usual it is a matter of personal preference.

Functionally the only difference you will notice is that the Qt method opens the Documents folder by default whereas the KDE methods open the current directory that the program is running from unless you specify a directory in the constructor. The bool variable bTest is provided so that you can see what string is being placed in the strDirectory string while testing the program.

Filling The Views

The views are filled in with the data in the fill views function. which starts with,

```
QDir dir( directory() );  
const QFileInfoList *fileInfoList = dir.entryInfoList();
```

First of all we create a QDir by passing in the saved directory and then call entryInfoList to get all the entries in the given directory which are stored in the returned QFileInfoList then all we need to do is set up the mechanism for cycling through the QFileInfoList.

```
QFileInfoListIterator it( *fileInfoList );  
QFileInfo *ptrFileItem;
```

```
while( ( ptrFileItem = it.current() ) != 0 )
```

which is done through the creation of an iterator to move through the list and a QFileInfo pointer to identify each item. It is then simply a matter of moving through the list and then extracting the information that we want to put in each view.

The following code is for the QListView,

```
QListViewItem *listViewItem = new QListViewItem( listView );  
KURL fileURL( ptrFileItem->absFilePath() );  
KFileItem fileItem( KFileItem::Unknown, KFileItem::Unknown, fileURL, true );  
listViewItem->setPixmap( ColumnOne, fileItem.pixmap( 0 ) );
```


Chapter 4 Containers And Views

```
listViewItem->setText( ColumnTwo, ptrFileItem->fileName() );
if( ptrFileItem->isSymLink() == true )
    listViewItem->setText( ColumnThree, "true" );
else
    listViewItem->setText( ColumnThree, "false" );
if( ptrFileItem->isDir() == true )
    listViewItem->setText( ColumnFour, "true" );
else
    listViewItem->setText( ColumnFour, "false" );
if( ptrFileItem->isFile() == true )
    listViewItem->setText( ColumnFive, "true" );
else
    listViewItem->setText( ColumnFive, "false" );
if( ptrFileItem->isReadable() == true )
    listViewItem->setText( ColumnSix, "true" );
else
    listViewItem->setText( ColumnSix, "false" );
if( ptrFileItem->isWritable() == true )
    listViewItem->setText( ColumnSeven, "true" );
else
    listViewItem->setText( ColumnSeven, "false" );
if( ptrFileItem->isHidden() == true )
    listViewItem->setText( ColumnEight, "true" );
else
    listViewItem->setText( ColumnEight, "false" );

++it;
```

We create a `QListViewItem` that has the current `QListView` as its parent and then to make getting the icon image easier a `KFileItem` is created which will give us a `QPixmap` of the files icon when we call `pixmap(0)`. The rest of the code simply tests for the required properties and adds true or false to the `QListViewItem` depending on their presence.

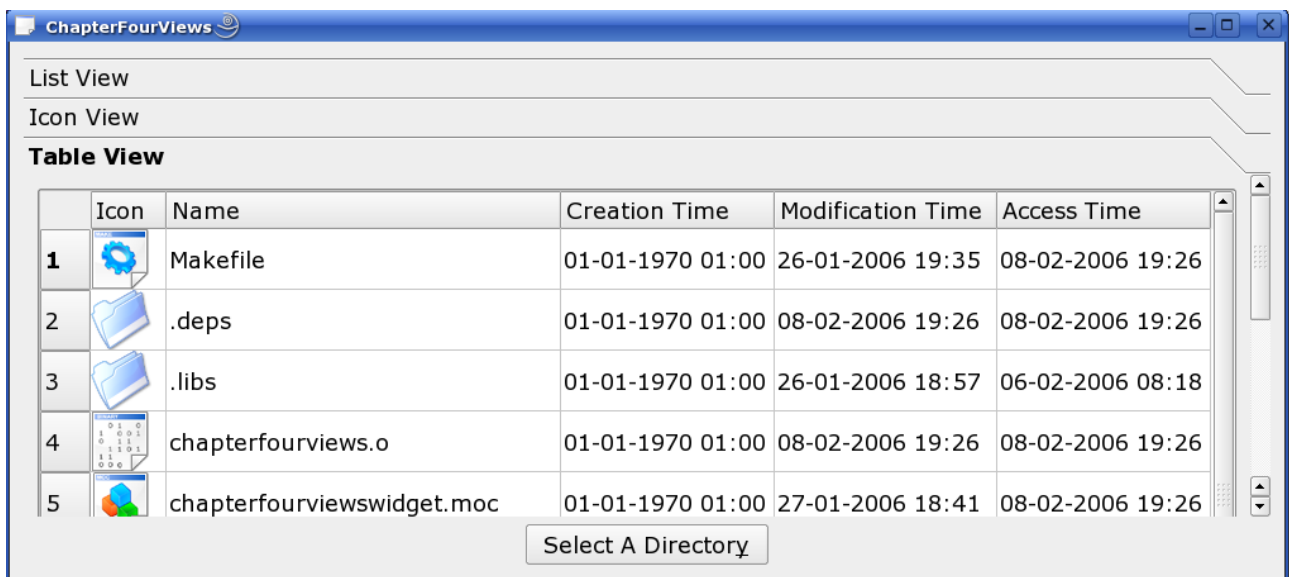
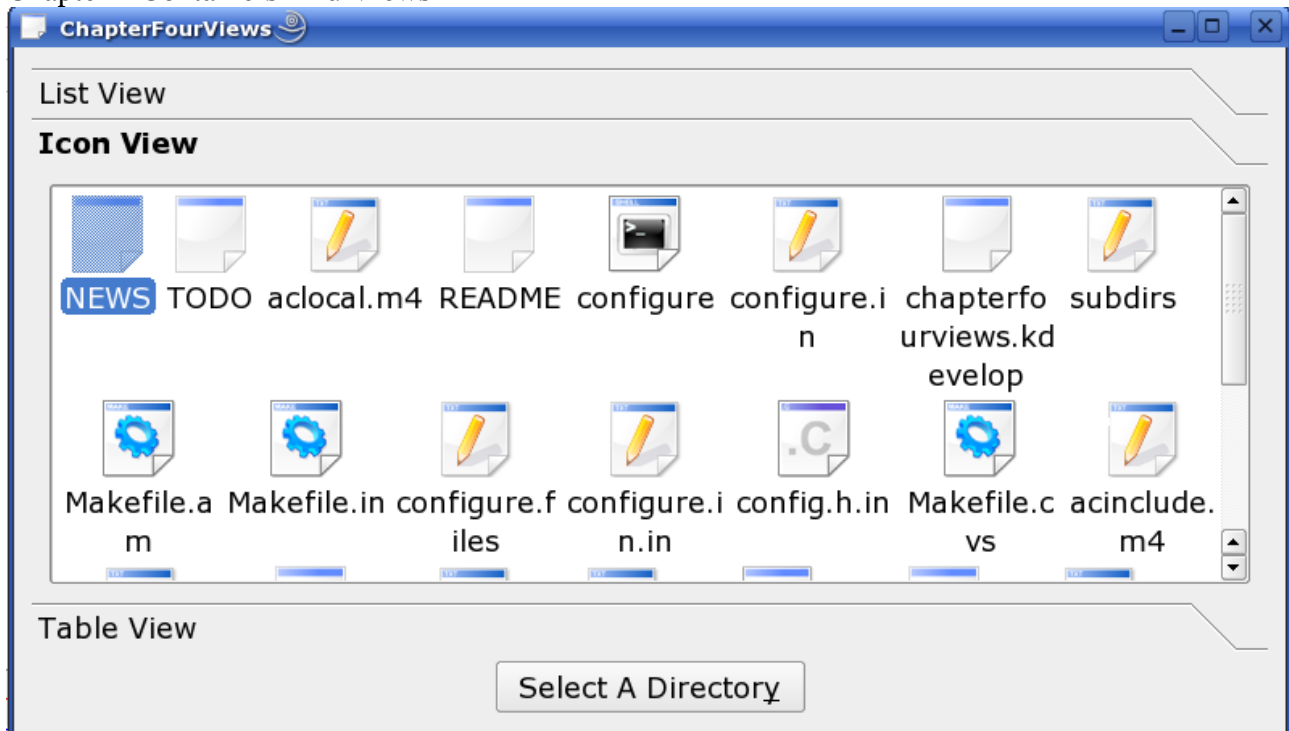
One thing to note codewise is the ending of the loop for the table

```
table->adjustColumn( ColumnOne );
table->adjustColumn( ColumnTwo );
table->adjustColumn( ColumnThree );
table->adjustColumn( ColumnFour );
table->adjustColumn( ColumnFive );
```

The calls to `adjustColumn` at the end automatically set the table column width so that it can accommodate the largest item that it needs to display.

What we end up with looks like this,

Chapter 4 Containers And Views



Summary

This brings us to the end of the containers and views in which we looked at how to set up some of the graphical views and containers that are available in the KDevelop GUI toolkit. We also and perhaps more importantly for future development saw how to add libraries that are not included by default into the project.

Chapter 5 Database Programming With MySQL

Love them or hate them there is one thing that you can pretty much guarantee if you are going to spend any time doing computer programming and that is that sooner or later you are going to have to either create or display data from a database. In this chapter we look at how to use the built in database widgets of QDataTable, QDataBrowser and QDataView, but first of all we need to set up some information to use in our database. For this demonstration we will be using MySQL which comes with Suse Linux and probably given it's popularity every other version of Linux as well.

Setting Up The Database

There seem to be two ways you can go about setting up MySQL. The easy way and the painful way and there seems to be only a slight difference that decides which way your set up is going to go. This way was worked out over several days of testing. If you want to do it another way or you do run into trouble you'll be needing this link. [MySQL Documentation](#)

Order

1. Install MySQL
2. MySQL_Install_DB
3. Start MySQL
4. MySQL_Fix_Privilege_Tables
5. Change Root Password (Recommended)
6. (Optional) Uninstall MySQL

1. Install My SQL

MySQL is installed with Suse 10 it is just not started as a service, but there are still reasons for checking the install with Yast. Mostly this is because the Graphical User Interface tools for MySQL are not installed by default so unless you set up the system knowing that you would need them you probably wont have them.

To install or to just check if you have everything you need open the Control Center (Yast) and select the Software Management option. It should open with the search option, type mysql, it doesn't matter about case and hit return.



Chapter 5 Database Programming With MySQL

The parts you are interested in here are the mysql-administrator and the mysql-query-browser, the query browser is for more advanced use than we will be using here but it can't hurt to have it.

The Yast setup should be familiar if you have setup the system yourself if not, click on the checkbox for the item you are interested in. A black tick will install it, a blue tick means it is already installed, a bin means remove it and a lightening bolt thingy means update it.

It should also be noted that these programs are included on the main menu at System/Service Configuration but if you want to set up the links on your desktop you can find them in `/usr/share/applications`

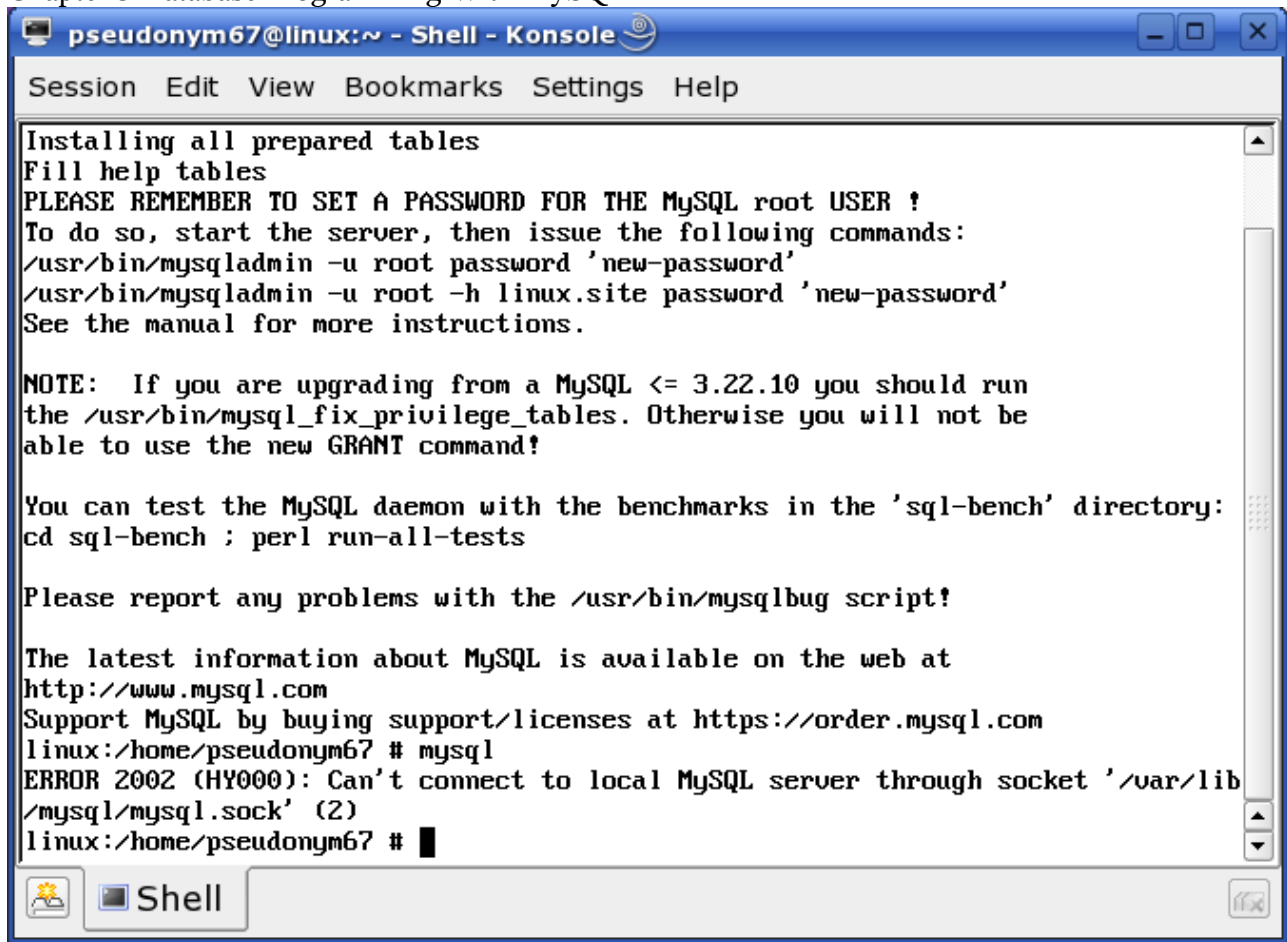


You can copy these to your desktop if you think you are going to need them regularly.

2. MySQL_Install_DB

The next thing you need to do is run the install db script which will setup the default MySQL users and databases. These are an anonymous user that has no password. A root user that has no password and the mysql database. There is also a test database that gets set up but this is empty and can just be ignored or deleted later.

To create the database and users open a console as superuser and type `MySQL_Install_DB`



```
pseudonym67@linux:~ - Shell - Konsole
Session Edit View Bookmarks Settings Help

Installing all prepared tables
Fill help tables
PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
To do so, start the server, then issue the following commands:
/usr/bin/mysqladmin -u root password 'new-password'
/usr/bin/mysqladmin -u root -h linux.site password 'new-password'
See the manual for more instructions.

NOTE: If you are upgrading from a MySQL <= 3.22.10 you should run
the /usr/bin/mysql_fix_privilege_tables. Otherwise you will not be
able to use the new GRANT command!

You can test the MySQL daemon with the benchmarks in the 'sql-bench' directory:
cd sql-bench ; perl run-all-tests

Please report any problems with the /usr/bin/mysqlbug script!

The latest information about MySQL is available on the web at
http://www.mysql.com
Support MySQL by buying support/licenses at https://order.mysql.com
linux:/home/pseudonym67 # mysql
ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/var/lib
/mysql/mysql.sock' (2)
linux:/home/pseudonym67 #
```

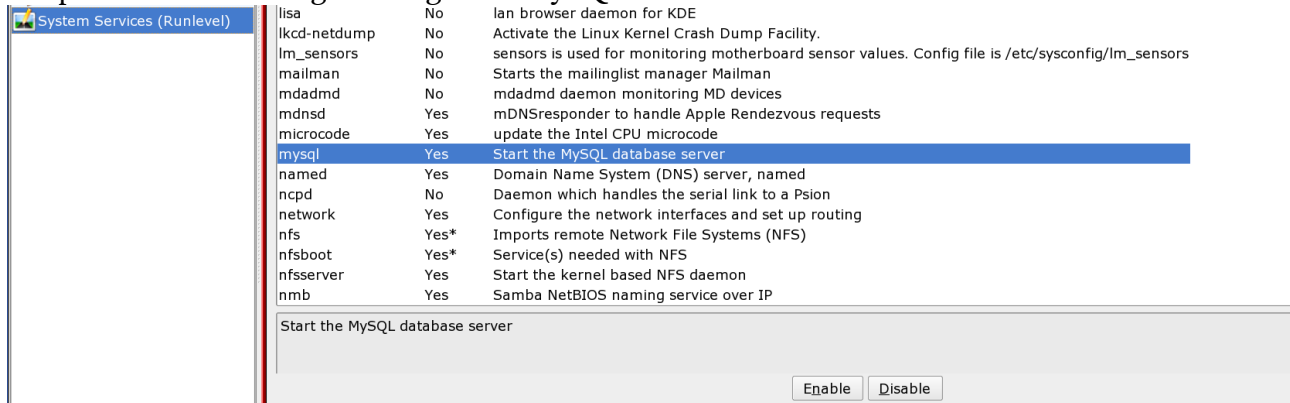
You should get a print out like the one above. Though of course for the moment you should ignore everything it says, unless it is reporting an error. The reasons for this are that to run the fix privilege tables script the service needs to be started and it has to log into mysql to set the tables which it is not going to be as simple as it could be if you have changed the password.

As you can see from the bottom of the print out any attempt to connect to MySQL at this point in time should be rejected as we haven't started it yet. We do that next.

3. Start MySQL

By default the MySQL server is disabled when you set up Suse so you need to start it yourself. To do this you need to access the Runlevel Services on whatever version of Linux you are using. On Suse this is done through the Control Center/YaST2 Modules/System/System Services (Runlevel). You will need Administrator privileges to access the services list.

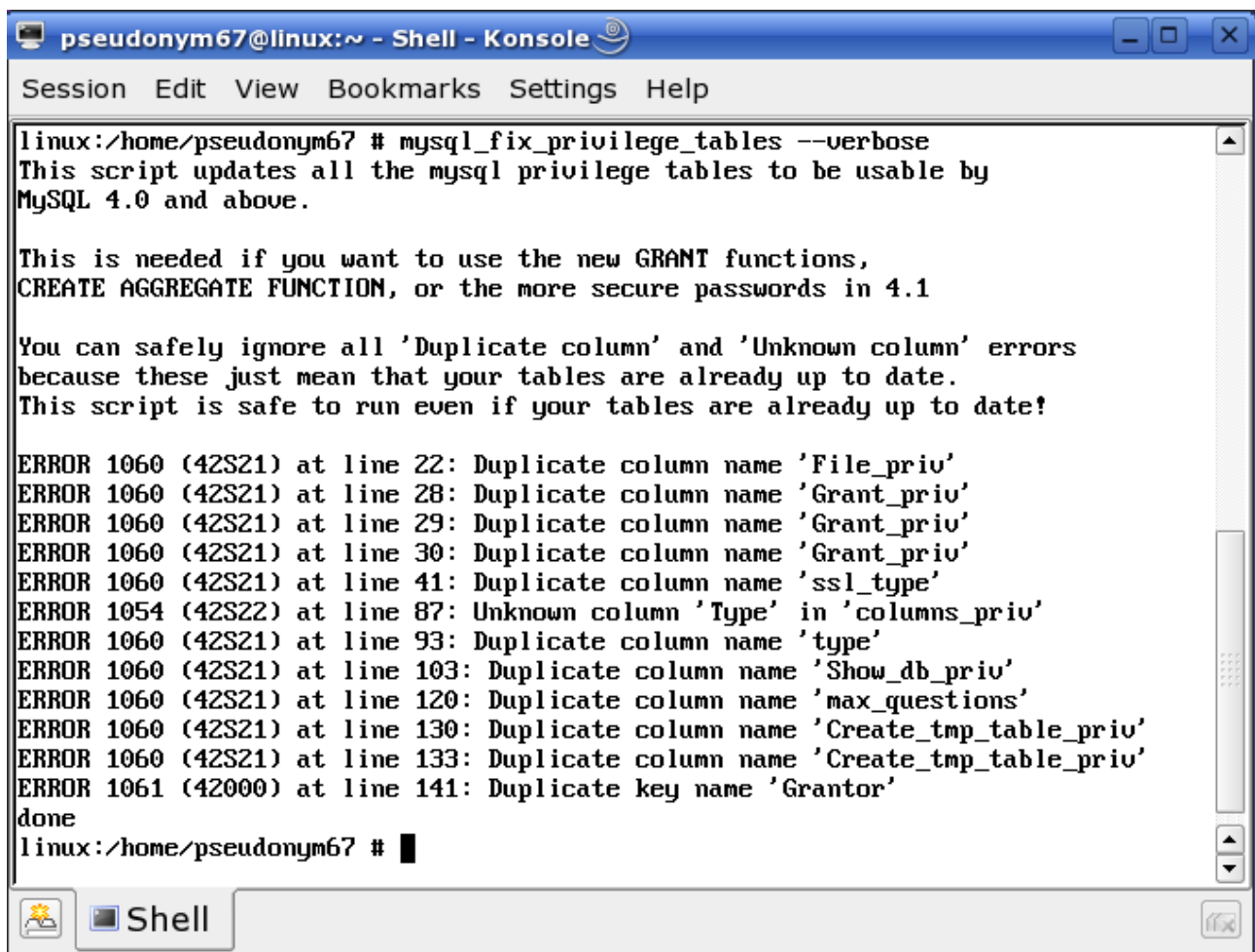
Chapter 5 Database Programming With MySQL



Once you enable the MySQL server here it will be started whenever you start your computer, so once you've done it you can just forget about it and use MySQL anytime you want.

4. MySQL_Fix_Privilege_Tables

Once the service is started open Konsole in superuser mode and type MySQL_Fix_Privilege_Tables.



5. Change Root Password

At this point it is recommended that you change the root password. The syntax is

```
mysqladmin -u root password "newpwd"
```

6. Uninstall MySQL

If you have got the the state where you are just getting nowhere and are having trouble getting answers from the MySQL website, the quickest solution, as long as you aren't using a machine that has a valid production database on it is just to delete MySQL and start again from scratch. You do this by going opening Yast, searching for mysql and setting the mysql option to be uninstalled which should show a bin at the checkbox.

The database files for MySQL on Suse are stored in var/lib/mysql. If you delete this folder you will have completely removed the MySQL information, including all databases and users, so like I say if the computer has ever been used as a MySQL production machine do not touch this folder or you will lose everything.

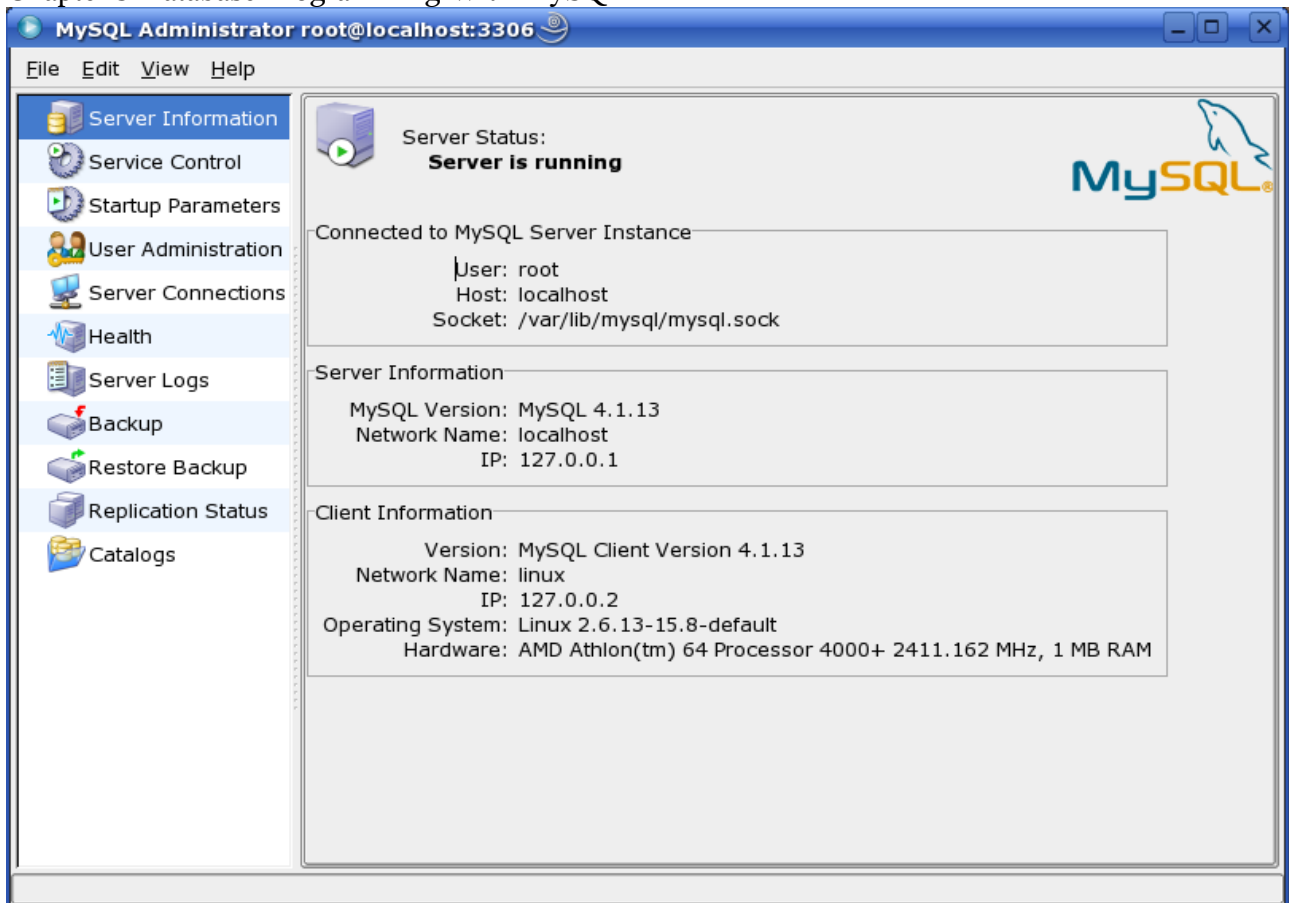
Administration

To setup MySQL exactly how you want it, start the MySQLAdministrator,



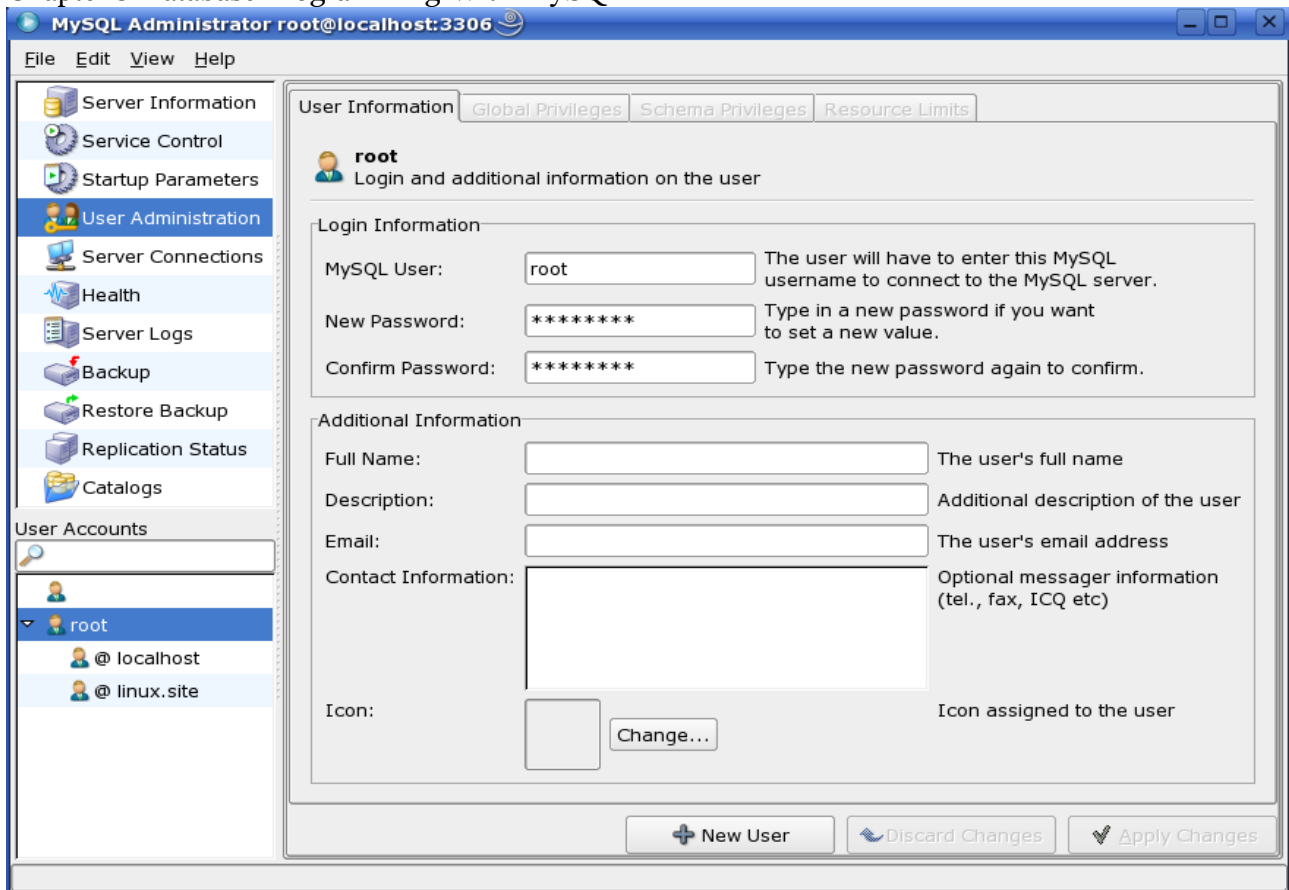
use root as the user and the password that you have set recently.

Chapter 5 Database Programming With MySQL



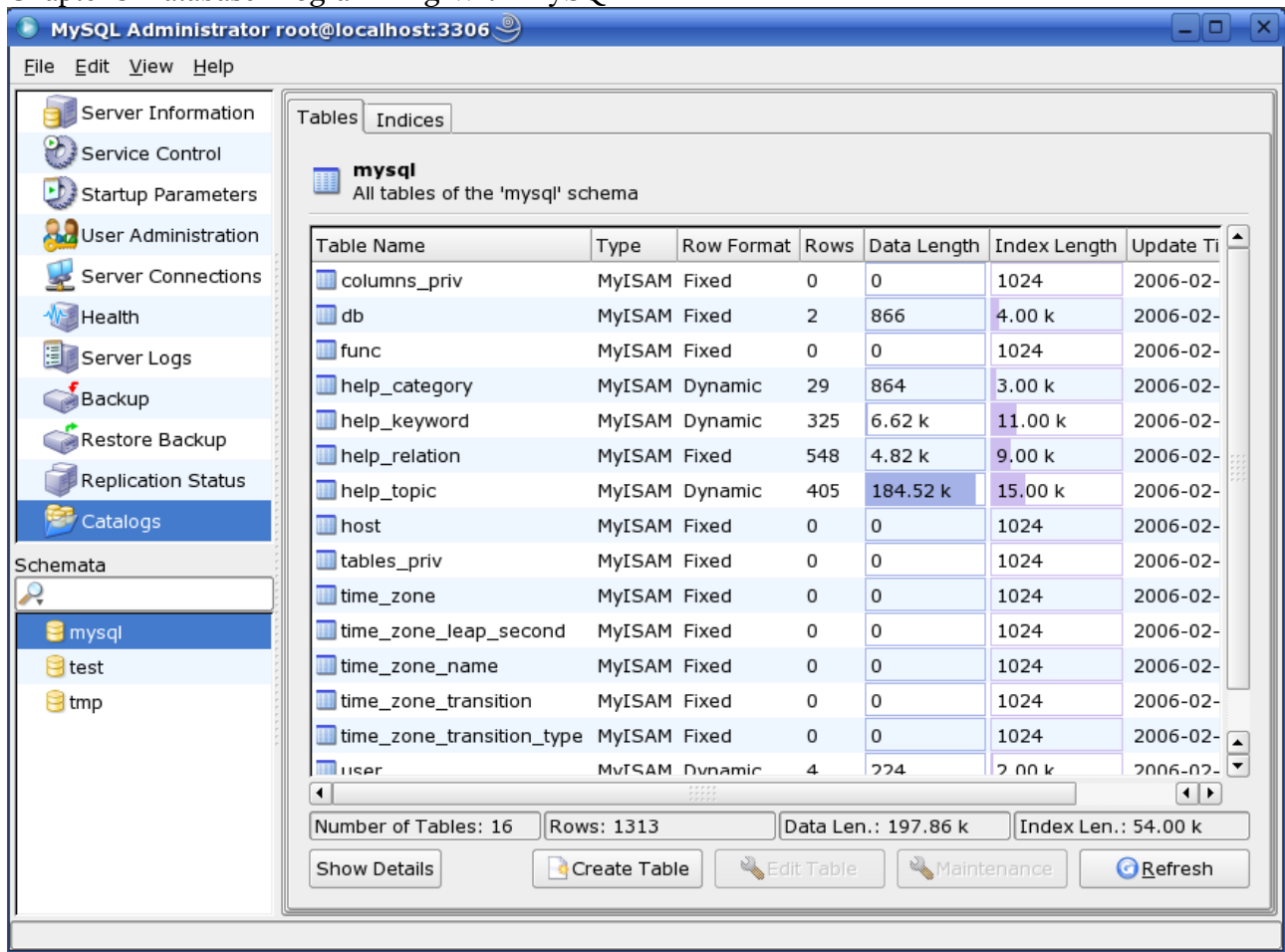
When it starts it should look something like the above. The important bits here are the users

Chapter 5 Database Programming With MySQL



Where you can add new users by using the button provided and the Catalogs section,

Chapter 5 Database Programming With MySQL



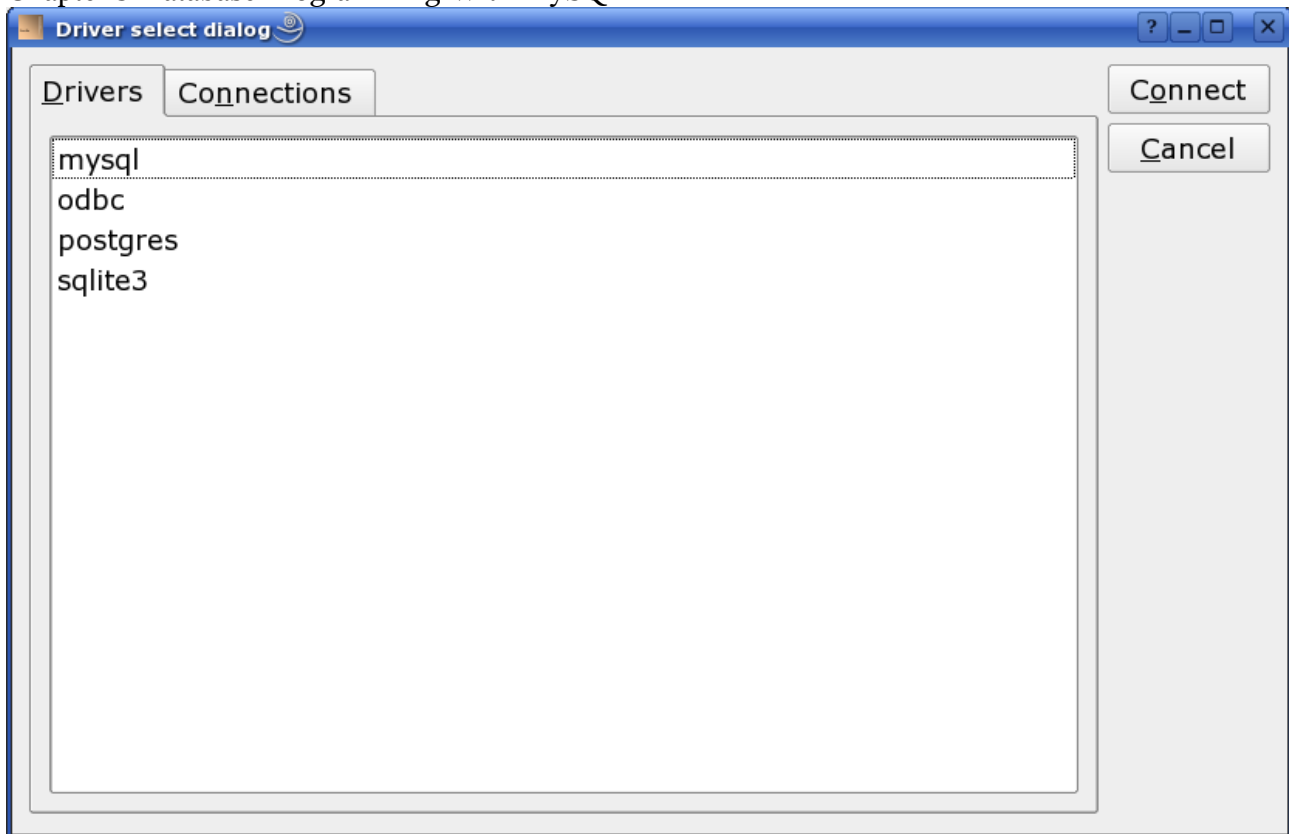
where you can see the databases that are on your system.

Knoda

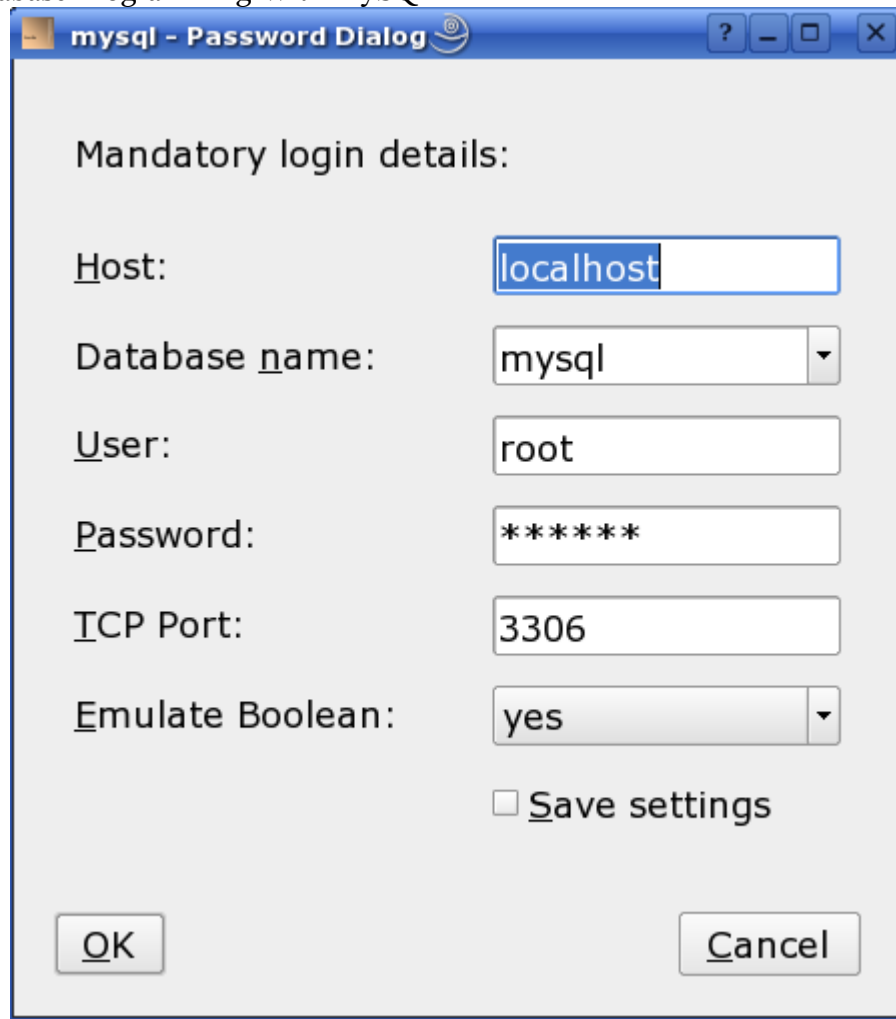
The default tool of choice for working with MySQL is going to be the KDE Knoda application for the simple reason that it allows us to set up the database without having to write all the SQL ourselves.

The link to the Knoda application can be found under the main menu Office/Database

Chapter 5 Database Programming With MySQL



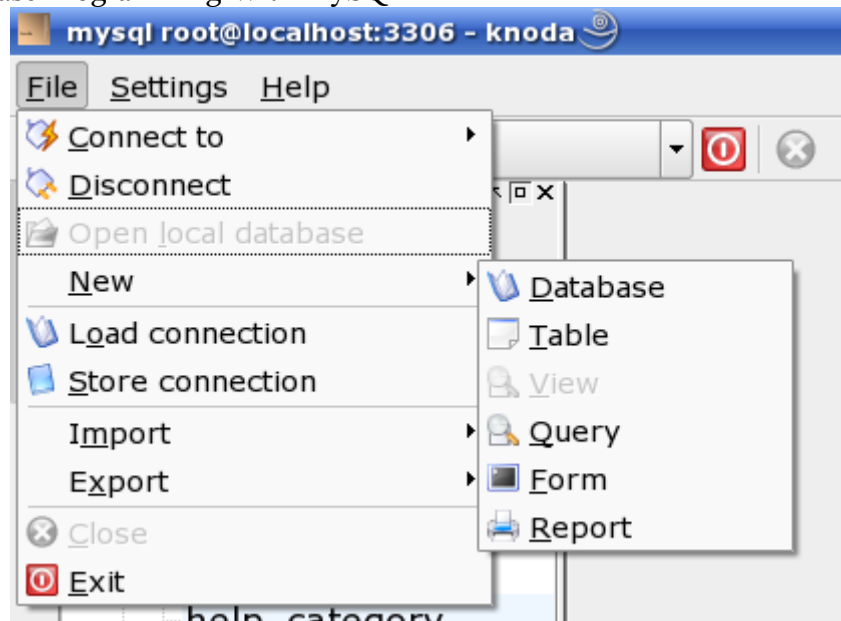
When started Knoda will ask which driver that we want to use, we select mysql and connect which gives us the connection dialog.



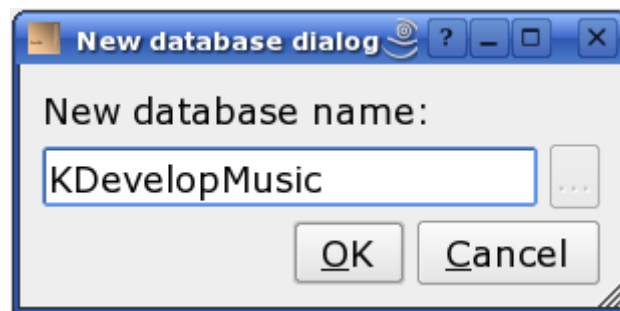
To setup and use our database we are just going to use the root connection with the password entered when we set up MySQL earlier. Once Knoda connects, assuming everything goes to plan you should have access to the MySQL databases that were installed during the setup process described above. We can ignore these as for our projects for this chapter we are going to setup a simple music database that we can then access through KDevelop and use within our own programs.

To create a completely new MySQL database with Knoda go to the File/New/Database menu.

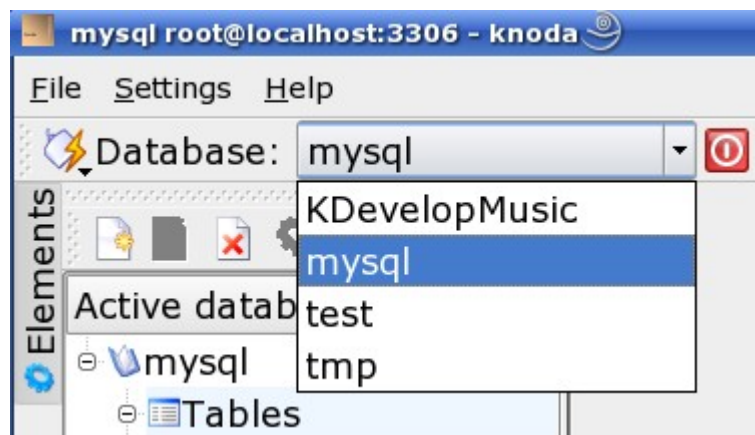
Chapter 5 Database Programming With MySQL



Select the New Database menu item and we get this dialog.



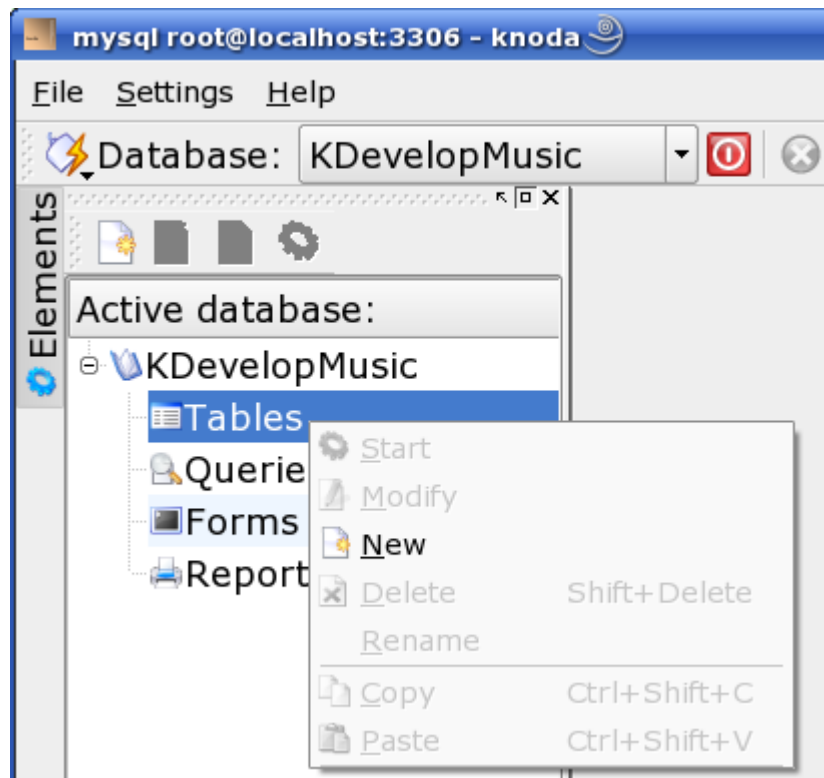
As you can see to setup a new database all we have to do is type in the name and select OK. At this point absolutely nothing will appear to happen as Knoda will not automatically change to the newly created database, nor does it ask you if you want to. This is not a major problem though as changing to the new database is easy.



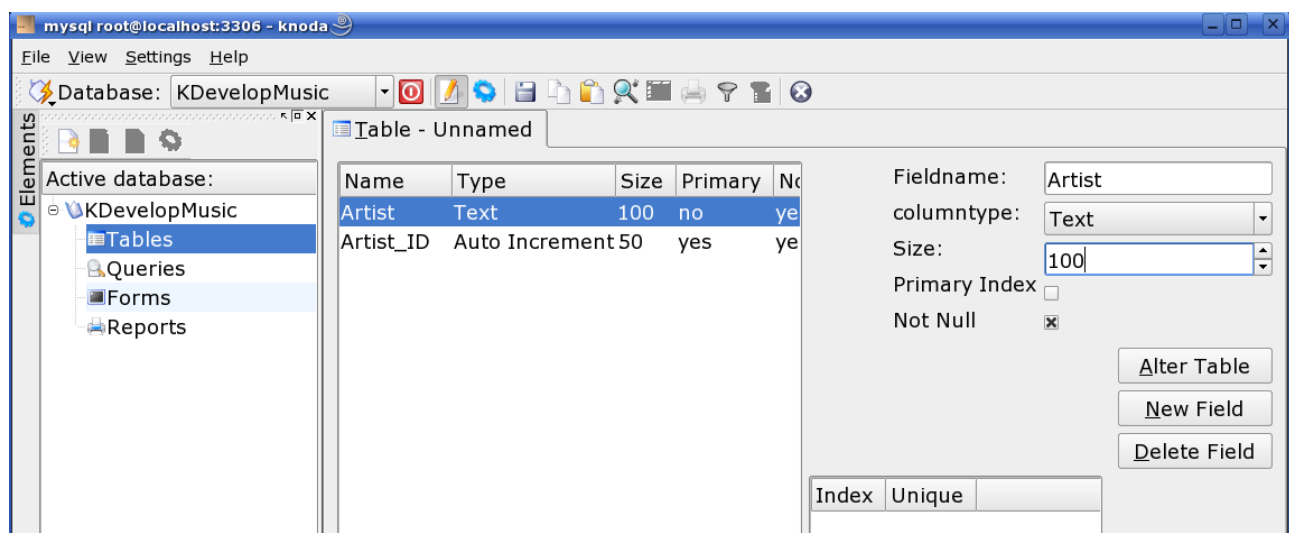
Select the Database combobox and select from any of the available databases. We want

Chapter 5 Database Programming With MySQL

KDevelopMusic which once selected Knoda will load our new, empty database. First of all we want to create three tables.



As you can see creating a new item in the table be it a table, Query, etc. Is as simple as right clicking and selecting new. For this example music database we are going to create three tables, these being the Artist table, the Album table and the Songs table.



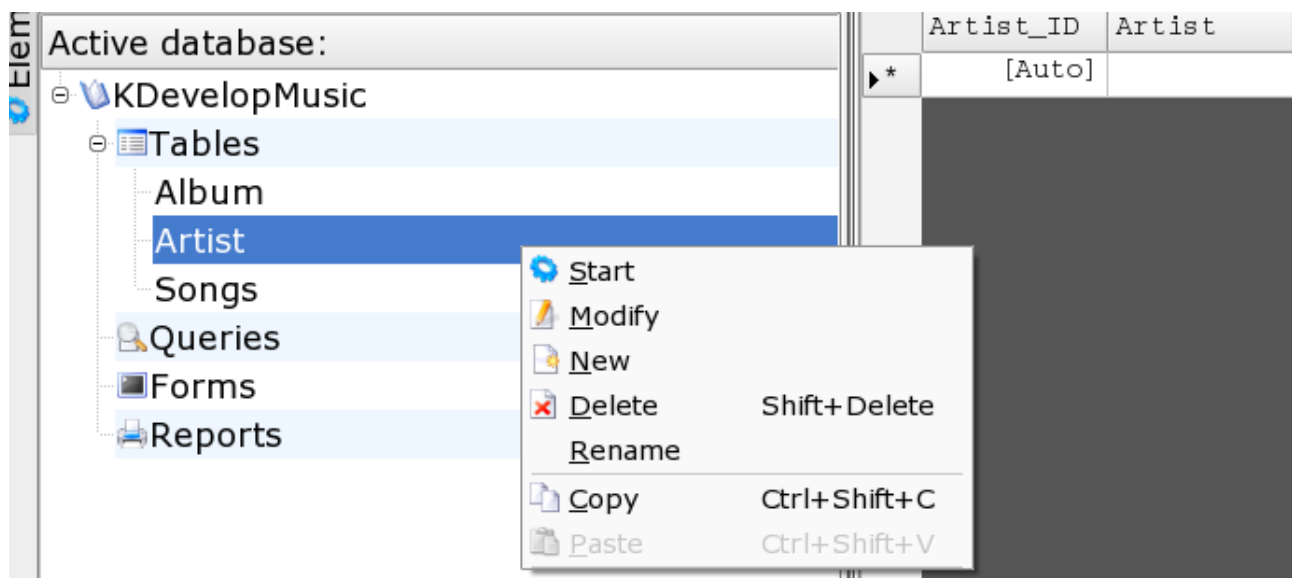
We can see here the preliminary setup for the Artist Table which contains two fields the Artist_ID and the Artist fields. To add a field to a table click on the New Field button and the fill out the options for the field in the options boxes on the top right. You can experiment with fields and tables here but you should really have a good idea of what your tables are going to contain and how they

Chapter 5 Database Programming With MySQL
are going to work together before you start.

One of the worst ways you can possibly design a database is by trying to cram everything into a single table. This is just asking for trouble because you will then me making the job of accessing the table far more difficult, for example if we know the id of an artist with this table we can use a pseudo sql statement along the lines of Select * from Artist where Artist_ID = knownvalue. If however, we also add the album names to this table we are going to end up trying to seperate out the individual album titles from the results of our select statement.

The tables created here are for a simple music database basically we are going to be storing the names of the artist the titles of the albums and the songs that each album contains. The database will be created in such a way that we can get the artist name from the song or the album name from the song but we wont include stuff like running times of songs, release dates or publishers. If you really want all those details you can always add them or record the album in amaroK.

Once you have created the tables you can start adding information to them. This is done in Knoda by right clicking the table and selecting start.



For now we will add some initial testing information in Knoda so that we can make sure that everything works as expected from this end. We can also play around with the data here so that when we get to developing programs with KDevelop we can concentrate on the programming side of things and have some sql statements already if we need them.

For test purposes I've added Three Bands with Four albums, containing sixty two songs between them. The test tables look like this, First the Artist Table,

| Table - Album | | Table - Artist | |
|---------------|-----------|----------------|--|
| | Artist_ID | Artist | |
| 1 | 1 | Korn | |
| 2 | 2 | Dinosaur Jr. | |
| 3 | 3 | Danielle Dax | |
| * | 0 | | |

Chapter 5 Database Programming With MySQL
 Followed by the Album table,

| | Album_ID | Artist_ID | Album_Title |
|---|----------|-----------|---------------------------|
| 1 | 1 | 1 | See You On The Other Side |
| 2 | 2 | 2 | Ear Bleeding Country |
| 3 | 3 | 1 | Greatest Hits Vol. 1 |
| 4 | 4 | 3 | Dark Adapted Eye |
| * | [Auto] | 0 | |

and a snippet from the Songs table,

| | Song_ID | Artist_ID | Album_ID | Song_Title |
|----|---------|-----------|----------|---------------------------|
| 24 | 24 | 2 | 2 | Not You Again |
| 25 | 25 | 2 | 2 | Out There |
| 26 | 26 | 2 | 2 | Start Choppin |
| 27 | 27 | 2 | 2 | Get Me |
| 28 | 28 | 2 | 2 | Feel The Pain |
| 29 | 29 | 2 | 2 | I Don't Think So |
| 30 | 30 | 2 | 2 | Take A Run At The Sun |
| 31 | 31 | 2 | 2 | Nothin's Goin' On |
| 32 | 32 | 2 | 2 | I'm Insane |
| 33 | 33 | 1 | 3 | Word Up |
| 34 | 34 | 1 | 3 | Another Brick In The Wall |
| 35 | 35 | 1 | 3 | Y'll Wanna Single |
| 36 | 36 | 1 | 3 | Right Now |
| 37 | 37 | 1 | 3 | Did My Time |
| 38 | 38 | 1 | 3 | Alone I Break |

In order to view this as a the result of a proper database query we need to create a new query by right clicking on Queries and selecting new.

Chapter 5 Database Programming With MySQL

Query - Standard Query

Artist0

- *
- Artist_ID
- Artist

Album1

- *
- Album_ID
- Artist_ID
- Album_Title

Songs2

- *
- Song_ID
- Artist_ID
- Album_ID
- Song_Title

| | | | | | | |
|------------|---------|---|-------------|---|------------|--------|
| Table: | Artist0 | ▼ | Album1 | ▼ | Songs2 | ▼ |
| Fieldname: | Artist | ▼ | Album_Title | ▼ | Song_Title | ▼ |
| Alias: | | | | | | |
| Order: | none | ▼ | none | ▼ | none | ▼ none |
| Show: | Yes | ▼ | Yes | ▼ | Yes | ▼ Yes |
| Criteria: | | | | | | |
| Or: | | | | | | |

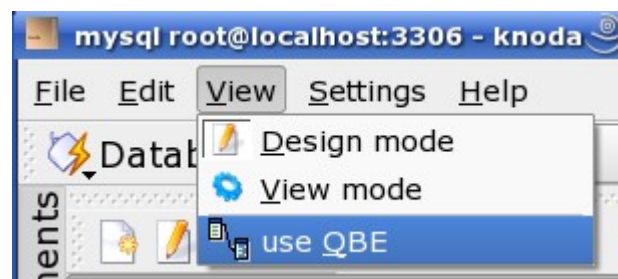
This is the Knoda graphical development for an sql query and the query itself is the standard query for this database. It basically gets all the songs according to album and artist, when you run it, like so,

Chapter 5 Database Programming With MySQL

| Query - Standard Query | | | |
|------------------------|--------------|---------------------------|---------------------|
| | Artist | Album_Title | Song_Title |
| 1 | Korn | See You On The Other Side | Twisted Transistor |
| 2 | Korn | See You On The Other Side | Politics |
| 3 | Korn | See You On The Other Side | Hypocrites |
| 4 | Korn | See You On The Other Side | Souvenir |
| 5 | Korn | See You On The Other Side | 10 Or A 2 Way |
| 6 | Korn | See You On The Other Side | Throw Me Away |
| 7 | Korn | See You On The Other Side | Love Song |
| 8 | Korn | See You On The Other Side | Open Up |
| 9 | Korn | See You On The Other Side | Coming Undone |
| 10 | Korn | See You On The Other Side | Getting Off |
| 11 | Korn | See You On The Other Side | Liar |
| 12 | Korn | See You On The Other Side | For No One |
| 13 | Korn | See You On The Other Side | Seen It All |
| 14 | Korn | See You On The Other Side | Tearjerker |
| 15 | Dinosaur Jr. | Ear Bleeding Country | Repulsion |
| 16 | Dinosaur Jr. | Ear Bleeding Country | Little Furry Things |
| 17 | Dinosaur Jr. | Ear Bleeding Country | In A Jar |
| 18 | Dinosaur Jr. | Ear Bleeding Country | Freak Scene |
| 19 | Dinosaur Jr. | Ear Bleeding Country | Budge |
| 20 | Dinosaur Jr. | Ear Bleeding Country | Just Like Heaven |
| 21 | Dinosaur Jr. | Ear Bleeding Country | The Wagon |

I've called it the Standard Query because any other queries that are used will basically be an edit to this query. There are two ways that you can develop SQL queries with Knoda the most obvious is through the graphical interface as shown above and the other for those who can write SQL off the top of their heads you can type in the SQL.

To get to the SQL editor select the View menu,



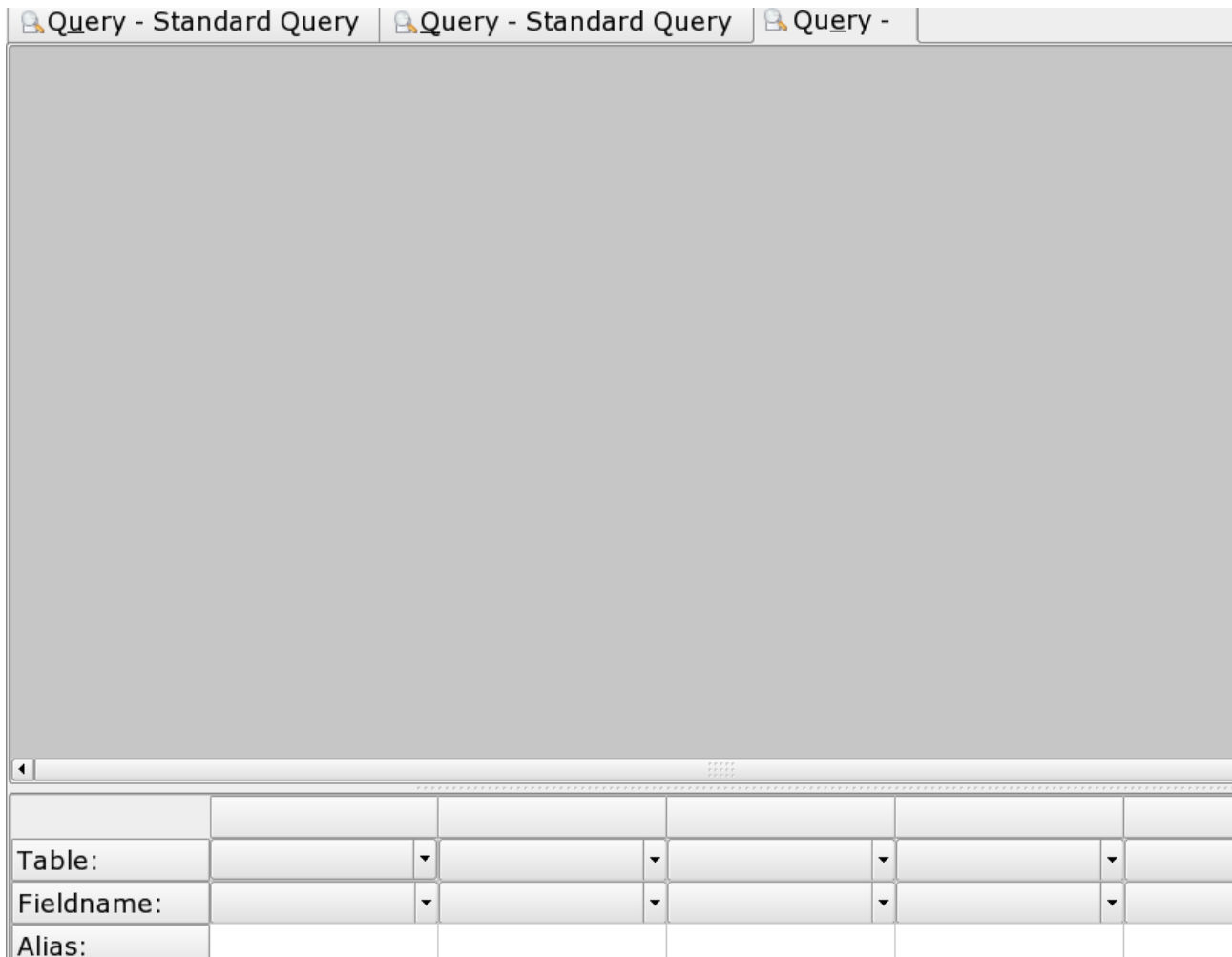
and select use QBE this will turn of the graphical development and allow you to type the SQL straight in to the editor. The SQL for the Standard Query is,

```
SELECT "Artist0"."Artist" , "Album1"."Album_Title" , "Songs2"."Song_Title" FROM "Artist"
"Artist0" , "Album" "Album1" , "Songs" "Songs2" WHERE
("Album1"."Artist_ID"="Artist0"."Artist_ID") AND
("Songs2"."Album_ID"="Album1"."Album_ID")
```

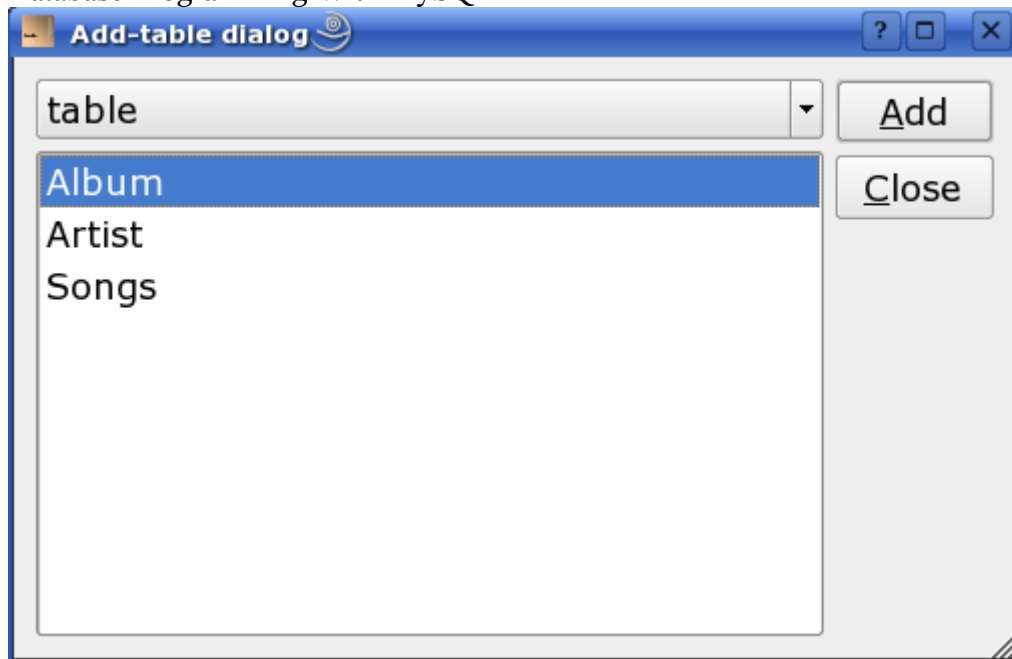
Though seeing as the graphical interface is there and it opens by default it seems kinda silly not to use it if it is going to make life easier.

Chapter 5 Database Programming With MySQL

When we first start a new query the tables shown in the picture above are not there so we need to add them we do this by right clicking in the empty space above the comboboxes.

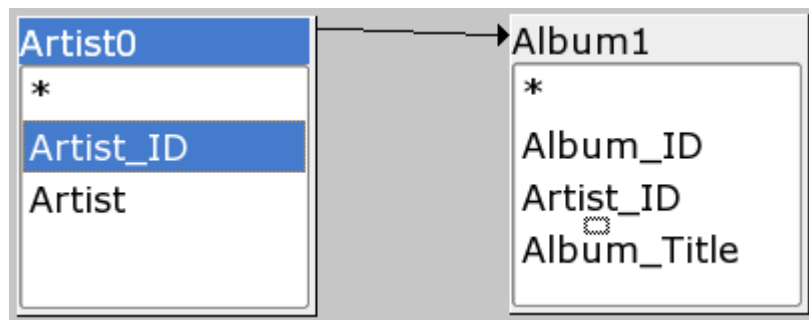


this will give you a menu with one option “Add datasource”. When you select this a dialog box will open showing you the tables that are available in the currently selected database.



You are free to add as many of the tables as you want and in the case of this example we add all of them before closing the dialog.

The first thing you'll want to do when you have added the tables to the query is set up the links between the tables. This is done by selecting the item in the table to link and dragging it to the appropriate item in another table.



for example in the above the `Artist_ID` in the `Artist` table should map to the `Artist_ID` in the `Album` table. It is good practice to have any links between tables to be named the same in each table, the idea being that anyone looking at the tables can then easily see what links to what. Whilst on the subject of names the alert ones will have noticed that Knoda has changed the table names adding a number beginning with zero to the table names as they were added. These are just aliases for the tables which are resolved in the from part of the SQL statement.

```
FROM "Artist" "Artist0" , "Album" "Album1" , "Songs" "Songs2"
```

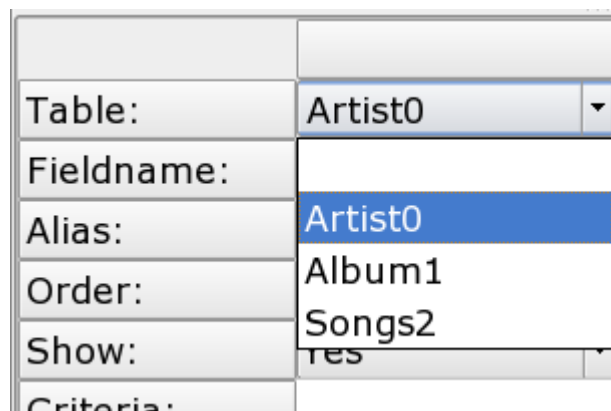
which tells the database that “Artist0” is an alias for “Artist”, etc.

When you release the mouse over the table that you wish to establish a link with you will get a dialog.



which shows you the link that Knoda thinks you are trying to make and allows you to change it if you meant to link to another item in the table.

Once you have established the links all you need to do is setup exactly what it is that you want to display. This is done through the drop down boxes at the bottom of the screen. If you are a complete beginner it can take a few attempts to get this to display exactly the way that you want to but as a basic rule of thumb you should have tables that display the least information on the left and work right with the tables displaying more information as you go.



As you can see once the tables have been added to the query they will be selectable from the drop down box for the table. Once we have selected the table we can select any fields from that table that we want to display.

| | | |
|------------|---------|-------------|
| | | |
| Table: | Artist0 | Album1 |
| Fieldname: | Artist | Album_Title |
| Alias: | | |
| Order: | none | * |
| Show: | Yes | Album_ID |
| Criteria: | | Artist_ID |
| Or: | | Album_Title |

In the Standard Query that we have set up in this example we only show the information that is relevant to the topic. All the items that end in ID are designed for us to use from within the database and are therefore not really important to anyone who is looking at the database simply to see what tracks are on a specific album.

KDevelop Database Support

KDevelop has three gui widgets that can be used for displaying database information these are the Qt class widgets DataTable, DataBrowser and DataView.

Using a DataTable

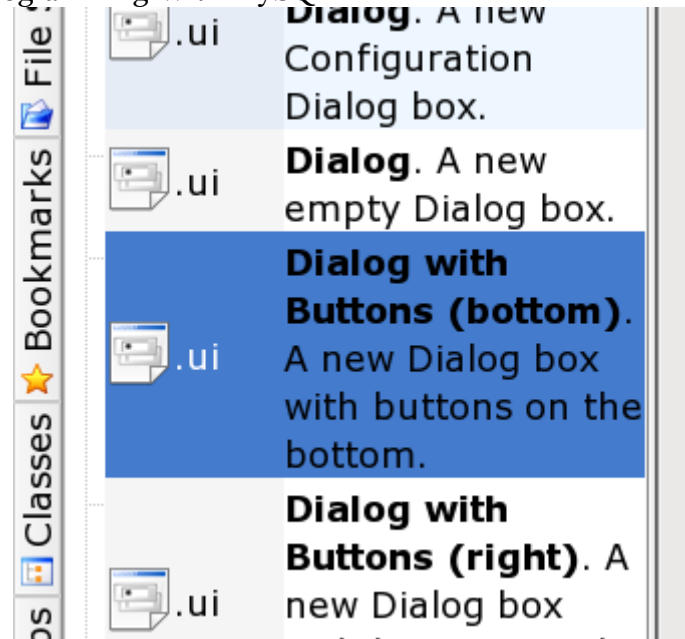
Unfortunatly database programming isn't as straight forward as it could be at the moment so it is better to start with the idea that we are going to have to do most of the work by hand. The main source of the problems being that while the DataTable will attempt to set up a database connection if you go through the wizard to set it up. It doesn't save the information that you use to log into the database which means that if you try to just use the wizards and run the code you will be told that there was an error connecting to the database.

The way to get the QDataTable to work is to drop it onto the form and then just cancel the wizard. This will add a blank QDataTable to you form. We can work with this.

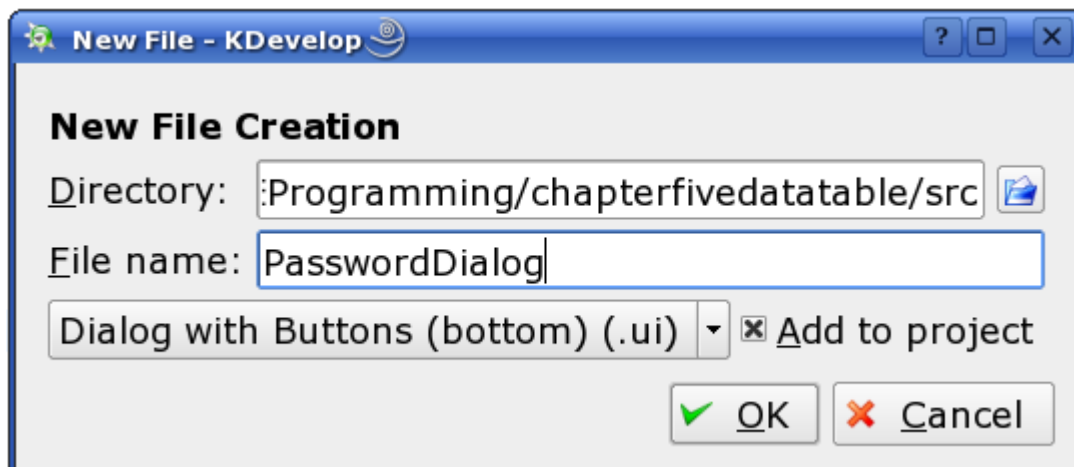
First of all we are going to have to add a dialog so that we can get the user name and the password from the user. We could hard code these but it's just plain nasty so in an effort not to encourage bad habits we'll do it properly from the start.

Adding A Dialog

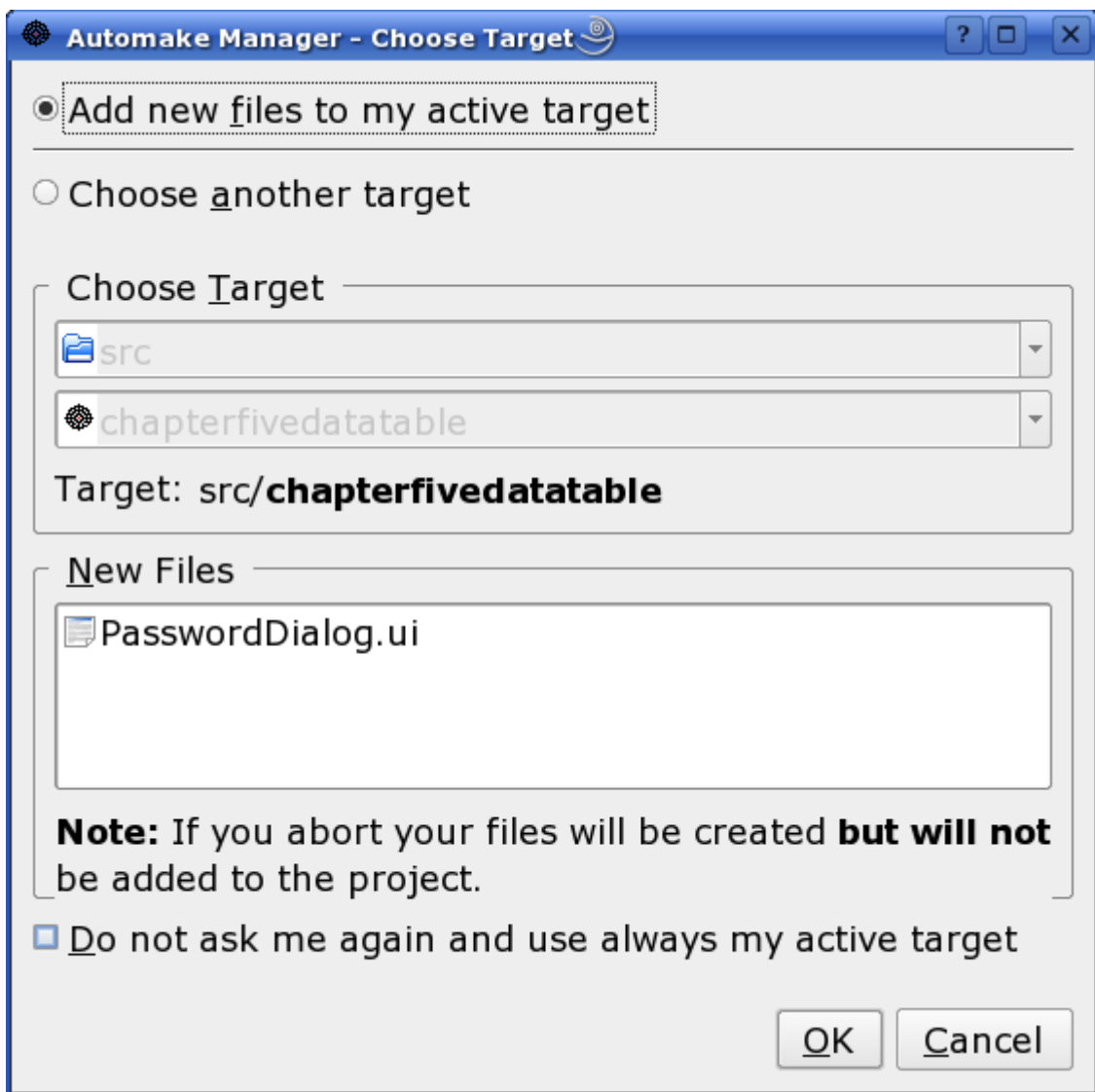
To add a new dialog to the project click on the "New File" tab on the left side of KDevelop and select the appropriate dialog,



for our simple user and password dialog we'll select the standard Dialog with Buttons (bottom). When we select the dialog we are given a setup dialog,



This just asks what the name of the file is to be and gives a final opportunity to change our minds about which dialog we want.



We are then presented with an automake dialog that asks which of the current projects we want to add this to. If you just want to add the dialog to the current project you are working on then accept the defaults by clicking OK.

Although we now have a new ui file we still do not have any class files to implement the new user interface, in this case our password dialog. There is however a need to be careful about class names here so don't go rushing in creating files called PasswordDialog.h and PasswordDialog.cpp or you could start making life really complicated and annoying. If however you remember what was said in earlier chapters about how the meta object compiler works and how it will create a header and .cpp file for a .ui file even though you never see it and it is this file that controls the class declarations for the form widgets. Any code implementation of a form needs to inherit from these classes, which admittedly is a bit tricky at the moment as they don't exist yet.

So first of all we need to get the meta object compiler to generate our class and header file. This is done by running Automake and friends followed by configure from the build menu and then building the project. Once the project is built you will see the newly created files in the projectname/debug/src directory. In this case they are the passworddialog.h which looks like

```
#ifndef PASSWORDDIALOG_H
```


Chapter 5 Database Programming With MySQL

```
#define PASSWORDDIALOG_H

#include <qvariant.h>
#include <qdialog.h>

class QVBoxLayout;
class QHBoxLayout;
class QGridLayout;
class QSpacerItem;
class QPushButton;

class passwordDialog : public QDialog
{
    Q_OBJECT

public:
    passwordDialog( QWidget* parent = 0, const char* name = 0, bool modal = FALSE, WFlags
fl = 0 );
    ~passwordDialog();

    QPushButton* buttonHelp;
    QPushButton* buttonOk;
    QPushButton* buttonCancel;

protected:
    QHBoxLayout* Layout1;
    QSpacerItem* Horizontal_Spacing2;

protected slots:
    virtual void languageChange();

};
```

and the passworddialog.cpp file.

```
#include <kdialog.h>
#include <klocale.h>
/*****
** Form implementation generated from reading ui file
** '/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chapterfivedatatable/src/PasswordDialog.ui'
**
** Created: Fri Mar 3 18:09:34 2006
** by: The User Interface Compiler ($Id: qt/main.cpp 3.3.4 edited Nov 24 2003 $)
**
** WARNING! All changes made in this file will be lost!
*****/

#include "PasswordDialog.h"

#include <qvariant.h>
#include <qpushbutton.h>
#include <qlayout.h>
#include <qtooltip.h>
#include <qwhatsthis.h>

/*
 * Constructs a passwordDialog as a child of 'parent', with the
 * name 'name' and widget flags set to 'f'.
 *
 * The dialog will by default be modeless, unless you set 'modal' to
 * TRUE to construct a modal dialog.
 */
passwordDialog::passwordDialog( QWidget* parent, const char* name, bool modal, WFlags
fl )
    : QDialog( parent, name, modal, fl )
{
    if ( !name )
        setName( "passwordDialog" );
```

Chapter 5 Database Programming With MySQL

```
setSizeGripEnabled( TRUE );

QWidget* privateLayoutWidget = new QWidget( this, "Layout1" );
privateLayoutWidget->setGeometry( QRect( 20, 240, 476, 33 ) );
Layout1 = new QHBoxLayout( privateLayoutWidget, 0, 6, "Layout1");

buttonHelp = new QPushButton( privateLayoutWidget, "buttonHelp" );
buttonHelp->setAutoDefault( TRUE );
Layout1->addWidget( buttonHelp );
Horizontal_Spacing2 = new QSpacerItem( 20, 20, QSizePolicy::Expanding,
QSizePolicy::Minimum );
Layout1->addItem( Horizontal_Spacing2 );

buttonOk = new QPushButton( privateLayoutWidget, "buttonOk" );
buttonOk->setAutoDefault( TRUE );
buttonOk->setDefault( TRUE );
Layout1->addWidget( buttonOk );

buttonCancel = new QPushButton( privateLayoutWidget, "buttonCancel" );
buttonCancel->setAutoDefault( TRUE );
Layout1->addWidget( buttonCancel );
languageChange();
resize( QSize(511, 282).expandedTo(minimumSizeHint()) );
clearWState( WState_Polished );

// signals and slots connections
connect( buttonOk, SIGNAL( clicked() ), this, SLOT( accept() ) );
connect( buttonCancel, SIGNAL( clicked() ), this, SLOT( reject() ) );
}

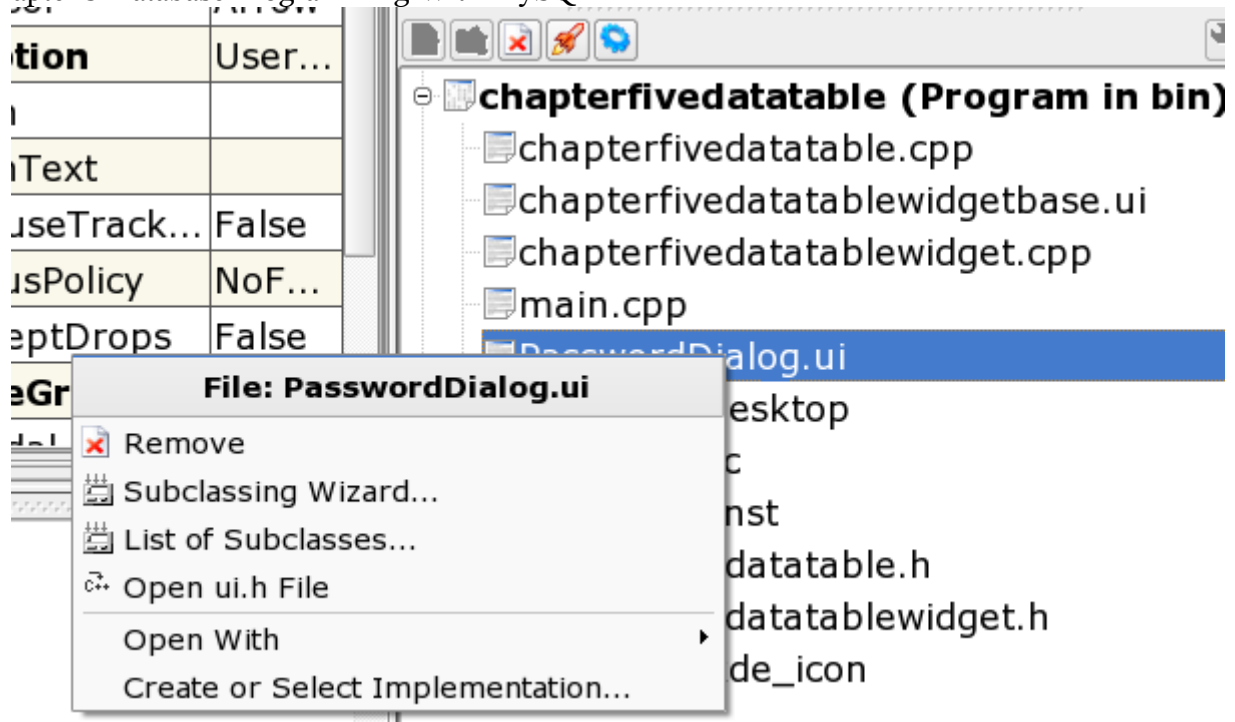
/*
 * Destroys the object and frees any allocated resources
 */
passwordDialog::~passwordDialog()
{
    // no need to delete child widgets, Qt does it all for us
}

/*
 * Sets the strings of the subwidgets using the current
 * language.
 */
void passwordDialog::languageChange()
{
    setCaption( tr2i18n( "UserName And Password" ) );
    buttonHelp->setText( tr2i18n( "&Help" ) );
    buttonHelp->setAccel( QKeySequence( tr2i18n( "F1" ) ) );
    buttonOk->setText( tr2i18n( "O&K" ) );
    buttonOk->setAccel( QKeySequence( tr2i18n( "Alt+K" ) ) );
    buttonCancel->setText( tr2i18n( "Ca&ncel" ) );
    buttonCancel->setAccel( QKeySequence( tr2i18n( "Alt+N" ) ) );
}

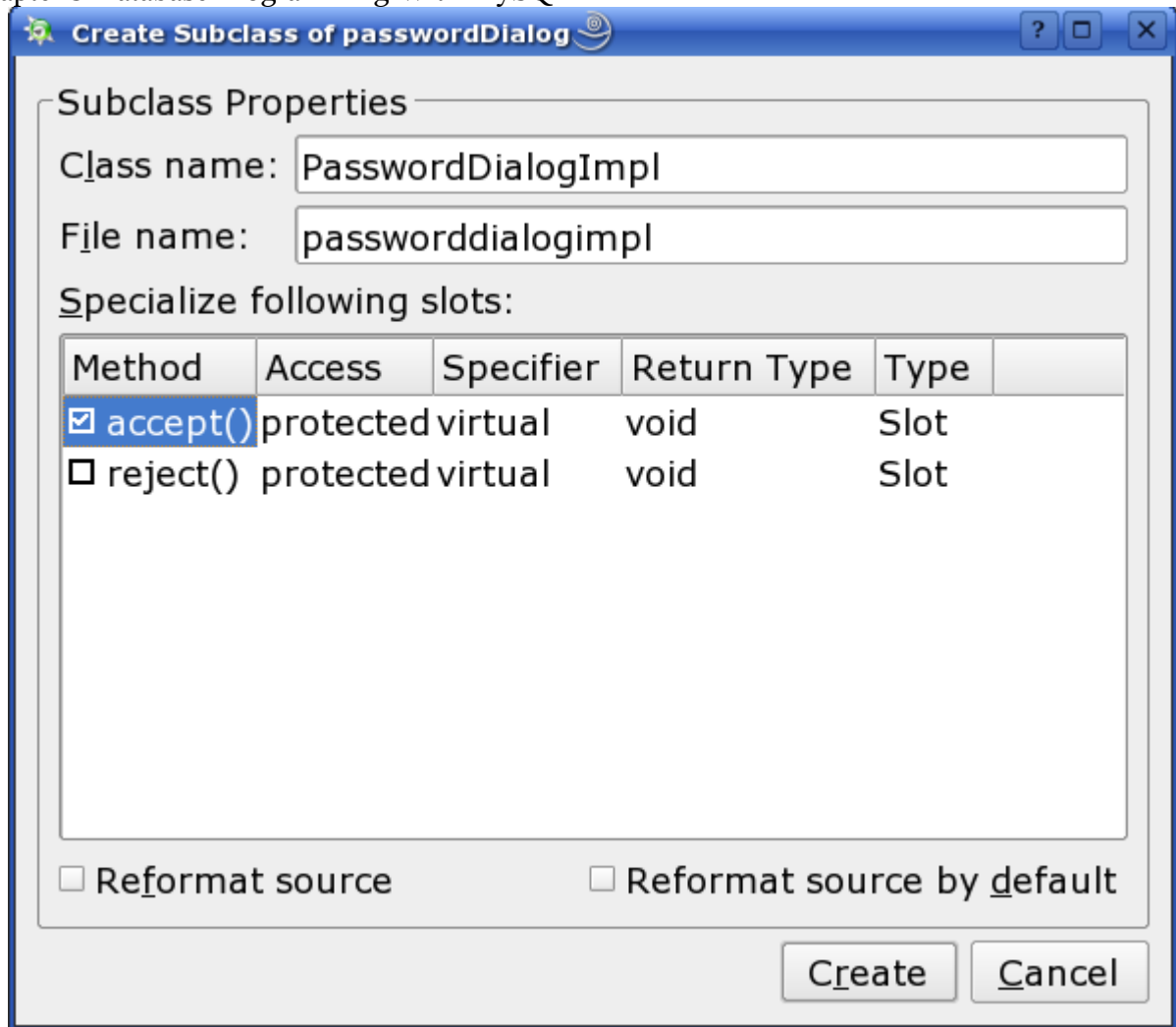
#include "PasswordDialog.moc"
```

Now we can set about getting our dialog up an running. Right click on the the .ui file in the Automake Manager.

Chapter 5 Database Programming With MySQL

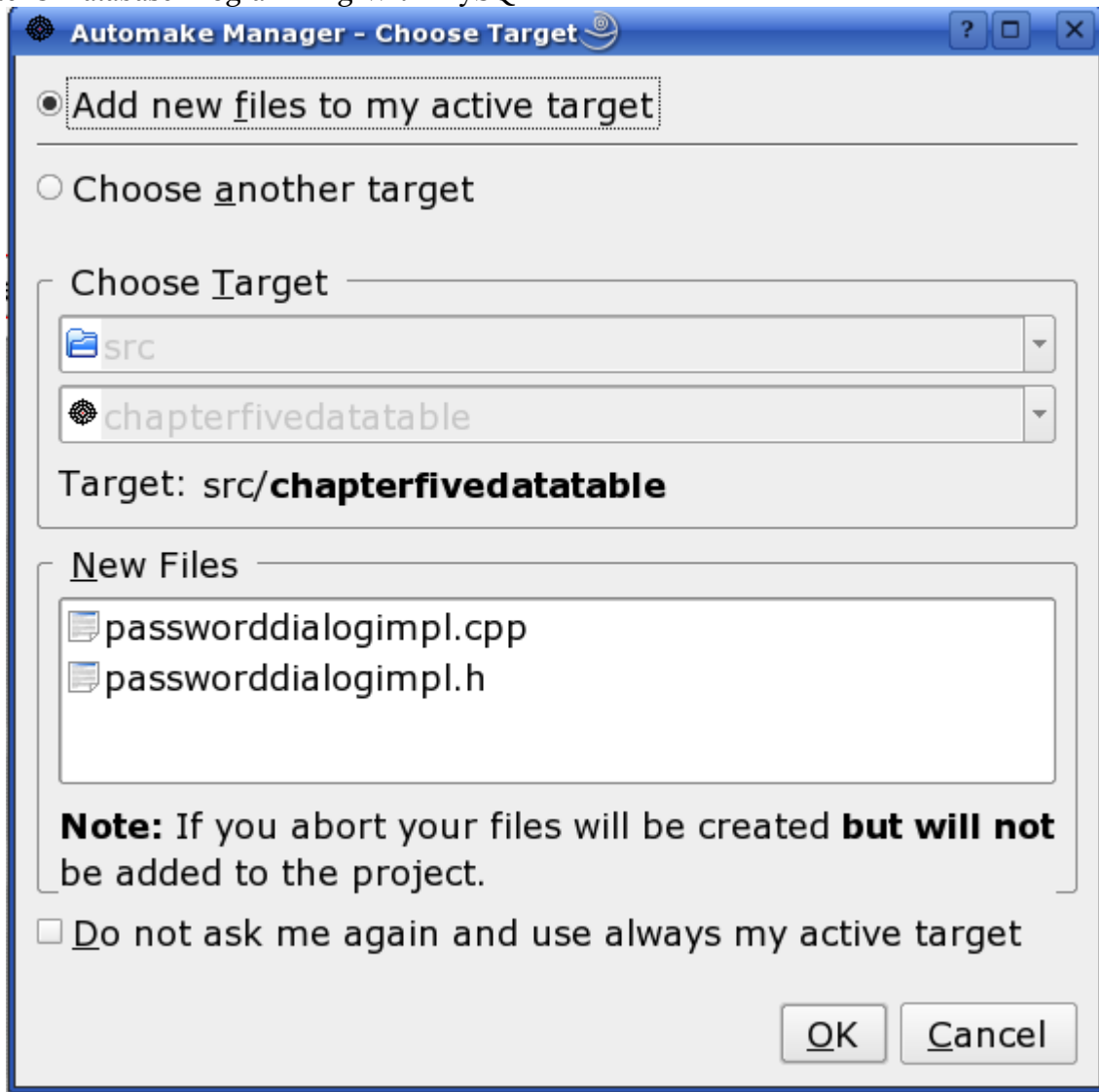


In order to create the classes to implement the PasswordDialog we need to select either the “Subclassing Wizard” or the “Create or Select Implementation”, the actual file creation is the same in either one,



The main difference is that with the Subclassing Wizard as you can see you can choose to specialise the implementation of some of the buttons in this case we are going to need to implement the accept function so that we can later save the strings added for the username and the password and use them to access the database. In order to avoid naming issues I've named the class PasswordDialogImpl to make it clear that this is the implementation of the password dialog.

The final dialog that we get is the Automake Manager dialog again,



Here we can see it adding the .cpp and .h files and if we look at the .h file we can see,

```
#ifndef PASSWORDDIALOGIMPL_H
#define PASSWORDDIALOGIMPL_H

#include "PasswordDialog.h"

class PasswordDialogImpl : public passwordDialog
{
    Q_OBJECT

public:
    PasswordDialogImpl(QWidget* parent = 0, const char* name = 0, bool modal = FALSE,
        WFlags fl = 0 );
    ~PasswordDialogImpl();
    /*$PUBLIC_FUNCTIONS$*/

public slots:
    /*$PUBLIC_SLOTS$*/

protected:
    /*$PROTECTED_FUNCTIONS$*/

protected slots:
    /*$PROTECTED_SLOTS$*/
    virtual void accept();
```

Chapter 5 Database Programming With MySQL

```
};
```

not only that it inherits from the MOC created passwordDialog class but provides and implementation of the accept() function. As we have just added new files to the project run automake and friends from the build menu followed by configure and then run build to make sure it all builds correctly. Now we can get on with implementing the dialog.

Implementing The Dialog

From the Dialog point of view the implementation is straightforward we declare a couple of strings, one for the user name and one for the password,

```
private:
    /** Store the UserName
    **/
    QString strUserName;
    /** Store the Password
    **/
    QString strPassword;
```

and add the get and set functionality, and then we add the following code to the overridden accept function,

```
setPassword( passwordLineEdit->text() );
setUserName( userLineEdit->text() );
QDialog::accept();
```

The technical bit appears when we realise that we want to have the dialog appear as the application starts but preferably before it displayed. This way when the application starts it will show the database table data as we want it without it obviously drawing it after the application has started. To do this we need to override the polish function. The polish function performs the initialisation of the widget before it is shown but the documentation comes with the warning that you should call the base class polish function before implementing your own code. This ensures that the default code for the widget is completed before you make your changes, as making the changes before the base classes have called polish can trigger further calls to polish, ending up with infinite recursion.

So our overriding of the polish function looks like this,

```
void ChapterFiveDataTableWidget::polish()
{
    QWidget::polish();

    PasswordDialogImpl *dlg = new PasswordDialogImpl();

    if( dlg->exec() == QDialog::Accepted )
    {
        loadDatabase( dlg->userName(), dlg->password() );
    }
}
```

Here we call the base class polish and then create a new dialog and execute it modally with the exec function. Note that calling show would display a modeless dialog that would have the main application widget displaying in the background which is exactly what we don't want it to do.

Query The Database

Now that we have the username and the password entered as the application starts there are three more parameters that we need to be able to connect to the database. These are the host computer and the database name and the driver that we are going to use to connect to the database. These are

Chapter 5 Database Programming With MySQL

the parameters that are entered into the Knoda connection dialog so they should be familiar by now. I've set up strings to hold the variables in the ChapterFiveDataTableWidget class and provided the appropriate access functions although this class is not accessed from anywhere so they could just be left as private variables that are initialised in the constructor as,

```
strName = "KDevelopMusic";
strDriver = "QMYSQL3";
strHost = "localhost";
```

These variables are all used in the loadDatabase function which is called at the end of our overridden polish function

The driver variable starts things off and is used with the line,

```
dbConnection = QSqlDatabase::addDatabase( strDriver );
```

This call initialises our database connection variable that is declared in the ChapterFiveDataTableWidget class declaration as,

```
QSqlDatabase *dbConnection;
```

The detailed listings of database drivers that can be used with Qt are in the SQL Module Drivers section of the help.

Now we set up the connection parameters to the database,

```
dbConnection->setDatabaseName( name() );
dbConnection->setUserName( user );
dbConnection->setPassword( password );
dbConnection->setHostName( host() );
```

```
if( dbConnection->open() == true )
```

As you can see we are simply using code to set exactly the same parameters we used for Knoda, so as long as we can get into Knoda there should be no reason for the dbConnection->open call to return false.

We are now in the position where we are ready to actually query the database and display it and when using a datatable as long as we have done our preparation properly the actual querying of the database is disgustingly easy. First of all a browse through the QSqlCursor class help file and it's derived class QSqlSelectCursor shows us that we can pass a table name to QSqlCursor or an SQL statement to QSqlSelectCursor constructor, along with our database collection and this will give us the results of our query in our newly constructed cursor object which we can then pass to the QDataTable object and let it do all the drawing and displaying for us. In code this looks like this,

```
///QSqlCursor *cursor = new QSqlCursor( "Artist", true, dbConnection );

QSqlSelectCursor *cursor = new QSqlSelectCursor( "SELECT Artist FROM Artist",
dbConnection );

///      QSqlSelectCursor *cursor = new QSqlSelectCursor( strSqlStatement, dbConnection );

dataTable1->setSqlCursor( cursor, true, false );
dataTable1->refresh();

for( int i=0; i<dataTable1->numCols(); i++ )
{
    dataTable1->adjustColumn( i );
}
```

Chapter 5 Database Programming With MySQL

The first call we use which is blanked out in the above code is the creation of a QSqlCursor object that takes the Artist table as a parameter and an autopopulate value of true along with the database connection pointer.

We then pass the cursor to the QDataTable by calling the setSqlCursor function, passing in the cursor itself and the value true for the autopopulate option which tells the table to automatically display the data. The final parameter is for the autodelete and as we don't want the table to be edited here we set it to false, which is the default. The following call to QDataTable refresh tells the table to redraw itself and it will use the now current cursor to do this.

The final piece of code just gets the number of columns and then tells all the table columns to adjust their width to that of the largest entry in the column.

The resulting view of the table is,



The screenshot shows a Qt application window titled "ChapterFiveDataTable". Inside the window is a QTableWidget displaying a table with three columns and three rows of data. The first column contains row indices (1, 2, 3), the second column contains Artist IDs (1, 2, 3), and the third column contains Artist names (Korn, Dinosaur Jr., Danielle Dax).

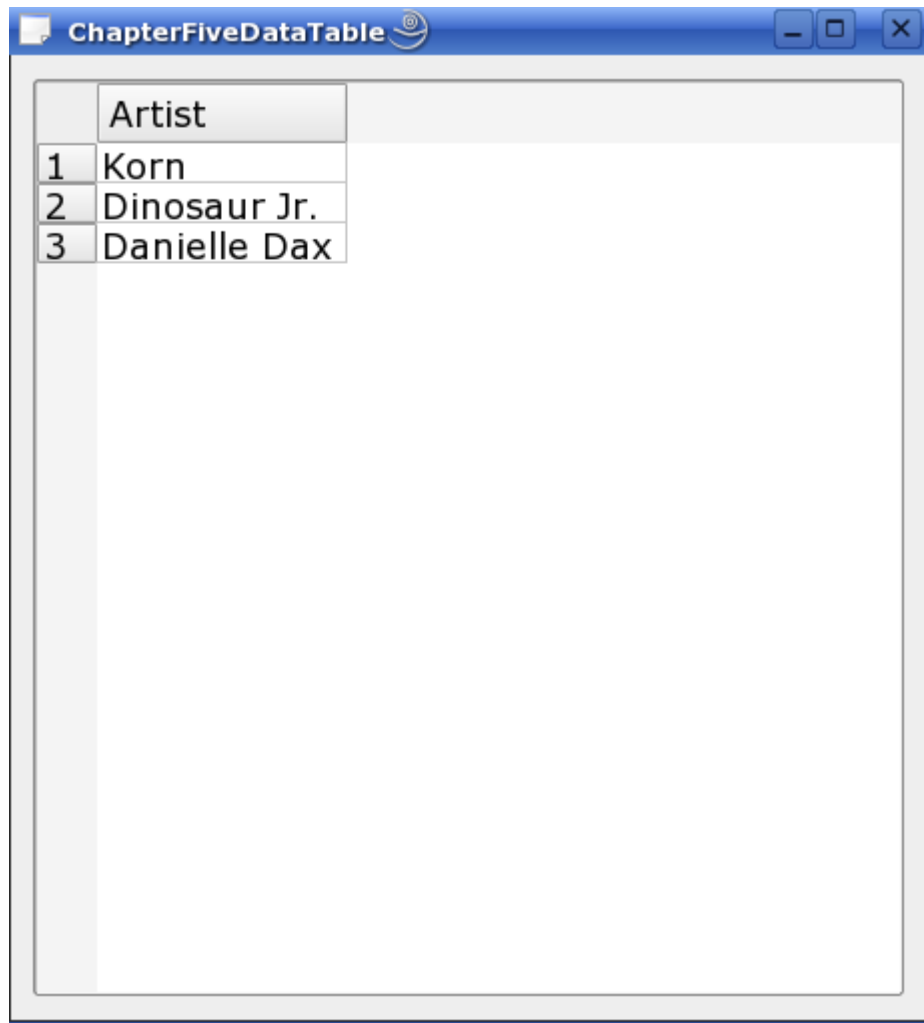
| | Artist_ID | Artist |
|---|-----------|--------------|
| 1 | 1 | Korn |
| 2 | 2 | Dinosaur Jr. |
| 3 | 3 | Danielle Dax |

which is fine and is exactly what we asked for as it shows all the data in the table, the only problem being that it is not really what we want as the Artist_ID is a key value and is of no interest to anyone who is simply trying to view the records in the database so we'll switch to the QSqlSelectCursor and see if we can get something a little more appropriate.

If we use the SQL statement

```
"SELECT Artist FROM Artist"
```


Chapter 5 Database Programming With MySQL
in the QSqlSelect constructor we get,



The image shows a screenshot of a Qt application window titled "ChapterFiveDataTable". Inside the window, there is a table widget displaying data from a database. The table has two columns: an index column and an "Artist" column. The data is as follows:

| | Artist |
|---|--------------|
| 1 | Korn |
| 2 | Dinosaur Jr. |
| 3 | Danielle Dax |

which is much more along the lines of what we want. But of course now we have to play just a little more and see what the StandardQuery we did in Knoda would look like,



| | Artist | Album_Title | Song_Title |
|----|--------------|---------------------------|---------------------|
| 1 | Korn | See You On The Other Side | Twisted Transistor |
| 2 | Korn | See You On The Other Side | Politics |
| 3 | Korn | See You On The Other Side | Hypocrites |
| 4 | Korn | See You On The Other Side | Souvenir |
| 5 | Korn | See You On The Other Side | 10 Or A 2 Wav |
| 6 | Korn | See You On The Other Side | Throw Me Away |
| 7 | Korn | See You On The Other Side | Love Sona |
| 8 | Korn | See You On The Other Side | Open Up |
| 9 | Korn | See You On The Other Side | Coming Undone |
| 10 | Korn | See You On The Other Side | Getting Off |
| 11 | Korn | See You On The Other Side | Liar |
| 12 | Korn | See You On The Other Side | For No One |
| 13 | Korn | See You On The Other Side | Seen It All |
| 14 | Korn | See You On The Other Side | Tearierker |
| 15 | Dinosaur Jr. | Ear Bleeding Country | Repulsion |
| 16 | Dinosaur Jr. | Ear Bleeding Country | Little Furry Things |
| 17 | Dinosaur Jr. | Ear Bleeding Country | In A Jar |
| 18 | Dinosaur Jr. | Ear Bleeding Country | Freak Scene |
| 19 | Dinosaur Jr. | Ear Bleeding Country | Budae |
| 20 | Dinosaur Jr. | Ear Bleeding Country | Just Like Heaven |
| 21 | Dinosaur Jr. | Ear Bleeding Country | The Wagon |

Tip Of The Day

On rare occasions the KDevelop environment can apparently get itself into an endless loop when trying to load a .ui on startup. If this happens it can make the project unaccessible and the only way to shut down the KDevelop to close is to force it by ending the process with KSysGuard which is found in System/Monitor/Performance Monitor (KSysGuard). To do this select the Process Table. Then find the KDevelop process and highlight it by clicking on it and then hit the kill button.

If this does happen and you can't get KDevelop to load the project then we can fix it with a quick bit of editing. KDevelop keeps track of what's in each project through a file with the extension .kdevses, in this case chapterfivedatatable.kdevses and it looks like,

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<!DOCTYPE KDevPrjSession>
<KDevPrjSession>
  <DocsAndViews NumberOfDocuments="4" >
    <Doc0 NumberOfViews="1"
URL="file:///home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chapterfivedatatable/src/chapterfivedatatablewidget.h" >
      <View0 line="55" Type="Source" />
    </Doc0>
    <Doc1 NumberOfViews="1"
URL="file:///home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chapterfivedatatable/src/chapterfivedatatable.h" >
      <View0 line="44" Type="Source" />
    </Doc1>
    <Doc2 NumberOfViews="1"
```

Chapter 5 Database Programming With MySQL

```
URL="file:///home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chapterfivedatatable/src/chapterfivedatatablewidget.cpp" >
```

```
<View0 line="51" Type="Source" />
```

```
</Doc2>
```

```
<Doc3 NumberOfViews="1"
```

```
URL="file:///home/pseudonym67/Dev/KDE/BeginningKDEProgramming/chapterfivedatatable/src/chapterfivedatatablewidgetbase.ui" >
```

```
<View0 Type="Other" />
```

```
</Doc3>
```

```
</DocsAndViews>
```

```
<pluginList>
```

```
<kdevdebugger>
```

```
<breakpointList/>
```

```
</kdevdebugger>
```

```
<kdevbookmarks>
```

```
<bookmarks/>
```

```
</kdevbookmarks>
```

```
<kdevvalgrind>
```

```
<executable path="" params="" />
```

```
<valgrind path="" params="" />
```

```
<calltree path="" params="" />
```

```
<kcachegrind path="" />
```

```
</kdevvalgrind>
```

```
</pluginList>
```

```
</KDevPrjSession>
```

As you can see this is a standard .xml file that KDevelop uses to keep track of the current files, breakpoints etc. that are being used by the project. The documents being viewed have the format of the name of the document and the viewtype that KDevelop is to render them in, so if for example the chapterfivedatatablewidgetbase.ui file was refusing to load properly then all we would have to is remove the information telling KDevelop to load that document which in this case is remove the xml for loading Doc3 and then restart KDevelop. The project should then load fine and you will be able to load the .ui file normally from the Automake Manager once KDevelop has started.

Using the DataBrowser

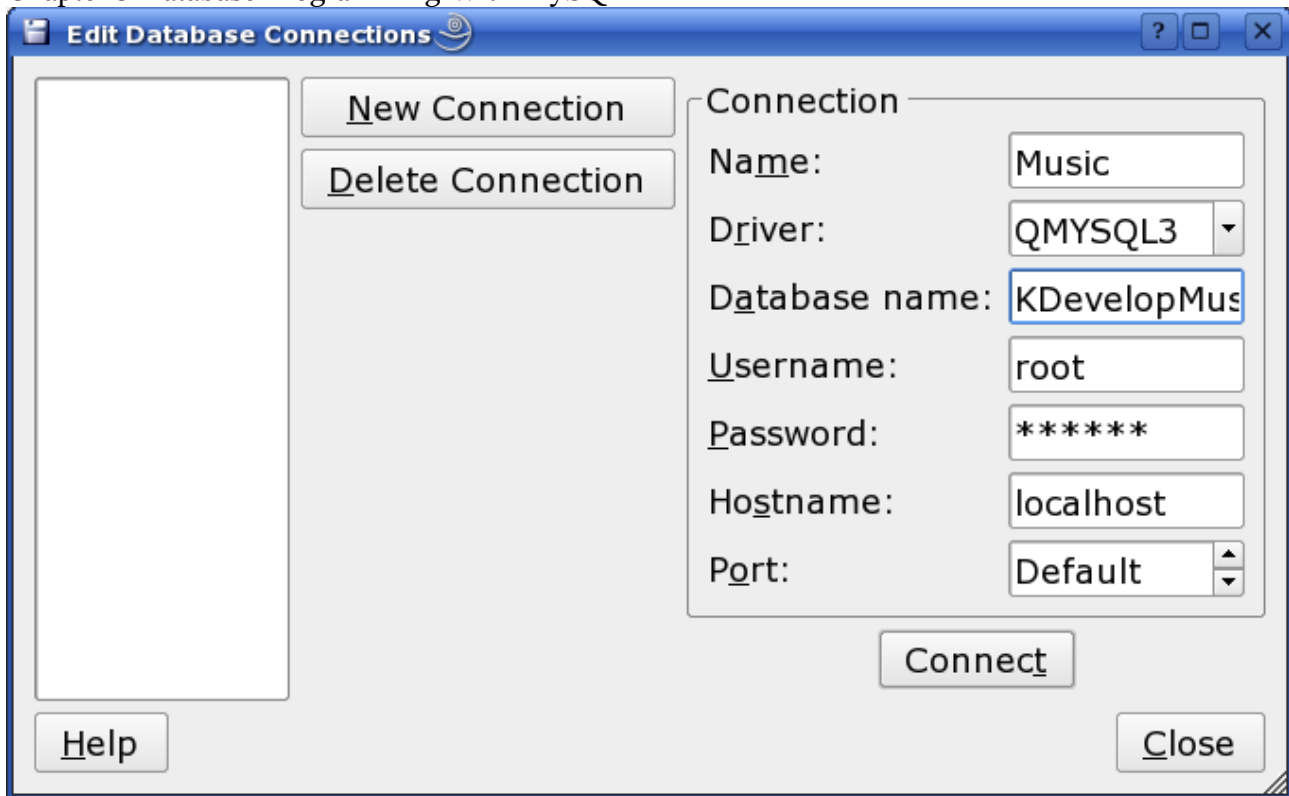
Setting up the DataBrowser project can be a little tricky so here's the least painful way to do it. Create the using the Simple Designer based KDE Application as with all the others and remove the label and the button as well as all the code. Then break the layout on the form and resize it so that you can fit the QDataBrowser object onto it. Then do a complete build of the project, you will need to run automake and friends and then configure, just to make sure that everything is satisfactory. Then shut down KDevelop. The object of this is so that when you load KDevelop this project is loaded on startup. Restart KDevelop and then add the QDataBrowser to the form. If the Database Connection wizard starts up,



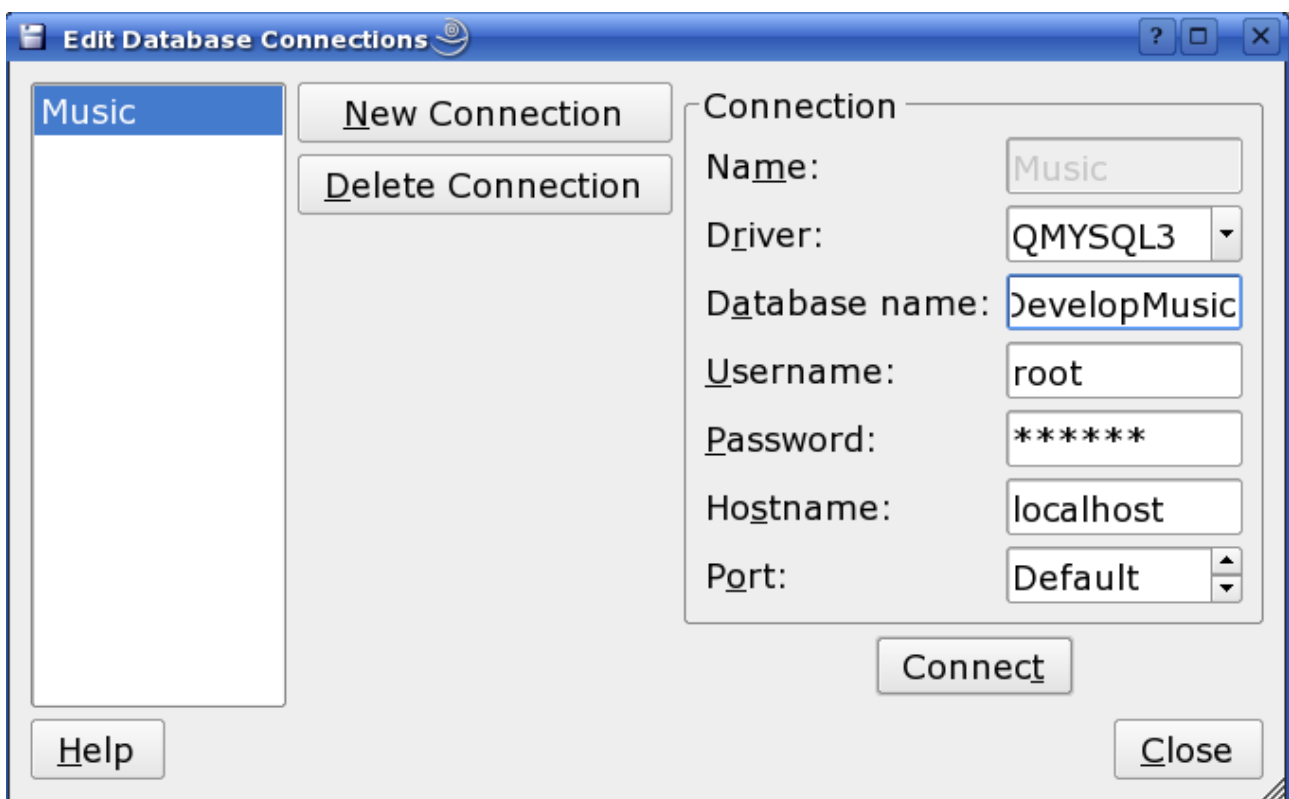
Then everything is fine, otherwise you are just going to have to restart KDevelop and try again.

For this project we will want to run through the wizard as although it does not retain the database connection information and we will have to code that by hand it does give us a gui for our query results and that saves us the task of having to do it.

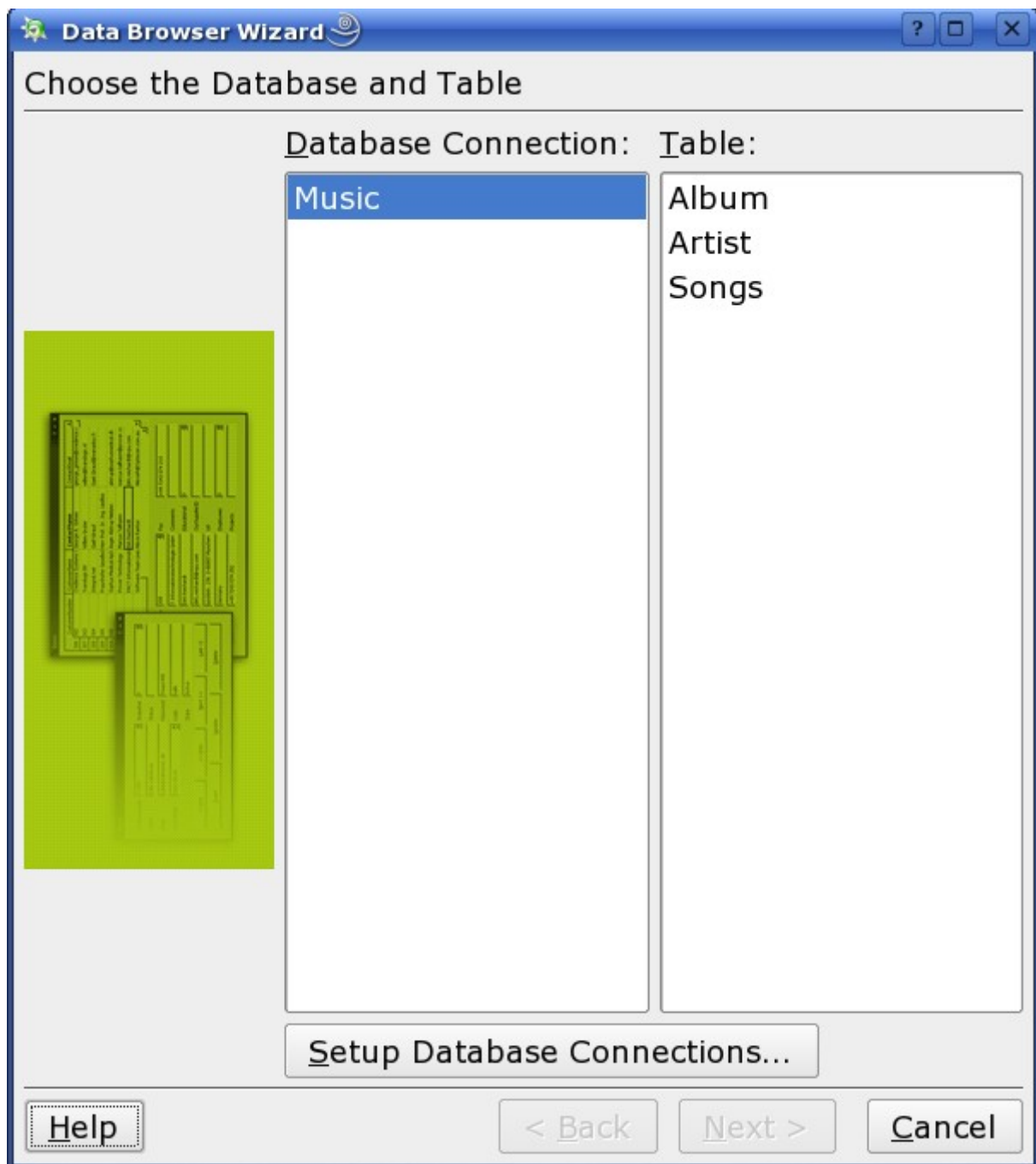
First of all we click on the Setup Database Connections button and fill out the connection dialog,



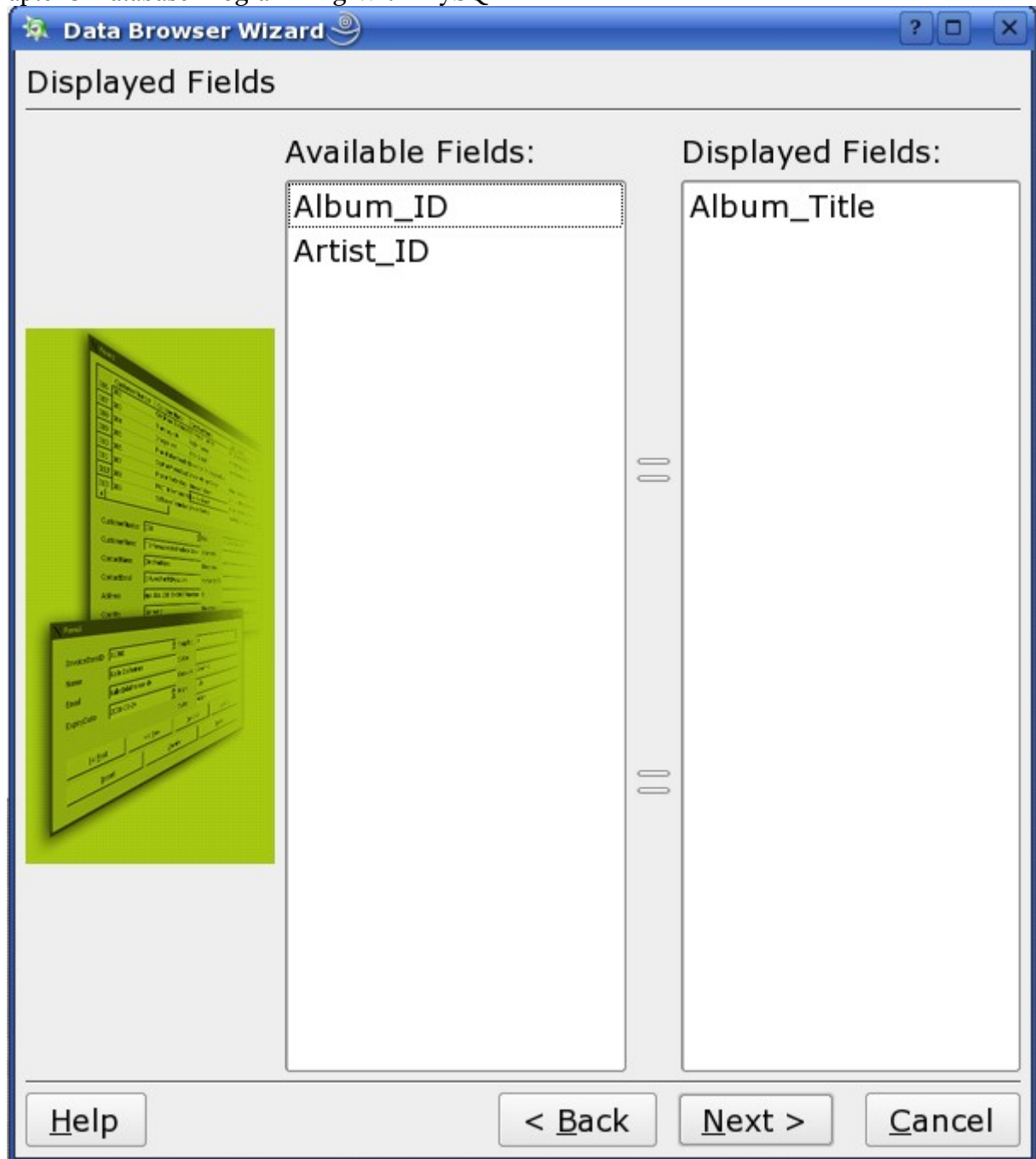
as you can see the only change here is that we have provided a name for the connection, the rest of the input is as it has been throughout this chapter. Clicking on the connect button gives us,



the Music connection has been established so we can click the close button,

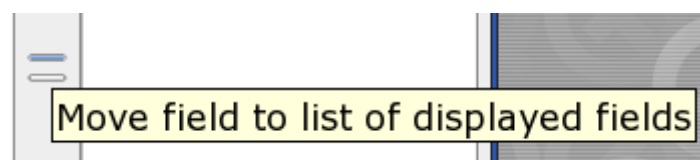


We used the artist table last time so this time we'll select the album table and click Next.

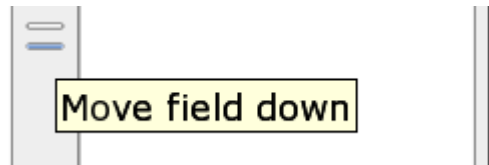
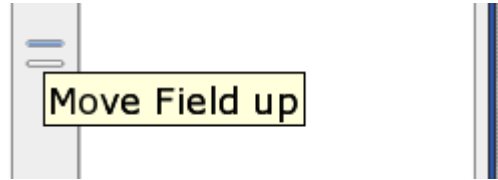
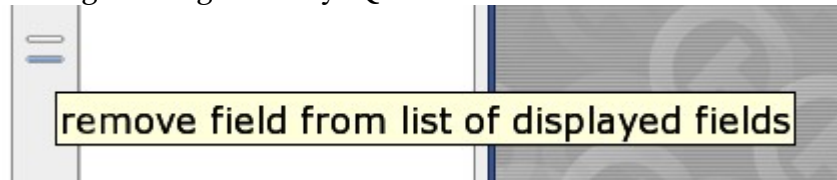


I'll admit at this point that this database is probably not ideal for demonstrating the browser as it would be more effective if the tables were to contain more items for display but even with one displayed field you should get the idea and the programming principals will apply no matter how many fields are present in the database table.

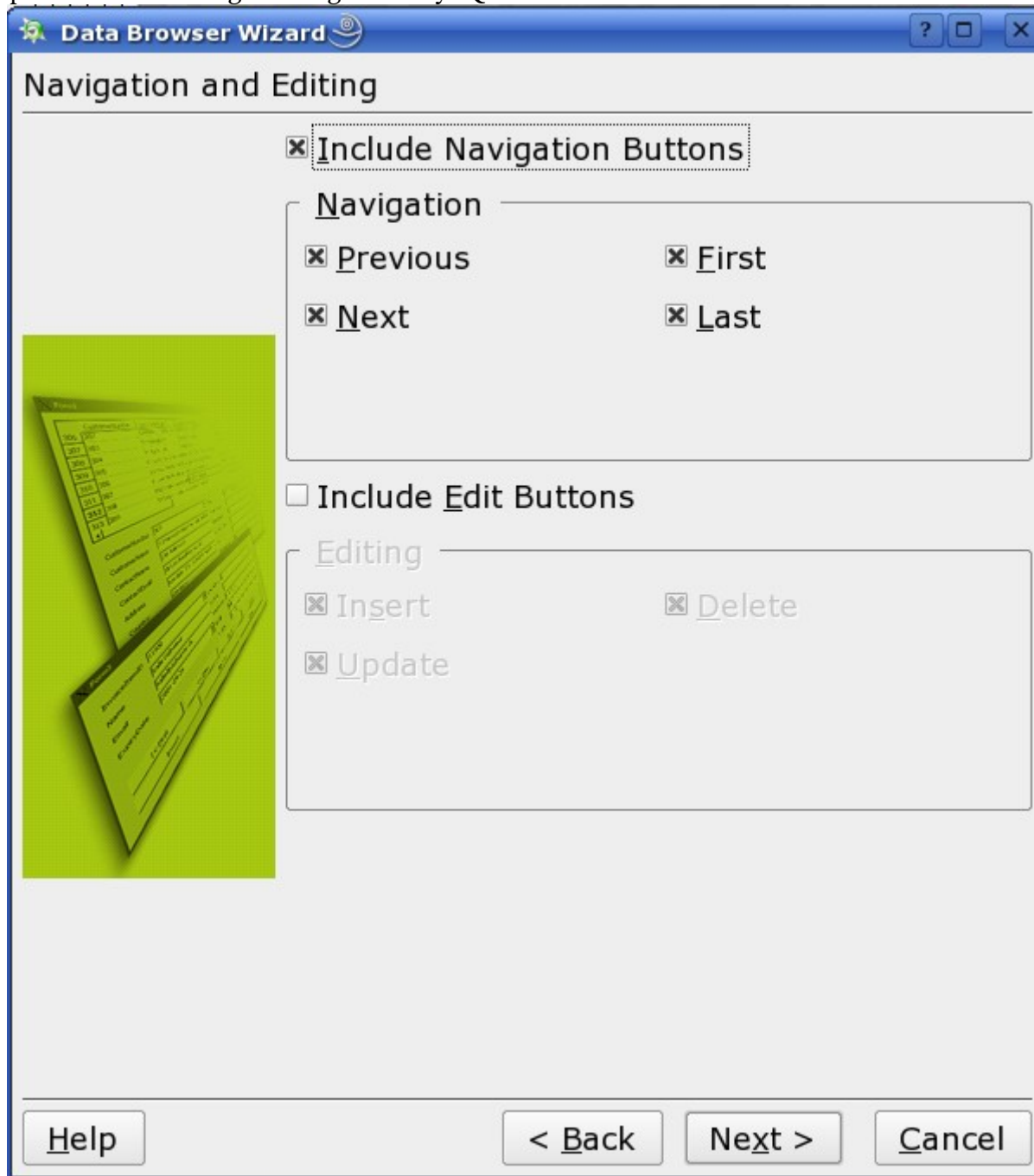
The selection of fields is controlled by the buttons in the middle which are not very clear here but they are from the top down,



Chapter 5 Database Programming With MySQL



Once we have the fields set up that we want we click next.



As you can see the Include Edit Buttons is disabled here as allowing people to add an album without being able to add the songs for the album.

The next screen in the wizard allows us to add a where clause to the SQL,

Data Browser Wizard

SQL

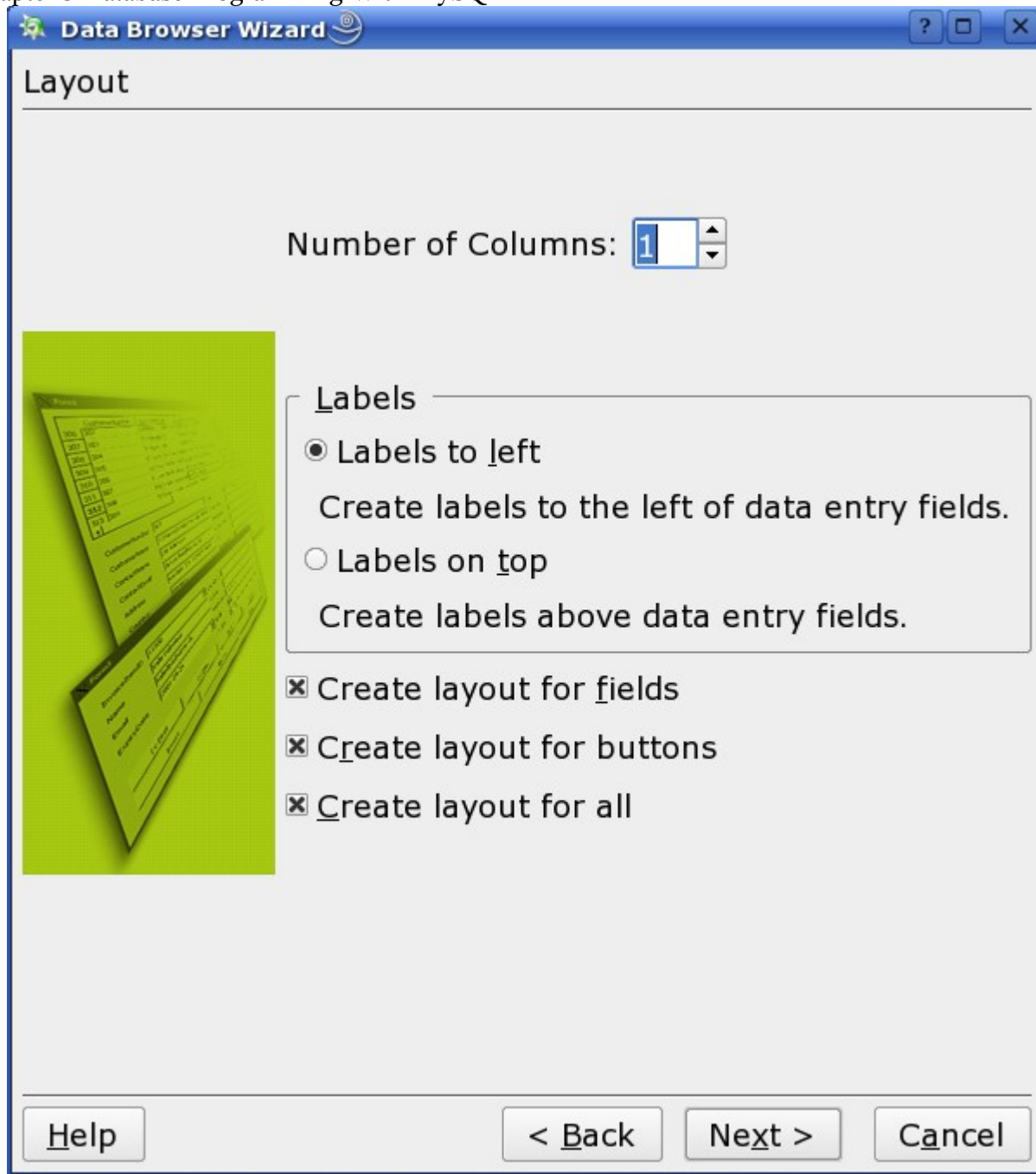
Filter: (a valid WHERE clause, e.g. id > 100)

Sort:

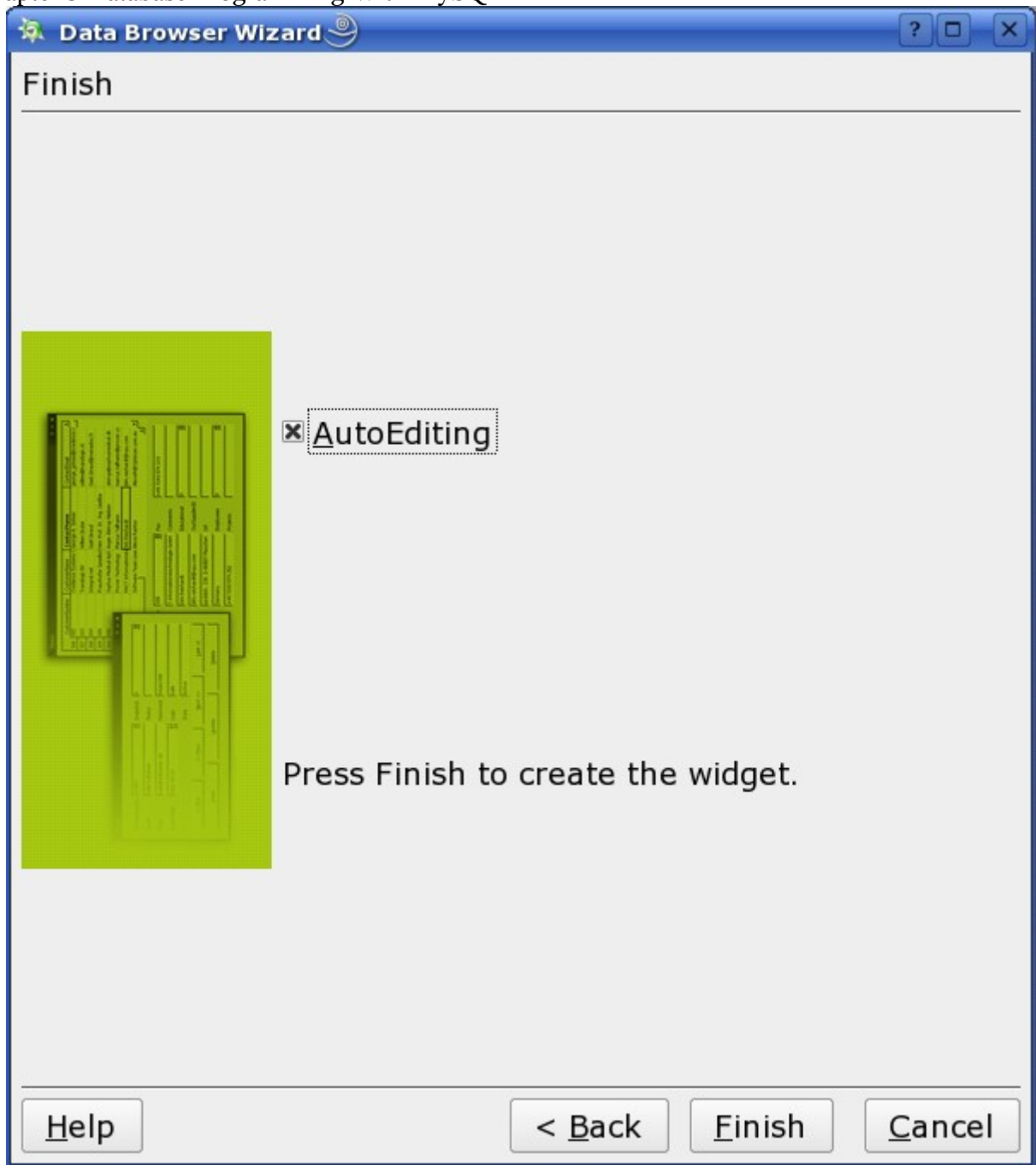
| Available Fields | Sort By |
|------------------|-----------------|
| Album_ID | Album_Title ASC |
| Artist_ID | |
| Album_Title | |

Help **< Back** **Next >** **Cancel**

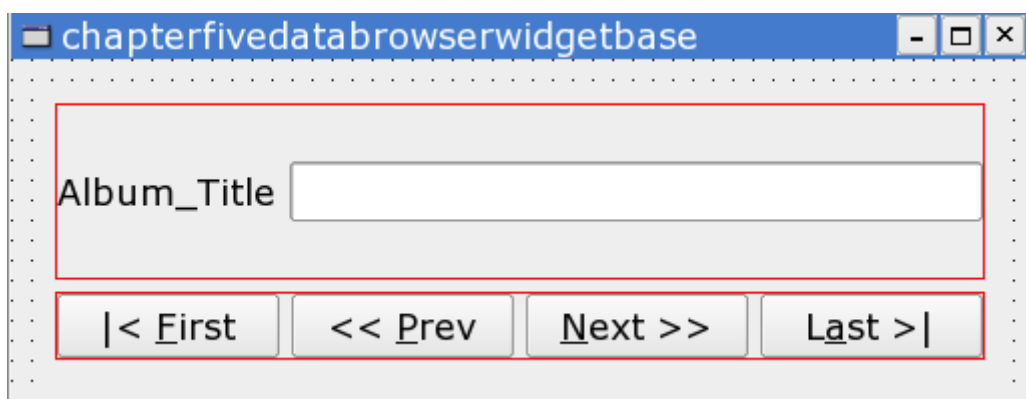
and to set the sort order for the displayed fields. Then we set the display options,



The final option allows us to set if we want the wizard to generate the SQL for updates,

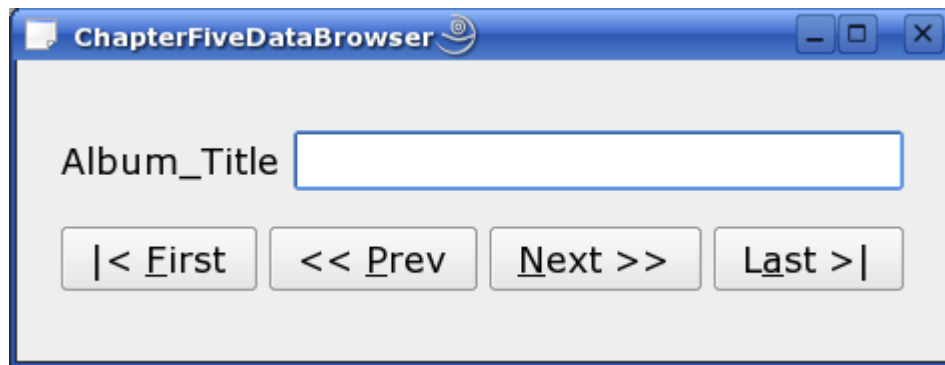


This gives us a gui that looks like



Chapter 5 Database Programming With MySQL

and a running program that looks like,



Reusing A Dialog

The next thing to do is add the polish function to our class that inherits from the “projectname”base class we do this in the usual way by adding

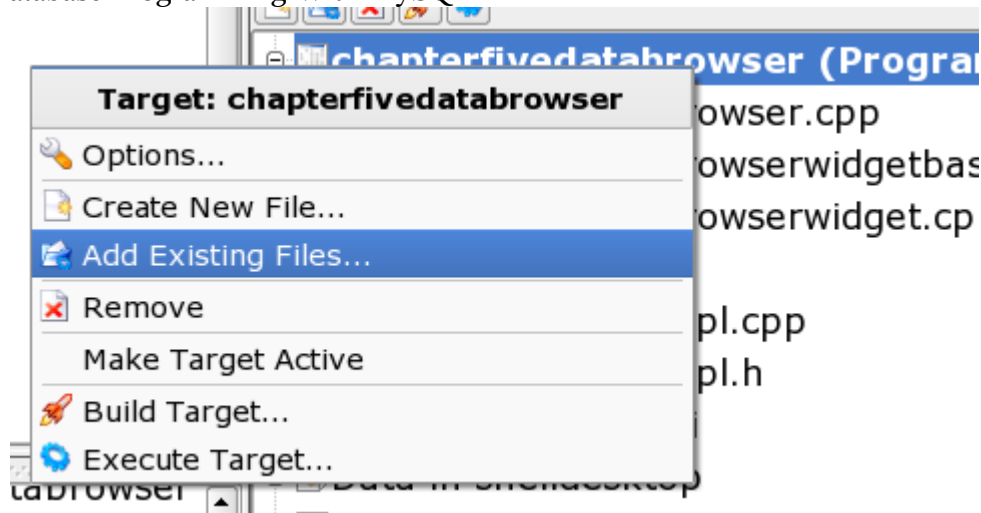
```
public slots:
    /*$PUBLIC_SLOT$*/
    virtual void polish();
```

to the header file and then adding the implementation to the cpp file. There is some code from the the wizard generated file that we can use. The implementation of the polish function in the ChapterFiveDataBrowserWidgetBase class is,

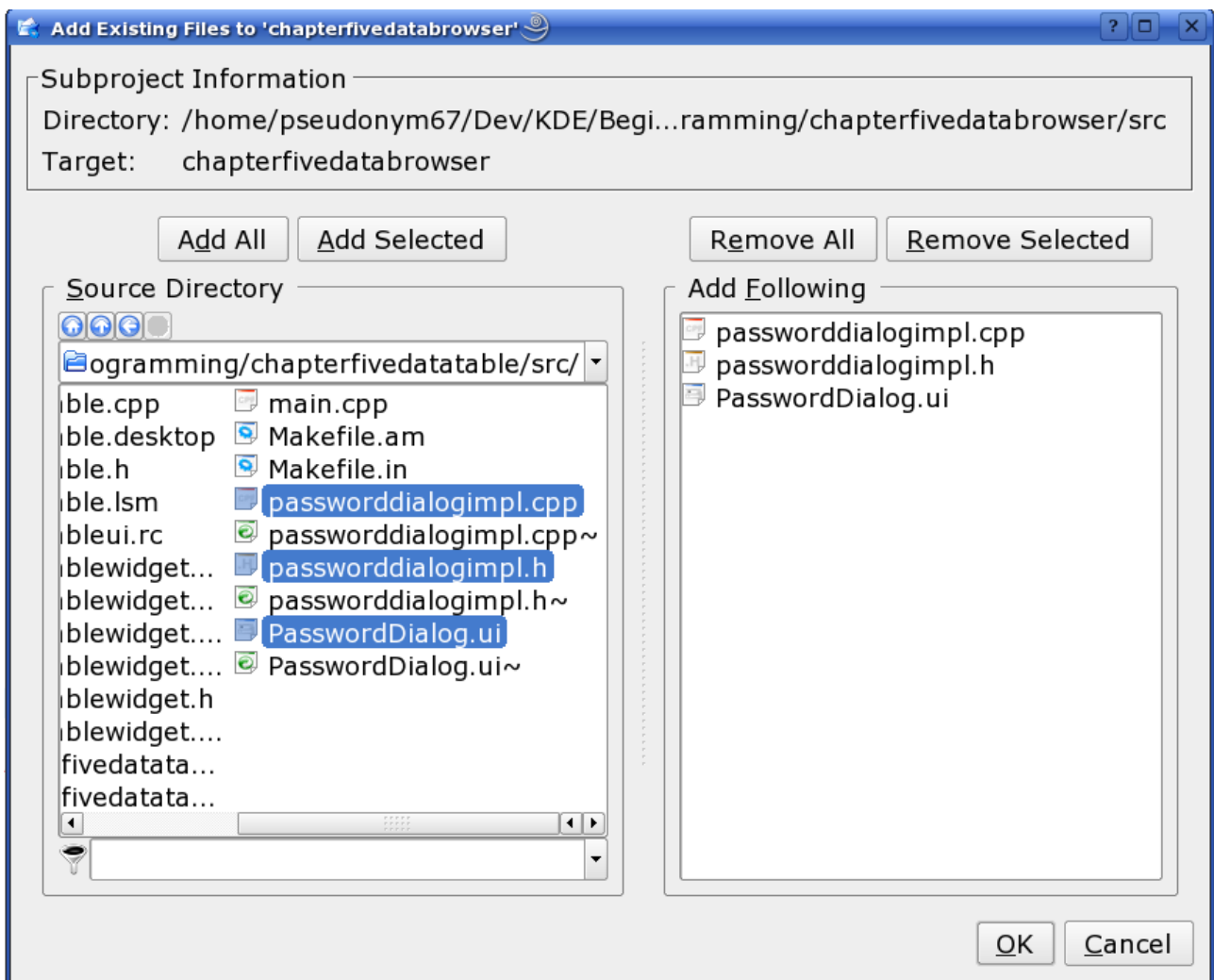
```
if ( dataBrowser3 ) {
    if ( !dataBrowser3->sqlCursor() ) {
        QSqlCursor* cursor = new QSqlCursor( "Album", TRUE, MusicConnection );
        dataBrowser3->setSqlCursor( cursor, TRUE );
        dataBrowser3->refresh();
        dataBrowser3->first();
    }
}
```

which is useful to us later but it wont work as it is because the necessary parameters for the MusicConnection which is the name given by the wizard to the QSqlDatabase, have not been set yet.

First of all we need to get the user name and password using the dialog that we used in the chapterfivedatatable project. We do this by right clicking on the project in the Automake Manager,

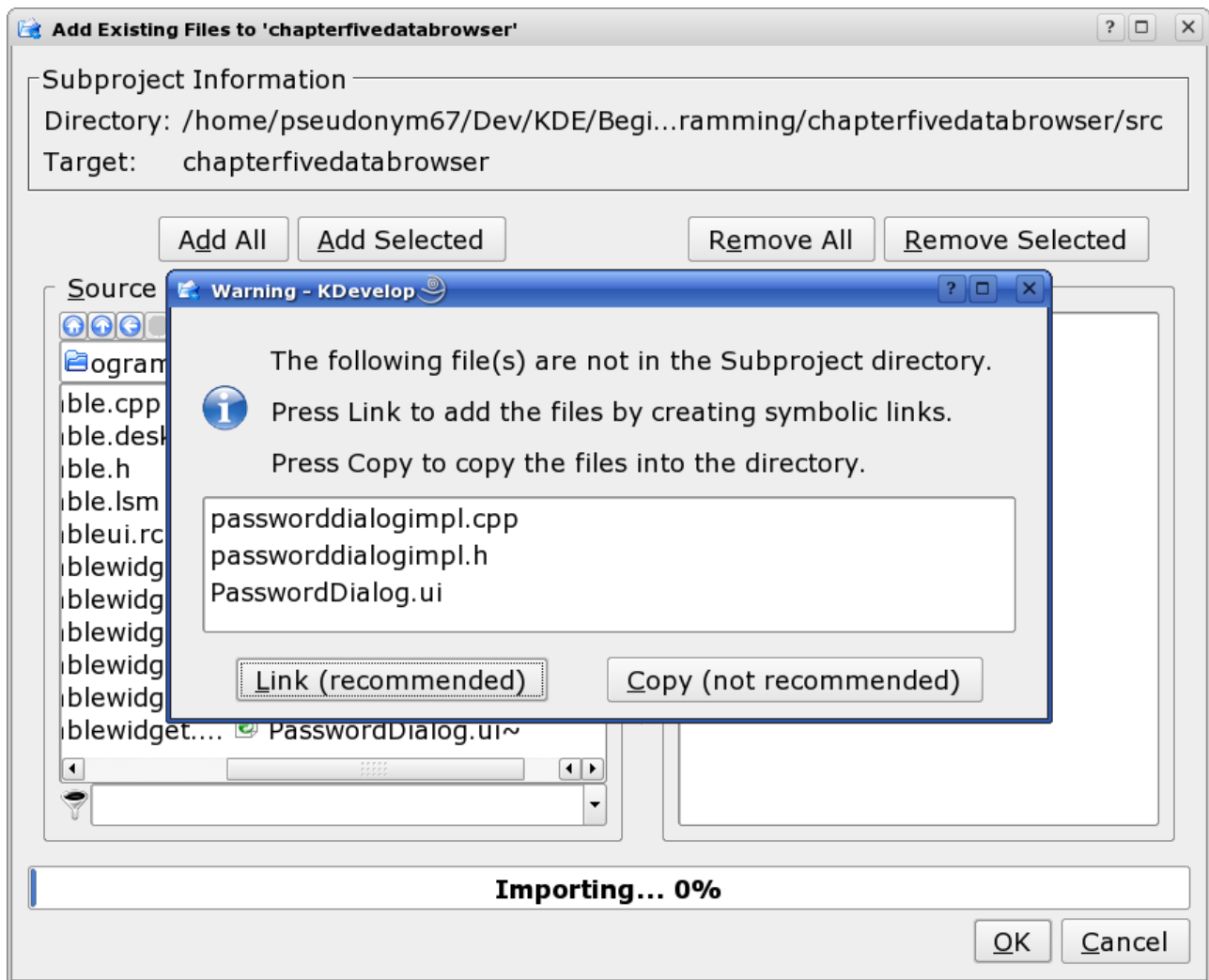


and selecting Add Existing Files.. which gives us this dialog.



Select the .ui file and the implementation header and definition file and click Add Selected. As long as you don't start changing the names of things the implementation should work without a hitch as the MOC will generate "projectname"widgetbase classes exactly as it did with the original project.

Chapter 5 Database Programming With MySQL
The next screen you will get will be.



You will be asked if you want to link or copy the files. In the normal scheme of things you would have the classes that you reuse in a standard directory and link to them rather than copy them this then means that if you do need to make any changes to the class files then they are automatically propagated through any projects and you don't need to start searching your hard disk for different implementations of the same class.

In this case though they have been copied to the project as in this case the project is a stand alone project that should be distributed in such a way as it can be built with as few issues as possible.

Now add the dialog to the polish function,

```
// Notice calling QWidget polish here as we
// are overriding the behaviour added by the wizard
// which initially overrode polish in the base class.
//
QWidget::polish();

PasswordDialogImpl *dlg = new PasswordDialogImpl();

if( dlg->exec() == QDialog::Accepted )
{
```

and build the project making sure that you have all the required headers in place. Then add the setup

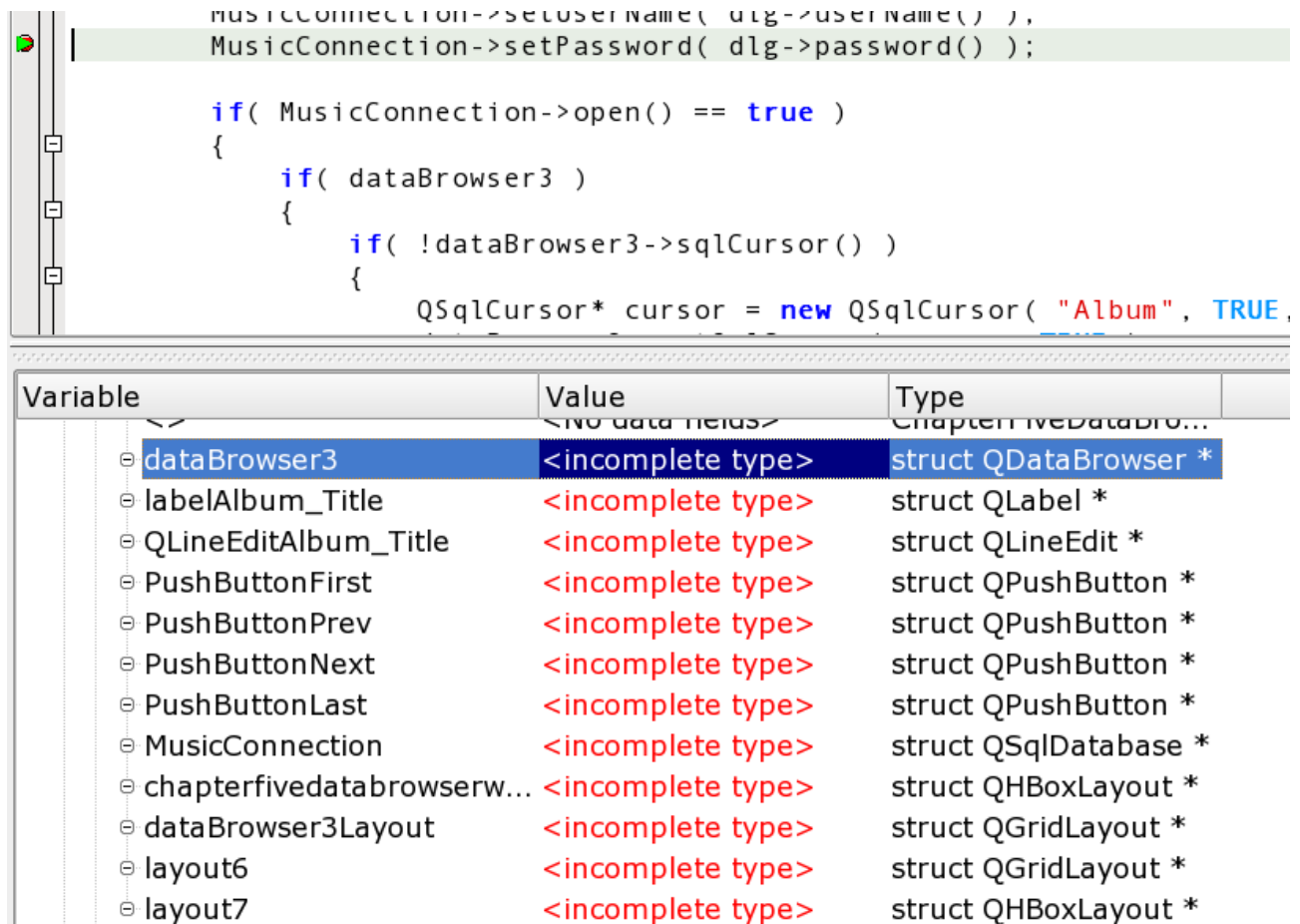
Chapter 5 Database Programming With MySQL code for the QSqlDatabase,

```
MusicConnection = QSqlDatabase::addDatabase( driver() );
if( MusicConnection == 0 )
{
    KMessageBox::error( this, "The database connection is invalid", "Data Browser
Demo" );
    return;
}

MusicConnection->setDatabaseName( name() );
MusicConnection->setHostName( host() );
MusicConnection->setUserName( dlg->userName() );
MusicConnection->setPassword( dlg->password() );

if( MusicConnection->open() == true )
{
```

Now the problem is that this code still doesn't work because of this,



```
MusicConnection->setUserName( dlg->userName() );
MusicConnection->setPassword( dlg->password() );

if( MusicConnection->open() == true )
{
    if( dataBrowser3 )
    {
        if( !dataBrowser3->sqlCursor() )
        {
            QSqlCursor* cursor = new QSqlCursor( "Album", TRUE,
```

| Variable | Value | Type |
|----------------------------|-------------------|-----------------------|
| dataBrowser3 | <incomplete type> | struct QDataBrowser * |
| labelAlbum_Title | <incomplete type> | struct QLabel * |
| QLineEditAlbum_Title | <incomplete type> | struct QLineEdit * |
| PushButtonFirst | <incomplete type> | struct QPushButton * |
| PushButtonPrev | <incomplete type> | struct QPushButton * |
| PushButtonNext | <incomplete type> | struct QPushButton * |
| PushButtonLast | <incomplete type> | struct QPushButton * |
| MusicConnection | <incomplete type> | struct QSqlDatabase * |
| chapterfivedatabrowserw... | <incomplete type> | struct QHBoxLayout * |
| dataBrowser3Layout | <incomplete type> | struct QGridLayout * |
| layout6 | <incomplete type> | struct QGridLayout * |
| layout7 | <incomplete type> | struct QHBoxLayout * |

As far as the debugger is concerned at this moment in time the gui objects have not been completely constructed which means that any tests for the data browsers objects are pretty hit or miss in fact on my computer the line,

```
if( !dataBrowser3->sqlCursor() )
```

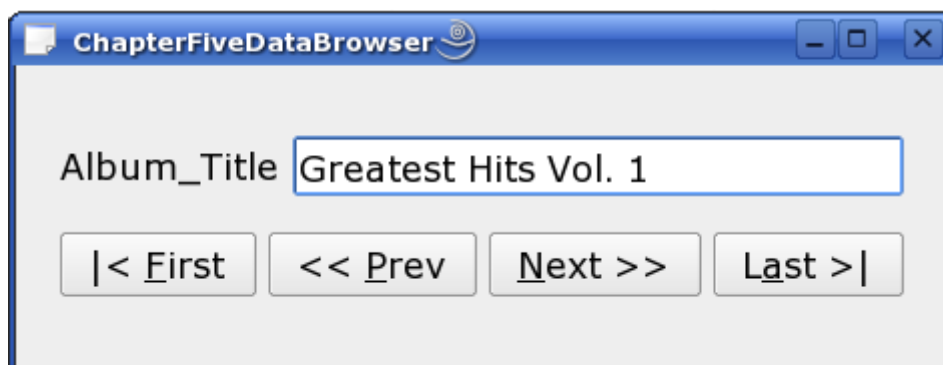
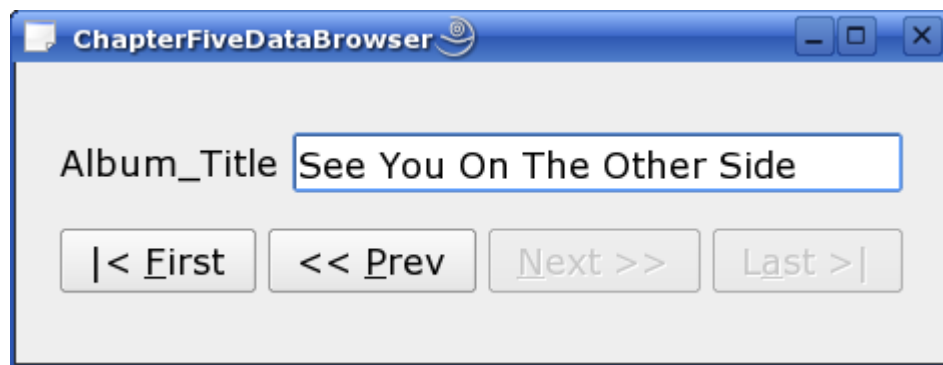
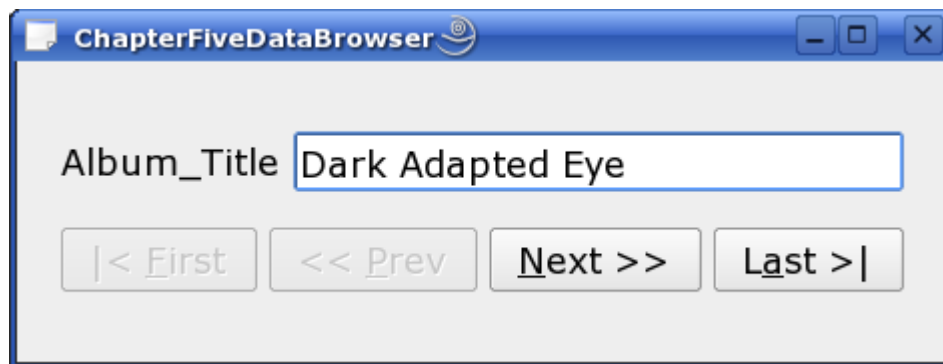
fails everytime because it does not give a false value for the sqlCursor, so what we do is remove this line and continue to generate the new QSqlCursor that points to our data. At the very worst here with the QSqlCursor inheriting from QObject we should only have an extra QSqlCursor hanging

Chapter 5 Database Programming With MySQL
around until the end of the function.

When we remove the test on the QSqlCursor we get,

```
if( MusicConnection->open() == true )
{
    if( dataBrowser3 )
    {
        QSqlCursor* cursor = new QSqlCursor( "Album", TRUE, MusicConnection );
        dataBrowser3->setSqlCursor( cursor, TRUE );
        dataBrowser3->refresh();
        dataBrowser3->first();
    }
}
```

and a program that gives us.



On a final note you should remember that we accepted the default setting for autoediting which is on. This means that any changes you make to the titles while running this program will be made to the database.

Chapter 5 Database Programming With MySQL

Using The DataView

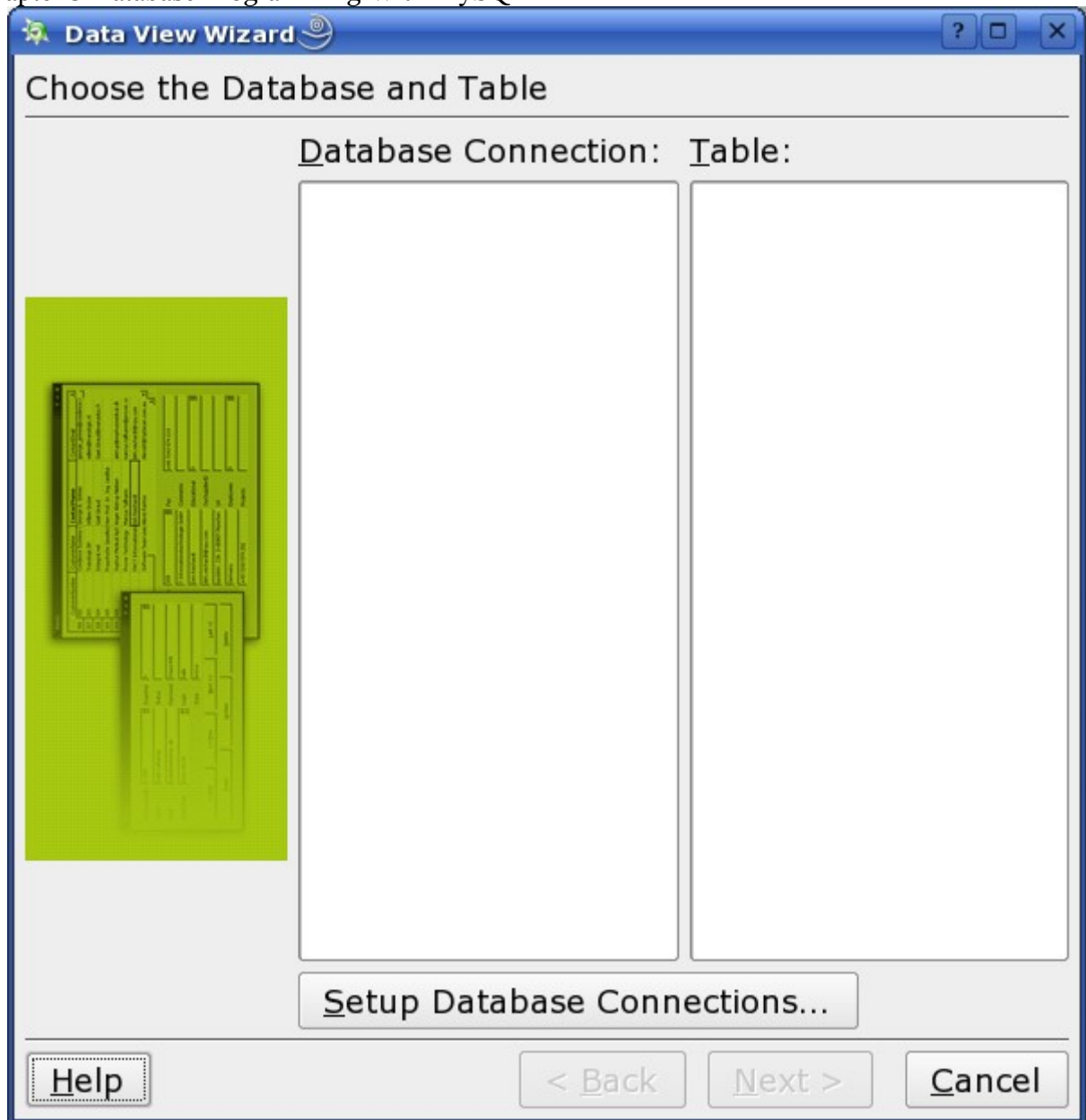
The QDataView is a cut down version of the QDataBrowser without the inbuilt ability to move through the records. It is designed more for paying attention to a particular section of a database rather than browsing through all the records in it. For say looking at personal records where the next record in the database doesn't necessarily have any connection with what you are doing with this record. For this reason the Music database just wasn't adequate to display the idea of how the QDataView works so I put together a quick fictional people database using Knoda using the same methods as described above.

The Knoda table looks like,

| Table - FictionalPeople | | | | | | | | | | |
|-------------------------|-----------|------------|-------------|------------|--------------|-------------|------------------|------|--------|---------|
| | Person_ID | First_Name | Middle_Name | Surname | House_Number | Flat_Number | Road_Name | Town | City | Country |
| 1 | 1 | Yukio | | Mishima | 23 | 2 | Oxford Street | | London | England |
| 2 | 2 | Emily | Jane | Bronte | 154 | 12 | Regent Street | | London | England |
| 3 | 3 | Charles | | Baudelaire | 65 | | Rue Saint Dennis | | Paris | France |
| * | [Auto] | | | | | | | | | |

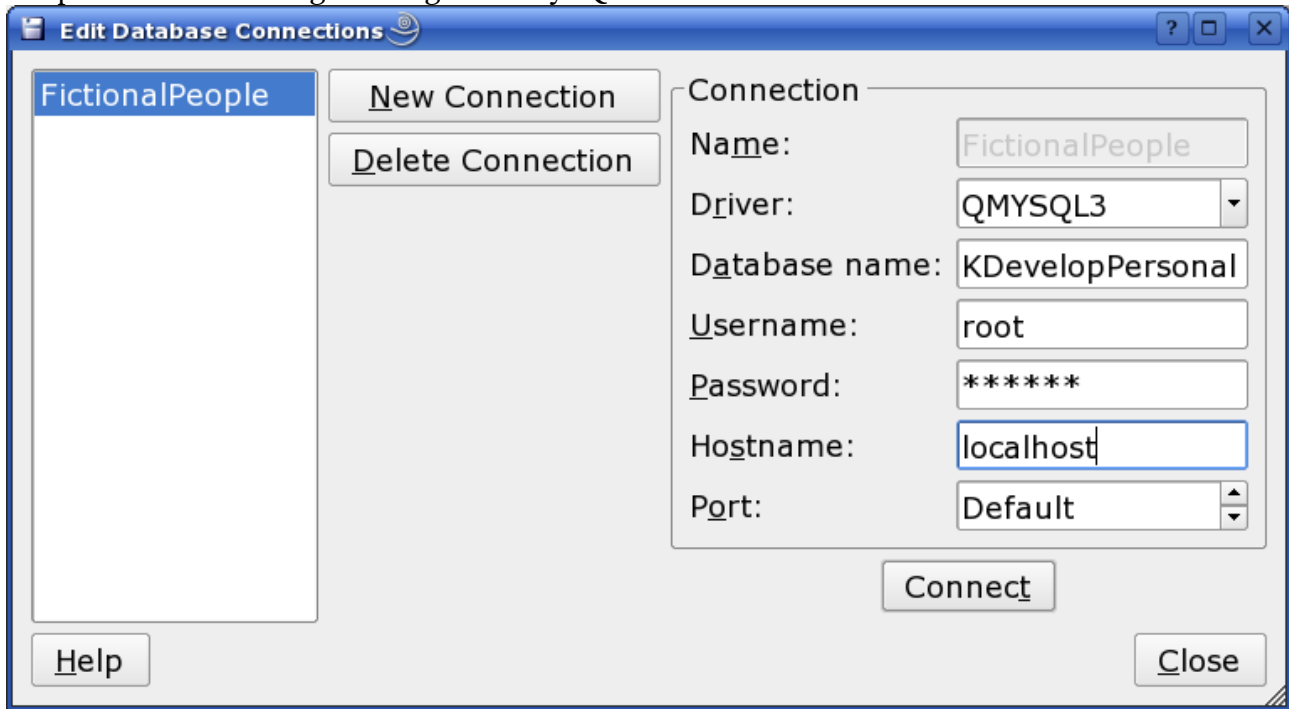
As you can see it is a simple enough database containing nothing more than the names and addresses of fictional people, well, the addresses are fictional anyway. It does however, give us enough fields to make our dataview project look a bit more respectable.

To start the wizard for creating the chapterfivedataview project, create the Simple Designer based KDE Application as before and remove the widgets and button_clicked functionality save and build the project and then add a DataView to the form,

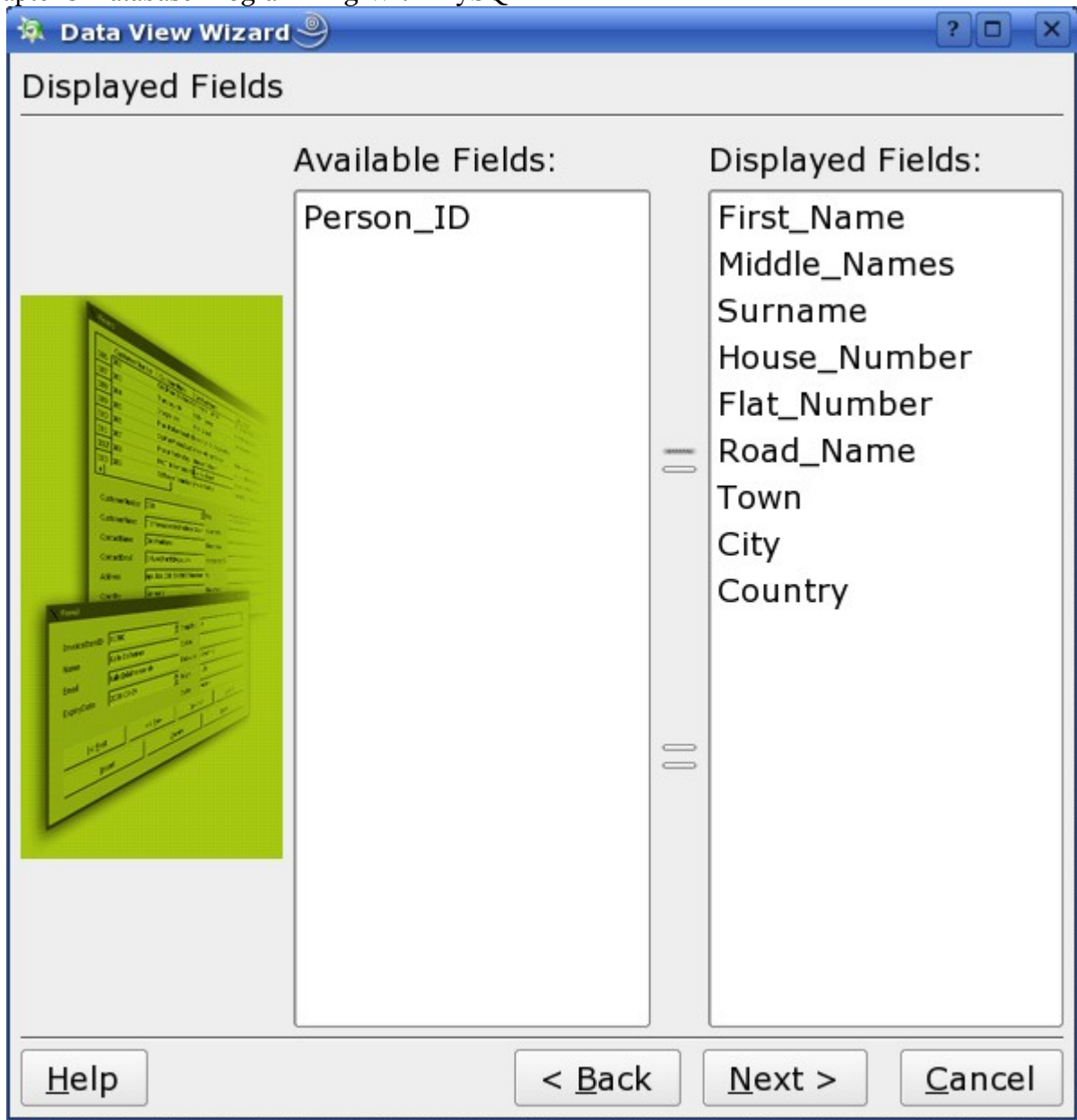


The wizard is almost identical to the DataBrowser wizard only slightly shorter.

Chapter 5 Database Programming With MySQL



With the only differences in the setup from previous projects being minor such as the change of the database name,



though there are more fields to display when we get to the fields dialog.

When we get to the code we notice that things are pretty much the same as they have been in recent projects. The basic structure is to set up a QSqlForm and add all the display widgets to the form. As a user of the project you don't directly use the QSqlForm yourself although if you wish to hand code a database form, looking at the wizard implementation is the way to do it as it sets the names for the widgets to table columns of the database and then maps the record you pass to the form to the appropriate columns. This is also how the previous QDataBrowser project works although in that project the QDataBrowser wraps the QSqlForm and controls things for us.

One thing you will notice if you look at the generated code is that the polish function is not overloaded automatically here. This is because the design of the QDataView isn't really suited to accessing the database immediately on start up and is why the chapterfivedataview project is developed the way it is.

The screenshot shows a Qt Designer window with a form layout. The form contains the following fields:

- First_Name (text input)
- Middle_Names (text input)
- Surname (text input)
- House_Number (text input)
- Flat_Number (text input)
- Road_Name (text input)
- Town (text input)
- City (text input)
- Country (text input)

Below the form, there is a button labeled "Connect To Database" and a QComboBox widget.

We start with the idea of a separate application that contacts the database when it needs to and displays a single entry from the database. Technically we could have overridden the polish function and connected to the database from there but I decided to separate everything out. So now we have a separate database function that is activated by clicking the button and at start up what is an empty QComboBox. The idea for the project being that the user connects to the database and a list of the available people is placed in the QComboBox. The user then selects the name of the person they wish to view the details of in the QComboBox and then the form is filled out.

The database connection starts when the button is clicked,

```

PasswordDialogImpl *dlg = new PasswordDialogImpl();

if( dlg->exec() == QDialog::Accepted )
{
    FictionalPeopleConnection = QSqlDatabase::addDatabase( databaseDriver() );
    FictionalPeopleConnection->setDatabaseName( databaseName() );
    FictionalPeopleConnection->setHostName( host() );
    FictionalPeopleConnection->setUserName( dlg->userName() );
    FictionalPeopleConnection->setPassword( dlg->password() );

    if( FictionalPeopleConnection->open() == true )
    {
        QSqlSelectCursor *cursor = new QSqlSelectCursor( "SELECT
FictionalPeople.First_Name, FictionalPeople.Middle_Names, FictionalPeople.Surname FROM
FictionalPeople", FictionalPeopleConnection );

        cursor->select();
        QString strName;

        while( cursor->next() == true )
        {
            strName = cursor->value( "First_Name" ).toString() + " ";

            if( cursor->value( "Middle_Names" ).toString().isEmpty() == false )
            {
                strName += cursor->value( "Middle_Names" ).toString() + " ";
            }

            strName += cursor->value( "Surname" ).toString();

            namesComboBox->insertItem( strName );
        }
    }
}

```

Chapter 5 Database Programming With MySQL

```
}  
}
```

As you can see there is nothing that should surprise you in this function, the password and database access is exactly the same as it has been in the previous two projects it is just initialised differently. The sql statement,

```
SELECT FictionalPeople.First_Name, FictionalPeople.Middle_Names, FictionalPeople.Surname  
FROM FictionalPeople
```

Gets the first name, the middle names and the last name from all the entries in the database. We then build up a name string by using the value function from the QSqlRecord class, we can do this because if you follow the class hierarchy in the help, you will see that the QSqlSelectCursor class inherits from the QSqlCursor class which in turn is derived from the QSqlRecord and the QSqlQuery classes. When this function has run we get,

chapterfivedataview

First_Name Flat_Number

Middle_Names Road_Name

Surname Town

House_Number City

Country

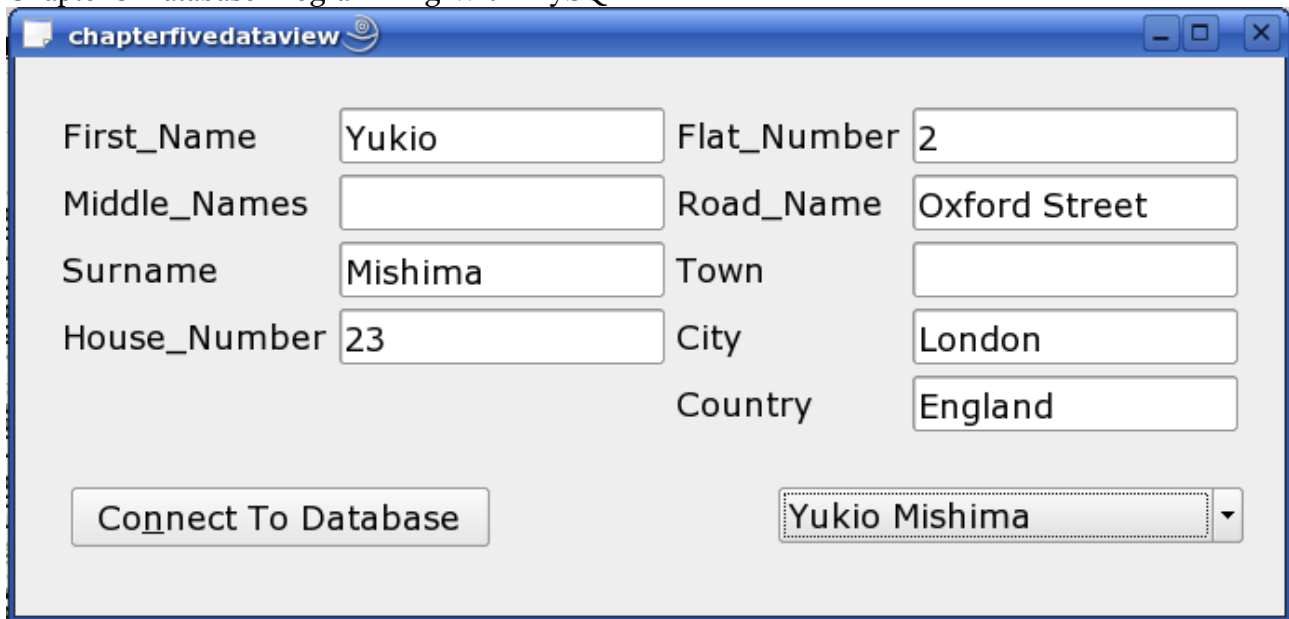
Connect To Database

Yukio Mishima
Yukio Mishima
Emily Jane Bronte
Charles Baudelaire

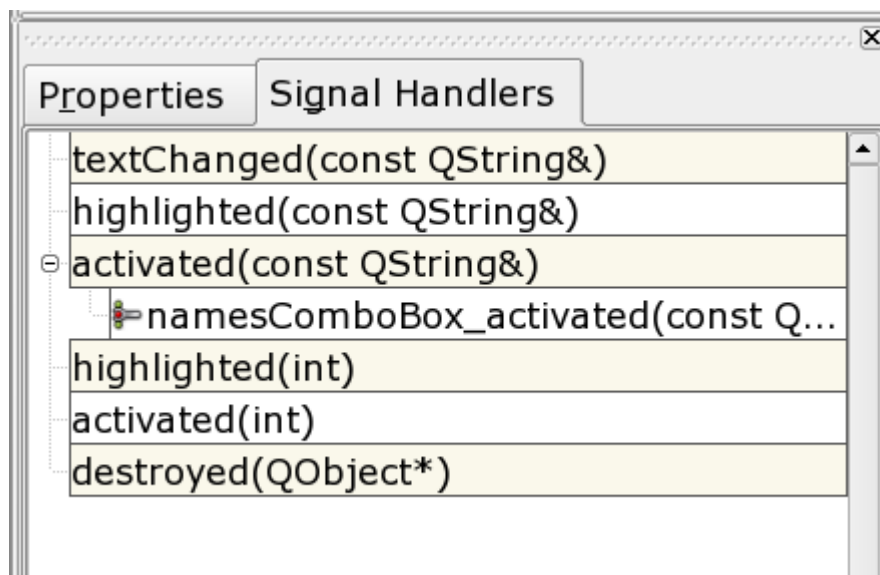
```
// KMessageBox::information  
int nFirstName = name.find()
```

which shows the three entries in the database in the QComboBox. You could call the fact that it doesn't automatically fill in the form as either a bug or a feature depending on your mood but when you think about if there is a long list of names why would you specifically want to see the first one any more than the others. Even when they are sorted.

Chapter 5 Database Programming With MySQL



The way to do this is to receive the activated signal from the QComboBox when a new item is selected in the list and displayed in the viewable area.



Adding the Signal Handler here adds the function to our widget class and then we need to add a variable name for the QString so that we can use it.

```
virtual void namesComboBox_activated( const QString& name );
```

and the function is,

```
void chapterfivedataviewWidget::namesComboBox_activated( const QString& name )
{
    int nSurname = name.findRev( ' ' );
    // KMessageBox::information( this, "The space for the surname is at " +
    QString::number( nSurname ) + "\nin name " + name );
    int nLength = name.length();
    setSurname( name.mid( nSurname+1, nLength ) );
    // KMessageBox::information( this, "The surname is " + surname() + " \nFrom the name "
    + name );
    int nFirstName = name.find( ' ' );
```


Chapter 5 Database Programming With MySQL

```
setFirstName( name.left( nFirstName ) );
// KMessageBox::information( this, "The first name is " + firstName() + "\nWith the
space at " + QString::number( nFirstName ) );
if( nFirstName != nSurname )
{
    setMiddleName( name.mid( nFirstName+1, nSurname - ( nFirstName+1 ) ) );
    // KMessageBox::information( this, "The middle names are " + middleName() + "\n
Starting from " + QString::number( nFirstName+1 ) + "\nEnding at " +
QString::number( nSurname ) + "\nFrom a length of " + QString::number( nLength ) );
}

QString strSqlString = "SELECT * FROM FictionalPeople WHERE FictionalPeople.First_Name
= '" + firstName() + "' ";
if( middleName() != 0 && middleName().isEmpty() == false )
    strSqlString += "AND FictionalPeople.Middle_Names = '" + middleName() + "' ";
strSqlString += "AND FictionalPeople.Surname = '" + surname() + "'";

// KMessageBox::information( this, strSqlString );

QSqlSelectCursor *cursor = new QSqlSelectCursor( strSqlString,
FictionalPeopleConnection );
cursor->select();
cursor->next();

dataView2->refresh( cursor );

// clear the middle name
strMiddleName.truncate( 0 );
}
```

A bit of a blob so we'll break it down into its three sections. The first section is the processing of the string that is passed to us from the QComboBox. This string is the full name of the person we want so we will have to break it up again. Without the KMessageBox calls that show the progress through the string it looks like this,

```
int nSurname = name.findRev( ' ' );
int nLength = name.length();
setSurname( name.mid( nSurname+1, nLength ) );
int nFirstName = name.find( ' ' );
setFirstName( name.left( nFirstName ) );
if( nFirstName != nSurname )
{
    setMiddleName( name.mid( nFirstName+1, nSurname - ( nFirstName+1 ) ) );
}
```

here we use a few of the QString functions to break up the string starting with the findRev function which starts at the back of the string and moves towards the start until it finds what you have specified in the call to the function. We then store the length of the string for future use and have also declared a few string variables in the class to keep the information. These take the, hopefully obvious names of FirstName, MiddleName and Surname. We then store the surname using the value plus one returned by findRev because we don't want the space. We then find the first space from the start of the string using the find function and then store the first name using the left function. Once we have the end and the beginning anything else falls into the middle names category so we get it or them using the mid function.

```
QString strSqlString = "SELECT * FROM FictionalPeople WHERE FictionalPeople.First_Name =
'" + firstName() + "' ";
if( middleName() != 0 && middleName().isEmpty() == false )
    strSqlString += "AND FictionalPeople.Middle_Names = '" + middleName() + "' ";
strSqlString += "AND FictionalPeople.Surname = '" + surname() + "'";
```

The next section of code builds the sql statement using the name values that we have just extracted

Chapter 5 Database Programming With MySQL

from the name QString. It should be noted that the variable values that we add are surrounded by ' marks. This is a small detail but essential as the query wont work without them.

The final section is,

```
QSqlSelectCursor *cursor = new QSqlSelectCursor( strSqlString,
FictionalPeopleConnection );
cursor->select();
cursor->next();

dataView2->refresh( cursor );

// clear the middle name
strMiddleName.truncate( 0 );
```

Here we run the query and make sure that we are at the start of the results and that they or it is ready to be read. Then we pass the cursor straight to the QDataview refresh function which takes a QSqlResult and automatically fills out the form.

One final tip for database development is never presume that the sql statement has worked. If something has failed it is more often than not a fault in the sql statement. When your sure that works check everything else.

Summary

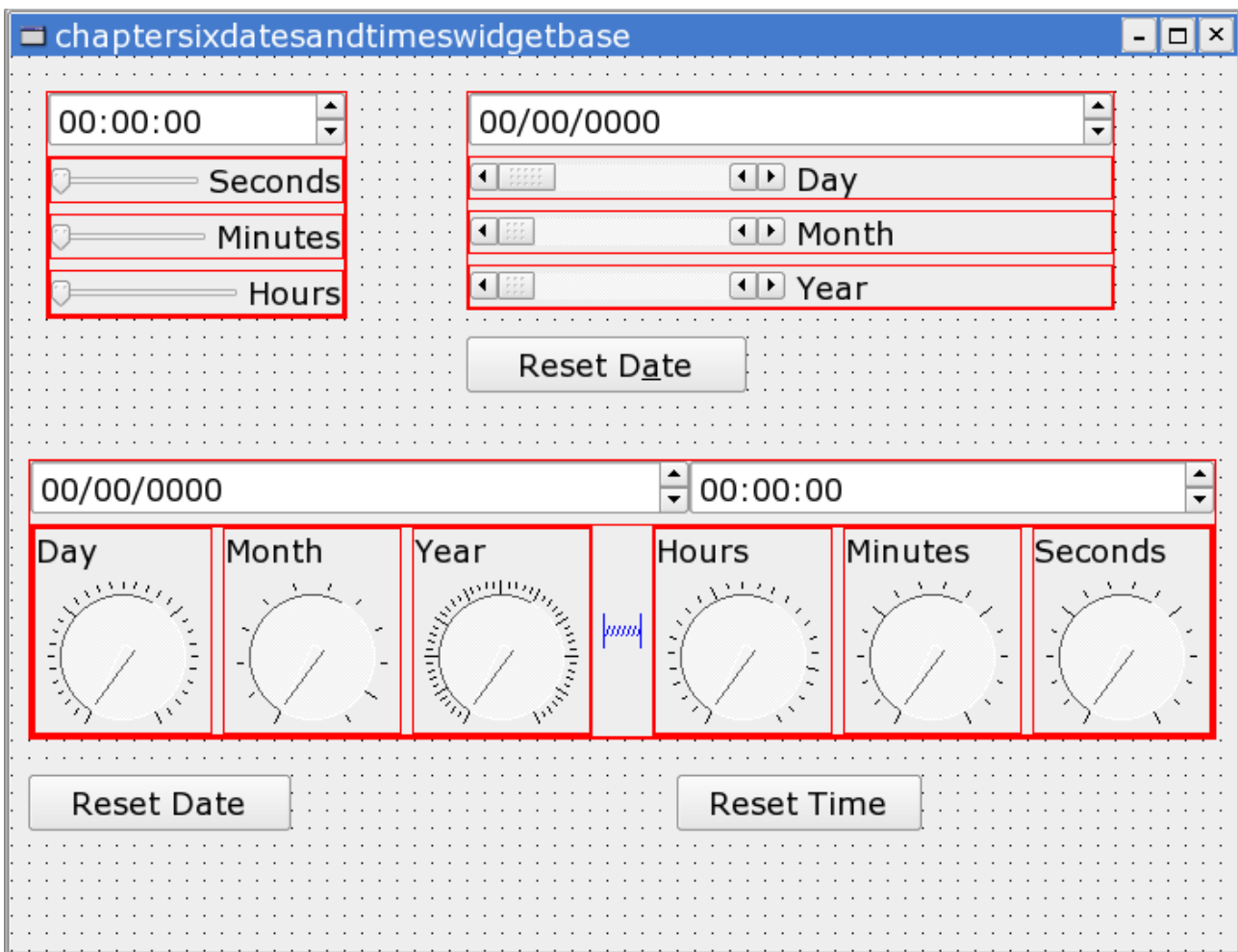
This Chapter has been a brief look at database development with KDevelop focusing on the available widgets. It has tried to cover a lot of ground but there is much left for the reader to experiment with. This chapter could go on for a few hundred more pages about transactions and saving data etc, but hopefully it will be enough to get people started.

Chapter 6 Input And Display

As we have already seen such widgets as the QComboBox and the QLineEdit, etc we shall restrict this chapter to the Widgets in the Input and Display panels that we haven't seen before. This gives us room to write a couple of programs that although perhaps not being dramatically useful will give an idea of how the widgets work.

Inputs

We start with the Inputs and have a design that looks like,



Here we see the development view of the ChapterSixDatesAndTimes example which is created as a Simple Designer based KDE Application like all the examples we have seen so far. The example is split into three obvious groups the first group being the QDateTimeEdit widget which is adjusted with QSlider widgets in the top left. The second is the QDateEdit which is controlled by the QScrollBar's on the top right and the QDateTimeEdit widget which is controlled through the QDial widgets along the bottom.

A lot of time is spent in some manuals going into all the details of times and dates but from a programming perspective unless you are going to be writing these sorts of widgets then the majority of the time you will spend using them will be to either read or write the data. Setting the current

Chapter 6 Input And Display

date and time is just as simple in KDE as it is in other systems

```
setDate( QDate::currentDate() );
```

```
setTime( QTime::currentTime() );
```

should have you covered most of the time, unless you want to get really specific and we'll see how to do that shortly.

The controls on the form be they QSliders, QScrollbar or QDials operate in the same way each one has the valueChanged signal handled in the code, so that the public slots section of the header is almost a list of valueChanged handlers,

```
virtual void resetDateButton_clicked();
virtual void yearScrollBar_valueChanged( int yearScollBarValue );
virtual void monthScrollBar_valueChanged( int monthScrollBarValue );
virtual void dayScrollBar_valueChanged( int dayScrollBarValue );
virtual void hoursSlider_valueChanged( int hoursSliderValue );
virtual void minutesSlider_valueChanged( int minutesSliderValue );
virtual void secondsSlider_valueChanged( int secondsSliderValue );
virtual void resetTime_clicked();
virtual void resetDateButtonDials_clicked();
virtual void secondsDial_valueChanged( int secondsDialValue );
virtual void minutesDial_valueChanged( int minutesDialValue );
virtual void hoursDial_valueChanged( int hoursDialValue );
virtual void yearDial_valueChanged( int yearDialValue );
virtual void monthDial_valueChanged( int monthDialValue );
virtual void dayDial_valueChanged( int dayDialValue );
```

with the exception of the three buttons that are used for automatically setting the dates.

When we implement these functions in the code we follow exactly the same process for both the dates and times

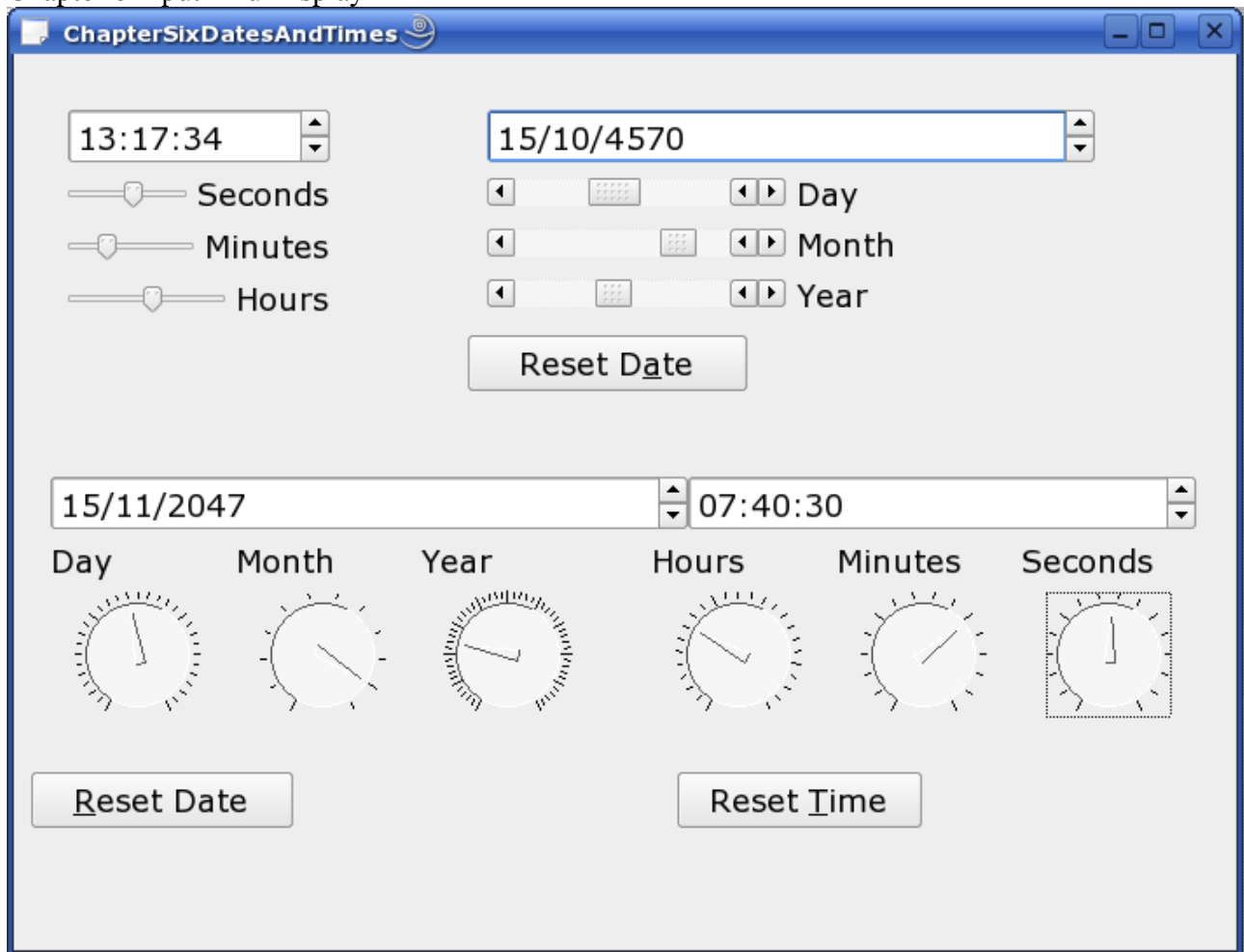
```
QTime time;
time.setHMS( timeEdit->time().hour(), minutesSliderValue, timeEdit->time().second() );
timeEdit->setTime( time );
```

This code is from the minutesSlider_valueChanged function and the same format is repeated throughout the class file. We start by creating a QTime object and then setting the time. as we are only changing the time for one item, in this case minutes we get in this case both the hours and seconds values from the control itself using the QTime second and the QTime hour functions that are accessed through the QTimeEdit time function. Once we have set the time object we pass it to the QTimeEdit widget with the setTime function.

The code from one of the date functions is,

```
QDate date;
date.setYMD( dateEdit->date().year(), monthScrollBarValue, dateEdit->date().day() );
dateEdit->setDate( date );
```

As you can see they are conceptually the same except we are dealing with dates instead of times. Which just leaves the buttons which set the dates and times in there widgets using the currentDate and currentTime functions in the setDate and setTime function calls.



Tip Of The Day

When you're overriding the signal handlers for your project it is possible to mess things up in a way that could be a pain if you don't know how to fix it. At one point I added a signal handler for a button click and realised that I hadn't named the button properly and it was using the default name. Because I prefer to have my variables explicitly named so there is no confusion, for me at least, about what they are and do. So I cancelled the dialog that asks you to select the class that you want to handle the signal. I was then faced with a dialog that told me that KDevelop could not find the class that implemented the .ui file.

What we need to do in this case is manually set up the implementation for the file this is done by right clicking on the .ui file in the Automake Manager,

```
const QString&)
```

```
bool)
```

```
e(bool)
```

```
,in
```

```
ge
```

```
pt
```

```
CE
```

```
-Wcast-align
```

```
-Wconvers
```

chaptersixdisplay (Program in bin)

chaptersixdisplay.cpp

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidget.cpp

chaptersixdisplaywidget.h

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

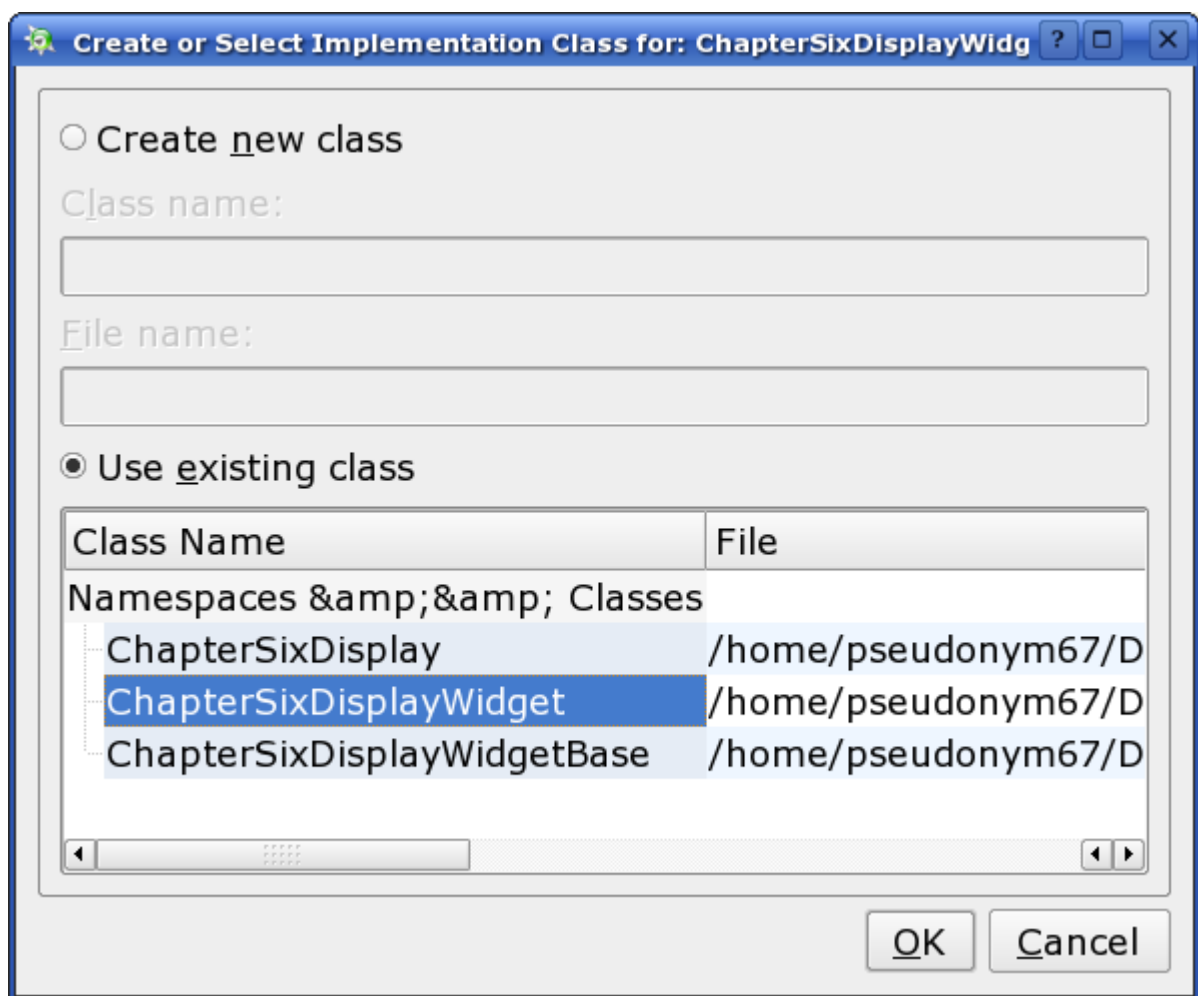
chaptersixdisplaywidgetbase.h

chaptersixdisplaywidgetbase.ui

chaptersixdisplaywidgetbase.cpp

chaptersixdisplaywidgetbase.h

and selecting the Create or Select Implementation option which will give us the implementation dialog we would normally get when starting to handle signals within a project.

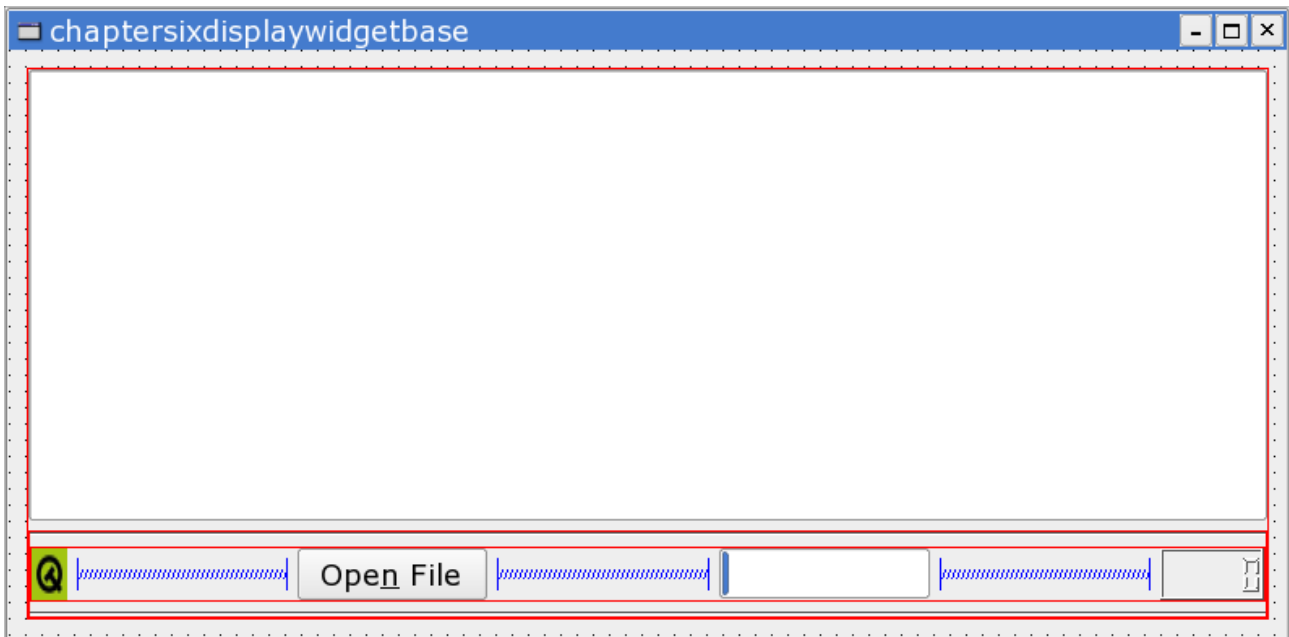


We select the class we wish to use to implement the signals and everything is back to normal.

Chapter 6 Input And Display

Displays

For the displays example we start with an application that looks like,



This is a simple enough layout in that the main area is covered with a QTextBrowser and a line of widgets below it. From left to right the little green Qt square is the QPixmapLabel which is next to a standard button and then there is a small progress bar (QProgressBar) and a LCD line counter, QLCDNumber.

The idea behind the application is that you browse for a file open it and then while it is opening the progress bar moves along, the LCD Number display counts the lines and the QPixmapLabel is replaced with the current icon for the local file. The driving functionality for the program is the open file button which contains the code,

```
KURL kurlFile = KFileDialog::getOpenURL();  
  
if( kurlFile.isEmpty() == false )  
    openFile( kurlFile );
```

This time we are using a static function of the KFileDialog class which just returns the KURL for the file. All we do with this function is show the KFileDialog to get a value and then call the function that does the work if a value is returned.

The main function is the openFile function which looks like this,

```
QFile file( fileName.pathOrURL() );  
  
// first of all get the number of lines in the file  
//  
int nNumberOfLines = 0;  
QString strLine;  
  
// Set up the bits and pieces  
//  
progressBar->setProgress( 0 );
```

Chapter 6 Input And Display

```
lcdNumber->display( 0 );

// get the pixmap for the opened file
//

KFileItem *fileInfo = new KFileItem( KFileItem::Unknown, KFileItem::Unknown, fileName,
true );
pixmapLabel1->setPixmap( fileInfo->pixmap( 0 ) );
delete fileInfo;

// now load the file

numberOfLines = 0;
textBrowser->clear();

if( file.open( IO_ReadOnly ) == true )
{
    QTextStream stream( &file );
    while( stream.atEnd() == false )
    {
        textBrowser->append( stream.readLine() );
        numberOfLines++;
        progressBar->setProgress( numberOfLines );
        lcdNumber->display( numberOfLines );
    }

    file.close();
}
else
{
    KMessageBox::error( this, "Unable to open the file " + fileName.pathOrURL() );
    return;
}
```

At the start of the function the file name, including path is associated with a QFile object, nothing is done yet this just initialises a QFile object so we initialise a few variables, set the progress bar and the lcd display to zero and then set the QPixmapLabel for the file.

The Pixmap for a file is discovered using the KFileItem class which is used quite extensively through the KDE code and can be set up using the

```
KFileItem::KFileItem( mode_t _mode, mode_t _permissions, const KURL &_url,
bool _determineMimeTypeOnDemand = false )
```

constructor. I point this out because it was a pig to work this one out because it is not immediately obvious that you can ignore the mode_t parameters by passing in KFileItem::unknown and then passing in true for the _determineMimeTypeOnDemand which has the effect of saying to the constructor here's the KURL for the file work the rest out for yourself which it does but I'd still have preferred a KFileItem(KURL url) constructor.

Once we have this information sorted out we can try to open the file.

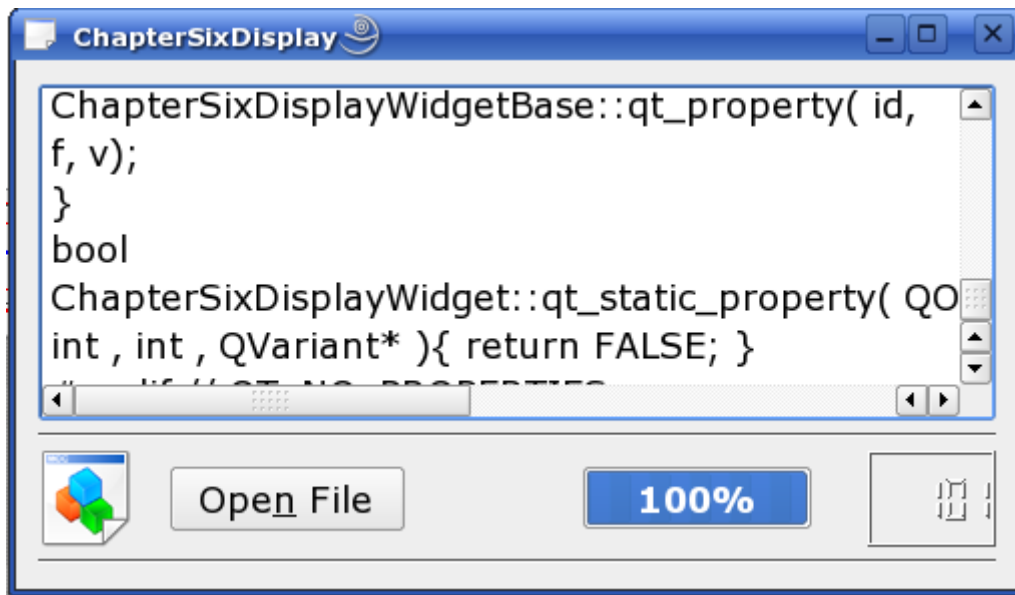
If the file does not open then we display a message and then exit the function. If the file opens successfully we create a QTextStream object to read the text from the file. Note that although we are using a QTextStream I have deliberately not added any filtering to restrict the type of file that is opened.

The main piece of code is in the while loop,

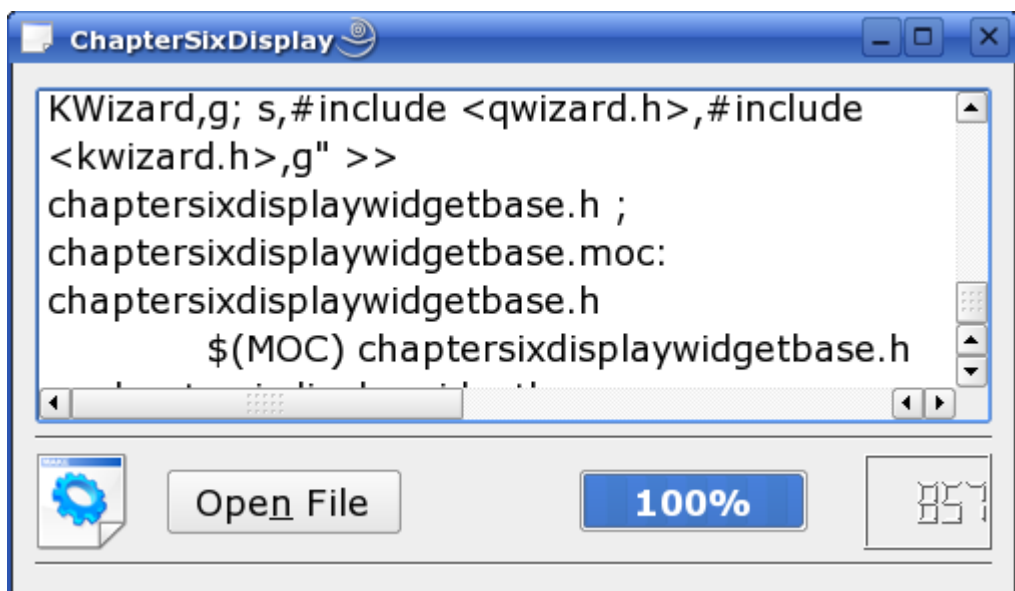
```
while( stream.atEnd() == false )
{
    textBrowser->append( stream.readLine() );
    numberOfLines++;
    progressBar->setProgress( numberOfLines );
    lcdNumber->display( numberOfLines );
}
```


Chapter 6 Input And Display

which reads in the file line by line and then adds each line to the QTextBrowser and increments both the QProgressBar and the QLCDNumber widgets.



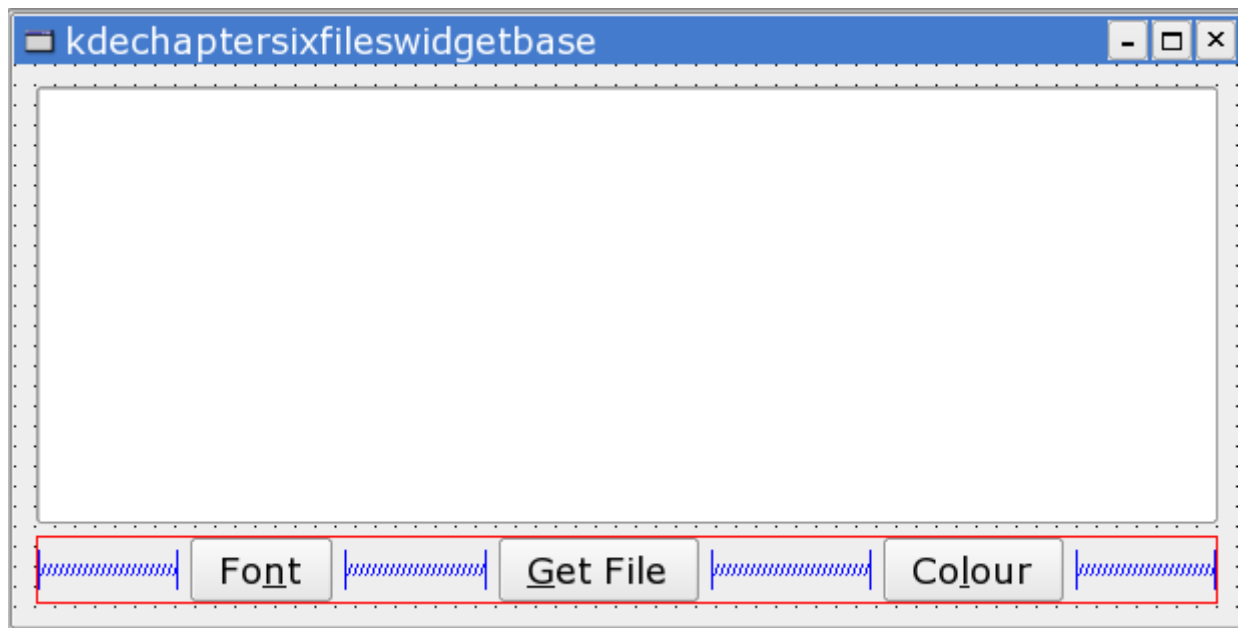
The image above is of the application running and displaying one of it's own .moc files. And the the image below is of the application displaying it's own makefile. The application will expand to full screen but the pictures are clearer if I don't have to shrink them.



Files And Some Common Dialogs

The final demonstration project for chapter six is the KDEChapterSixFiles project which displays some of the information that we get from using the KFileItem class that we used in the previous example and we'll also see how to use some of the common dialogs, such as the KFileDialog which we have already seen and introduces the KFontDialog and the KColorDialog classes. The design for

Chapter 6 Input And Display
the program looks like,



It is a simple layout with the main portion of the program being taken up by a QTextBrowser with some buttons at the bottom to get the file and set the font and the colour for the text. It should be noted from the start that although the colour can be set first if you have already opened the file it will not redraw the file and therefore change the colour until you open another file.

This is a picture of the standard program,

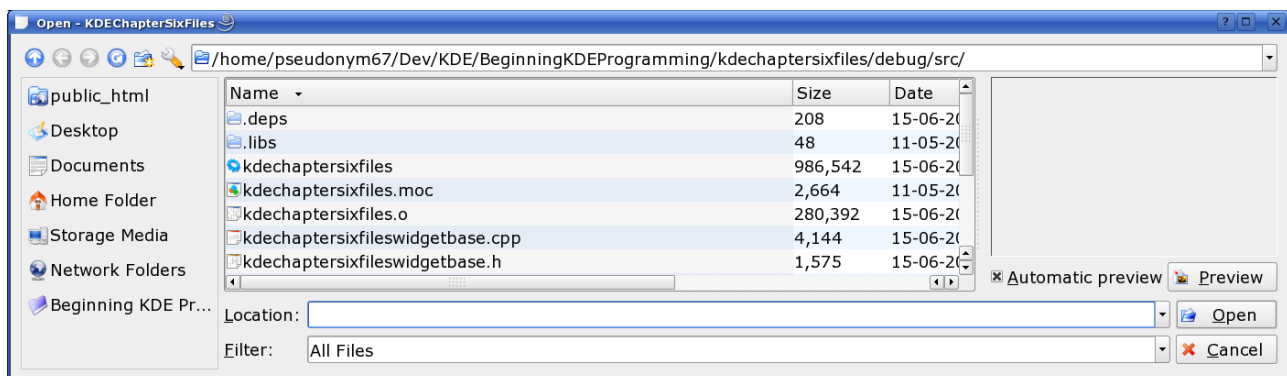


As you can see the program is giving a readout on it's own executable, it doesn't try to list all the information available to the KFileItem only the bits that would be meaningful to someone using the file. As you can see I've included the Status Bar Info and the Tool Tip Text which give some insight to how things are done from within KDE itself.

As with the previous code we get the file information from the KFileDialog using,

```
KURL fileName = KFileDialog::getOpenURL();
if( fileName.isEmpty() == true )
{
    KMessageBox::error( this, strNoFile, strChapterSix );
    return;
}
```

Which gives us the dialog,



and then get the KFileItem using,

Chapter 6 Input And Display

```
KFileItem *fileInfo = new KFileItem( KFileItem::Unknown, KFileItem::Unknown, fileName,
true );

if( fileInfo == 0 )
{
    KMessageBox::error( this, strAllocationWarning, strChapterSix );
    return;
}
```

The code then reads through the KFileItem and writes the information to the QTextBrowser, a small sample of the technique used is shown,

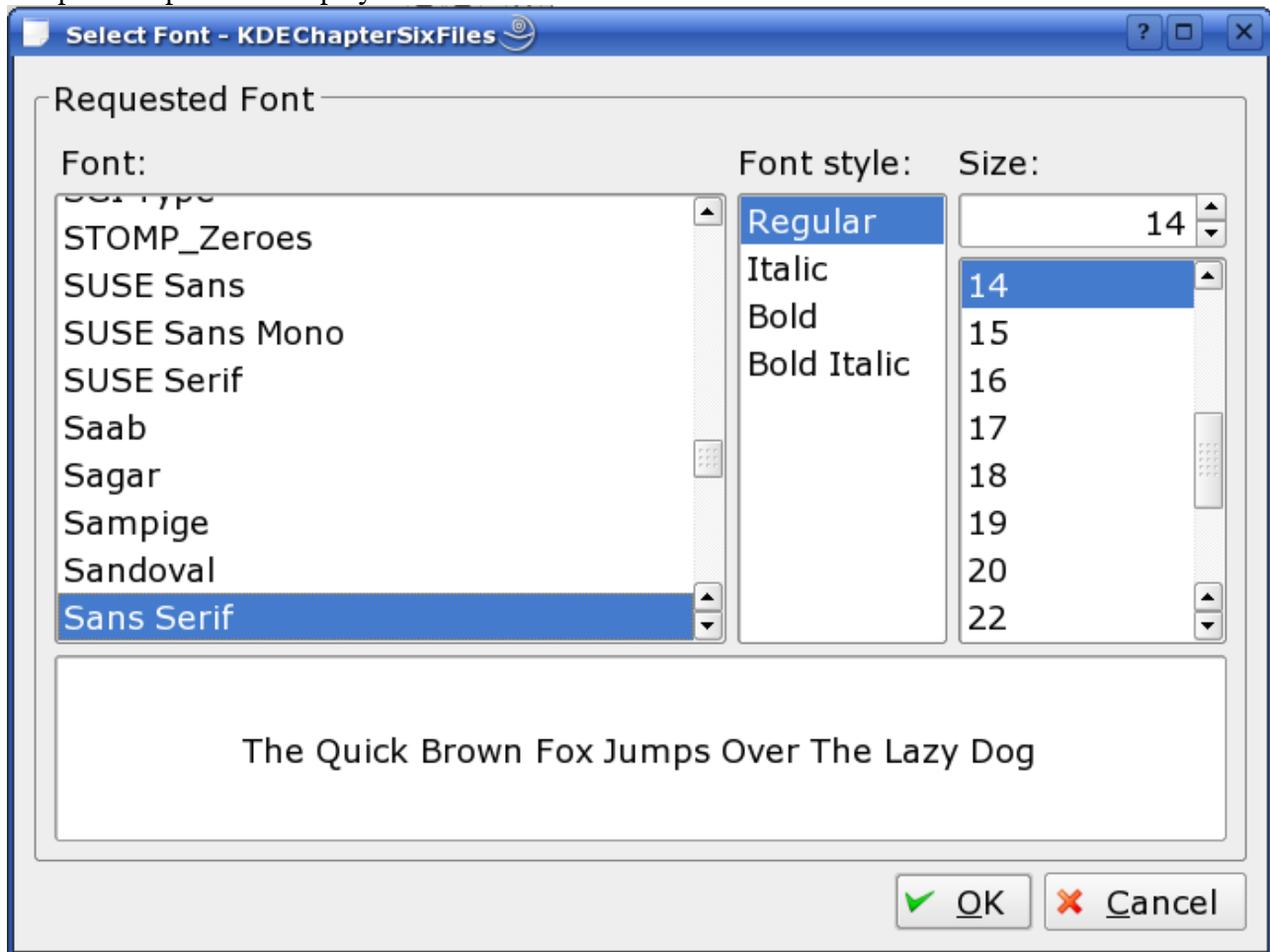
```
textBrowser2->clear();
textBrowser2->setColor( QColor( itemNameColour() ) );
textBrowser2->append( strFileName );
textBrowser2->moveCursor( QTextEdit::MoveLineEnd, false );
textBrowser2->setColor( QColor( itemTypeColour() ) );
textBrowser2->insert( fileInfo->name() );
textBrowser2->setColor( QColor( itemNameColour() ) );
textBrowser2->append( strDirectory );
textBrowser2->moveCursor( QTextEdit::MoveEnd, false );
textBrowser2->setColor( QColor( itemTypeColour() ) );
textBrowser2->insert( fileInfo->localPath() );
textBrowser2->setColor( QColor( itemNameColour() ) );
textBrowser2->append( strPermissions );
```

This code is from the very start, so the first thing that we do is clear the previous file, if any from the QTextBrowser and then set the colour of the text to the itemNameColour This is always black as there is no way from the running program to change this. We then append the string of the filename which is declared as,

```
QString strFileName = i18n( "File Name = " );
```

and then move the cursor to the end of the current line before changing the text colour once again and then using file name obtained from the KFileItem. The next couple of hundred lines are basically a case of wash, rinse and repeat the above.

To change the font we click the Font button and get the dialog,



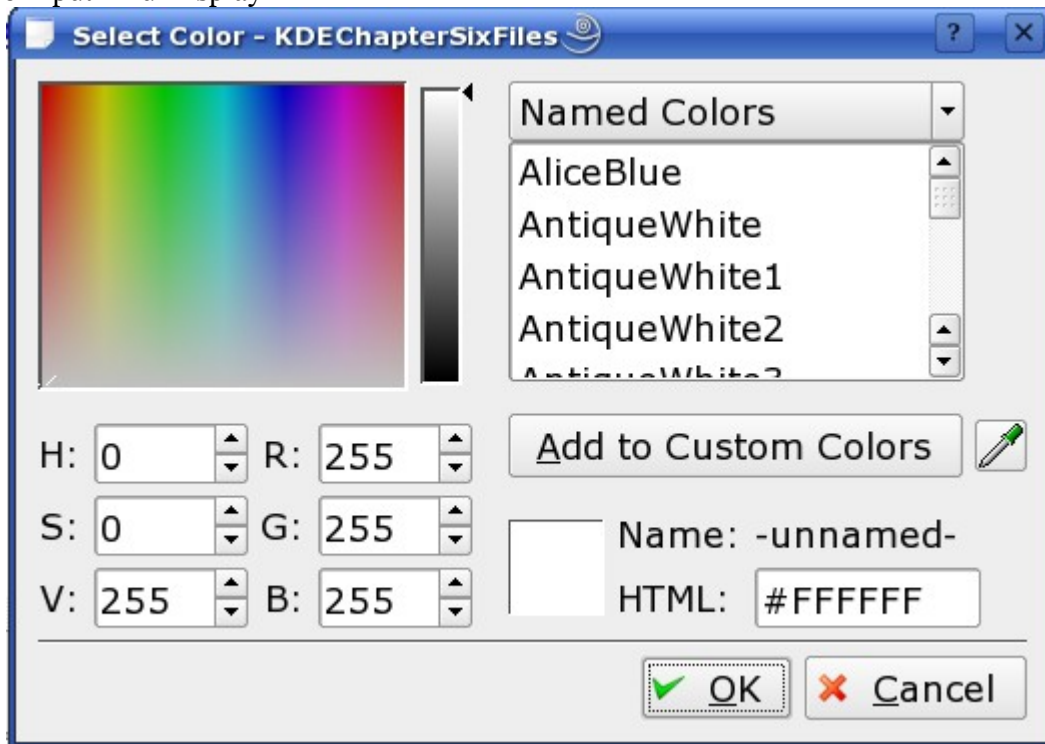
which is the standard type of font dialog you would expect on any system showing the available fonts, their styles and available sizes.

As with the `KFileDialog` getting the fonts is simply a case of using the `getFont` function, in this case `getFont`,

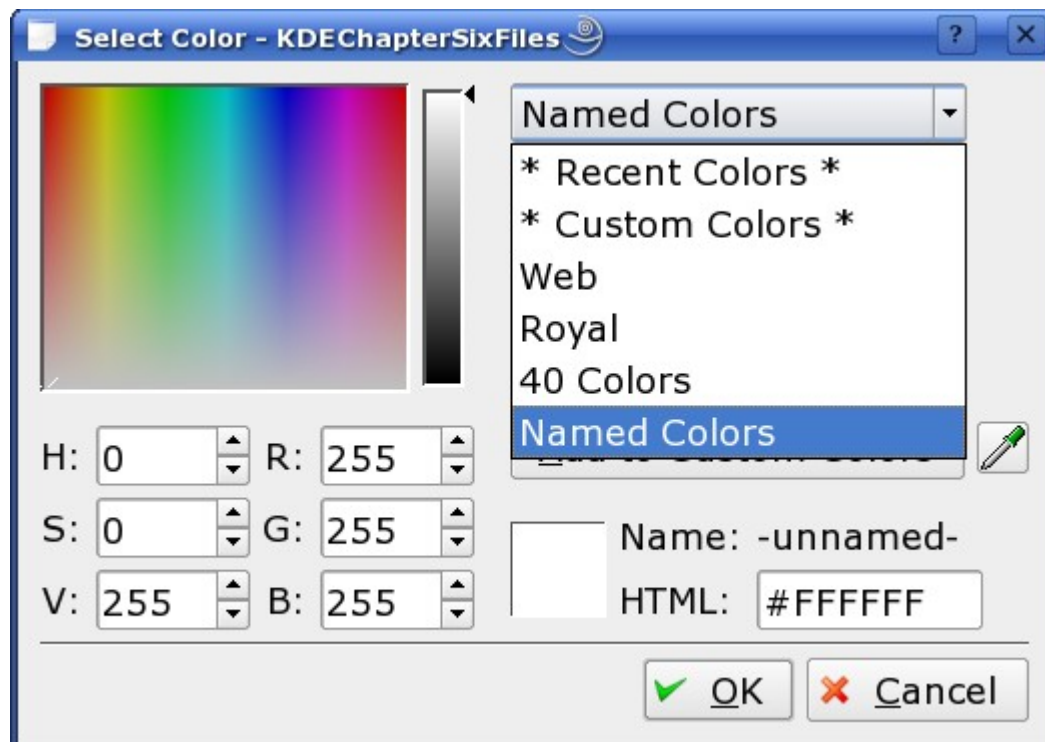
```
QFont tempFont;
if( KFontDialog::getFont( tempFont ) == KFontDialog::Accepted )
{
    setAppFont( tempFont );
}
```

When `OK` is clicked on the dialog the font details are saved in the temporary font, creatively named `tempFont`. This is then passed to the `setAppFont` function which just stores the font details so that we can use it when we write out the information for the next file that is opened.

If we click on the `Colour` button we get,



Which isn't quite the standard KColorDialog because we have,



A number of ways of representing the colours in the dialog and KDE automatically saves the setup for the dialog so if you set it for Web colours one time the next time you open the dialog it will default to the Web colours you set previously. You can of course ignore these and just enter the colour values you want or simply select from the colour chooser on the left.

Chapter 6 Input And Display

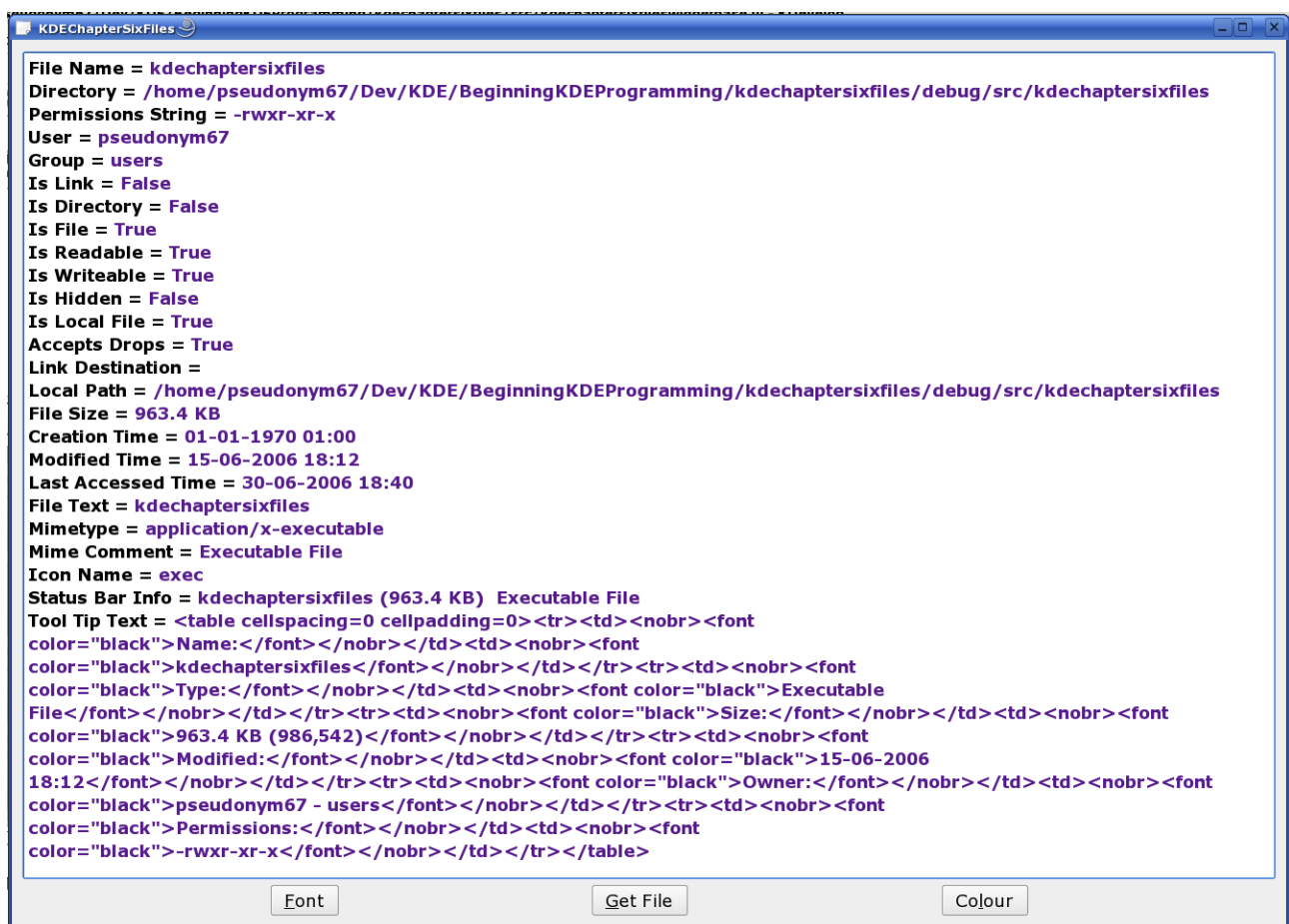
I personally find the Named Colors are fine for what I need so I mostly stick with them. The available colours are listed in `rgb.txt` which on Suse 10 is found in `usr/X11R6/lib/X11` or a good visual representation can be found at <http://sedition.com/perl/rgb.html>

The code for getting the colour is,

```
QColor tempColour;  
if( KColorDialog::getColor( tempColour ) == KColorDialog::Accepted )  
{  
    setItemTypeColour( tempColour.name() );  
}
```

And follows exactly the same format that was used for setting the font.

This is the program running after the font has been set to bold and the colour has been changed to purple.



Summary

In this chapter we have looked at the inputs and displays which apart from showing how to use the controls themselves have presented us with little difficulty and little that we haven't already seen but you may have noticed that most of the classes up until now have begun with a Q and are part of the Qt library. In the next section we will start to look at the KDE classes that are built both on top of the Qt library and designed to work with it.

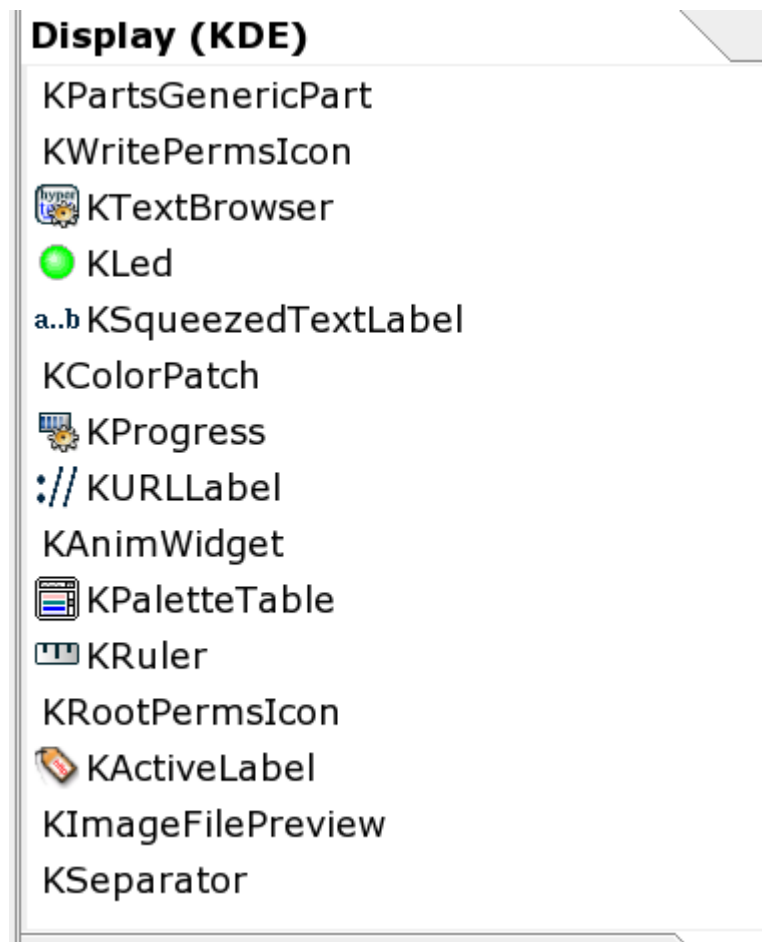
Part Two

KDE Widgets

Chapter 7 KDE Display Widgets

From this point onwards all the widgets we use will be part of the KDE system which means unlike a lot of the code that we have seen to date which relies largely on Qt the following code will require KDE to be installed if it is to run on any Linux system.

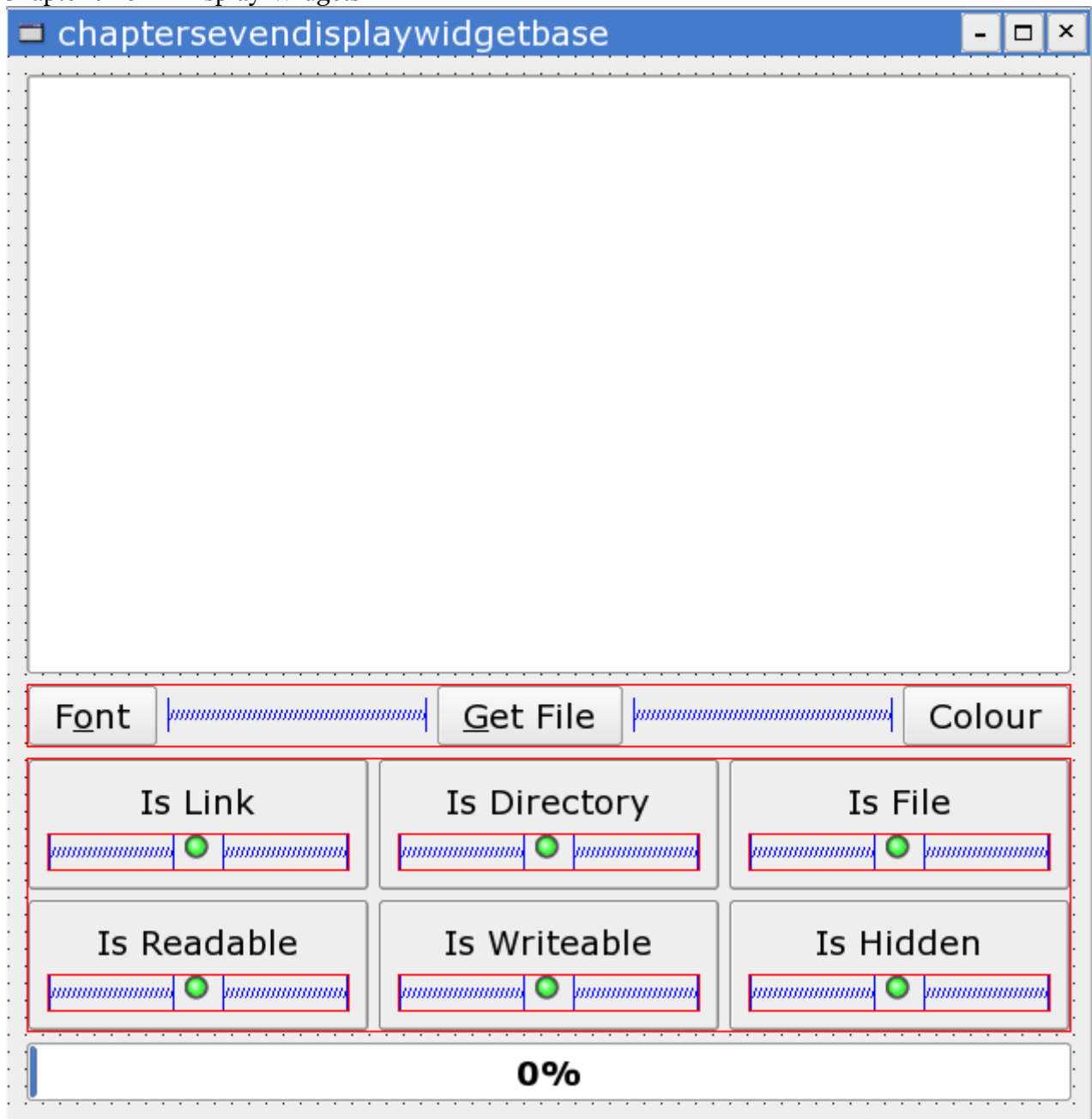
The Display (KDE) section looks like,



The first section that you will notice from the picture is the KPartsGenericPart widget. This widget takes us places that are beyond the scope of this book and into far more advanced topics than we are ready for. If you wish to know about this widget look up Chapter 12 of the KDE2 Development Book in the help.

KDE Display

The first example that we look at in the KDE Display section is a slight modification of what we have seen before and uses the KTextBrowser, the KProgress bar and the KLed displays as well as some of the common dialogs that we saw earlier,



As you can see with the exception of the KLed lights to represent the file properties and the KProgress bar across the bottom the application is almost identical to the one we saw earlier. The main difference between the KTextBrowser and the QTextBrowser that it extends is the handling of web links like,



so although the display above will appear identical in both the QTextBrowser and the KTextBrowser if you click on one of the links in a running KTextBrowser then the default web Browsing application on your system will open the page.

The Buttons for the Font, File and Colour start the standard dialog boxes that we have seen earlier in fact the code is pretty much a direct cut and paste of the previous code.

The KLed widget represents a Led light switch that we are using here to affirm if the file status corresponds to the above text. The KLed will be bright green if true and dark green if it is false. The KLed's are set with the following code,

```
QFileInfo fileInfo( kurl.pathOrURL() );

if( fileInfo.isSymLink() == true )
{
    linkLED->setState( KLed::On );
}
else
    linkLED->setState( KLed::Off );
```

We get a QFileInfo from the KURL and then test it's state and set the KLed appropriately.

The KProgress widget is derived from the QProgressBar that we saw earlier and the only differences are the way in which it sends signals when changes are made which will not usually make any difference to standard operation unless you want to change things based on the percentage of completion of the progress bar.

The only difference in our code is that the progress bar does work better than it did in the previous example but this is because I do this,

```
// get the number of lines in the file

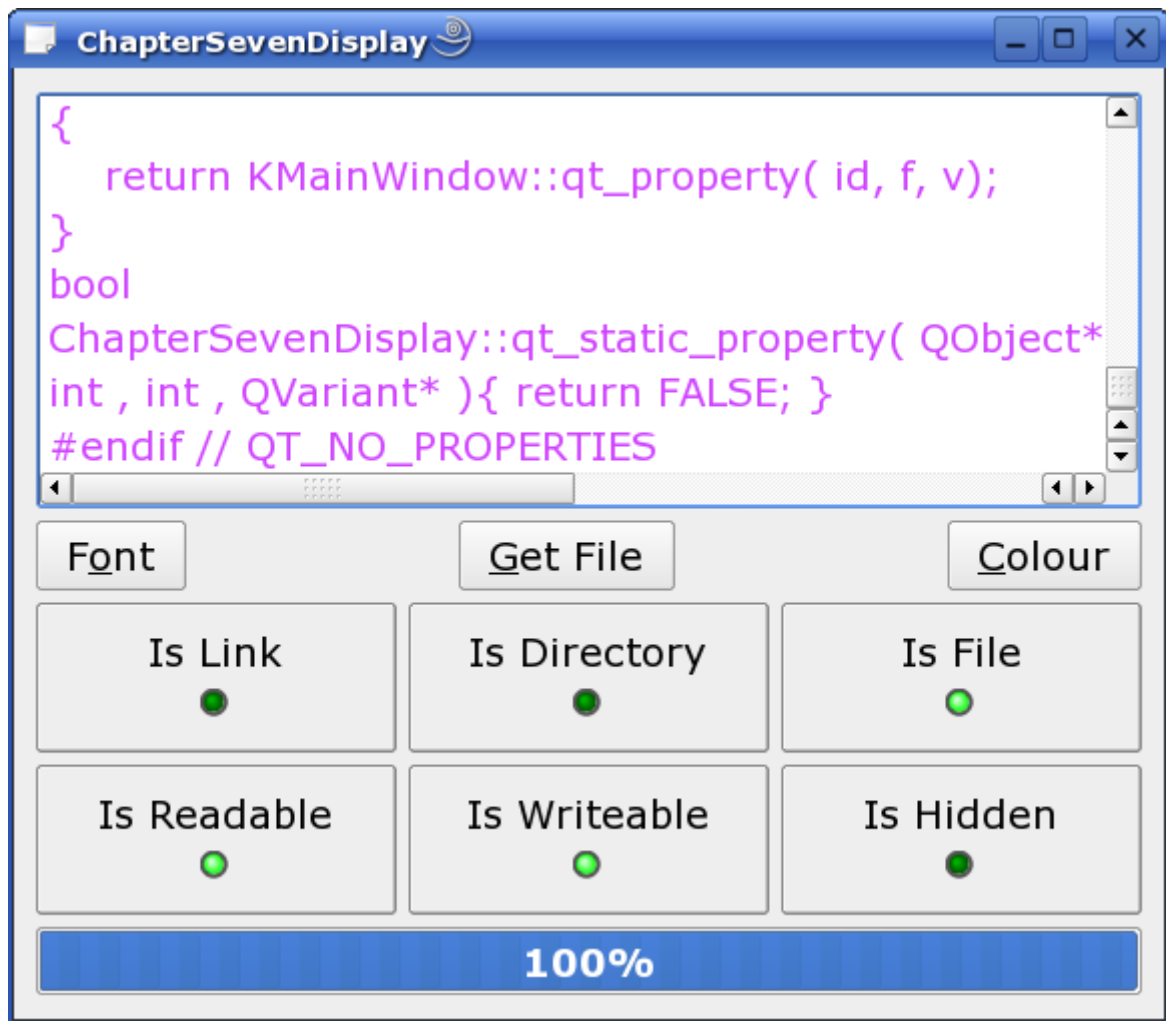
numberOfLines = 0;
if( file.open( IO_ReadOnly ) == true )
{
    QTextStream stream( &file );
    while( stream.atEnd() == false )
    {
        stream.readLine();
        numberOfLines++;
    }

    file.close();
}

progressBar->setTotalSteps( numberOfLines );
```

Chapter 7 KDE Display Widgets

I run through the file and count the number of lines and then call `setTotalSteps` to the number of lines in the file. This gives the progress bar a definite number to aim for, making it more accurate.

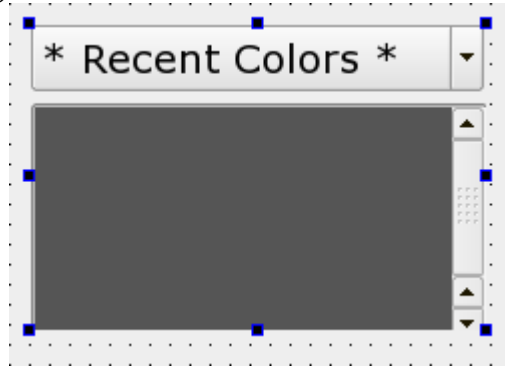


The finished project looks like the above picture showing the application displaying it's own .moc file.

Tip Of The Day

Occasionally you may find that one of the widgets is not set up right automatically when you drop it on to the form. There are a number of these in the following examples so here is how you identify and fix the problem. When you drop a widget onto the form such as a `KPaletteTable`,

Chapter 7 KDE Display Widgets



It looks fine but when you save the .ui file and compile you get,

```
chapterseventestwidgetbase.cpp: In constructor
'chapterseventestWidgetBase::chapterseventestWidgetBase(QWidget*, const char*, uint)':
chapterseventestwidgetbase.cpp:51: error: invalid conversion from 'const char*' to 'int'
chapterseventestwidgetbase.cpp:51: error: initializing argument 2 of
'KPaletteTable::KPaletteTable(QWidget*, int, int)'
```

If we open the generated “projectname”widgetbase. h file the KPaletteTable is declared as

```
KPaletteTable* unnamed;
```

and in the generated “projectname”widgetbase.cpp file constructor the KPaletteTable is initialised as,

```
unnamed = new KPaletteTable( this, "unnamed" );
unnamed->setGeometry( QRect( 130, 100, 228, 153 ) );
```

As you can see from the above compiler print out the constructor is complaining about the second argument which is the “unnamed” text. If you want to you can delete this and check to see if the program is running properly but remember this is not a permanent solution as soon as you change the form this file will be regenerated and your changes will be lost. The permanent solution is to copy the code that sets up the offending widget into the “projectname”widget files so in order to get the KPatletteTable working we would add

```
class KPaletteTable;
```

before the class declaration and

```
public:
    KPaletteTable *paletteTable;
```

within the class file add the header for the class.

```
#include "kcolordialog.h"
```

and,

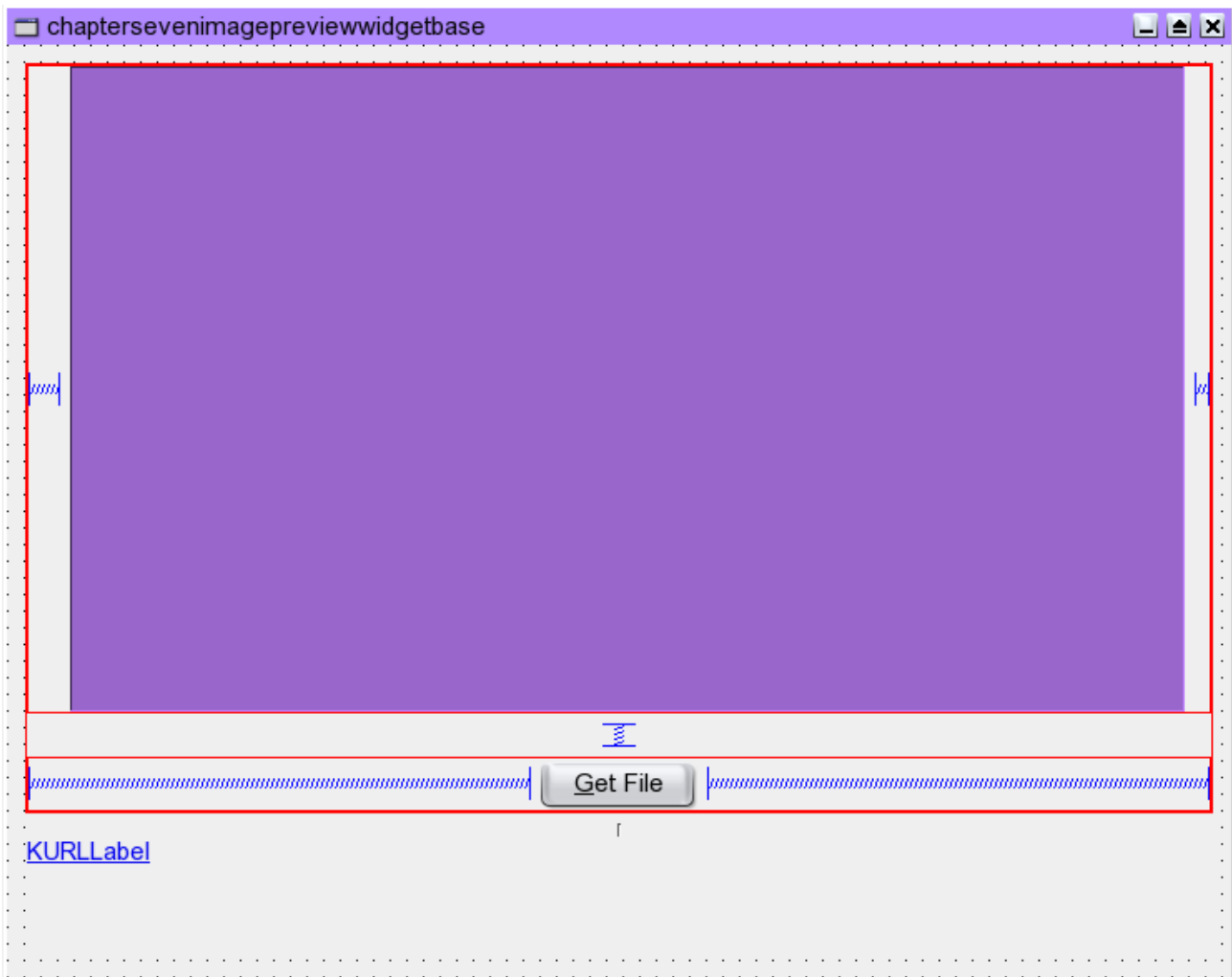
Chapter 7 KDE Display Widgets

```
paletteTable = new KPaletteTable( this );  
paletteTable->setGeometry( QRect( 130, 100, 228, 153 ) );
```

within the constructor. Of course this is going to make the layout of the widget form a complete pain so as you know where you want the offending widget to go, set a frame around that area and then you can adjust the layout of the widget as normal.

KDE Image Preview

For the second example in this chapter we mainly concentrate on the KImageFilePreview widget and then throw in a KURLLabel and a KActiveLabel more for good measure than for any specific purpose.



As you can see from the picture above there is no KImageFilePreview on the page this is because the generated code if we try to drop a KImageFilePreview widget on the form calls non-existent constructors so we use a standard QFrame as a place holder for the KImageFilePreview and implement it ourselves, as discussed in the Tip Of The Day section above. The two widgets below the Get File button are the KURLLabel and the KActiveLabel, these are used here as standard labels to show the path and filename of the file that we are previewing.

To set up the KImageFilePreview widget we add an object of the type into the header file for our implementation class which in this case is the file chaptersevenimagepreviewwidget.h and then in the constructor for the class we add

Chapter 7 KDE Display Widgets

```
ChapterSevenImagePreviewWidget::ChapterSevenImagePreviewWidget(QWidget* parent, const
char* name, WFlags fl)
    : ChapterSevenImagePreviewWidgetBase(parent, name, fl)
{
    pKImageFilePreview = new KImageFilePreview( this );
    pKImageFilePreview->setGeometry( QRect( 37, 10, 670, 388 ) );
    pKImageFilePreview->setMinimumSize( 630, 388 );
    imagePreviewLayout->addWidget( pKImageFilePreview );
}
```

Once the widget is created using it is fairly simple.

```
void ChapterSevenImagePreviewWidget::fileButton_clicked()
{
    KURL kurlFile = KFileDialog::getOpenURL();

    if( kurlFile.isEmpty() == true )
    {
        KMessageBox::error( this, "Error the file is empty, please choose another file" );
        return;
    }

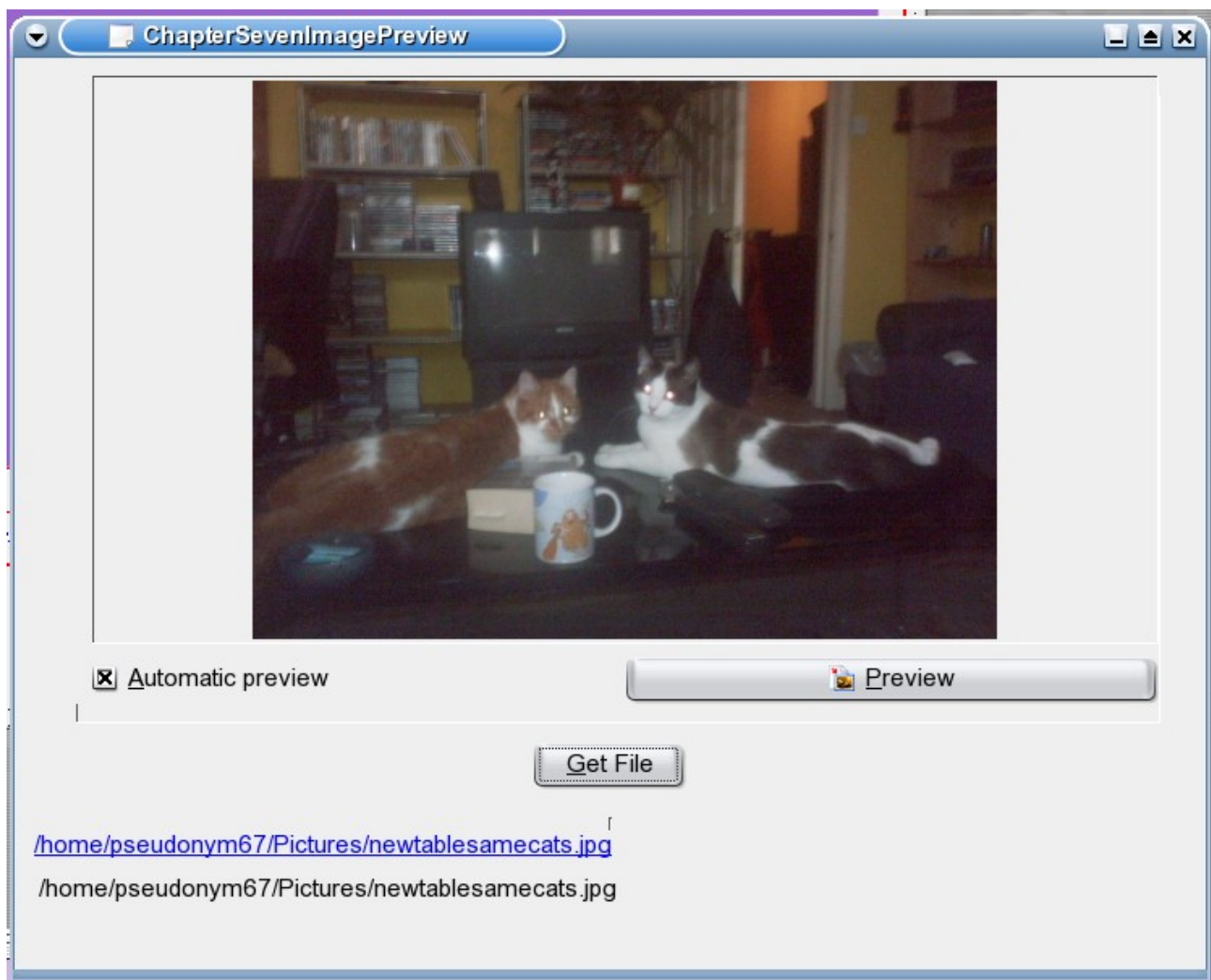
    pKImageFilePreview->showPreview( kurlFile );
    QString string = kurlFile.pathOrURL();
    urlLabel->setText( kurlFile.pathOrURL() );
    activeLabel->setText( kurlFile.pathOrURL() );
}
```

We run a KFileDialog as seen previously to get the file and then pass the filename to the KImageFilePreview widget with a call to showPreview. Then we add the filename to the labels.

When the KImageFilePreview widget is running it has two modes the first which is the default is to display the preview of the file as soon as it is passed a filename,



If we uncheck the Automatic preview option the KImageFilePreview widget will wait until we click the Preview button before it displays the preview.



Once we have set the widget up and passed it a filename then there is nothing else for us to do as the widget takes care of the rest for us.

As for the labels the KURLLabel is a label with idea that we can use it to provide links to other web pages or files by simply clicking on it, although the handling of this is left up to us and is not implemented here, but it means that by default the label changes colour when the mouse is hovered over it indicating that it is an active widget ready for use.

</home/pseudonym67/Pictures/newtablesamecats.jpg>

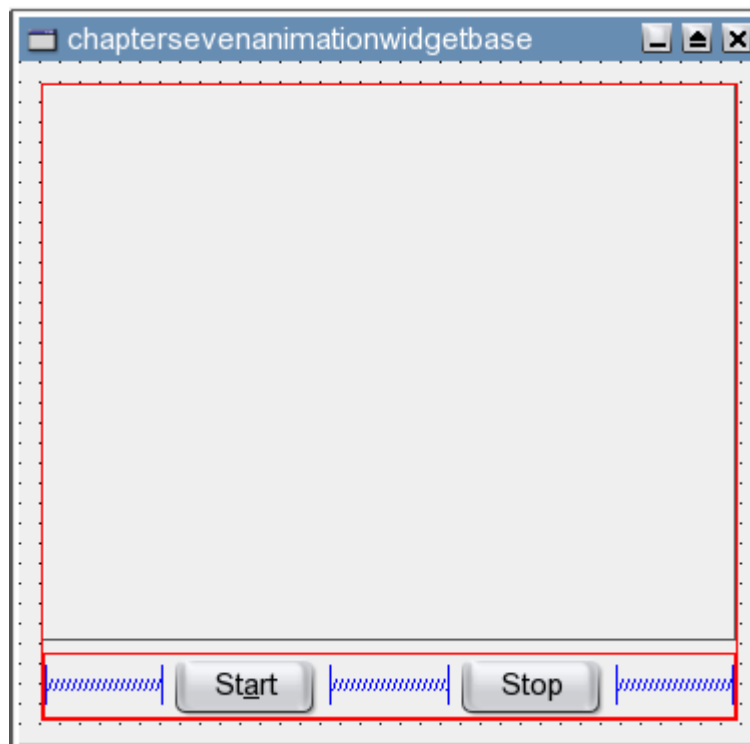
The KActiveLabel widget is a different implementation of the same idea although it doesn't give the same visual clues it does have an openLink function that can be called with the file name.

KDE Animation

The next example just looks at the KAnimWidget which is included in the Display (KDE) section of the Toolbox.

Chapter 7 KDE Display Widgets

The setup of the project looks like this,



As you can see there is no widget viewable on the form which means that the drag and drop aspect of KDevelop adds the wrong constructor again so we set up as with the other widgets seen previously and add the code,

```
pAnimWidget = new KAnimWidget( "kde", 0, this, "kAnimWidget1" );
pAnimWidget->setGeometry( QRect( 140, 90, 90, 90 ) );
pAnimWidget->setSize( 88 );
```

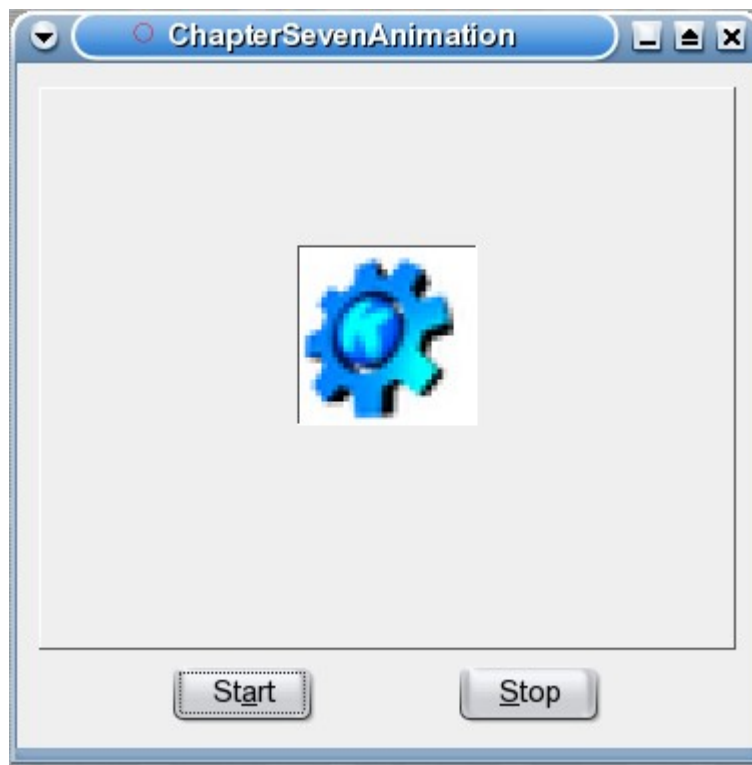
to the constructor of the ChapterSevenAnimationWidget class. And here you can also see the problem with this widget from the outset. Notice that the name passed to the constructor for the KAnimWidget is “kde” actually when running the program you can pretty much change this to the name of any KDE application and you will get the icon for that application, although it is unlikely to be animated.

The thinking behind it as far as I can tell is that all animations are just a series of icons and therefore the widget can use the KIconLoader to load the icons. This is fine when developing applications that are part of KDE but not so good when trying to develop independent applications because it only allows you to use the KDE system for loading the icons. The point being that if you create a test project in `usr/share/` called `chaptersevenanimation` and add a `icons` folder and a `pics` folder placing your icons, named `chaptersevenanimation` there then the program will not find them if you run it with the name “`chaptersevenanimation`” replacing the “`kde`” as the first parameter to the program.

So unless you are actually developing for kde I'd steer clear of this widget as it won't let you pass a directory name in for the icon or icons that you wish to load but as I said if you want to run it and change the “`kde`” it will load just about any KDE icon here as it gets them from `opt/kde3/share/icons/hicolor` (on Suse).

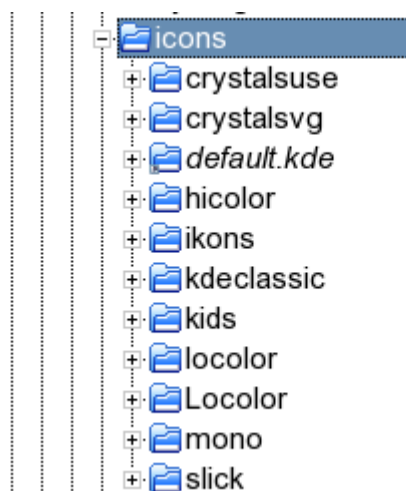
Chapter 7 KDE Display Widgets

The rest of the application is just the adding of the start and stop buttons to test the animation. Note if you are playing around with it, not all the icons are animated so if you load an icon that isn't animated and hit start the icon will just flicker a bit as it is redrawn to the screen.



Icons

All this looking into how the animation widget works cannot help but bring us round to the subject of icons and understanding how they work in KDE. If you go to `opt/kde3/share/icons` (on Suse) you will see.

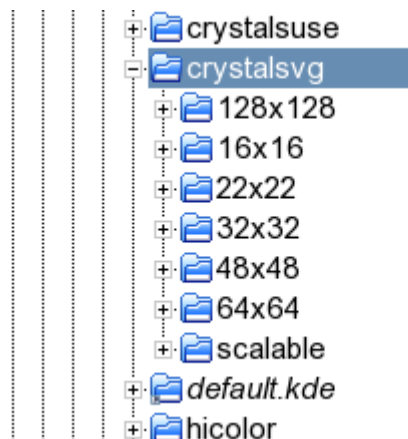


Chapter 7 KDE Display Widgets

which is the directory structure for the options that are displayed when you go to Control Center/Appearance and Themes/Icons

| Name | Description |
|--------------|--|
| Crystal SUSE | Icon Theme by Everaldo.com Design Studio |
| Crystal SVG | Icon Theme by Everaldo.com Design Studio |
| iKons | iKons Icon Theme by Kristof Borrey (kborrey@skynet.be) |
| KDE-Classic | KDE Classic Icon Theme |
| KDE-LoColor | Lowcolor Icon Theme |
| Kids | Icon Theme by Everaldo.com Design Studio |
| Monochrome | By Danny Allen (danny@dannyallen.co.uk) |
| Slick Icons | Slick Icons Version 1.5 |

The icons are then stored by size in the corresponding directories.

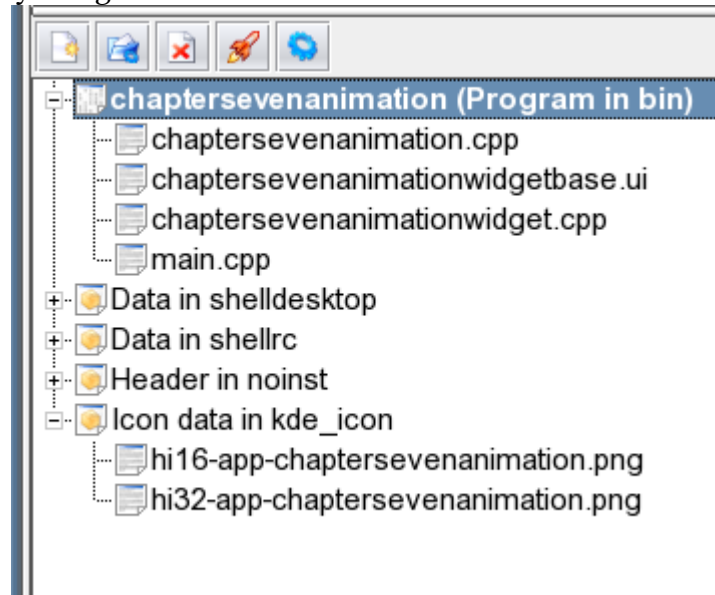


The icons size to be used on your system can be set through the Control Centre and it is here that the KIconloader used mentioned above is used throughout KDE to find the correct type and size of Icon for the system.

KDevelop Icons

One thing that's bugged me since I started looking into programming with KDevelop has been the icons it produces.

Chapter 7 KDE Display Widgets



I've no idea what these are supposed to be and have completely failed to find an application including KDevelop that can open them, so in the tradition of programmers everywhere I thought stuff 'em and did it myself.

If you look at the image for the running application of chaptersevenanimation you will see that the icon in the top left is set to a little red circle. Whereas none of the previous applications have had the icon set this is because I implemented the code myself. To do this all you do is create an icon for you project. You can overwrite the icons generated by KDevelop, it doesn't matter though as they are still not used when you run the program.

To implement the icon open the main.cpp file and add the include file

```
#include <qpixmap.h>
```

Then add the icon code to the file, just after the main widget has been set.

```
else
{
    // no session.. just start up normally
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    /// @todo do something with the command line args here

    mainWin = new ChapterSevenAnimation();
    app.setMainWidget( mainWin );
    QPixmap icon;
    icon.load( "chaptersevenanimation.png" );
    mainWin->setIcon( icon );
}
```

Chapter 7 KDE Display Widgets

```
mainWin->show();

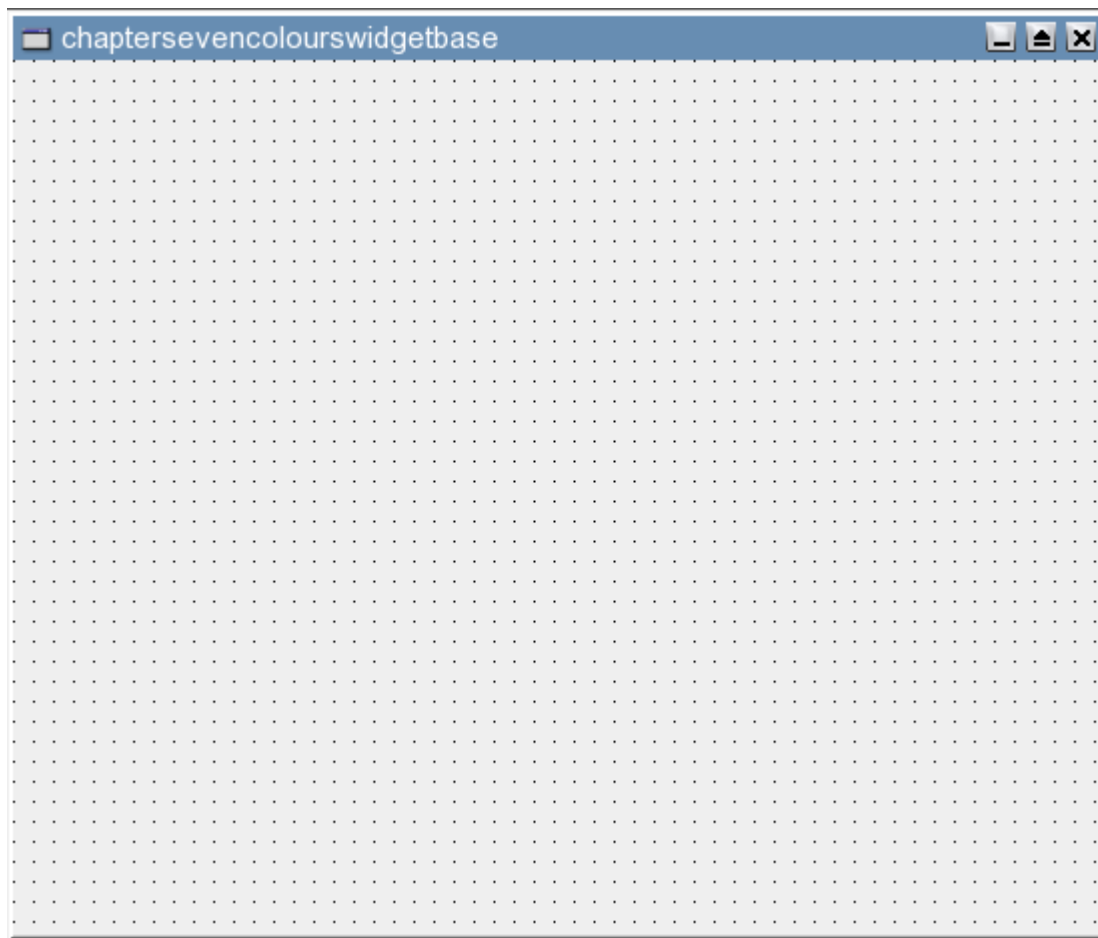
args->clear();

}
```

Loading the icon and setting the main windows icon before you show it will allow you to set any icon you want for your application.

KDE Colours

The final little application we are going to look at for this chapter is the ChapterSevenColours which looks at using the KColorPalette widget and the KColorPatch widget. As neither of these will add the correct constructor code from the gui we must add them manually which means our gui development looks like this.



Which gives absolutely nothing away so we will have to look at the chaptersevenwidget constructor to see what is going on here.

```
colorPatch = new KColorPatch( this );
colorPatch->setGeometry( QRect( 0, 0, 533, 437 ) );

paletteTable = new KPaletteTable( this );
paletteTable->setGeometry( QRect( 140, 130, 127, 151 ) );
paletteTable->setPalette( "Web" );
```

Chapter 7 KDE Display Widgets

```
connect( paletteTable, SIGNAL( colorSelected(const QColor&,const QString&) ), this, SLOT(
paletteTable_colorSelected(const QColor&,const QString&) ) );
```

We create a new KColorPatch object and set the geometry to the size of the dialog. Then we create the KPaletteTable afterwards, doing it the other way means that the KColorPatch will completely cover the KPaletteTable and it is important that we are able to see all of that widget. Once we have set the geometry for the KPaletteTable we call setPalette with the string “Web” this selects the web colour options when the KPaletteTable loads.

The options available are,

Recent Colors

Custom Colors

Web

Royal

40 Colors

Named Colors

The KPaletteTable is the same widget that we looked at when we were looking at the colour file dialog in chapter six.

Implementing Our Own Connections

The final line in the constructor is

```
connect( paletteTable, SIGNAL( colorSelected(const QColor&,const QString&) ), this, SLOT(
paletteTable_colorSelected(const QColor&,const QString&) ) );
```

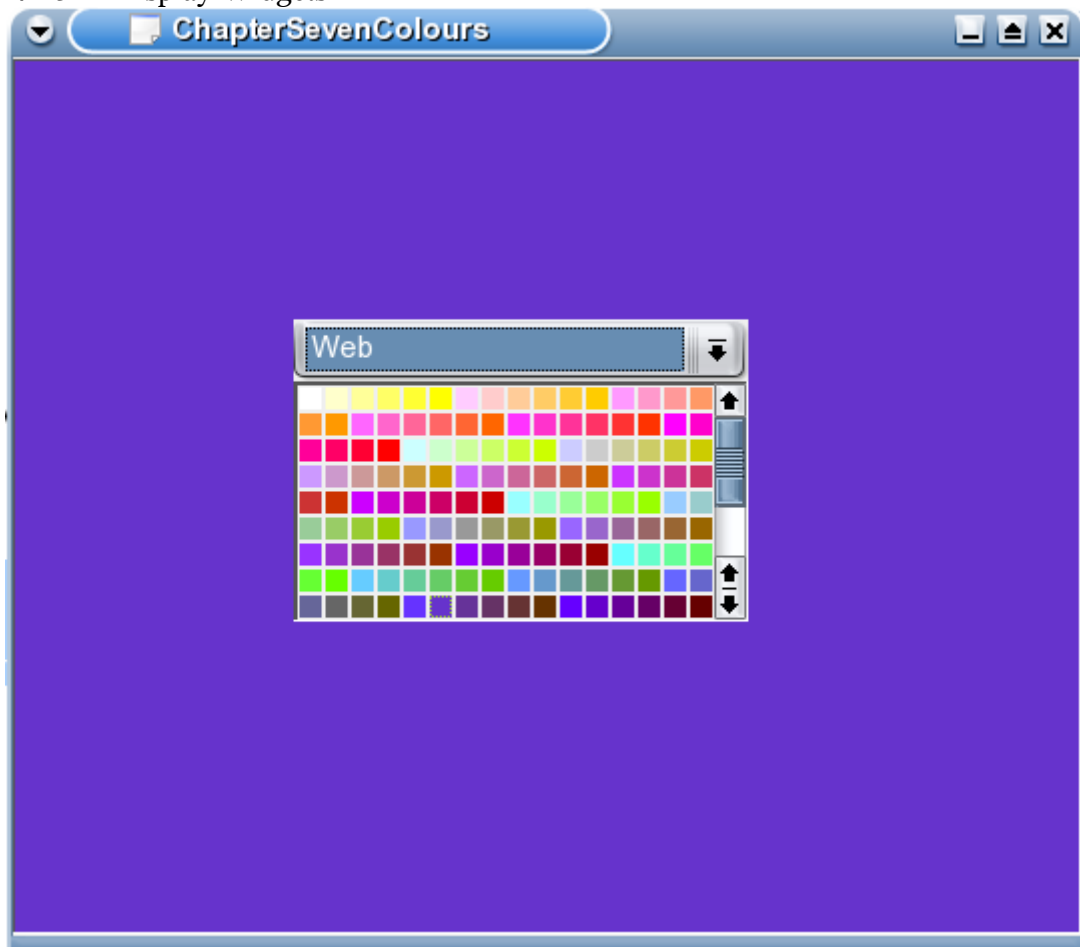
This is the connection to call the ChapterSevenColours paletteTable_colorSelected function which is implemented as,

```
void ChapterSevenColoursWidget::paletteTable_colorSelected(const QColor& color, const
QString& colorText )
{
    colorPatch->setColor( color );
}
```

This simply sets the KColorPatch widget to the colour that was selected in the KPaletteTable, however as we are unable to use the gui to set up the connection we must do it manually so once again. We call the connect function passing in the name of the widget that will be sending the signal and then state that the signal to be sent is the colorSelected signal and list the parameters before stating which widget will receive the signal which in the current case is this which refers to the ChapterSevenWidget class and then we list the slot that will receive the signal and list the parameters. In the heading we declare,

```
public slots:
    /*$PUBLIC_SLOT$*/
    virtual void paletteTable_colorSelected(const QColor& color,const QString&
colorText);
```

and then in the cpp file we add the function as listed above. Ending up with,



Summary

In this chapter we have taken our first look at some of the widgets that are exclusive to KDE beginning with the Display widgets and have taken a quick look at icons in KDE as well.

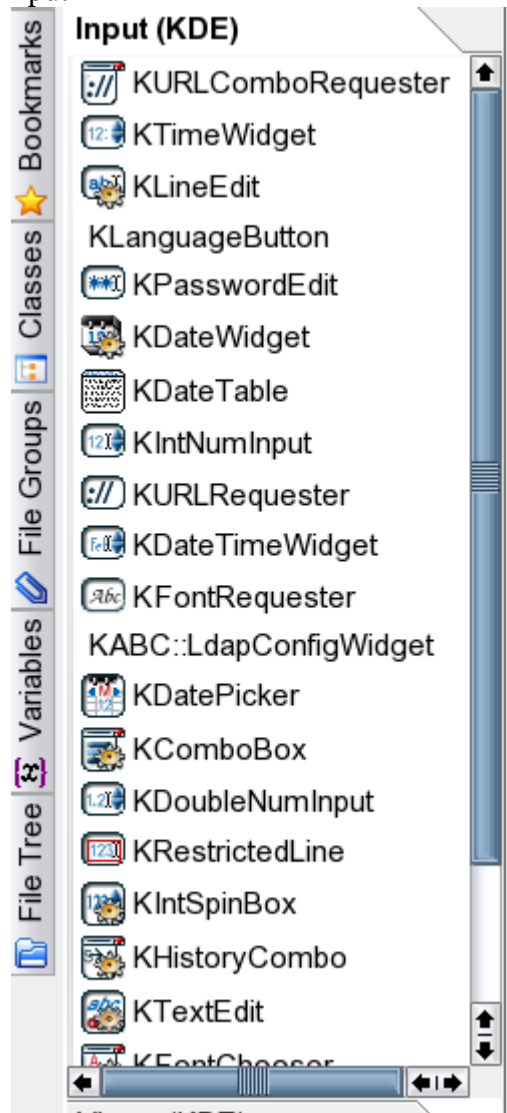
Chapter 8 KDE Buttons And Input

As we progress through the KDE specific widgets we are starting to see more and more familiar widgets from your day to day use of KDE and in this chapter as it is concerned with the inputs we will come across a few widgets that anyone who uses KDE will see on a day to day basis.

While the KDE Buttons section is simple containing only four items,



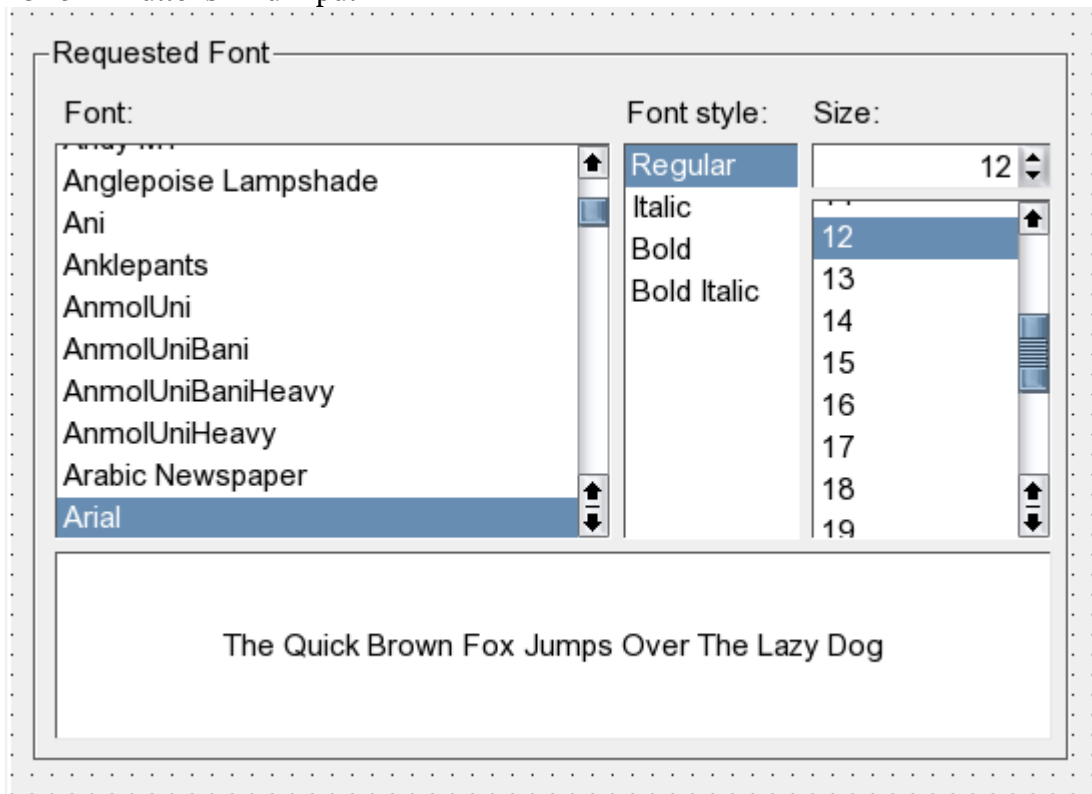
The KDE Input section is somewhat larger containing twenty three items, a few of which are shown here.



So let's get started.

Text And Fonts

We'll start by looking at the KTextEdit and use that to look at the Input options with fonts using the KFontCombo and also the KPushButton. We won't be explicitly using the KFontChooser as it looks like,



which if you were paying attention a couple of chapters ago is simply the KFontDialog without the dialog widget. We could implement it in our own dialog but then we would just be rewriting the KFontDialog. So if you can ever think of a reason to change the font without being able to open a dialog widget then here it is. Seeing as I can't we'll move swiftly on.

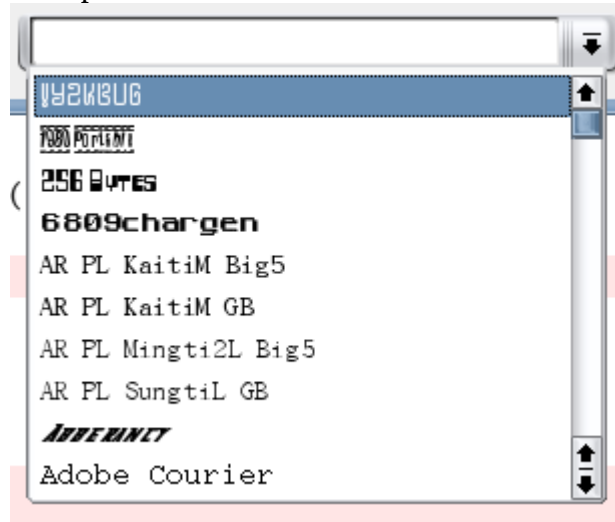
The first example for this chapter is the ChapterEightText example which uses the KTextEdit, KPushButton and the KFontCombo widgets. The KPushButton is essentially a QPushButton that inherits the KGuiItem class. If you look at the KGuiItem class in the help you can see that it's function is to provide the extensions to basic Q classes that are used as standard throughout the KDE environment. Clicking on the KPushButton will open the standard font selection dialog widget which contains the KFontChooser widget shown above using the code,

```
QFont tempFont;
if( KFontDialog::getFont( tempFont ) == KFontDialog::Accepted )
{
    textEdit->setFont( tempFont );
}
```

As with the KPushButton the KTextEdit is derived from the appropriate class, in this case QTextEdit, and extended to fit in with the requirements of the KDE environment, so setting the font is as simple as calling the QTextEdit setFont function with the new font.

The KFontCombo widget is a standard drop down Combo Box widget that is prefilled with the available KDE system fonts changing the font using this widget is just a matter of selecting it from the list.

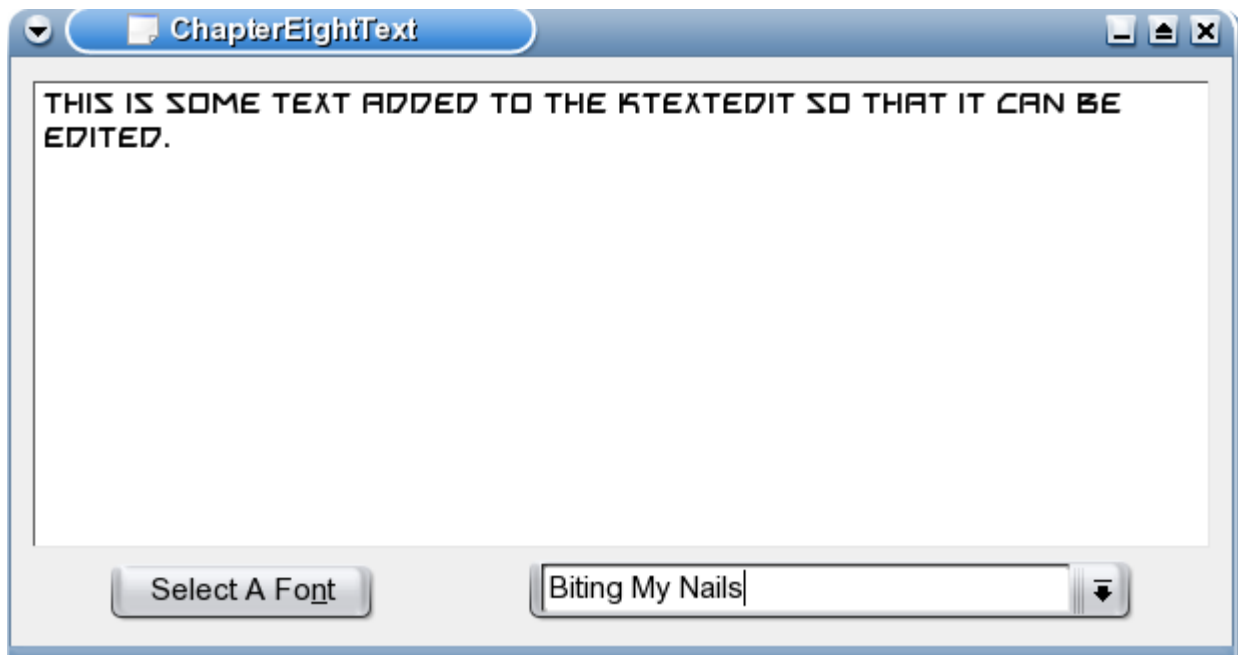
Chapter 8 KDE Buttons And Input



In order to get the selected font we respond to the KFontCombo textChanged signal,

```
void ChapterEightTextWidget::fontCombo_textChanged( const QString &font )
{
    QFont tempFont( font );
    textEdit->setFont( tempFont );
}
```

As the textChanged signal passes the name of the font we create a new QFont object using the font name before passing the font to the setFont function.

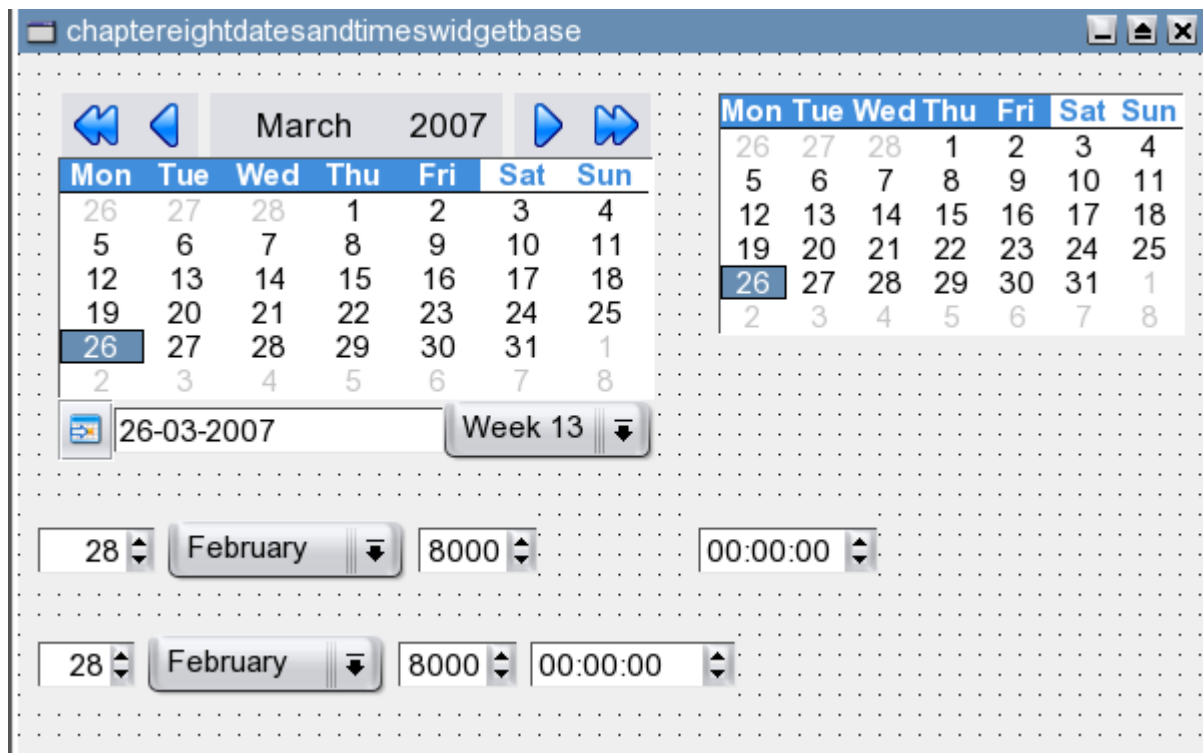


Which means the text in the KTextEdit is changed to the selected font, although whether it is readable or not depends really on which font you choose.

Dates And Times

For the dates and times example we look at the four date widgets which are the KDatePicker, the KDateTable, the KDateWidget and the KDateTimeWidget which is just a KDateWidget with the

Chapter 8 KDE Buttons And Input
KTimeWidget which is also included.



Fortuanetly for this project the constructors for the controls are set up correctly by KDevelop so we can concentrate on what it is they can do.

As can be seen from the image above the KDatePicker is a fully functioning calender widget, whereas the KDateTable is a cut down version of the same widget. The functions of these are pretty self explanatory. The only thing that we really need to know here is if we were say writing a diary where people could click on the date then we would do it by catching the tableClicked() signal in the editor and handling the signal in the ChapterEightDatesAndTimesWidget class. So the header would contain something like.

```
virtual void kDatePicker1_tableClicked();
```

and the function would be handled in the cpp file with

```
void ChapterEightDatesAndTimesWidget::kDatePicker1_tableClicked()
{
    QDate dateSelected = kDatePicker1->getDate();
    KMessageBox::information( this, "The Date you have selected is " + dateSelected.toString() );
}
```

As you can see all I do is display a message confirming the date that has been clicked on. At this point if you were writing a diary say you would display a text widget so that the user could enter the text they required.

The three controls on the bottom of the form are the KDateWidget which allows the selection of a date the KTimeWidget that allows the user to enter a time value but in this case is used to display the current time and the KDateTimeWidget which is the two previous widgets linked together.

Chapter 8 KDE Buttons And Input Timers

Timers are a useful tool to be aware of when programming and can help in many time dependant situations. Simply put in linux terms a timer is a signal that will be fired at whatever interval you tell it to or only once if you wish.

To set up a timer you need to add somethings to your class header file.

```
class QTimer;

class ChapterEightDatesAndTimesWidget : public ChapterEightDatesAndTimesWidgetBase
{

public slots:
    /*$PUBLIC_SLOT$*/
    virtual void updateClock();

private:
    QTimer *clockTimer;
};
```

The above is an edited version of the class header showing just the requirements for the timer to be set up. First of all we declare the timer class at the top of the header file. This just tells the compiler to allow the use of the timer class for now and it will be derfined later. It is defined by including the header file for the QTimer class in the cpp file. Next we declare a new slot that we are going to use whenever the timer is fired and finaly we declare the timer itself.

In the cpp file we first of all include the correct header file.

```
#include <qtimer.h>
```

Then in the constructor we set the timer up.

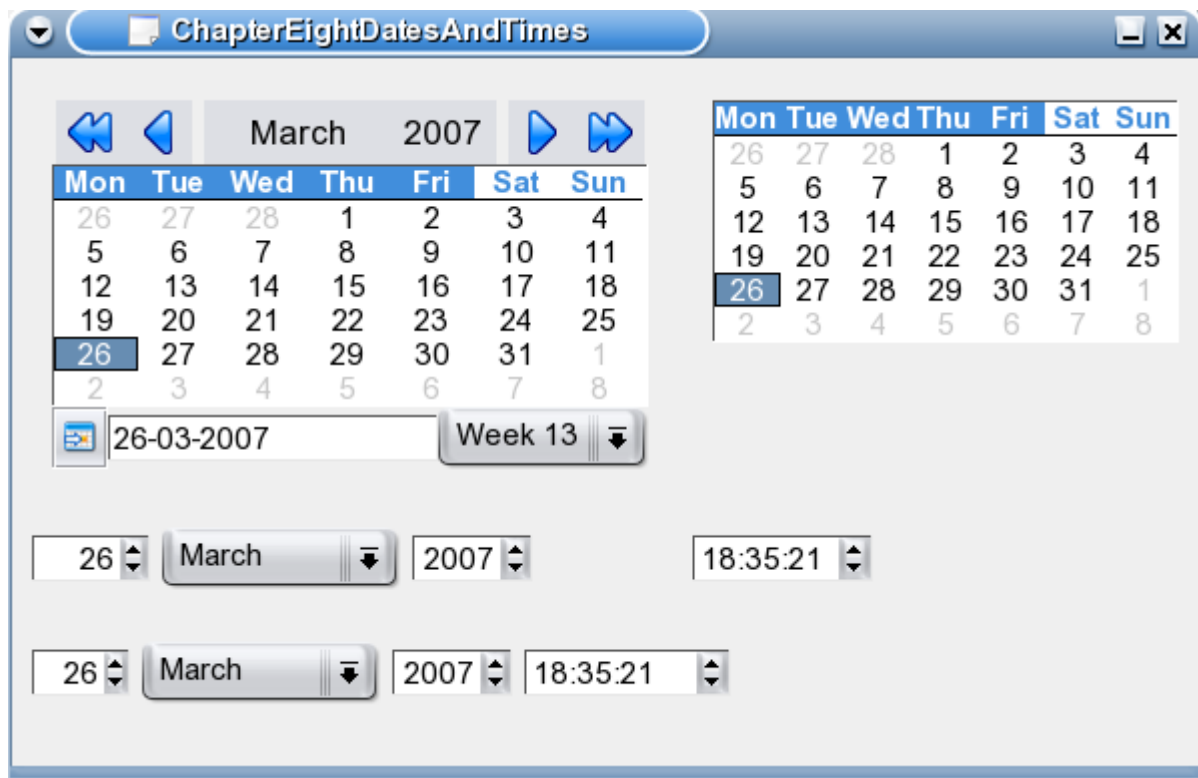
```
clockTimer = new QTimer( this );
connect( clockTimer, SIGNAL( timeout() ), this, SLOT( updateClock() ) );
clockTimer->start( 1000, FALSE );
```

First we create the timer telling it that this class will be it's parent and as such will be responsible for deleting the timer when it's constructor is called. Then we connect the timer signal with the correct slot by calling the connect function with the clockTimer that we have just created and stating that we want to catch the signal timeout(), which is the only signal a QTimer will emit so no chance of confusion there. We then set this class as the reciever for the signal and the updateClock() function as the receiving slot. Finally we start the timer telling it to fire every second or to be exact every 1000 milliseconds and then set the single shot parameter to false to tell the timer to continue firing for as long as the program is running or until we tell it to stop.

The implementation of the updateClock slot is then

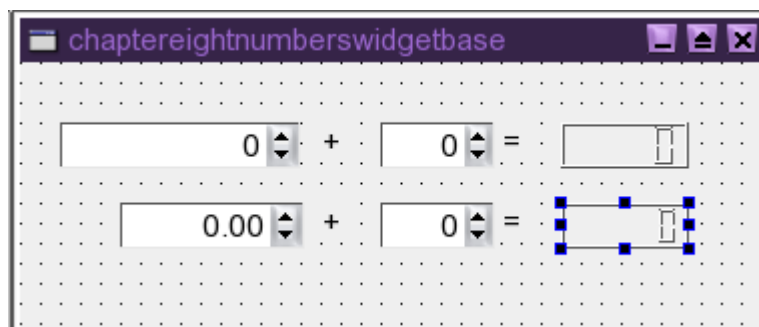
```
void ChapterEightDatesAndTimesWidget::updateClock()
{
    clockWidget->setTime( QTime::currentTime() );
    dateTimeWidget->setDateTime( QDateTime::currentDateTime() );
}
```

The implementation simply sets the time controls of the widget to the current time.



Numbers

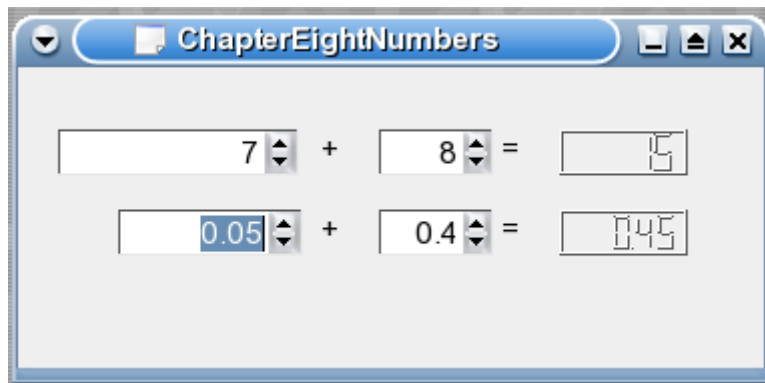
This project is by far the simplest project in the book and illustrates the for number spin box widgets, these are the KIntNumInput widget, the KIntSpinBox, the KDoubleNumInput widget and the KDoubleSpinBox widget. These are all straight forward number displays that if they are working properly there isn't really much of interest to be said about them.



As you can see all we are doing is displaying the widgets and then adding the integers and the double values and displaying the results in QLCDNumber displays. The way this is done is by accepting the valueChanged signal for each box and then updating the display. So the signal handler that catches the valueChanged signal for the KIntSpinBox widget would read.

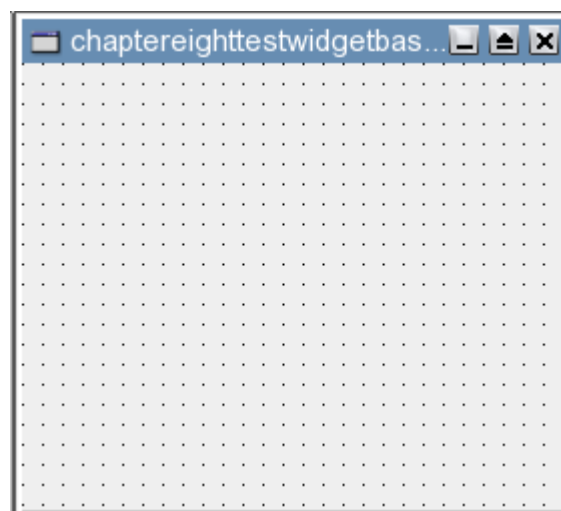
```
void ChapterEightNumbersWidget::kIntSpinBox_valueChanged( int number )
{
    intValue->display( kIntNumInput->value() + number );
}
```

Chapter 8 KDE Buttons And Input
Which gives a running program that looks like,



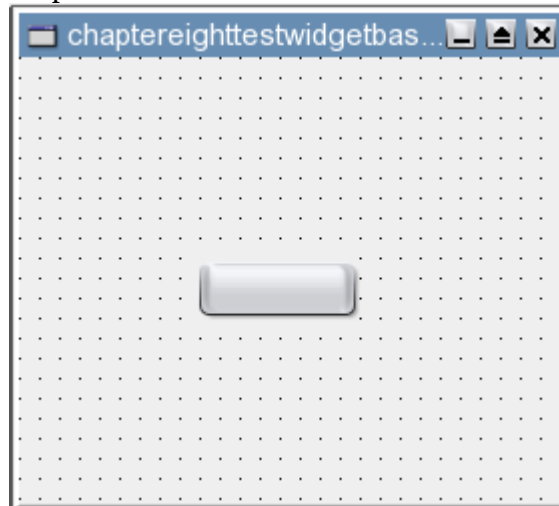
Tip Of The Day

This tip is more of a familiarity breeds contempt issue than any real problem with anything else and it can be easily duplicated but is really frustrating if you do it to yourself without realising what it is. Setup a start project like so,

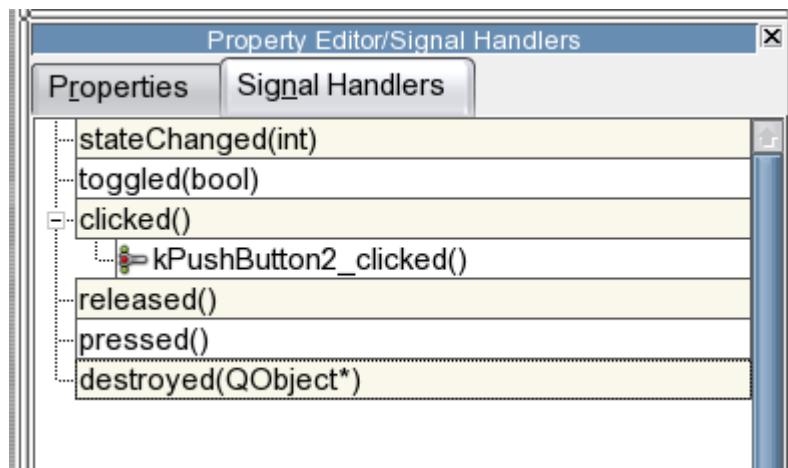


Now add a button. Note though that any widget will do I'm just picking on the buttons here.

Chapter 8 KDE Buttons And Input



Now Add a signal handler.



Now add a message box, not forgetting to add the include for the header.

```
KMessageBox::information( this, "The button has been clicked" );
```

Now if you run the code and click the button you will get nothing, no messagebox nothing. This is because although you have saved the code in your implementation classes you haven't saved the form that tells the code generator to basically generate the connect call that will wire in the signal to the slot you have set up in the implementation class with the result that you are looking at perfectly reasonable but unworking code and trying to work out what is wrong with it when the back plumbing hasn't been done.

Of course this is just a simple demonstration the place this is more likely to show up is when you are in the process of implementing a gui and get called away or have to stop for some other reason.

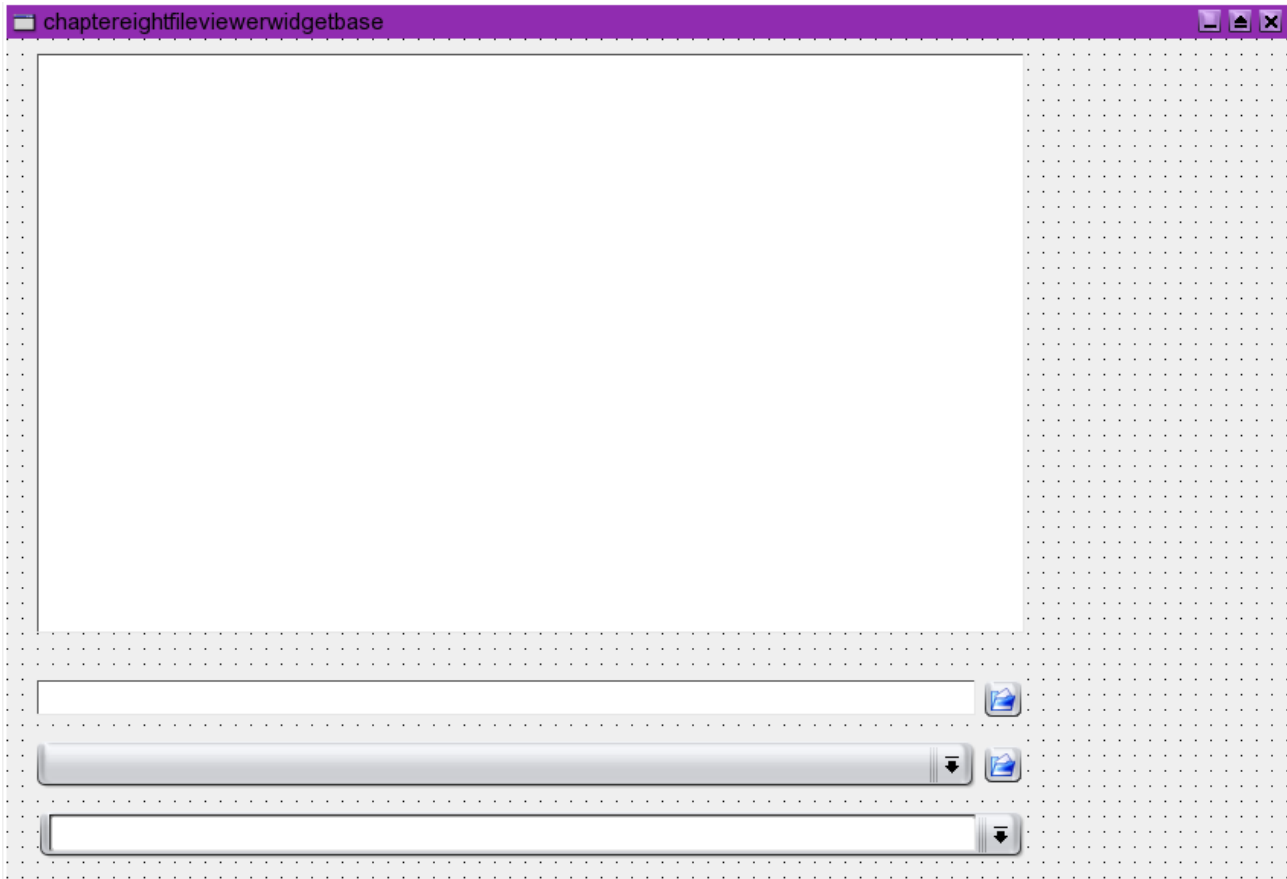
File Viewer

For the final project in this chapter we are going to implement a small file viewer to show how some of the widgets work. These widgets are the KURLRequester, the KHistoryCombo and the

Chapter 8 KDE Buttons And Input

KURLComboRequester. The project also implements the KArrowButton widget but we don't implement the KIconButton or the KKeyButton widgets. The reason these buttons are left out is because in the case of the KIconButton I cannot think of a single use for a button whose only function is to choose the icon that it is going to display and in the case of the KKeyButton I was simply unable to get it working effectively.

The project in development looks like this,



The project simply displays a text box for the files and the three widgets that we are going to use to select the files to be displayed. The top widget is the KURLRequester, the middle widget is the KURLComboRequester and the bottom widget is the KHistoryCombo. The reason for the space on the right hand side is because this is where the four KArrowButton widgets that we also use will appear. As with some of the other widgets the KArrowButton will not work from the Gui interface so we have to add the buttons manually with the code.

```
kArrowButtonUp = new KArrowButton( this, Qt::UpArrow, "ArrowUp" );
kArrowButtonUp->setGeometry( QRect( 750, 80, 20, 20 ) );
kArrowButtonUp->setProperty( "arrowType", 0 );

connect( kArrowButtonUp, SIGNAL( pressed() ), this, SLOT( kArrowButtonUp_pressed() ) );
```

This is the code from the ChapterEightFileViewerWidget class that implements the Up Button. There are the four buttons up, down, left and right that are used to allow the user to move the cursor around the text file once it is loaded.

Chapter 8 KDE Buttons And Input

The values for the call to `setGeometry` were taken by adding the `KArrowButton` to the Gui as normal and then copying the setup information from the generated `ChapterEightFileViewerWidgetBase` class in the debug directory before deleting the `KArrowButton` from the Gui as the constructor code generated is incorrect.

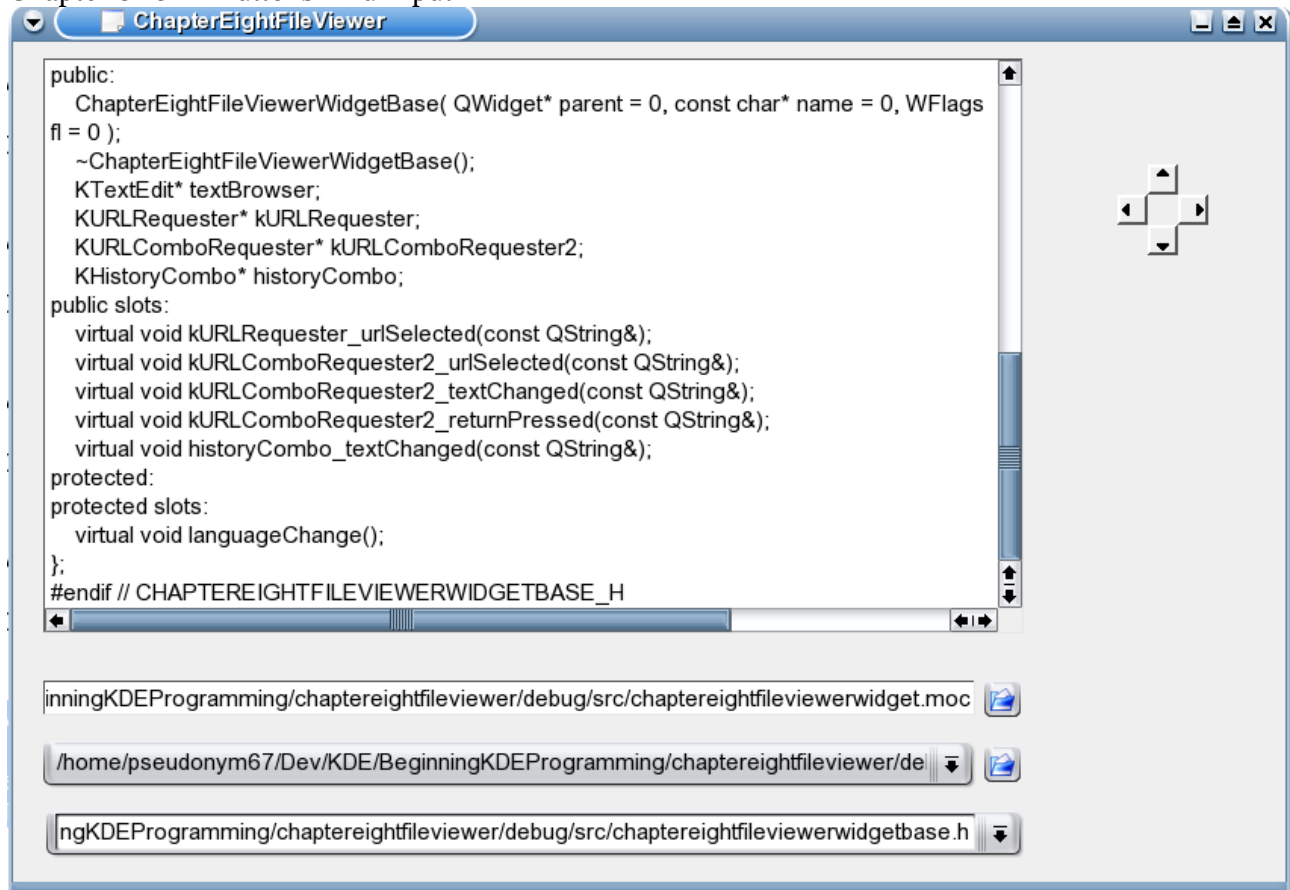
The call to `setProperty` is to tell the `KArrowButton` which way the pointer it draws should be pointing and the connect sets up the receiver for the pressed signal emitted by this button as the `kArrowButtonUp_pressed` slot.

The idea behind `KURLRequester` and `KURLComboRequester` is that you click the button to the right and a standard file dialog opens so that you can select the file with the selected file then being displayed in the box to the left. So whenever a file is selected the Requester widget will send a `urlSelected` signal which we receive,

```
void ChapterEightFileViewerWidget::kURLComboRequester2_urlSelected( const QString&
fileName )
{
    openFile( fileName );
    historyCombo->addToHistory( fileName );
}
```

We then open the filename, including the path to the history combo and continue. There is, however, just one problem and that is that when we try to change the file displayed using the combo widget nothing happens yet when we use the combo box in the history combo things change. The reason for this is that the `urlSelected` signal is only sent when the file is changed by the dialog and not when you use the drop down box. In fact the only signal we can catch is the `textChanged` signal which is only sent from an editable combo box widget. The confusing thing about this is that while the `KURLComboRequester` could arguably be editable there is no logical reason for the `KHistoryCombo` to be editable at all. This is in all honesty just plain confusing and hopefully will be sorted in KDE 4.0 until then it's probably better to just ignore these.

Chapter 8 KDE Buttons And Input



Summary

In this chapter we've briefly ran through the bulk of the KDE specific widgets that largely are just extensions of the widgets that we have seen before. As a result for the main parts these widgets have very specific tasks that either fit with a specific project or not rather than being general purpose. For the next chapter we look at the larger widgets that are contained in the Views (KDE) and Container (KDE) sections.

Chapter 9 KDE Containers and Views

In this chapter we will be looking at the KDE supplied containers and views although there will be a couple of exceptions, these being the KCMModule container which is a special type of widget that is used in the KDE Control Center. It is the KDE equivalent of a Windows Control Panel module and as such deals with the way KDE as a whole works. Once again if you are at that level of writing for KDE then you probably won't need to read this project.

The second widget to be avoided is the KTabWidget. This at first appearances seems to be set up as a wizard type widget as it doesn't allow access to the tab labels through the gui and on display only shows a single tab. So I decided to use it as a wizard type widget but it didn't respond to

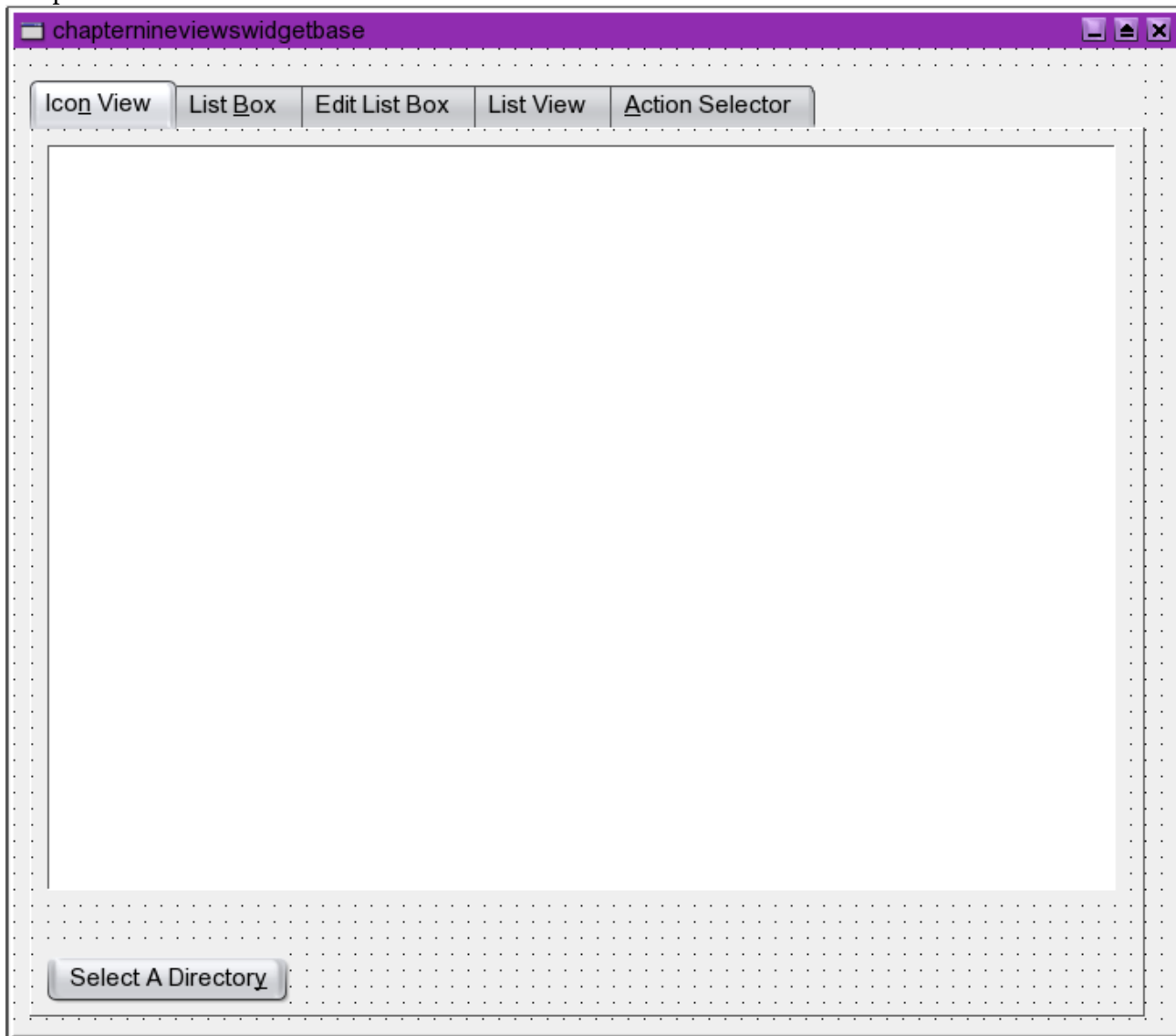
```
if( nIndex < 4 )  
    kTabWidget->setCurrentPage( nIndex + 1 );  
else  
    kTabWidget->setCurrentPage( 0 );
```

So I decided to use QTabWidget for the first project for this chapter instead.

Views

This project is very similar to what we saw in Chapter Four where a series of views are displayed within a single project. This project uses the KIconView, the KListBox, the KEditListBox, the KListView and the KActionSelector all used as separate tabs in a QTabWidget like so.

Chapter 9 KDE Containers and Views



There are no problems with setting up any one these widgets and the code is almost identical for some of the views as we saw in Chapter Four. So the code that fills the KIconView is,

```
QString strDirectory = KFileDialog::getExistingDirectory( QString::null, this, "Select A
Directory" );

QDir dir( strDirectory );
const QFileInfoList *fileInfoList = dir.entryInfoList();
QFileInfoListIterator it( *fileInfoList );
QFileInfo *ptrFileItem;

iconView->clear();

QFileInfoListIterator iconIt( *fileInfoList );
while( ( ptrFileItem = iconIt.current() ) != 0 )
{
    // only using the files
    //

    if( ptrFileItem->isFile() == true )
    {
        KURL fileURL( ptrFileItem->absFilePath() );
        KFileItem fileItem( KFileItem::Unknown, fileURL, true );
        new QIconViewItem( iconView, ptrFileItem->fileName(), fileItem.pixmap( 0 ) );
    }
}
```

Chapter 9 KDE Containers and Views

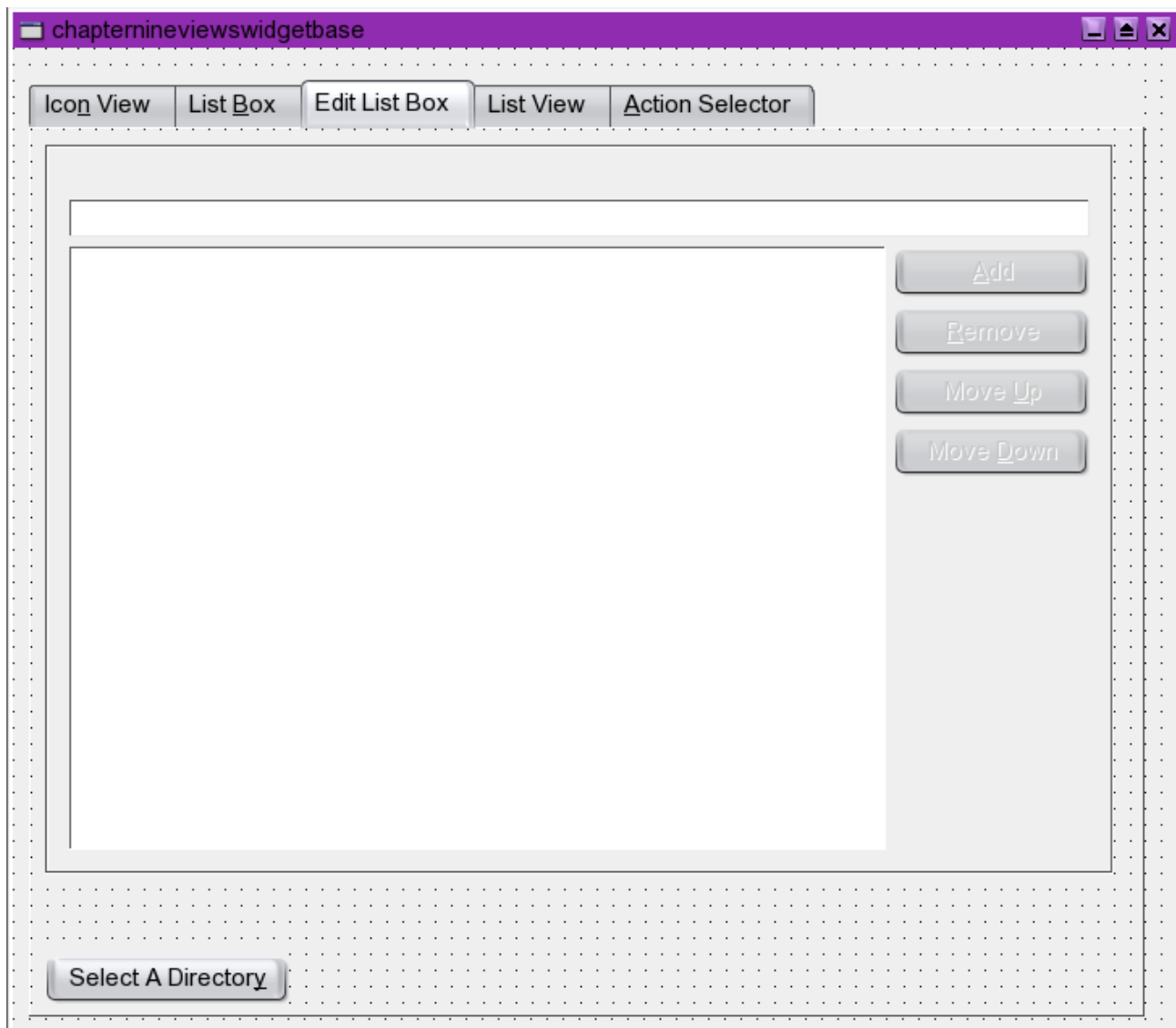
```
    }  
  
    ++iconIt;  
}
```

Here we use the template that is used for the views with exception of the KActionSelector. By using the KFileDialog we select a directory and then iterate through the items in the directory in this case creating a new QIconViewItem within the KIconView for each item present.

The KListBox follows the same format with the exception that the important part of the iteration reads,

```
if( ptrFileItem->isFile() == true )  
{  
    listBox->insertItem( ptrFileItem->fileName() );  
}
```

The KEditListBox is exactly the same from the coding point of view as the list box. The main exception here is how the KEditListBox looks and works.



As you can see the KEditListBox widget is a standard list box with a edit box widget across the top and four buttons on the right hand side. These are the Add, Remove, Move Up and the Move Down buttons. You can type directly into the KEditBox and then click the Add button to add new items. If

Chapter 9 KDE Containers and Views

you select an item from the list box the name will appear in the KEditBox at the top and you can change what is there and these changes will be reflected in the KListBox widget. You can delete any selected item in the KListBox with the Remove button and change the position of any item within the list with the Move Up and the Move Down buttons. As there is no code in the project to save any changes, should you run this program you can play about with it as much as you want.

The code for the KEditListBox is identical to that for the KListBox

```
if( ptrFileItem->isFile() == true )
{
    editListBox->insertItem( ptrFileItem->fileName() );
}
```

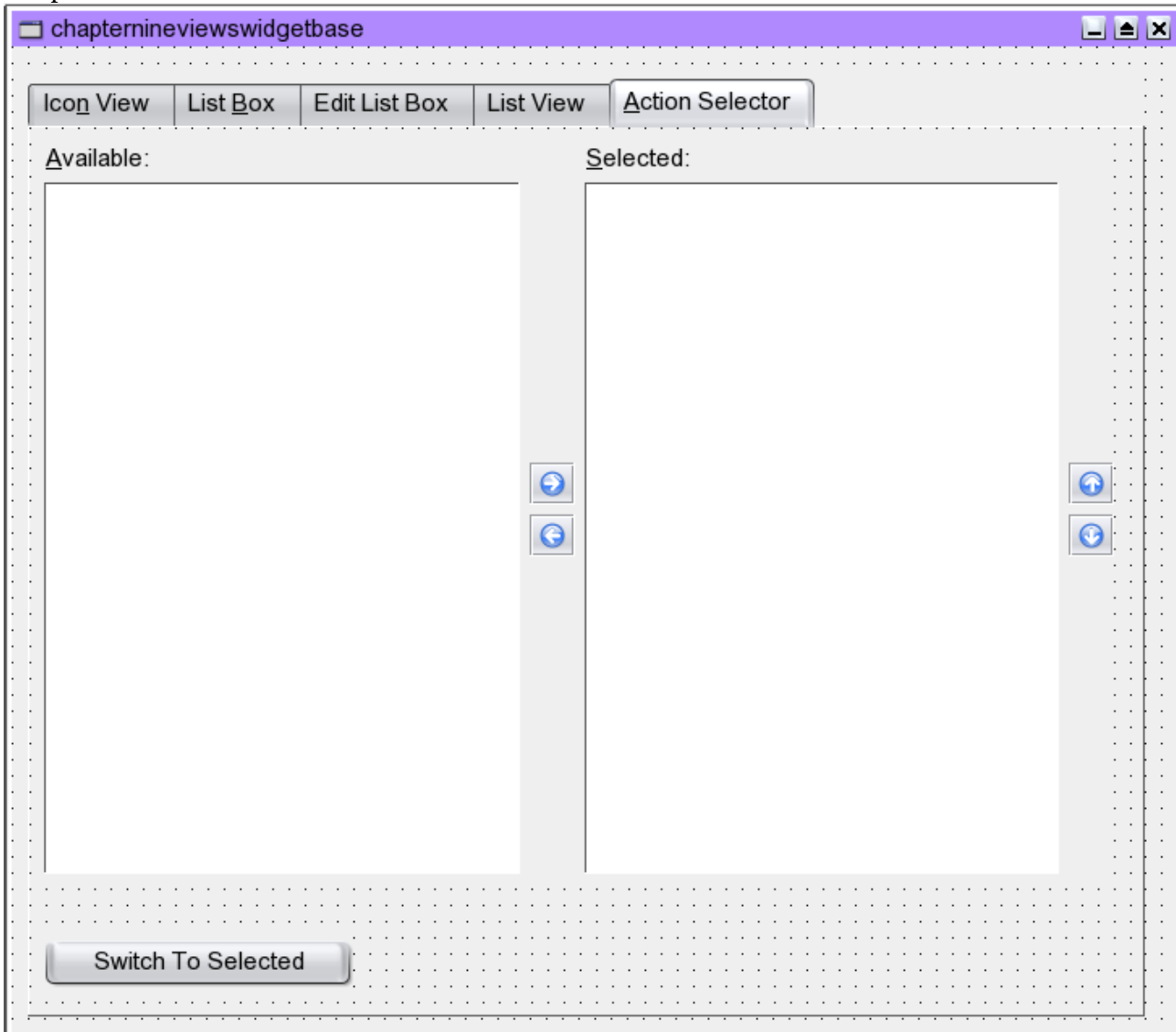
The code for the KListView is more complicated but is still the code we have seen previously

```
QListViewItem *listViewItem = new QListViewItem( listView );
KURL fileURL( ptrFileItem->absFilePath() );
KFileItem fileItem( KFileItem::Unknown, KFileItem::Unknown, fileURL, true );
listViewItem->setPixmap( ColumnOne, fileItem.pixmap( 0 ) );
listViewItem->setText( ColumnTwo, ptrFileItem->fileName() );
if( ptrFileItem->isSymLink() == true )
    listViewItem->setText( ColumnThree, "true" );
else
    listViewItem->setText( ColumnThree, "false" );
if( ptrFileItem->isDir() == true )
    listViewItem->setText( ColumnFour, "true" );
else
    listViewItem->setText( ColumnFour, "false" );
if( ptrFileItem->isFile() == true )
    listViewItem->setText( ColumnFive, "true" );
else
    listViewItem->setText( ColumnFive, "false" );
if( ptrFileItem->isReadable() == true )
    listViewItem->setText( ColumnSix, "true" );
else
    listViewItem->setText( ColumnSix, "false" );
if( ptrFileItem->isWritable() == true )
    listViewItem->setText( ColumnSeven, "true" );
else
    listViewItem->setText( ColumnSeven, "false" );
if( ptrFileItem->isHidden() == true )
    listViewItem->setText( ColumnEight, "true" );
else
    listViewItem->setText( ColumnEight, "false" );

++it;
```

For each file we create a new QListViewItem in the KListView and then after loading the Icon for the file and setting the filename we check the true or false value of certain aspects of the file for the remaining columns.

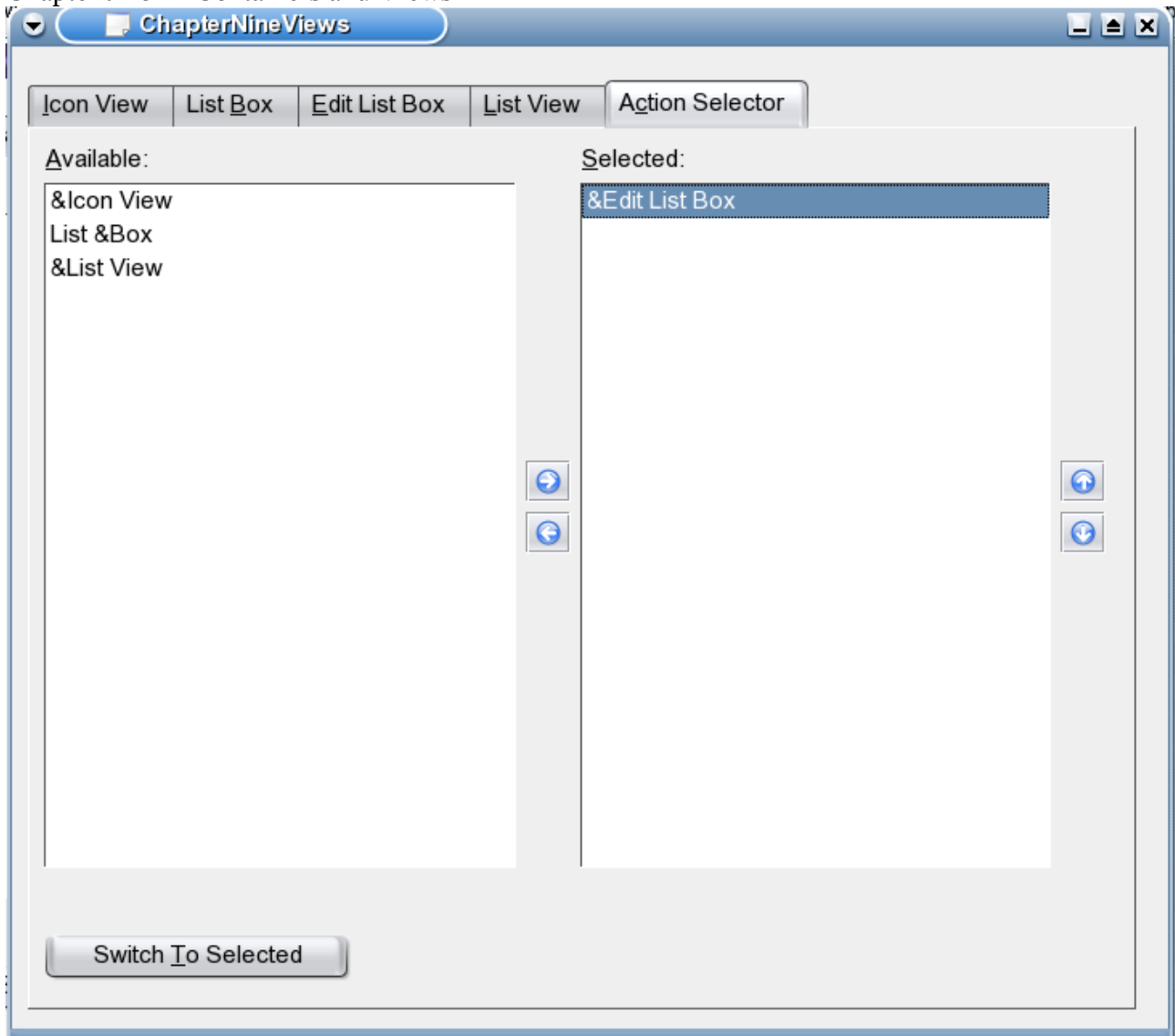
The final Item in the program is the KActionSelector,



The idea here is that you have a number of available actions or tasks to do in the Available side of the, not all of which will need to be done at any given time. The ones that are required at the present time will be moved into the Selected side of the box. It is then up to the programmer the decide how and when these should be run. One important thing to note here is that the Available and Selected are standard list boxes so when using them you need to be aware that you you are using words or phrases that both mean something to people reading them but can also be picked up in the code.

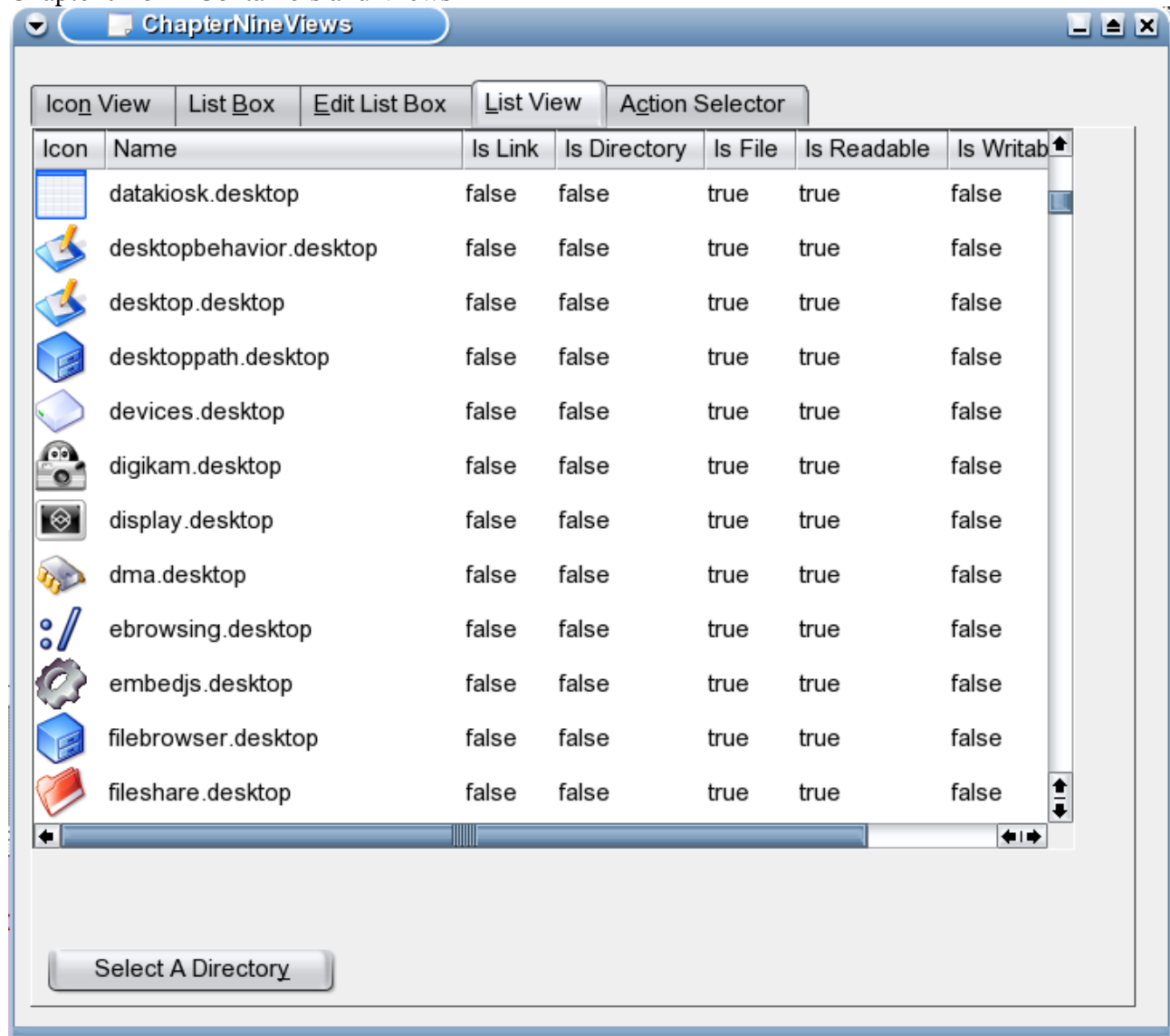
For this test implementation I've added the names of the other views and then added a button that can be used to to take the user straight to the highlighted view.

Chapter 9 KDE Containers and Views

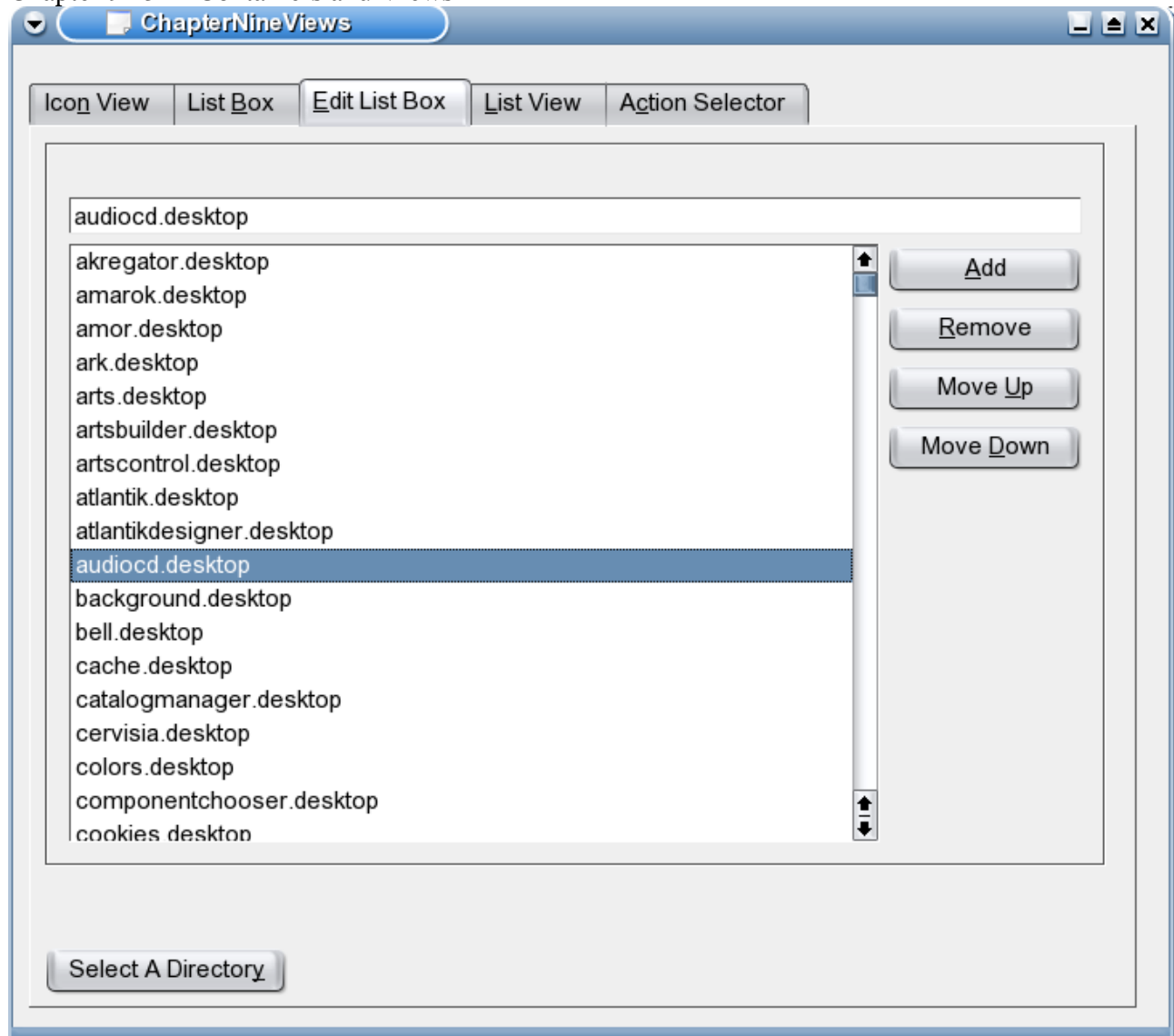


For the running application images I've used the `Opt/kde3/applications/kde` directory which holds all the desktop icons for the kde applications.

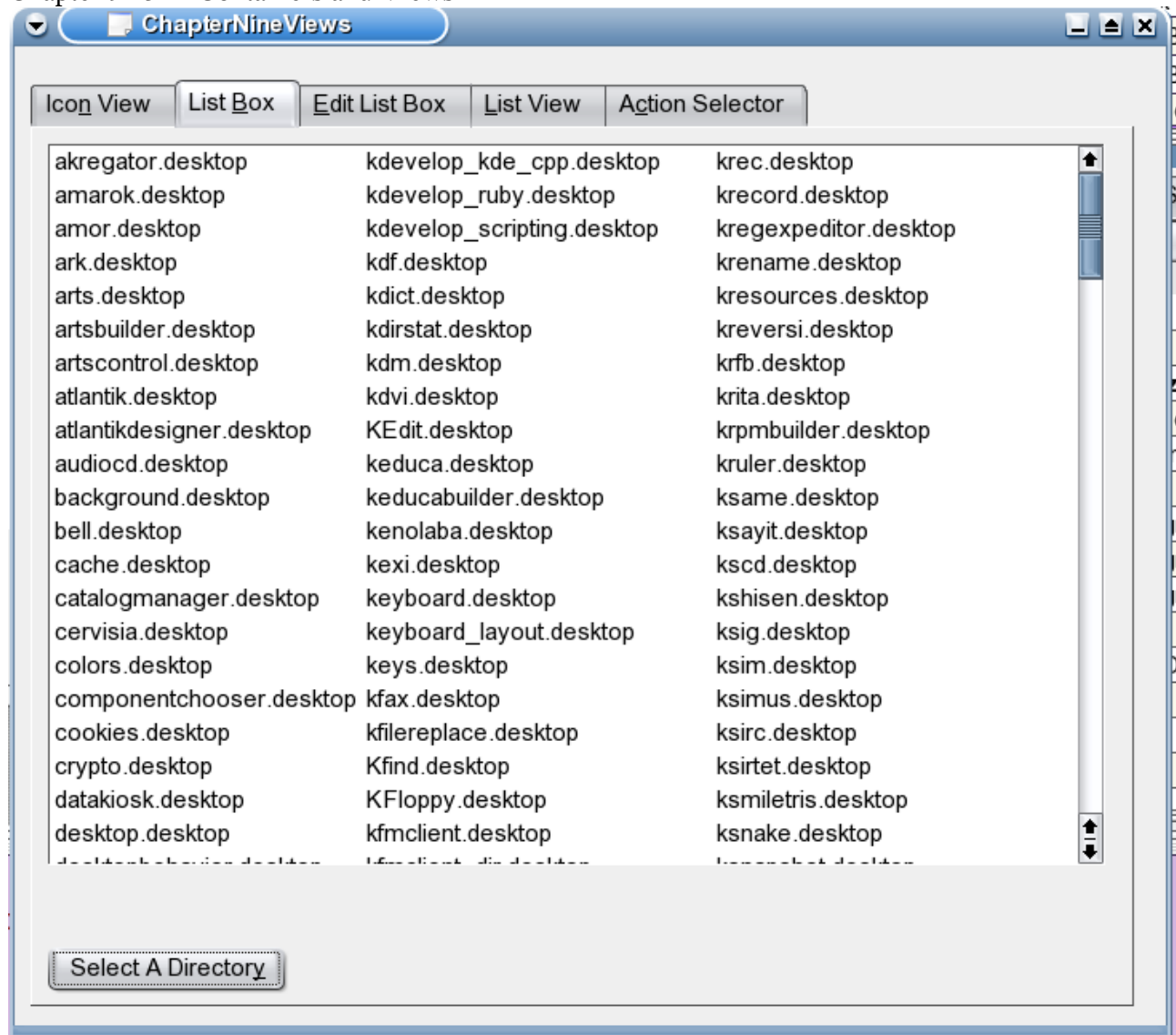
Chapter 9 KDE Containers and Views

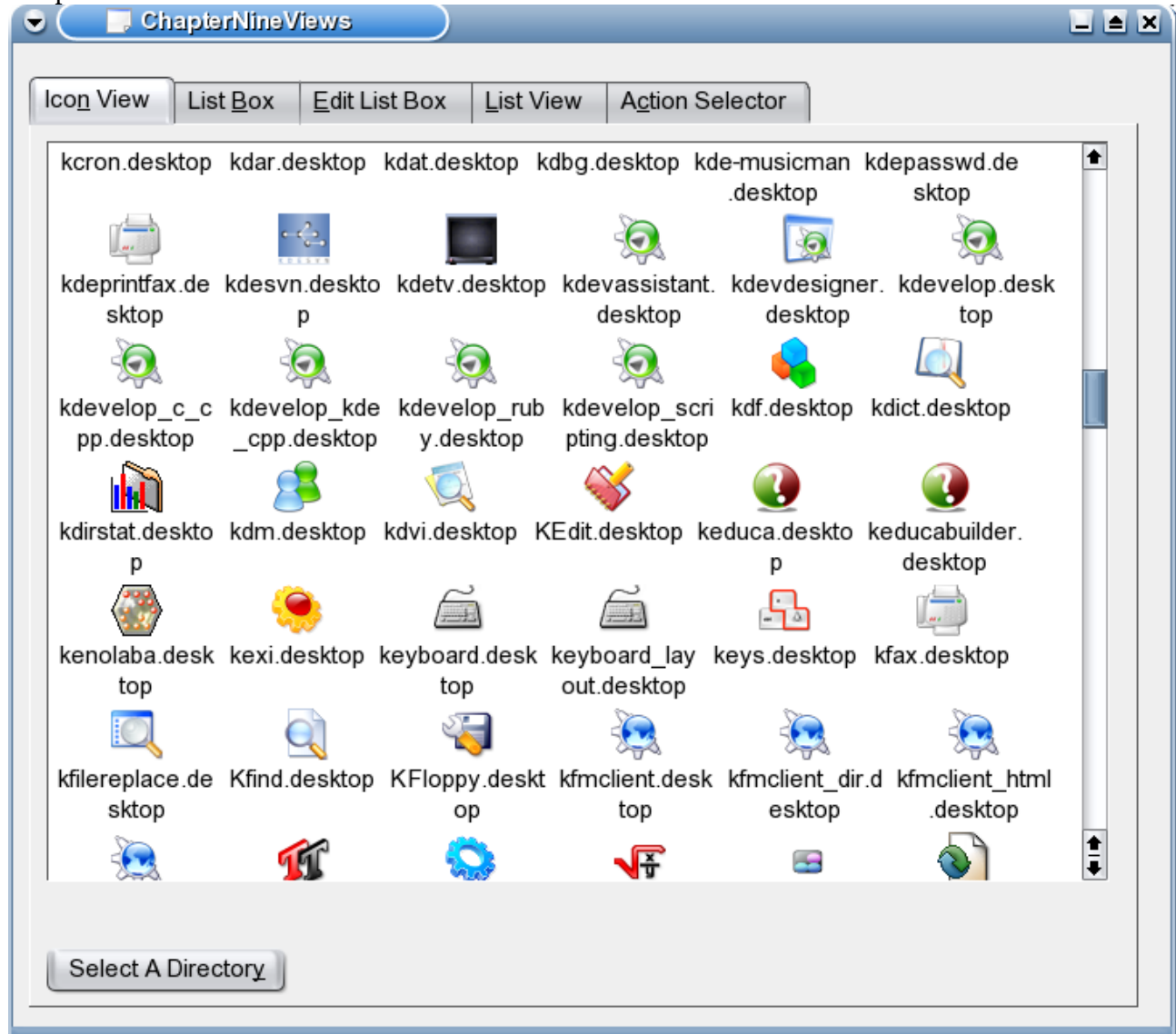


Chapter 9 KDE Containers and Views



Chapter 9 KDE Containers and Views





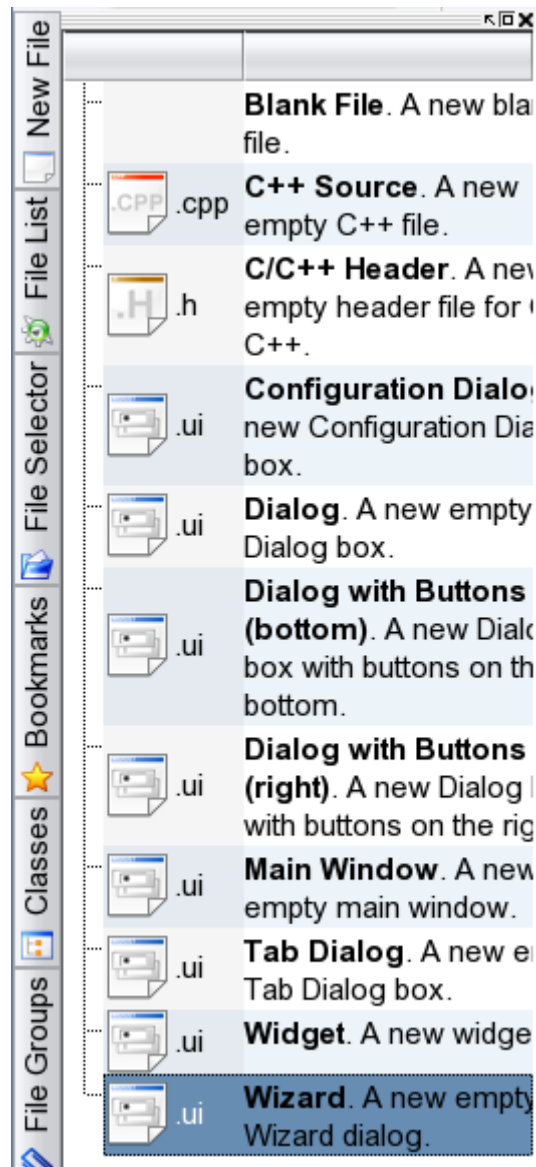
Containers

For this project we are going to be looking at the kinds of containers that you can add to your KDE project. There is a minor but here because I mentioned previously why I wasn't going to be looking at the,



Chapter 9 KDE Containers and Views

KCModule and the KTabWidget but we are not going to be dealing with the KDialog at least not the one shown in the toolbox Container (KDE) section for the simple reason that attempting to add a dialog to your widget form in this way will just add a frame on the existing widget. When what we really want are separate Dialog forms. To get those we need to look at the New File Section.



which not only allows us to add new C++ source and header files but a range of new widgets to our program.

Tip Of The Day

When dealing with these dialogs remember that the ui file you are working with creates the class at compile time. This means that you cannot handle the dialog functions or any buttons that you may add within the dialog code. If you want to keep the code for your dialog separate from the main code of your program then create a file within the project to handle the dialog and any signal responses that you wish to add by subclassing the .ui file in the automake manager you do this by right clicking on the .ui file and selecting the subclassing wizard. Ultimately though it is up to you where you choose to handle any signals emitted by your dialog.

Chapter 9 KDE Containers and Views

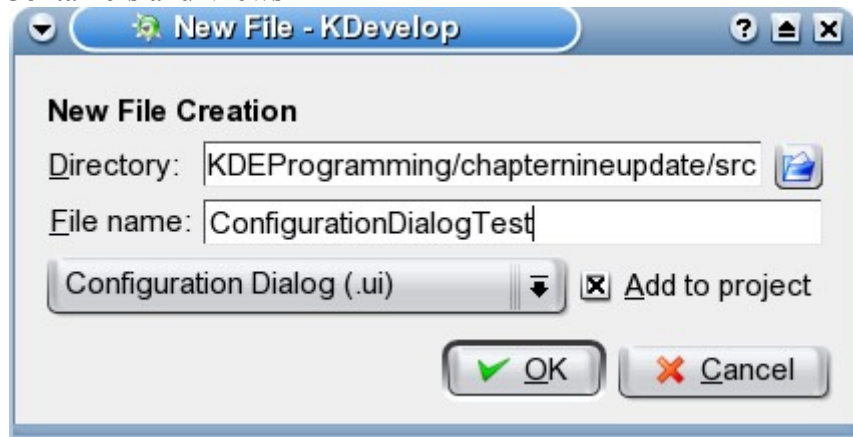
This project is another no frills this is simply how you do it project showing how the various dialogs above work or not as the case may be. The project itself simply consists of a row of buttons that call the various containers. usually with the code

```
ConfigurationDialog *dlg = new ConfigurationDialog();  
dlg->show();
```

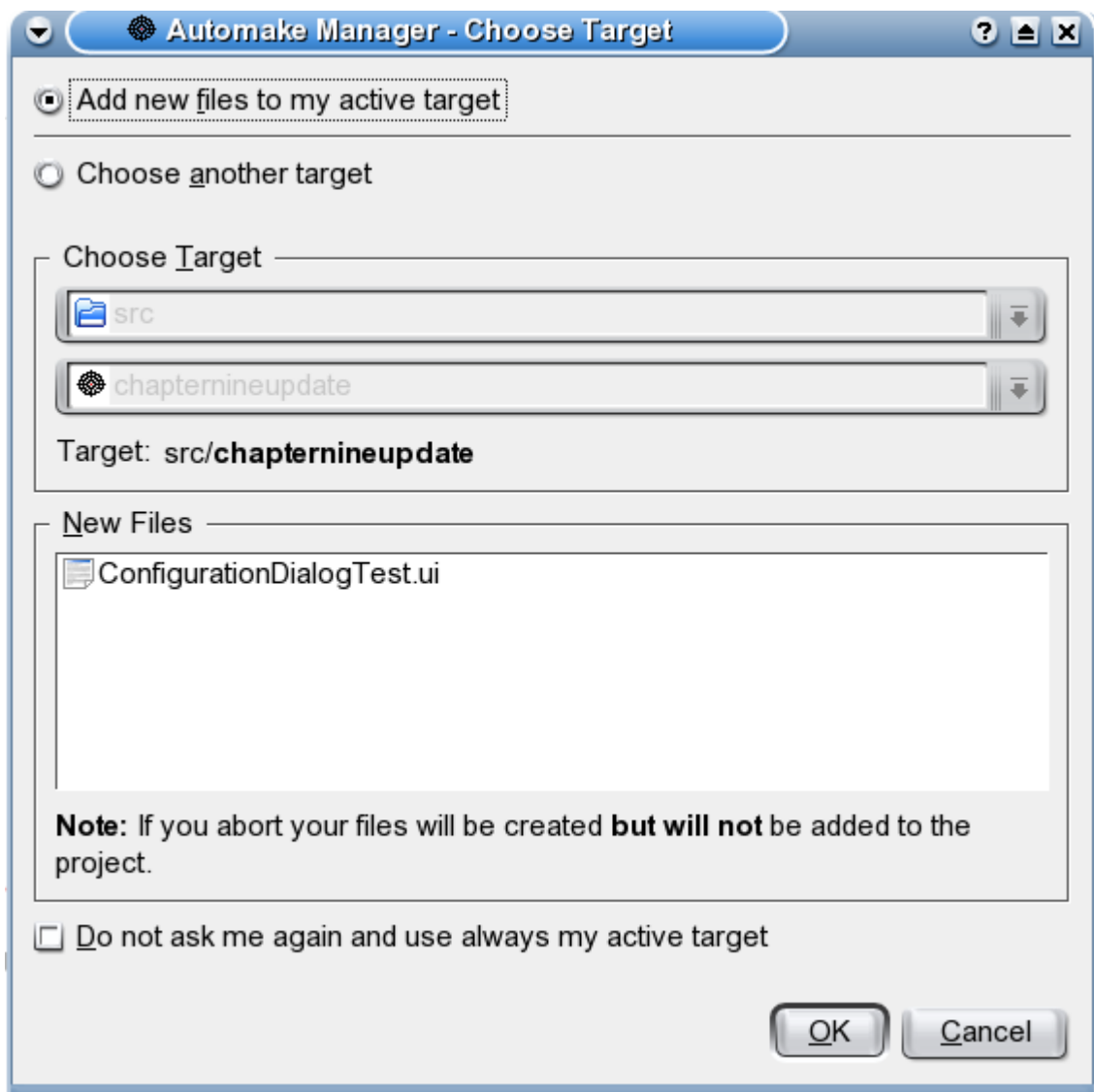
which shows the call for the configuration dialog.



When adding these widgets to your code you will start off with the New File Dialog that requests the name for your widget



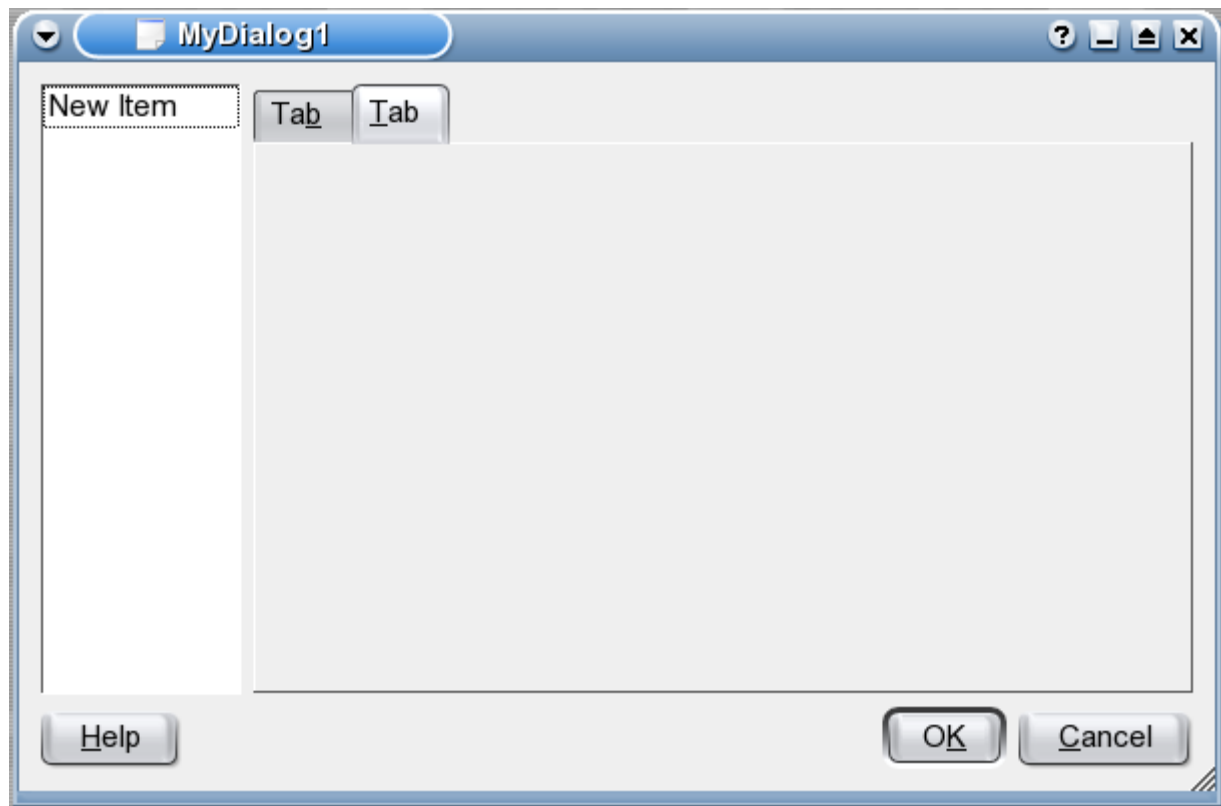
You will then be given another dialog that asks what project you would like the files to be placed in.



If you accept the default then the files will be added directly to your current project.

Chapter 9 KDE Containers and Views

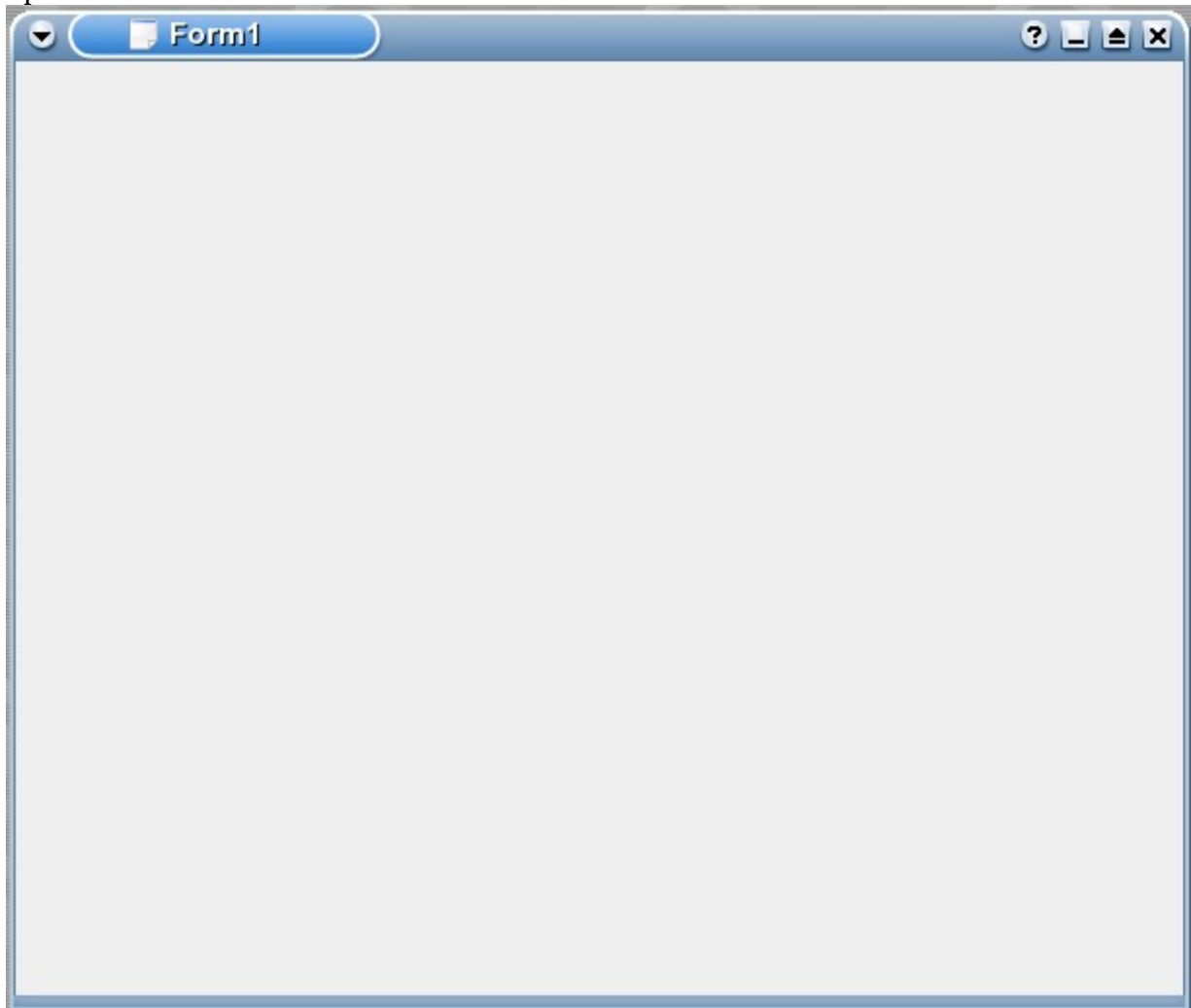
The first item in the above list is the configuration dialog that running as it comes looks like,



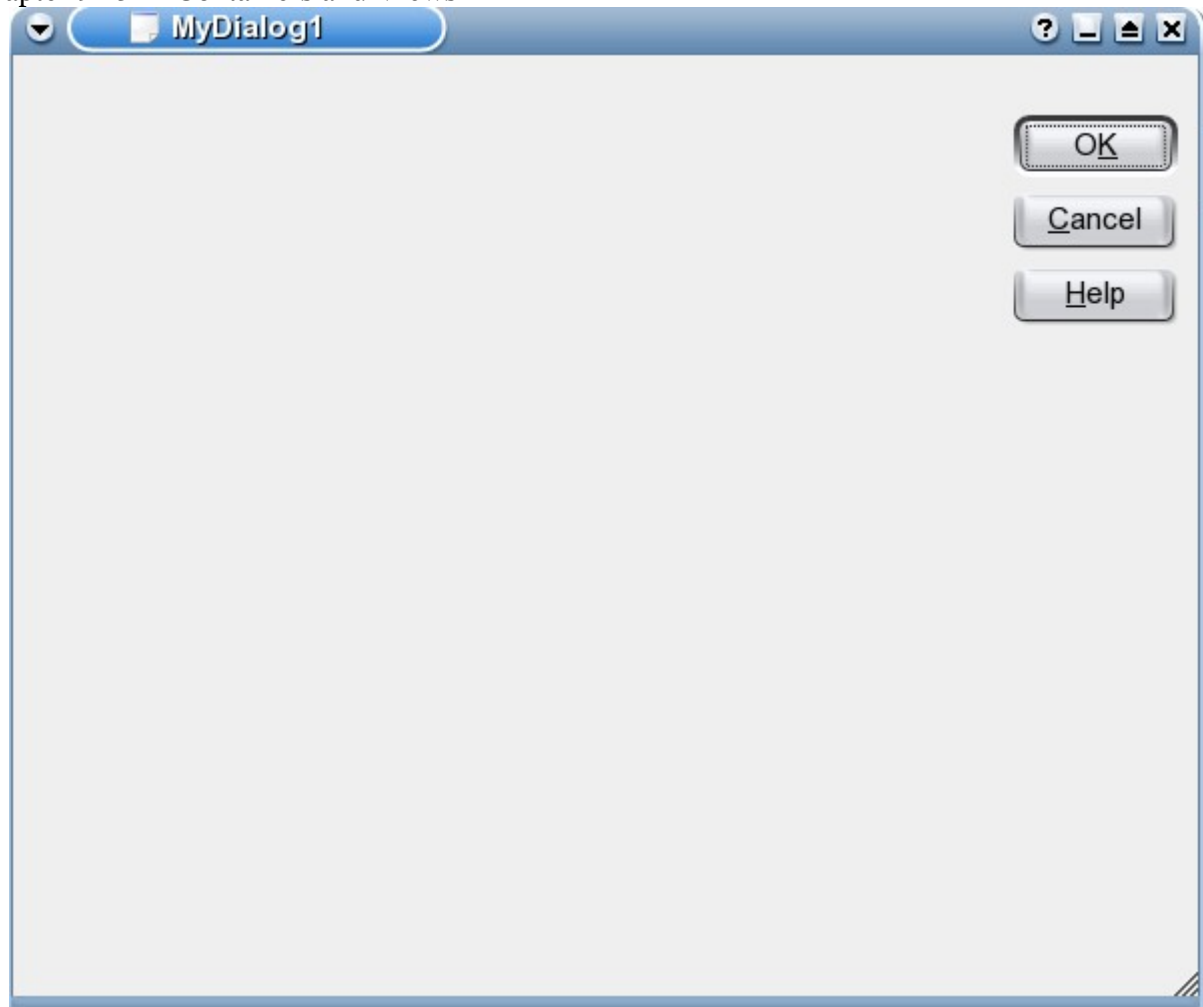
As the name suggests this dialog is a template that you can use to adjust any settings for your project.

The second button starts the empty dialog which is basically the form that we have been using for the starting point for our projects.

Chapter 9 KDE Containers and Views

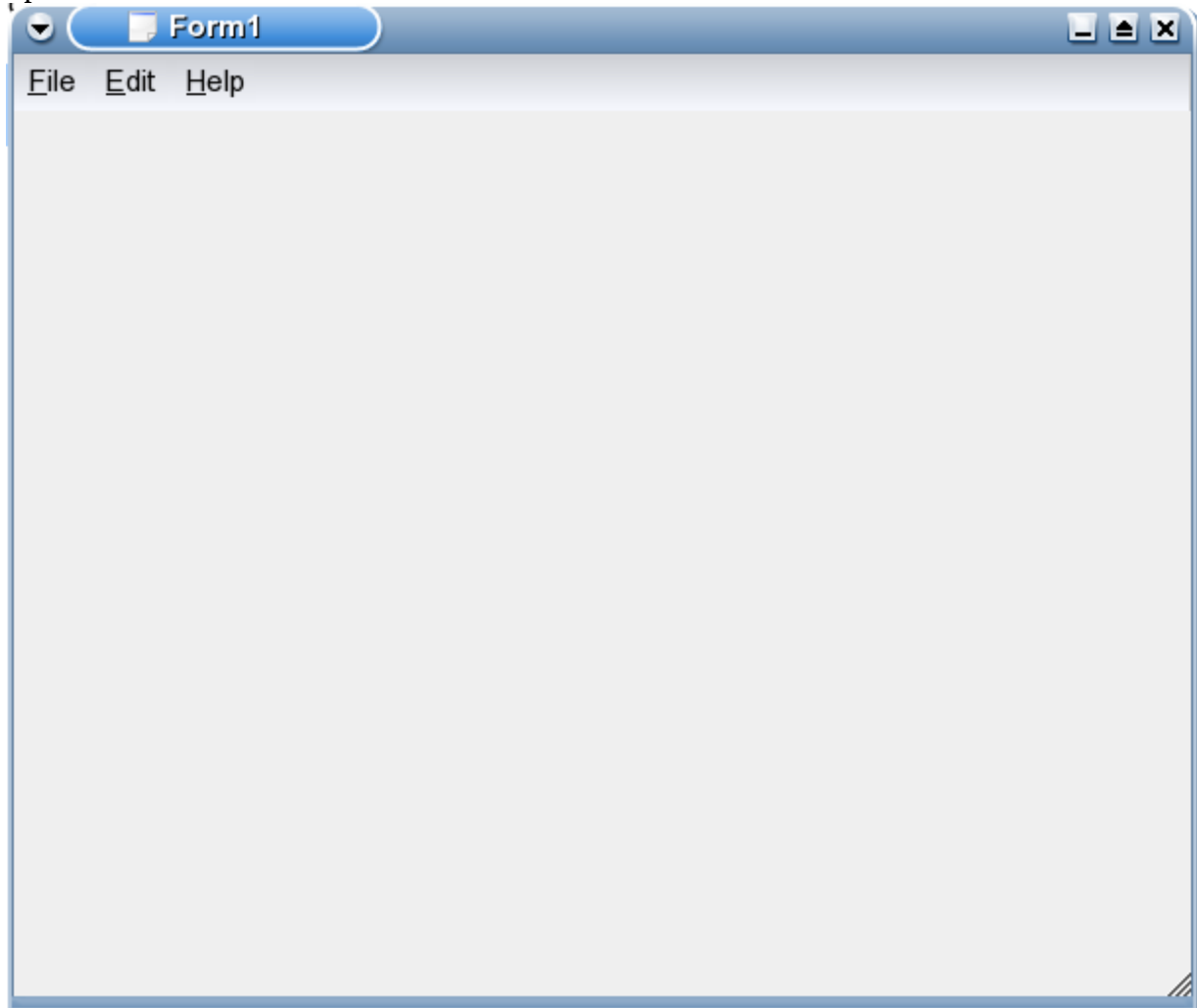


For some reason the next dialog the buttons bottom dialog doesn't show but it's not like it would be any great difficulty to create this yourself as you can see from it's companion dialog the buttons right dialog



The next form is the Main Window form which is interesting from the point of view that you might expect it to be the default form for the type of application that we have been looking at so far. It is basically the same form only with menus added.

Chapter 9 KDE Containers and Views



So if you want a form with menus create the project as normal then add one of these and call this form first. This would be done in the main.cpp file where you would replace the definition of `mainWin`

```
ChapterNineContainers *mainWin = 0;

if (app.isRestored())
{
    RESTORE(ChapterNineContainers);
}
else
{
    // no session.. just start up normally
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    /// @todo do something with the command line args here

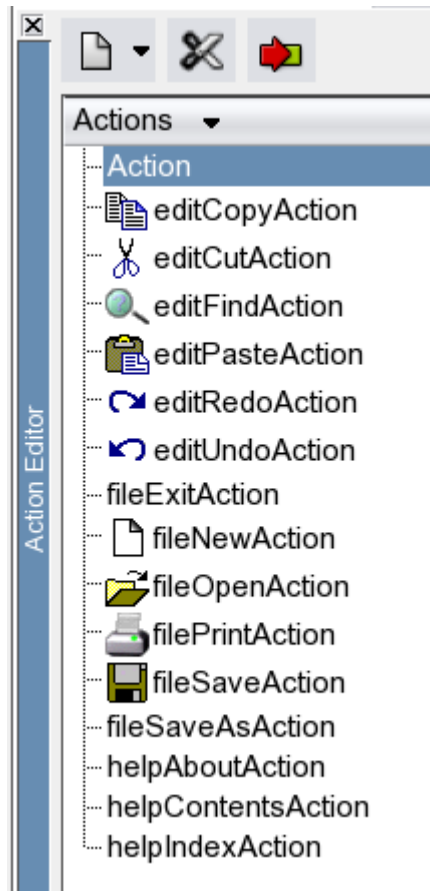
    mainWin = new ChapterNineContainers();
    app.setMainWidget( mainWin );
    mainWin->show();

    args->clear();
}
```

So for example if you created a project and added a new MainWindow called for arguments sake `MyNewMainWindow`, you would just replace the occurrences of `ChapterNineContainers` with `MyNewMainWindow`.

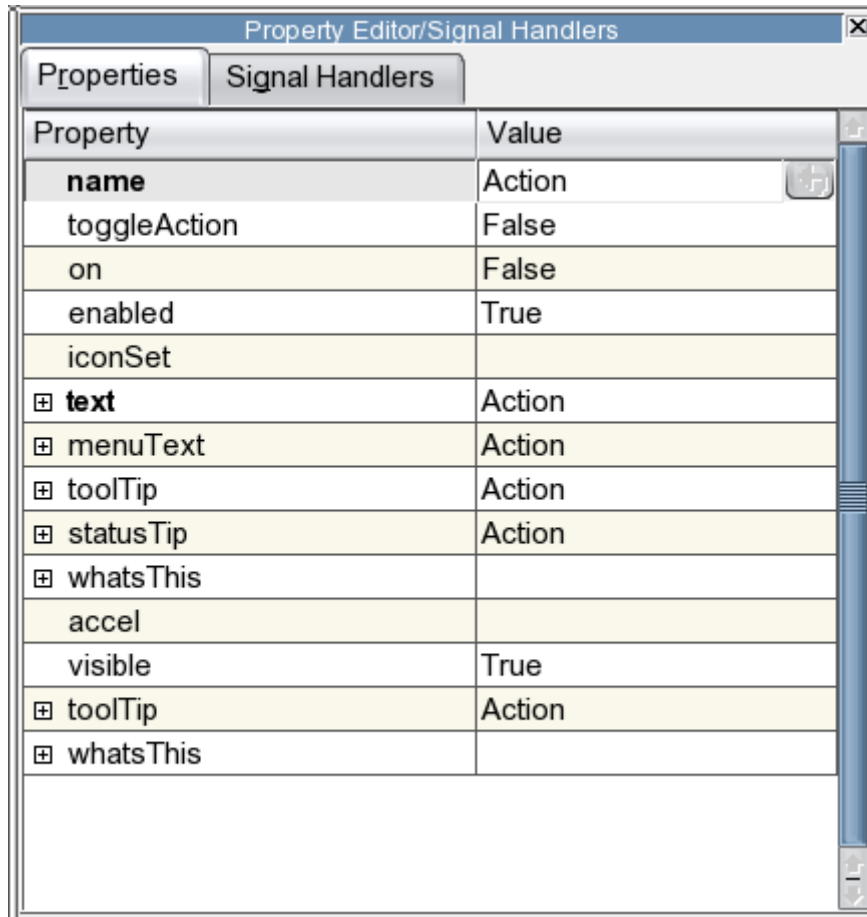
Menu Items

If you open the .ui file after adding it to your project you will be met with the Action Editor.

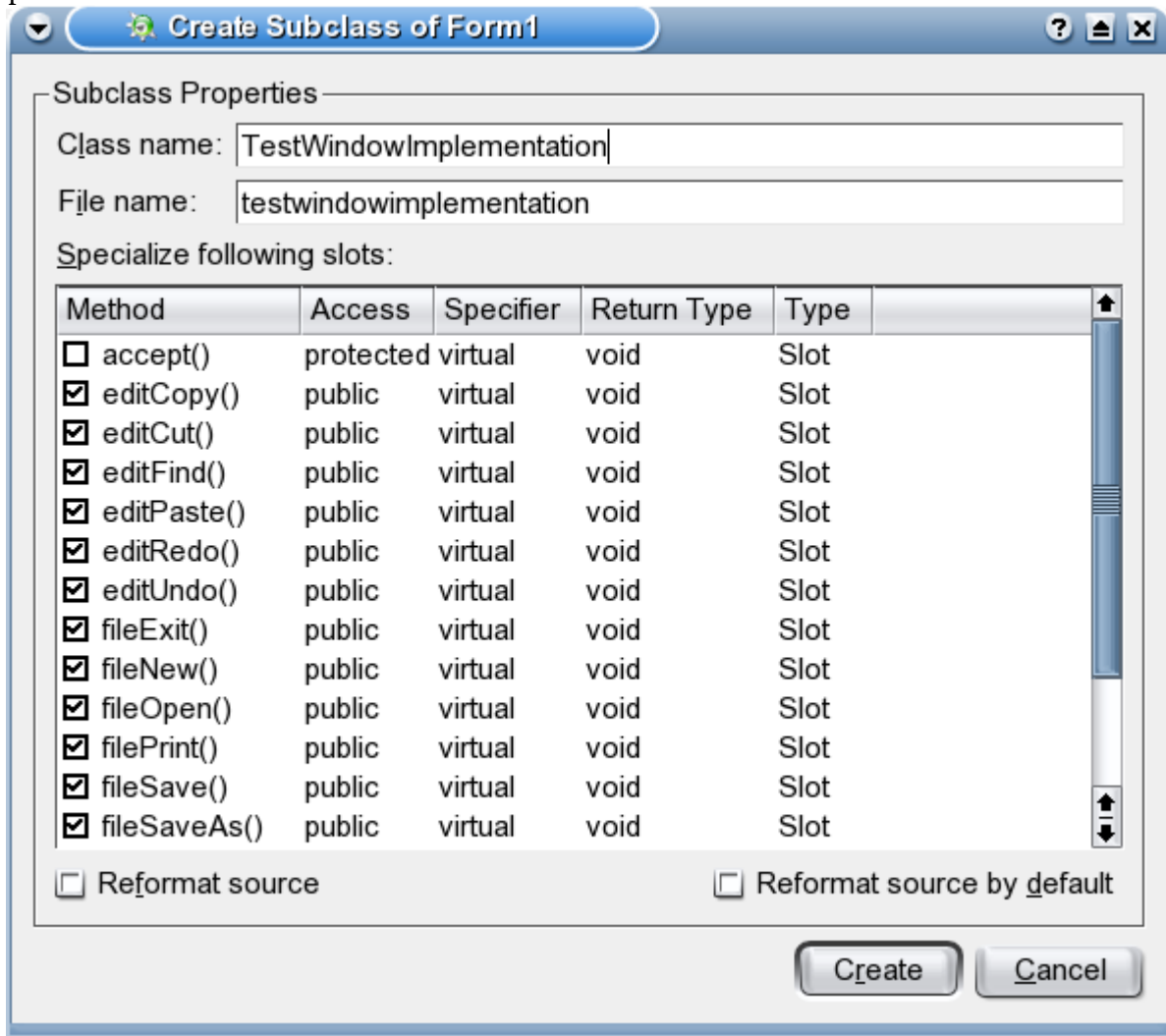


As you can see this is the list of the standard menu items that you would expect to find on any main window. The menu items are objects of the QAction class which is a class designed to make the implementation of menu items for menus and toolbars easier by grouping the required functionality together. You select a new action by selecting the option from the button on the left of the toolbar or by right clicking on the editor itself. When you have added a new action you select the properties and edit it as you would any other form object.

Chapter 9 KDE Containers and Views



In the case of the Main Window widget you should always subclass it as you want the signals for the messages,



As you can see the subclass wizard gives you the option of which signals to handle in the derived class. Which will then add the selected ones to you header file,

```
virtual void    helpAbout();
virtual void    helpContents();
virtual void    helpIndex();
virtual void    editFind();
virtual void    editPaste();
virtual void    editCopy();
virtual void    editCut();
virtual void    editRedo();
virtual void    editUndo();
virtual void    fileExit();
virtual void    filePrint();
virtual void    fileSaveAs();
virtual void    fileSave();
virtual void    fileOpen();
virtual void    fileNew();
```

with the corresponding functions defined in the .cpp file. You will of course be required to fill these in yourself but you can't have everything.

One word of warning is that in the subclass dialog above you can see at the top that I am subclassing form1 which is what all newly created forms are called. This is a big mistake as you then have to manually track down the references to form one and change them to the base class in

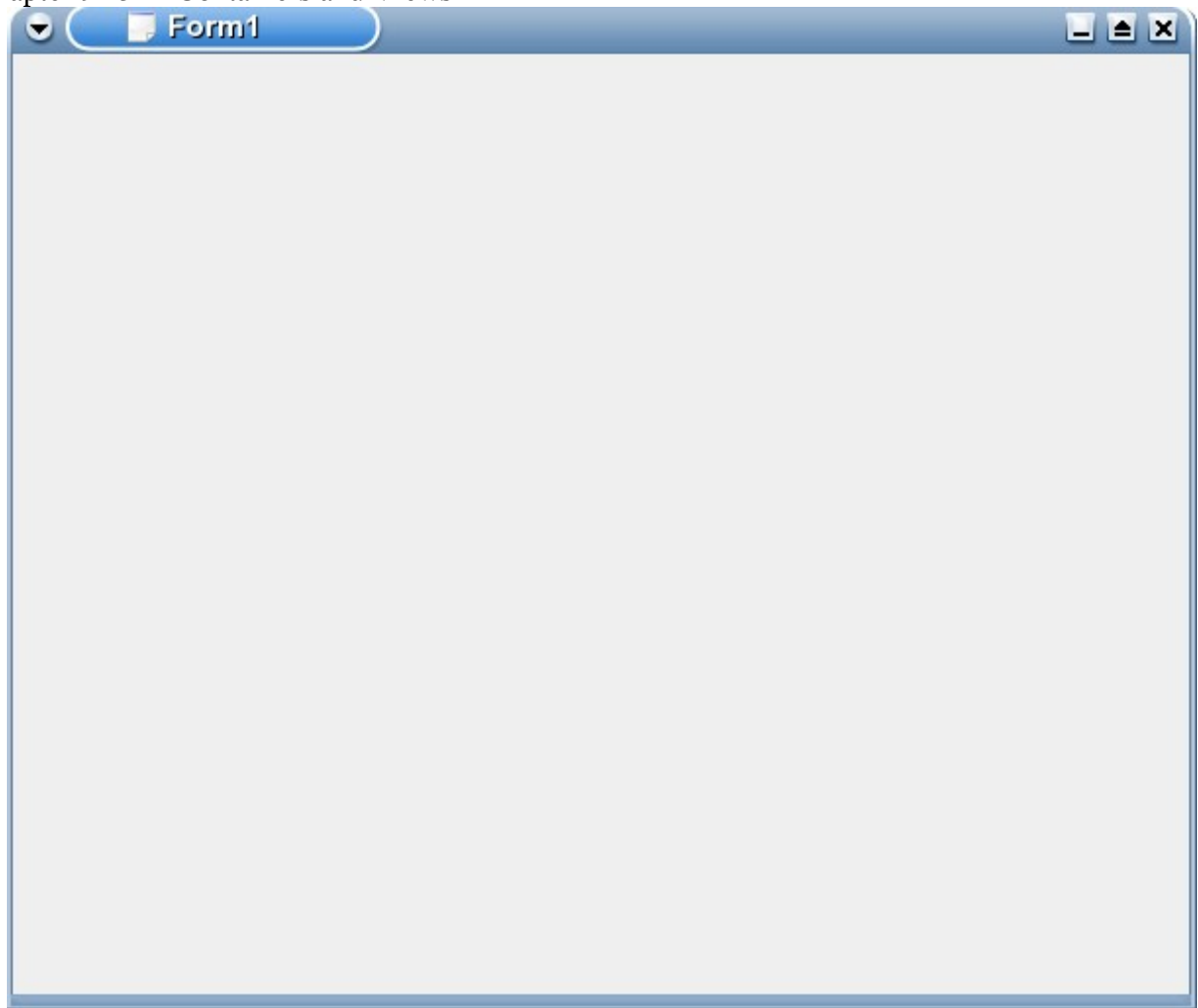
Chapter 9 KDE Containers and Views
this case TestWindow.

The tab dialog not surprisingly is a dialog with tabs,

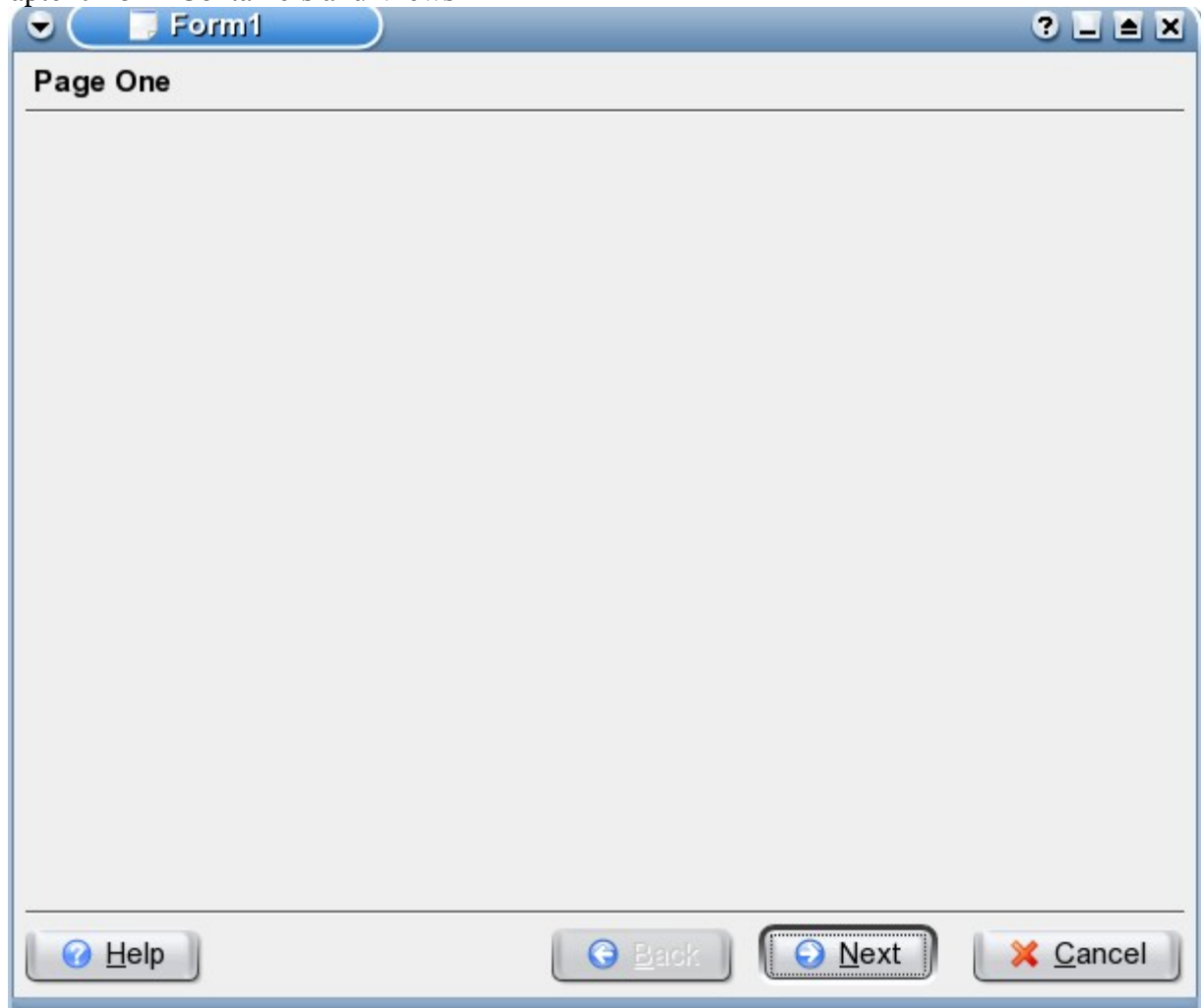


while the widget is just another form the same as the empty dialog

Chapter 9 KDE Containers and Views



The wizard is probably the most useful of the bunch, I've added some pages and edited the page names so you can see that it works,



Summary

That wraps up our tour of the basic widgets available in KDE. Hopefully I've achieved my goal of showing how to use almost all of the available widgets, with one or two exceptions. The more observant will have noticed that I've missed out the graphics section. The reason for this is that most of the widgets in the Graphics section, such as the KHSSelector for example are best used in the KColorDialog and their user interfaces aren't designed for use outside of the KDE main code. If you want to play around with these to see they all work from the interface.

Chapter 10 Custom Widgets

At some point or other when developing software you find that the provided tools don't quite do what you would like them to do. Personally I liked the look of the KKeyButton but wasn't over impressed with it's built in functionality wanting instead a button that looked like that but behaved differently. So the only option was to rewrite the existing button and create the KSquareButton class.

Building Your Own Widgets

One thing about open source is that by using the help you can see the source code for any KDE class that you want to copy/change the behaviour of so using the KKeyButton as a template I wrote the KSquareButton. In this case the main task was not implement more functionality to the class but to remove the shortcut key code from the widget.

So where the KKeyButton declares a lot of functions that manage the shortcut our class header is a lot simpler

```
#include <qpushbutton.h>

class KSquareButton : public QPushButton
{
    Q_OBJECT

public:
    KSquareButton( QWidget *parent, const char *name );
    virtual ~KSquareButton();

    void setText( const QString &text );
    void drawButton( QPainter *painter );

};
```

The main things that we are concerned with in this class is to set the text on the button and to draw the button itself. The setText function is simplicity itself in that all we do is,

```
QPushButton::setText( text );
setFixedSize( sizeHint().width() + 12, sizeHint().height() + 8 );
```

use the parent version of setText to set the actual text and then call setFixedSize which resizes the button so that the text fits within the button space, if you wanted a fixed size button you would remove this line but would then be running the risk of having them look stupid because longer text strings would not display properly.

The paint function takes a bit more thought though, as with any widget we build up the image in layers so we start with,

```
QPointArray pArray( 4 );
pArray.setPoint( 0, 0, 0 );
pArray.setPoint( 1, width(), 0 );
pArray.setPoint( 2, 0, height() );
pArray.setPoint( 3, 0, 0 );

/// draw the top and the left light border
///

QRegion regionOne( pArray );
```

Chapter 10 Custom Widgets

```
painter->setClipRegion( regionOne );
painter->setBrush( backgroundColor().light() );
painter->drawRoundRect( 0, 0, width(), height(), 20, 20 );
```

which defines a point array that draws the first region, this region is a diagonal region from the top left to the top right and down to the bottom left of the button. The colour white in the picture is set by setting the brush to the `backgroundColor().light()`. Note that we don't use an exact colour here as the colours are set by the system.



Notice that although in the call to `drawRoundRect` we passed in the parameters for the entire button size the area drawn is only the region specified by the point array and passed to the `setClipRegion` function.

Next we draw the bottom right triangle which is shaded blue grey in the above picture,

```
pArray.setPoint( 0, width(), height() );
pArray.setPoint( 1, width(), 0 );
pArray.setPoint( 2, 0, height() );
pArray.setPoint( 3, width(), height() );

/// draw the bottom right triangle dark
///

QRegion regionTwo( pArray );
painter->setClipRegion( regionTwo );
painter->setBrush( backgroundColor().dark() );
painter->drawRoundRect( 0, 0, width(), height(), 20, 20 );
```

The only difference between this code and the above code is the definition for the region that we are drawing and the setting of the brush to `backgroundColor().dark()`, which gives us,



This completes the background for the button, now all we need to do is draw the button, we start by turning off the clipping as we actually want to draw over what we have drawn so far.

```
painter->setClipping( false );
```

and then draw the button surface with,

```
painter->setPen( backgroundColor().dark() );
painter->setBrush( colorGroup().button() );

if( width() > 14 && height() > 10 )
    painter->drawRect( 7, 5, width() - 14, height() - 10 );
```

which will draw a standard button coloured panel with a dark border.



Chapter 10 Custom Widgets

which is pretty much the button image we want. We just need to add a few finishing details. First we draw the button label

```
drawButtonLabel( painter );
```

which is the QPushButton function that draws whatever is set to appear on the button be a picture or text. Then we draw the little rectangle that you see appear when you click on the button.

```
painter->setPen( colorGroup().text() );
painter->setBrush( NoBrush );

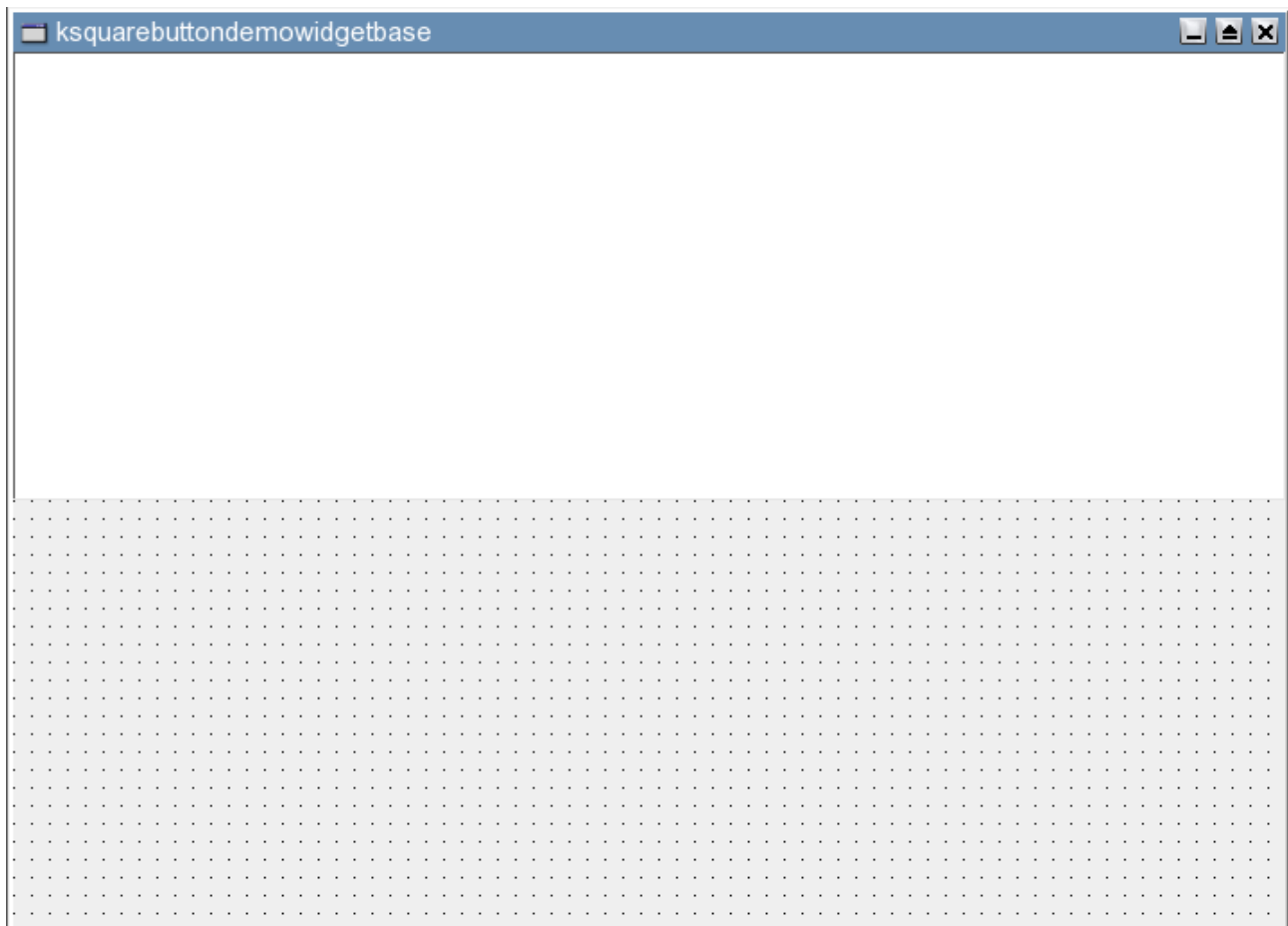
if( hasFocus() == true )
{
    if( width() > 16 & height() > 12 )
        painter->drawRect( 18, 12, width() - 16, height() - 12 );
}
```

giving us,



So all we need to do now is test it.

Testing The Widget



Chapter 10 Custom Widgets

We'll start with a simple application that has a KTextEdit widget and then add some buttons with the code. As the inspiration for this class came from the KKeyButton it's only fitting that we do a keyboard,

```
KSquareButton *kSquareButtonQ;  
KSquareButton *kSquareButtonW;  
KSquareButton *kSquareButtonE;  
KSquareButton *kSquareButtonR;
```

declaring the buttons in the header file for the widget and adding the definitions for slots for when the button is clicked.

```
void kSquareButtonQ_clicked();  
void kSquareButtonW_clicked();  
void kSquareButtonE_clicked();  
void kSquareButtonR_clicked();  
void kSquareButtonT_clicked();
```

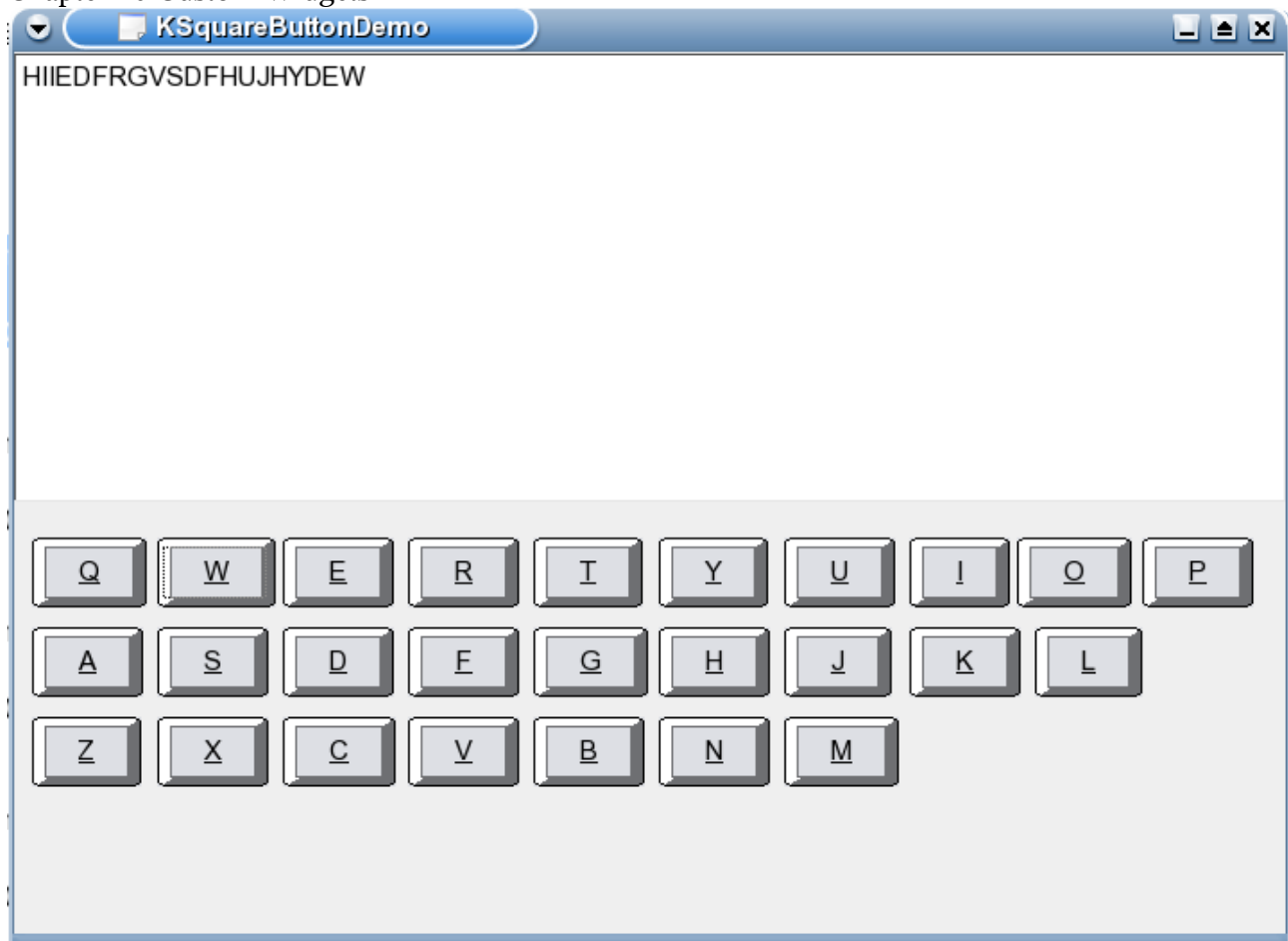
Now we need to create the widgets and set up the connections, in the constructor for the KSquareButtonDemoWidget class

```
kSquareButtonQ = new KSquareButton( this, "KSquareButtonQ" );  
kSquareButtonQ->setGeometry( QRect( 10, 270, 64, 39 ) );  
kSquareButtonQ->setText( "Q" );  
  
connect( kSquareButtonQ, SIGNAL( clicked() ), this, SLOT( kSquareButtonQ_clicked() ) );
```

Each button is created and setGeometry is called to place the button on the widget. I should point out that coding the QRect's by hand at this point would be a total pain so I set up a test project and placed some KKeyButtons on the form and then compiled the code and took the QRect values from the generated widget base cpp file. The text for the button is then set with a call to setText and finally a connection is made. All the button does when clicked is add the letter to the KEditBox with,

```
void KSquareButtonDemoWidget::kSquareButtonQ_clicked()  
{  
    textEdit->insert( "Q" );  
}
```

and the running application is,



Tip Of The Day

it is important to note that if you set up a slot connection incorrectly the compiler will not tell you anything about it. What you will get is a message when you run your application.

```
./ksquarebuttondemo
QObject::connect: No such slot KSquareButtonDemoWidget::kSquareButtonM_clicked()
QObject::connect: (sender name: 'kSquareButtonM')
QObject::connect: (receiver name: 'ksquarebuttondemowidgetbase')
```

This example shows that the `kSquareButtonM` is not set up properly and that if we click it then we will not receive the signal. the message tells us that there is something wrong with the `kSquareButtonM` connection so let's take a look at it.

```
void kSquareButtonM_Clicked();
```

The error is that the slot is defined as `Clicked` with a capital `c` and the connection is defined as

```
connect( kSquareButtonM, SIGNAL( clicked() ), this, SLOT( kSquareButtonM_clicked() ) );
```

which is what it should be and is in keeping with all the other definitions so we need to change the header and the implementation. and once this is done the progem runs without any error messages.

Chapter 10 Custom Widgets

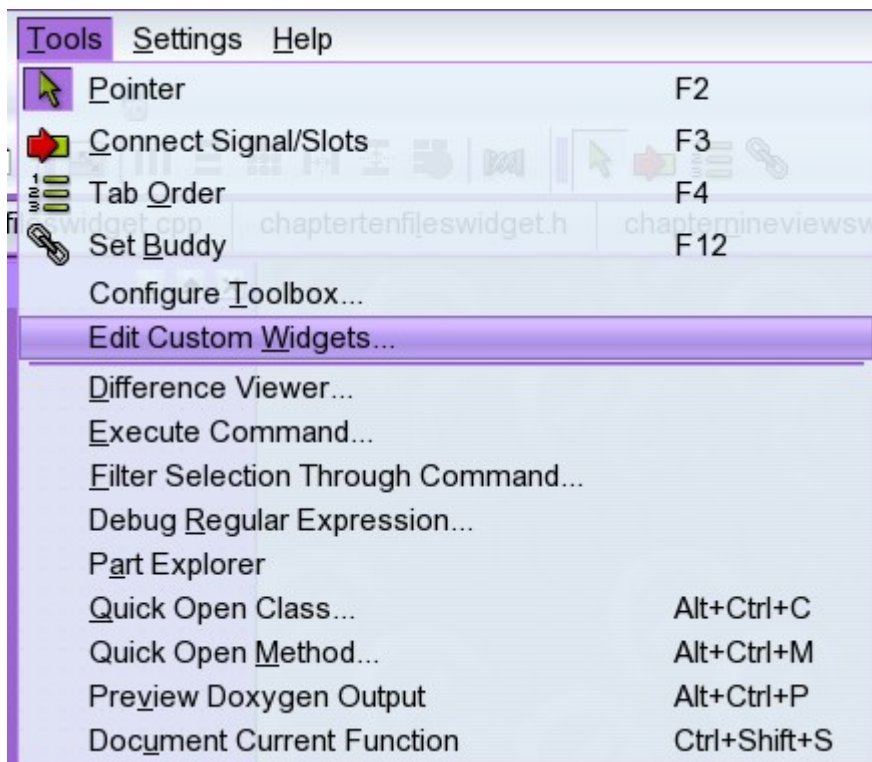
Of course now that we've tested the new button there is one thing that is immediately apparent and that is that the demo widget is rubbish so we shall have to see if we can turn it into a proper functional widget.

Adding A Custom Widget

Initially the Custom Widgets panel is empty when we start KDevelop,



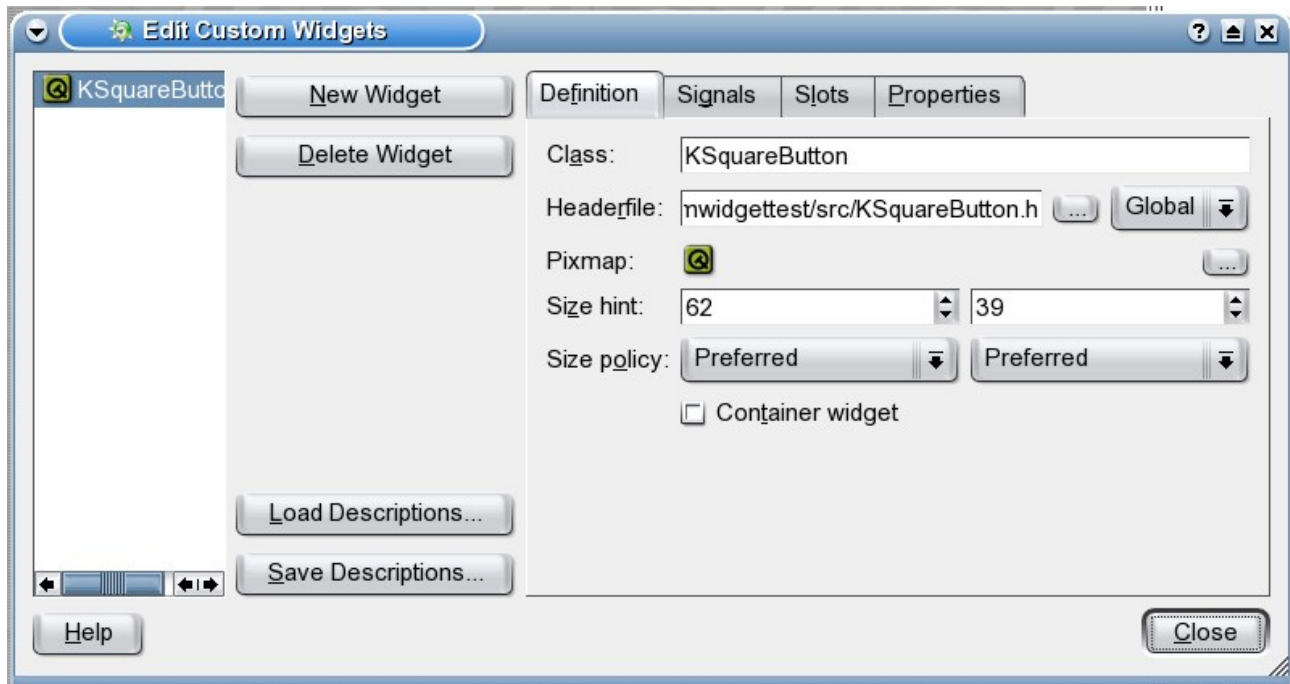
and it will stay that way unless you add something yourself, so let's see if we can turn the KSquareButton into a custom widget to start with. We start at Tools\Edit Custom Widgets in KDevelop,



which gives us this dialog,

Chapter 10 Custom Widgets

Defining The Widget



As you can see we start by defining the widget, the class we are using is the KSquareButton class and we point the file to the header file for the class. The pixmap option allows us to set an image for the widget that will be displayed in the Custom Widgets section of KDevelop. You can see the default to the left of the above image. For the size hints we add the size that we want the button to be, these figures are taken from the generated file for the KKeyButton I mentioned earlier. There is also a checkbox to select if the widget is going to be used as a Container Widget this means will you be adding widgets to your widget after it is created, if the widget is a completely self contained object we can leave this unchecked.

The other important button this screen is the save descriptions button which saves the information about your widget. This is the file that KDevelop or rather the Qt Designer embedded in KDevelop will read to add your widget to the custom widgets panel. You should remember where you save this as the custom widget will not be loaded automatically into the development environment for projects that you haven't specified it for, although adding it once you have it setup is simply a matter of loading the saved file and adding the class files to your project.

Defining the Signals

Chapter 10 Custom Widgets



Here you can see that the slot for clicked is defined. Even though this is a standard signal from our QPushButton base class I've added it here to show how it's done as this will have no impact on the widget whatsoever.

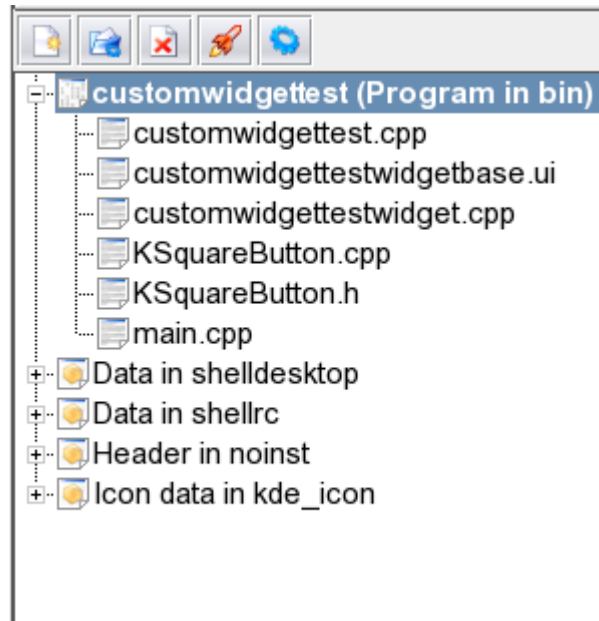
Technically there is room for confusion here if you get into a situation where you are defining every signal and slot that your widgets handles then you should rethink it and follow two simple rules. Basically only define a Signal if the custom widget emits it and it will be handled by an outside class. And only define a slot if the signal for the slot is emitted from a widget that is not a part of your custom widget.

The handling of the emitted signal for clicked is as you would expect,

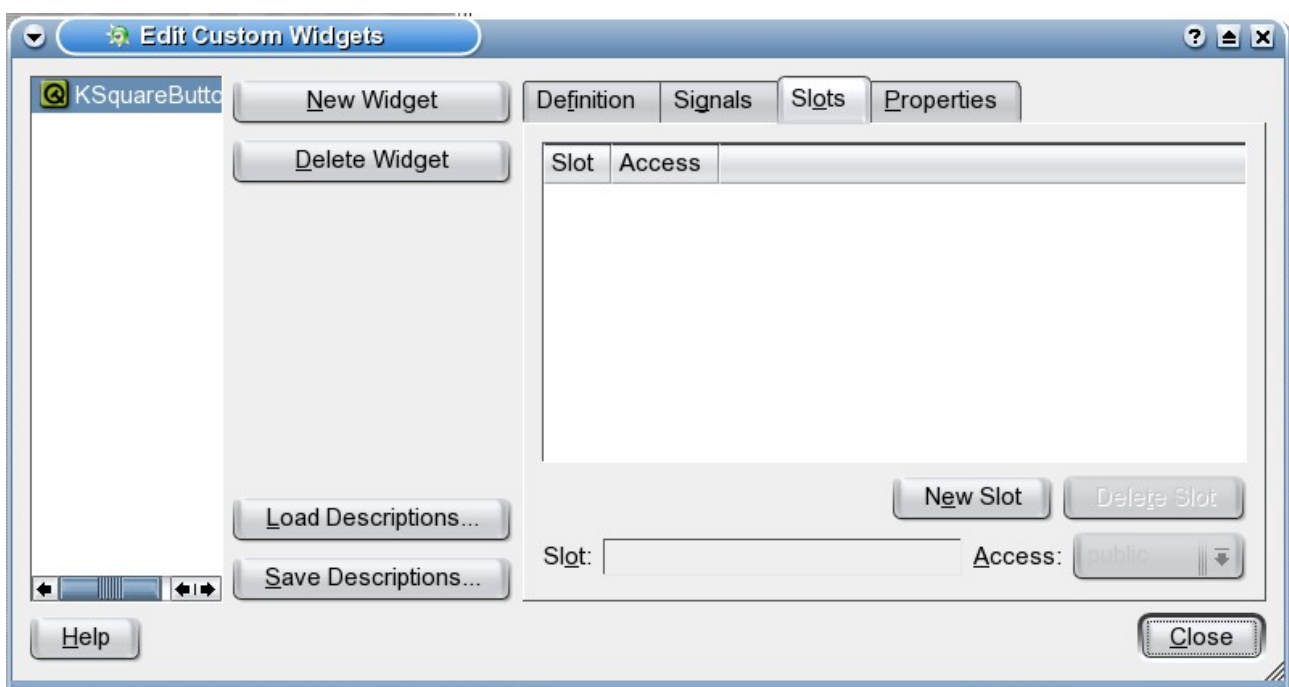
```
connect( kSquareButtonTest, SIGNAL( clicked() ), this, SLOT( kSquareButtonTest_clicked() ) );
```

Is used in the constructor of the .cpp file and I should mention that you need to add the header for the widget to the .cpp file or the compiler will say that the call to connect is invalid. In fact you need to add all the implementation and header files for the widget to the project in which you intend to use it.

Chapter 10 Custom Widgets



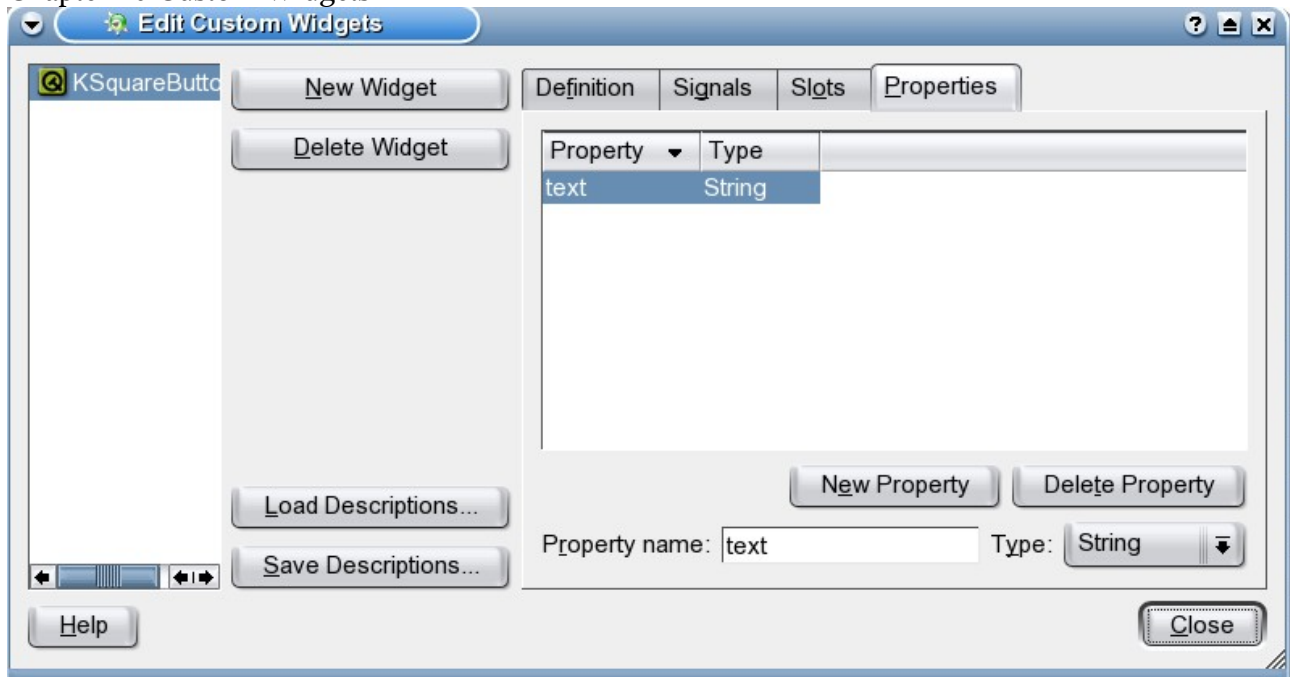
Defining The Slots



There are no slots implemented for this widget but as you can imagine you would implement a slot in exactly the same way that you normally would in any widget class.

Defining The Properties

Chapter 10 Custom Widgets



To set up a property in a widget you would use the standard methods for implementing your property so the text that will be placed on the button we would get and set the properties with the code,

```
void setText( const QString &text );
QString text();
```

With the implementations being,

```
void KSquareButton::setText( const QString &text )
{
    QPushButton::setText( text );
    setFixedSize( sizeHint().width() + 12, sizeHint().height() + 8 );
}

QString KSquareButton::text()
{
    return QPushButton::text();
}
```

We can now set and get our property as normal. The thing is we are using a Gui environment and Qt provides us with a mechanism where we can get the property in question to show up in the Gui. We do this using the `Q_PROPERTY` macro which takes the form `Q_PROPERTY(type property_name read_function read_function_name write_function write_function_name)` and is written into the header file as,

```
Q_PROPERTY( QString text READ text WRITE setText )
```

So when we look at the KSquareButton in the designer we get,

Chapter 10 Custom Widgets

| Property Editor/Signal Handlers | |
|---------------------------------|------------------|
| Properties | Signal Handlers |
| Property | Value |
| minimumSize | [0, 0] |
| maximumSize | [32767, 32767] |
| paletteForegroundColor | |
| paletteBackgroundColor | |
| paletteBackgroundPixmap | |
| palette | |
| backgroundOrigin | WidgetOrigin |
| font | Arial-12 |
| cursor | Arrow |
| mouseTracking | False |
| focusPolicy | NoFocus |
| acceptDrops | False |
| toolTip | |
| whatsThis | |
| text | a |
| comment | |

and if we run it,

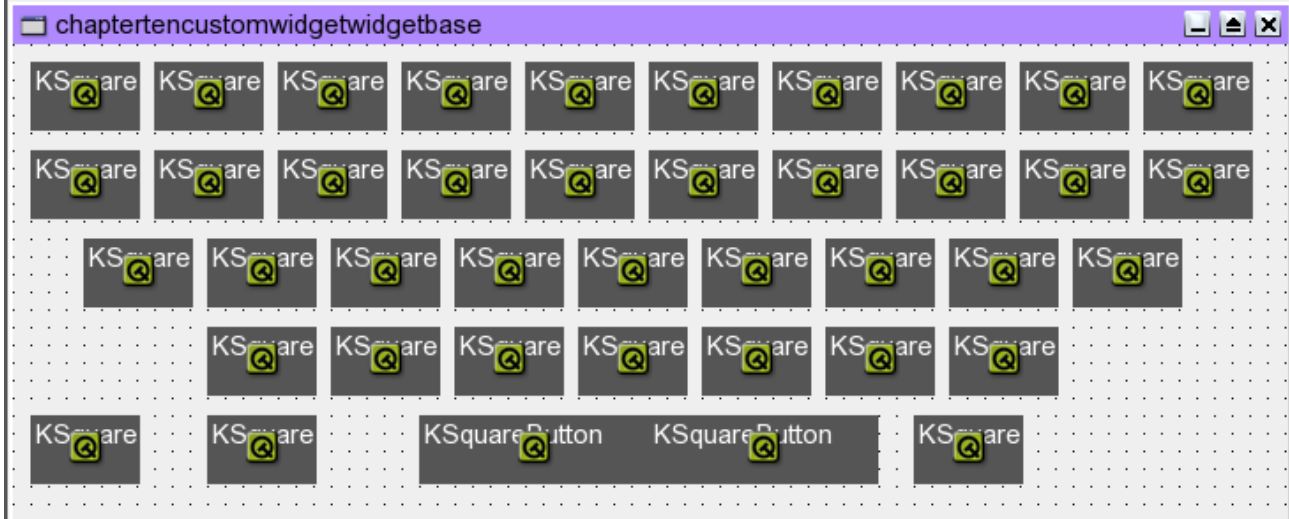


OK we now have a functional custom widget so let's see if we can use it in anger.

Create A Virtual Keyboard

To start off turning the KSquareButtons into a virtual keyboard we could if we had loads of time and patience sit and code everything by hand, or alternatively we could simply do the layout on a widget let the moc compiler generate the code and then copy it, like so,

Chapter 10 Custom Widgets



We then create a KVirtualKeyBoard class and copy the generated widget it code to it,

```
class KSquareButton;

class KVirtualKeyBoard : public QFrame
{
    Q_OBJECT
```

There are more than twenty declarations of KSquareButtons so I haven't included them here, if you really want to see them look at the header file, kvirtualkeyboard.h in the source code.

I've also added the functions.

```
inline void setCapsLocked( bool capsLocked ) { bCapsLocked = capsLocked; };
inline bool capsLocked() { return bCapsLocked; };
```

for setting the caps lock and the signal,

```
signals:
    void virtualKeyPressed( QString key );
```

which is the signal that is emitted by the widget when a button is pressed.

```
void KVirtualKeyBoard::kSquareButtonB_clicked()
{
    if( capsLocked() == true )
        emit virtualKeyPressed( i18n( "B" ) );
    else
        emit virtualKeyPressed( i18n( "b" ) );
}
```

Changes To KSquareButton

When using the KSquareButton there are a couple of refinements that needed to be made, firstly they aren't exactly square until we add

```
setFixedSize( 45, 45 );
```

To the constructor. Secondly the code,

```
void KSquareButton::setText( const QString &text )
{
```

Chapter 10 Custom Widgets

```
QPushButton::setText( text );
setFixedSize( sizeHint().width() + 12, sizeHint().height() + 8 );
}
```

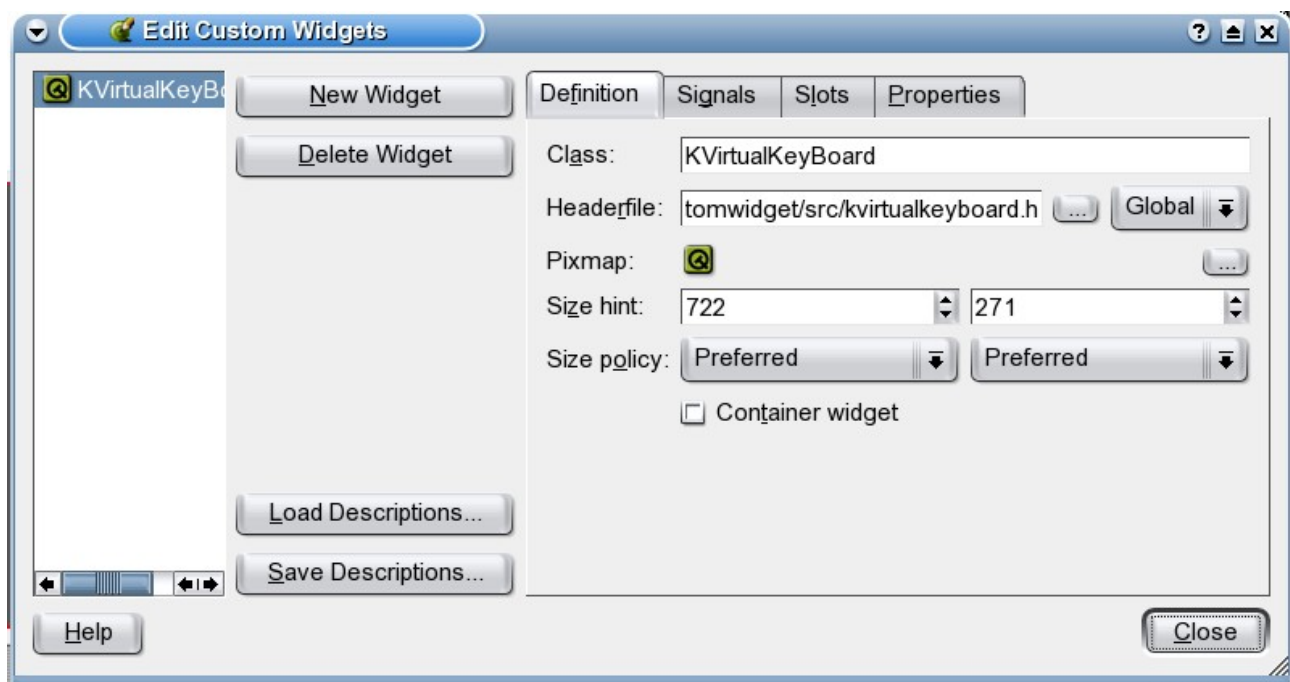
Causes the code to resize every time the text on the button is changed. This is ugly and not strictly necessary for example when the caps button is pressed all the buttons adjust their sizes, so the code was changed to.

```
void KSquareButton::setText( const QString &text )
{
    QPushButton::setText( text );
    if( text.length() > 3 )
        setFixedSize( sizeHint().width() + 12, sizeHint().height() + 8 );
}
```

which means that the buttons are only resized if there are more than three characters in the text string which means that the buttons won't resize unless it is strictly necessary.

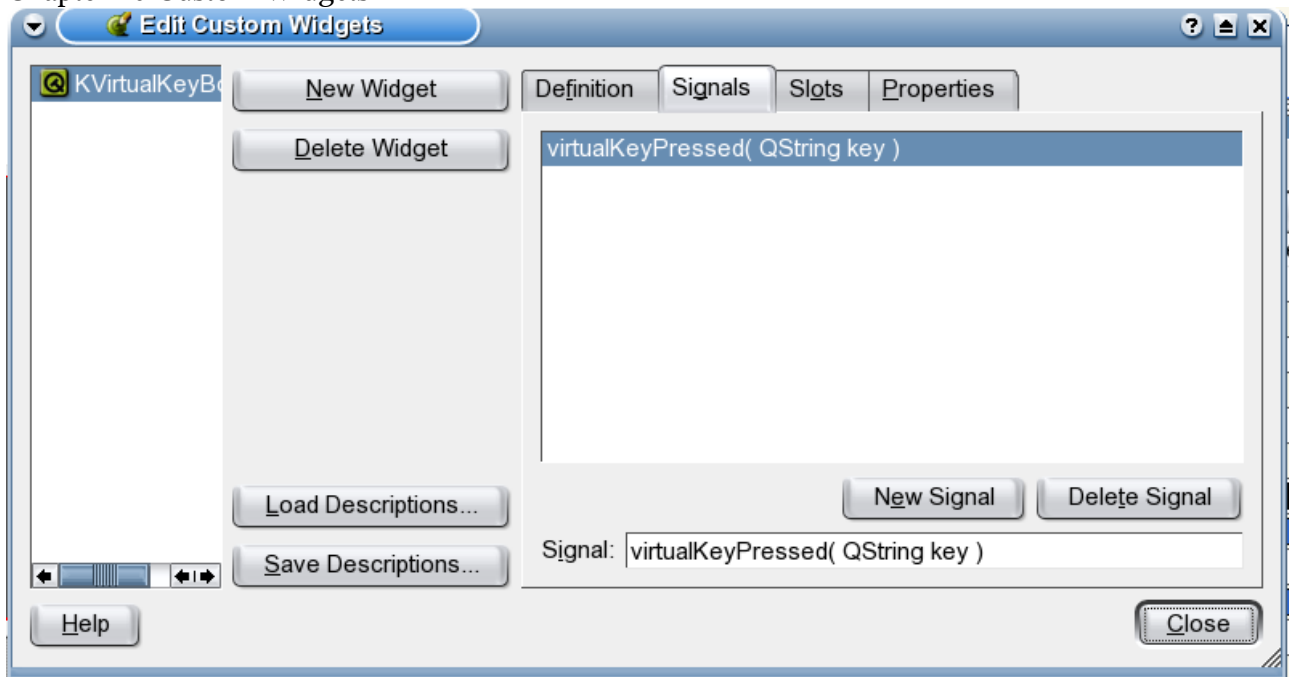
Creating the KVirtualKeyBoard Widget

The creation for the widget once we have the class is exactly the same as with the KSquareButton described earlier,

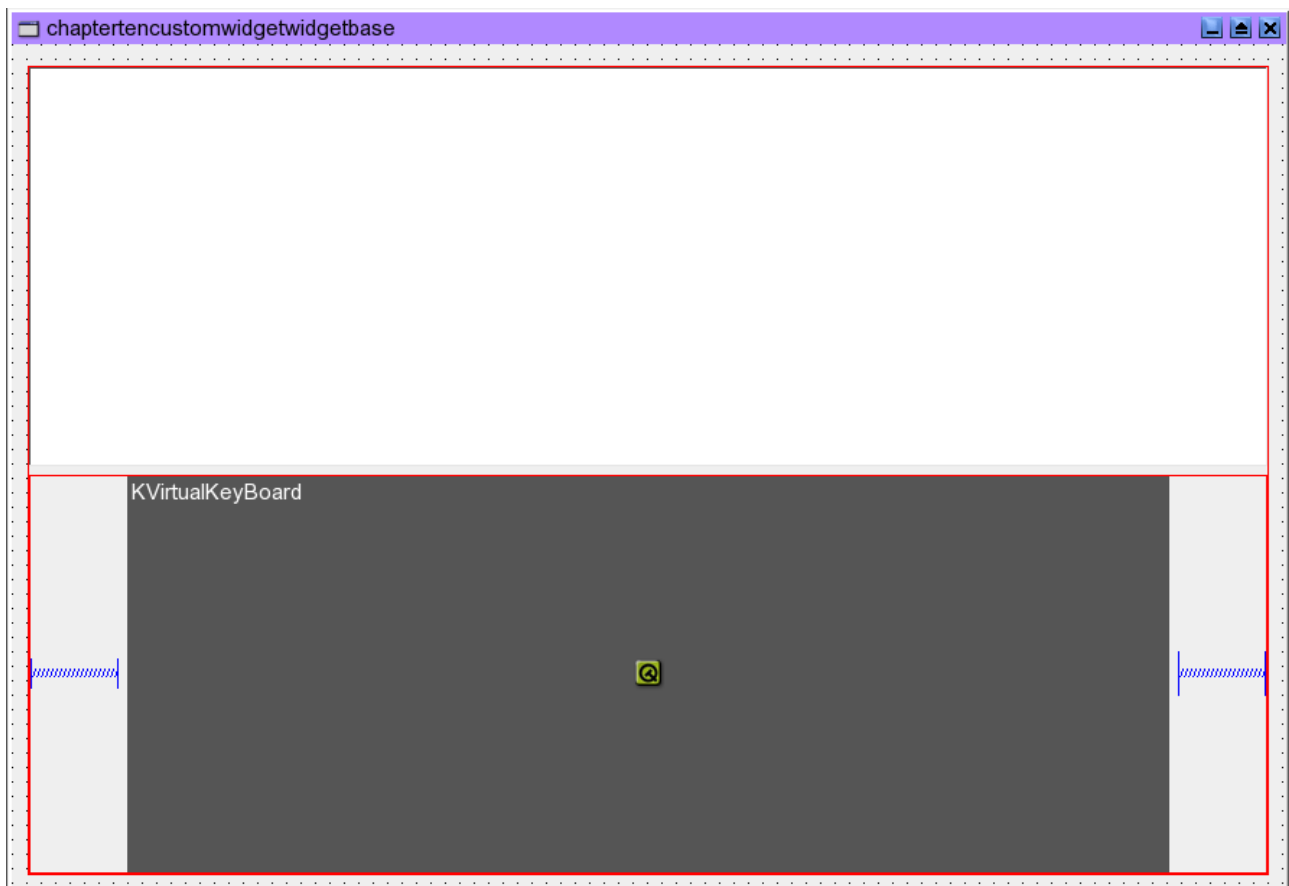


Of course as we are emitting a signal we have to define it.

Chapter 10 Custom Widgets



Now all we need to do is test it with,



We place the virtual keyboard widget on the form and add a text widget, it is then a simple matter to tie them together with the code,

Chapter 10 Custom Widgets

```
public slots:  
    /*$PUBLIC_SLOTS$*/  
    void virtualKeyPressed( QString key );
```

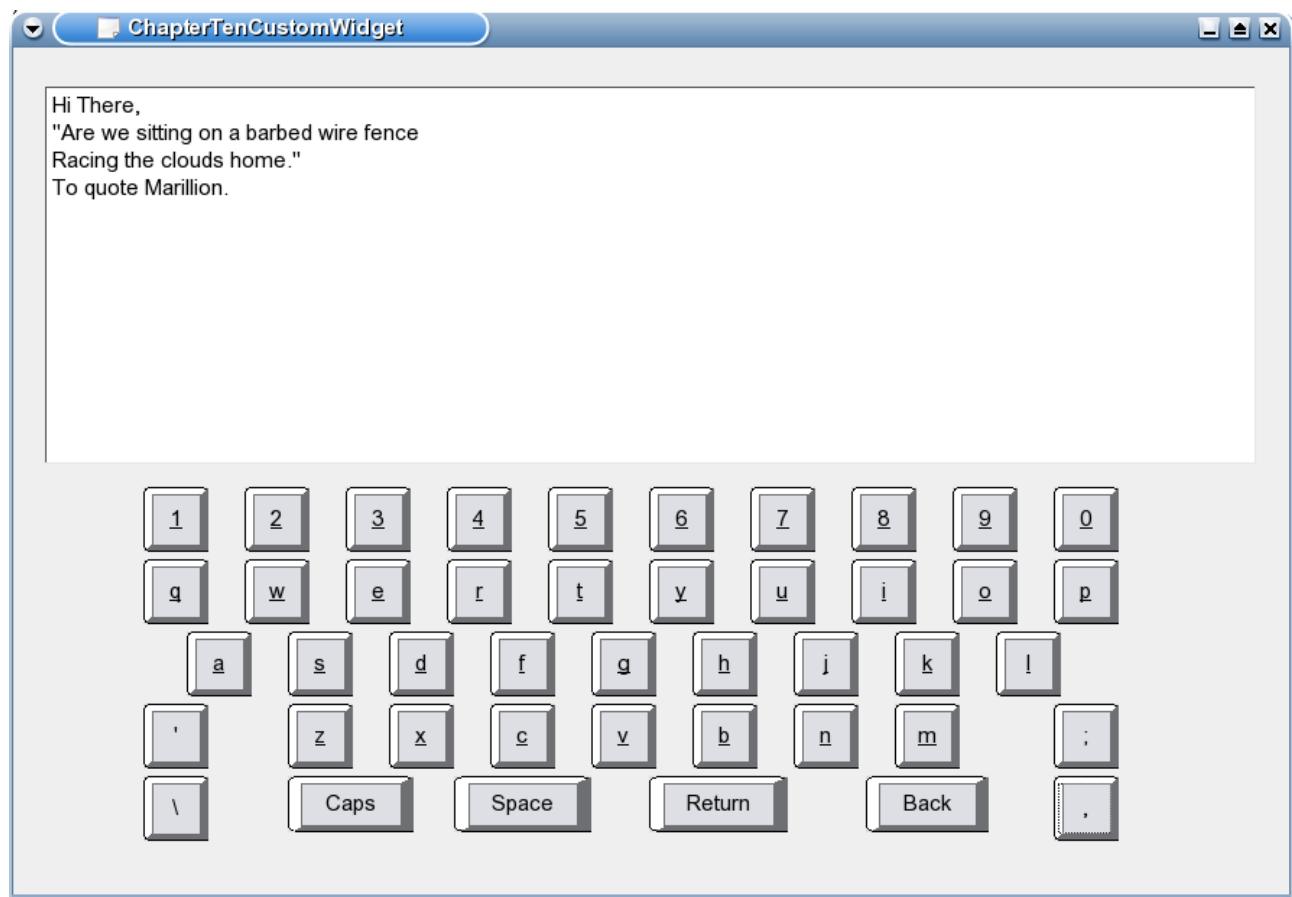
added to the header and the function,

```
void ChapterTenCustomWidgetWidget::virtualKeyPressed( QString key )  
{  
    if( key != 0 && key == "BS" )  
    {  
        textEdit->moveCursor( QTextEdit::MoveBackward, false );  
        textEdit->del();  
    }  
    else  
    {  
        textEdit->insert( key );  
        textEdit->moveCursor( QTextEdit::MoveForward, false );  
    }  
}
```

which if the Back button is pressed the code moves one space back and calls delete which deletes the character directly in front of it, leaving the cursor in the correct position for the next character. Then there's the connection,

```
connect( kVirtualKeyBoard1, SIGNAL( virtualKeyPressed( QString ) ), this,  
        SLOT( virtualKeyPressed( QString ) ) );
```

in the constructor. Which gives us,



Chapter 10 Custom Widgets

Summary

That should be enough to get you going with you're own custom widgets, of course this one still isn't perfect it doesn't have a question mark for a start and there's no way to move the cursor through the text.

Part Three

Getting Things Done

Chapter 11 Events

Events are generally the low level inputs from the computer the most common of which are generated from the computer hardware itself. If you remember from the chapter eight dates and times project we set up a timer and created a function that was triggered everytime the timer fired. Well in the background the timer triggered an event that called the function that we had specified through the connect function. If we look through the QWidget class we can see a whole list of events declared as protected virtual functions.

```
virtual bool event ( QEvent * e )
virtual void mousePressEvent ( QMouseEvent * e )
virtual void mouseReleaseEvent ( QMouseEvent * e )
virtual void mouseDoubleClickEvent ( QMouseEvent * e )
virtual void mouseMoveEvent ( QMouseEvent * e )
virtual void wheelEvent ( QWheelEvent * e )
virtual void keyPressEvent ( QKeyEvent * e )
virtual void keyReleaseEvent ( QKeyEvent * e )
virtual void focusInEvent ( QFocusEvent * )
virtual void focusOutEvent ( QFocusEvent * )
virtual void enterEvent ( QEvent * )
virtual void leaveEvent ( QEvent * )
virtual void paintEvent ( QPaintEvent * )
virtual void moveEvent ( QMoveEvent * )
virtual void resizeEvent ( QResizeEvent * )
virtual void closeEvent ( QCloseEvent * e )
virtual void contextMenuEvent ( QContextMenuEvent * e )
virtual void imStartEvent ( QIMEvent * e )
virtual void imComposeEvent ( QIMEvent * e )
virtual void imEndEvent ( QIMEvent * e )
virtual void tabletEvent ( QTabletEvent * e )
virtual void dragEnterEvent ( QDragEnterEvent * )
virtual void dragMoveEvent ( QDragMoveEvent * )
virtual void dragLeaveEvent ( QDragLeaveEvent * )
virtual void dropEvent ( QDropEvent * )
virtual void showEvent ( QShowEvent * )
virtual void hideEvent ( QHideEvent * )
virtual bool macEvent ( MSG * )
virtual bool winEvent ( MSG * )
virtual bool x11Event ( XEvent * )
virtual bool qwsEvent ( QWSEvent * )
```

The paintEvent we have already come across when we did our own custom drawing in previous chapters and we'll be dealing more with paintEvent later in this chapter and again when we get to the chapter on drawing.

KeyBoard Basics

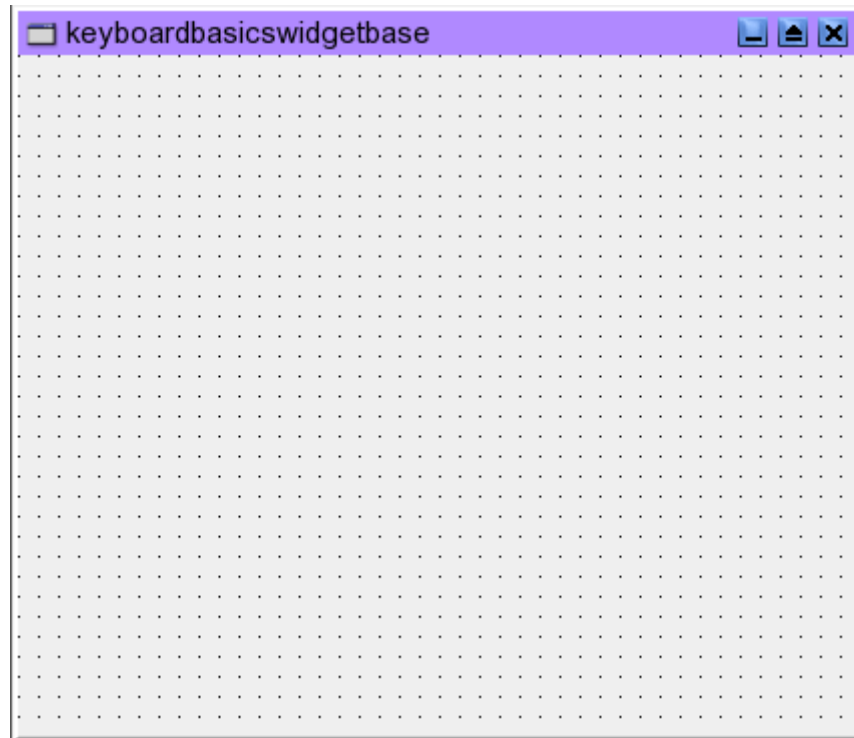
For this section we will concentrate on the standard keyboard and mouse events and pretty much ignore the rest, unless we find we need them in later projects.

The functions are declared as protected virtual functions which means that if we want to override them by subclassing the widget that implements them. In the case of the key demonstration code we will override the QTextEdit class with the class,

```
class KKeyDemoEdit : public QTextEdit
{
    Q_OBJECT
```

which somewhat unfortunately gives us a starting project that looks like,

Chapter 11 Events



with the subclassed Edit widget being added in the constructor with the code,

```
KKeyDemoEdit *textEdit = new KKeyDemoEdit( this, "demoEdit" );
textEdit->setGeometry( QRect( 11, 16, 390, 300 ) );
textEdit->setSizePolicy( QSizePolicy( QSizePolicy::SizeType)7, (QSizePolicy::SizeType)7,
                                0, 0, textEdit->sizePolicy().hasHeightForWidth() ) );
textEdit->setMinimumSize( QSize( 390, 300 ) );
```

All the functionality for the demo program takes place within the KKeyDemoEdit which overrides the functions KeyPress and Key Release

```
protected:
    virtual void keyPressEvent( QKeyEvent *keyEvent );
    virtual void keyReleaseEvent( QKeyEvent *keyEvent );
```

it also declares some simple state variables,

```
private:
    bool bControlPressed;
    bool bShiftPressed;
    bool bAltPressed;
```

and implements the gets and sets for them.

The main part of the program takes place when a key is pressed and the keyPressEvent function is called.

The function starts with the code,

```
bool bStateSet = false;
int nKey = keyEvent->key();
append( "KeyPressed" );
append( "Key = " + QString::number( keyEvent->key() ) );
append( "KeyText = " + keyEvent->text() );
append( "Ascii Code = " + QString::number( keyEvent->ascii() ) );
append( "Button State = " );
```

Chapter 11 Events

The code adds text to our edit control that tells us about the key that was pressed. To start with the data is collected using the functions available from the `QKeyEvent` pointer passed to it. The values returned are pretty simple in that the key is the integer value for the key the text is the text of the key which is the letter or number that was pressed and the ascii code represents the ascii value of the key. Typically though life isn't always simple as the state value which in the `qevent` header file reads as,

```
ButtonState state() const    { return ButtonState(s); }
```

should return the state for the button so that we can tell if the Control, Shift or the Alt have been pressed, this doesn't appear to ever get set as in the debugger the value `s` is almost always zero, I say almost because if you add the code,

```
if( keyEvent->state() == QKeyEvent::ControlButton )
{
    int i=0;
}
```

and set a breakpoint at `i = 0` you will occasionally catch it though on my development system it misses more than it catches and therefore comes under the heading of unreliable, still we have the key value as an integer so we can manage.

```
if( nKey == 4128 )
{
    moveCursor( QTextEdit::MoveEnd, false );
    insert( "Shift pressed " );
    bStateSet = true;
    setShiftPressed( true );
}
```

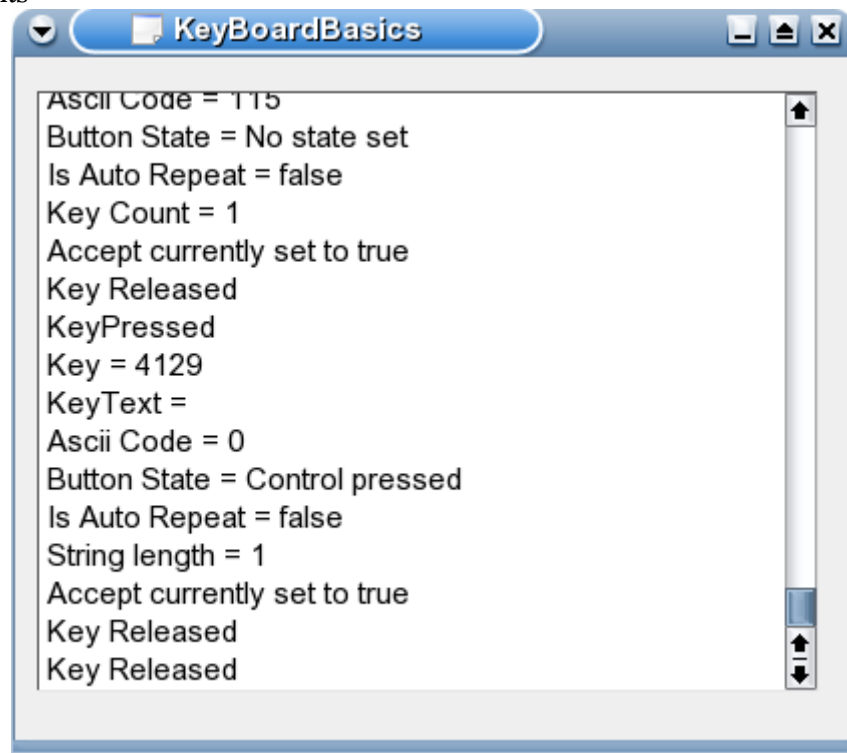
This code checks to see if the shift button has been pressed and if so sets the `ShiftPressed` value to true which means if the key is still pressed when another key is pressed the code at the end of the function,

```
if( nKey != 4128 && shiftPressed() == true )
{
    moveCursor( QTextEdit::MoveEnd, false );
    append( "Shift + " + keyEvent->text() + " pressed" );
}
```

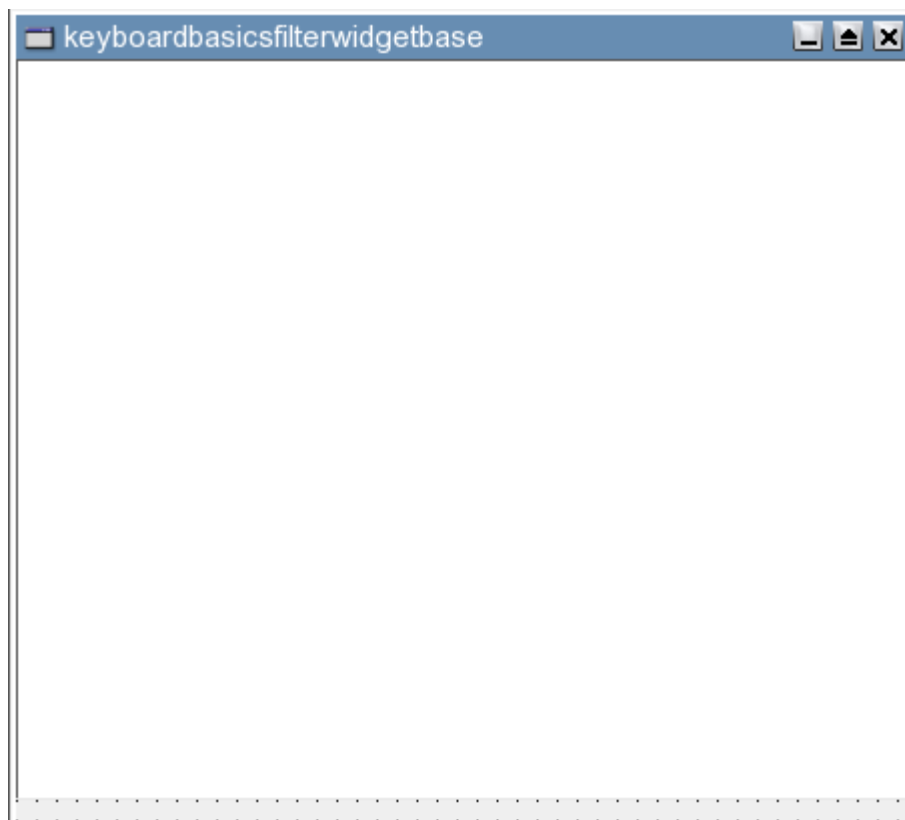
is activated. Other parts of the function check to see if the keypad has been pressed or if the key is repeating. The final part is the `isAccepted` function. This if set to true, tells the system that we have dealt with the key press and effectively swallows it. If it is set to false then the key stroke is passed through the system to anything that wants to use it.

The running program looks like,

Chapter 11 Events



There is however, another way of doing getting the keyboard input and various other events from within a class that doesn't rely on subclassing the widget that you are using. This is done through a process called event filtering. What happens is that as event such as keystrokes are passed through the system then you can tell Qt that you wish to handle events within your own code. We use this technique in the keyboardBasicsFilter project.



Chapter 11 Events

This project is almost identical to the previous project except that we are using a standard edit widget and in the constructor we call,

```
textEdit->installEventFilter( this );
```

The event filter itself is declared as.

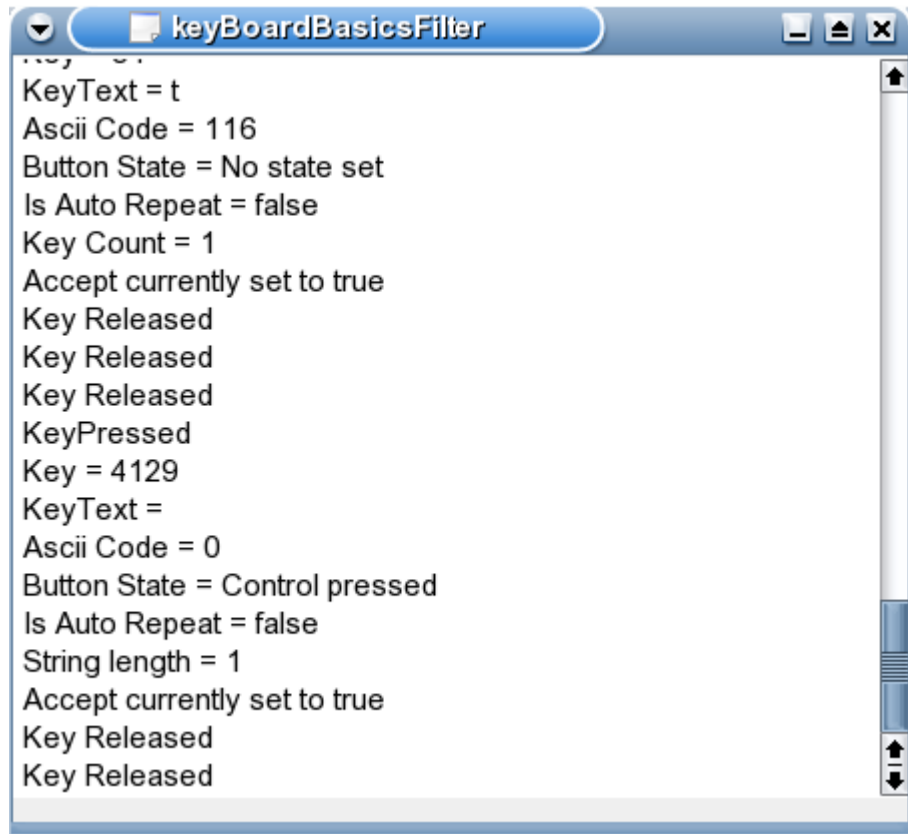
```
virtual bool eventFilter( QObject *watched, QEvent *e );
```

Note that we are overriding the eventFilter function we are not declaring a user function to handle the events, we are simply saying by calling installEventFilter that we have overridden the eventFilter function.

In the eventFilter function we have this code.

```
if( watched == textEdit )
{
    if( event->type() == QEvent::KeyPress )
    {
        QKeyEvent *keyEvent = static_cast< QKeyEvent * >( event );
        return FilterKeyPressedEvent( keyEvent );
    }
    if( event->type() == QEvent::KeyRelease )
    {
        QKeyEvent *keyEvent = static_cast< QKeyEvent * >( event );
        return FilterKeyReleasedEvent( keyEvent );
    }
}
else
{
    return this->eventFilter( watched, event );
}
```

What we are doing here is testing to see if the event occurred in the textEdit widget which is the one that we wanted the keyboard input for and then we check the event type. If the type is a QEvent KeyPress then we cast the passed in QEvent to a QKeyEvent and pass it to the function that we are using to deal with the keypresses. We do exactly the same for the key release and end up with a program that looks suspiciously like the previous program.



And that's how we get the keyboard input.

Tip Of The Day

When we talk about event filters the theory makes it sound as if we are adding an event filter that will be able to filter all events that are available on the computer. This is not entirely true in that certain items and widgets get certain events and certain others don't. On the whole this makes sense but it can be a stumbling block if you are trying to do something different, whether it is considered good GUI design or not. You will notice that the MouseBasic code contains,

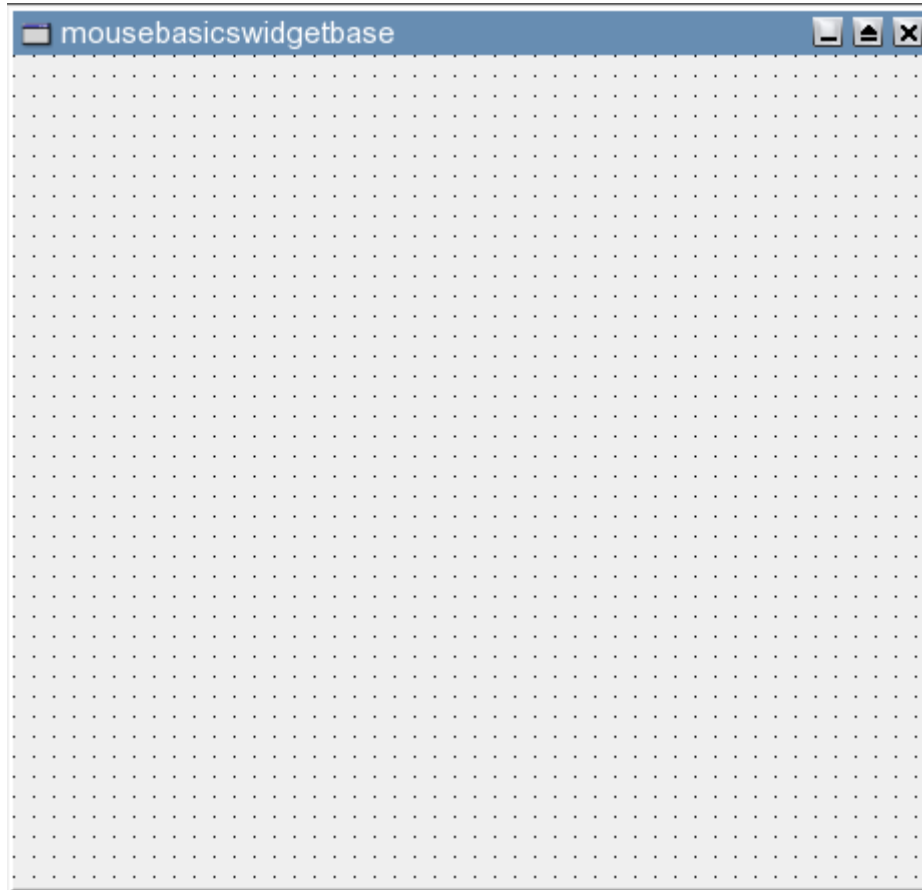
```
if( e->type() == QEvent::KeyPress )
{
    QMessageBox::information( this, "info", "key pressed" );
    return true;
}
```

This code is never executed, which when you think about it, it is not unreasonable to assume that a standard widget will not require keyboard input.

Mouse Basics

When dealing with the mouse there are three events that we are going to focus on these are the MouseButtonPress and the Wheel events, which we will then test to see which button was pressed or which way the wheel was turned. Of course as we are focusing on the mouse there isn't much to look at so we start off with an uninspiring looking project.

Chapter 11 Events



As with the previous project this project is written using the `installEventFilter` way of handling event filters.

Popup Menus

Everyone who has ever used a computer knows what a popup or a context menu is. You right click on the form or widget and a relevant menu pops up giving you a limited range of options that you can perform on your current task. For instance you may be able to set some item data within the program or choose a shape or pen but you will rarely be able to access the file save menu from a popup menu.

In this project we will use the mouse to place text characters on to the form, using the popup menu to select the characters and the mouse wheel to select between upper and lower case characters.

To create a popup menu declare it in the header file.

```
QPopupMenu *popupMenu;  
QPopupMenu *capsPopupMenu;
```

Note here that we are creating two menus one for standard characters and one for capital characters, which means that in the constructor we have,

```
popupMenu = new QPopupMenu();  
popupMenu->insertItem( "a", this, SLOT( setA() ) );  
popupMenu->insertItem( "b", this, SLOT( setB() ) );
```

Chapter 11 Events and

```
capsPopupMenu = new QPopupMenu();
capsPopupMenu->insertItem( "A", this, SLOT( setA() ) );
capsPopupMenu->insertItem( "B", this, SLOT( setB() ) );
```

As you can see the popup menus are allocated the same as any other QObject and then we call insertItem to add the menu item that will be viewed when the popup is shown. There are a number of insertItem overrides defined in the QPopupMenu class and you should look at them in order to choose the one that best suits what you want to do. The one used in the project is takes the form of

```
int insertItem ( const QString & text, const QObject * receiver, const char * member,
const QKeySequence & accel = 0, int id = -1, int index = -1 )
```

With the text string being the text to be shown, the receiver being the class that will receive the signal when the item is clicked, the member being the slot that will be called when the menu item is clicked, the QKeySequence being the accelerator keys to quickly access the menu item, the id being the identification number and the index being the index number for placing the item in the menu which if left to its default will add the item at the end.

The slots are defined in the header as you would expect,

```
public slots:
    /*$PUBLIC_SLOTS$*/

    void setA();
    void setB();
    void setC();
```

with standard implementations,

```
void MouseBasicsWidget::setA()
{
    if( capsLocked() == true )
        setText( 'A' );
    else
        setText( 'a' );
}
```

which set the internal text value to the character required. As mentioned earlier the program uses the installEventFilter function in the constructor so we check for the right button click with the code

```
if( e->type() == QEvent::MouseButtonPress )
{
    QMouseEvent *mouseEvent = static_cast< QMouseEvent * >( e );
    if( mouseEvent->button() == Qt::LeftButton )
    {
        drawText( mouseEvent->x(), mouseEvent->y(), QString( QChar( text() ) ) );
        TextData textData;
        textData.setText( QChar( text() ) );
        textData.setXpos( mouseEvent->x() );
        textData.setYpos( mouseEvent->y() );
        textList.append( textData );

        return true;
    }

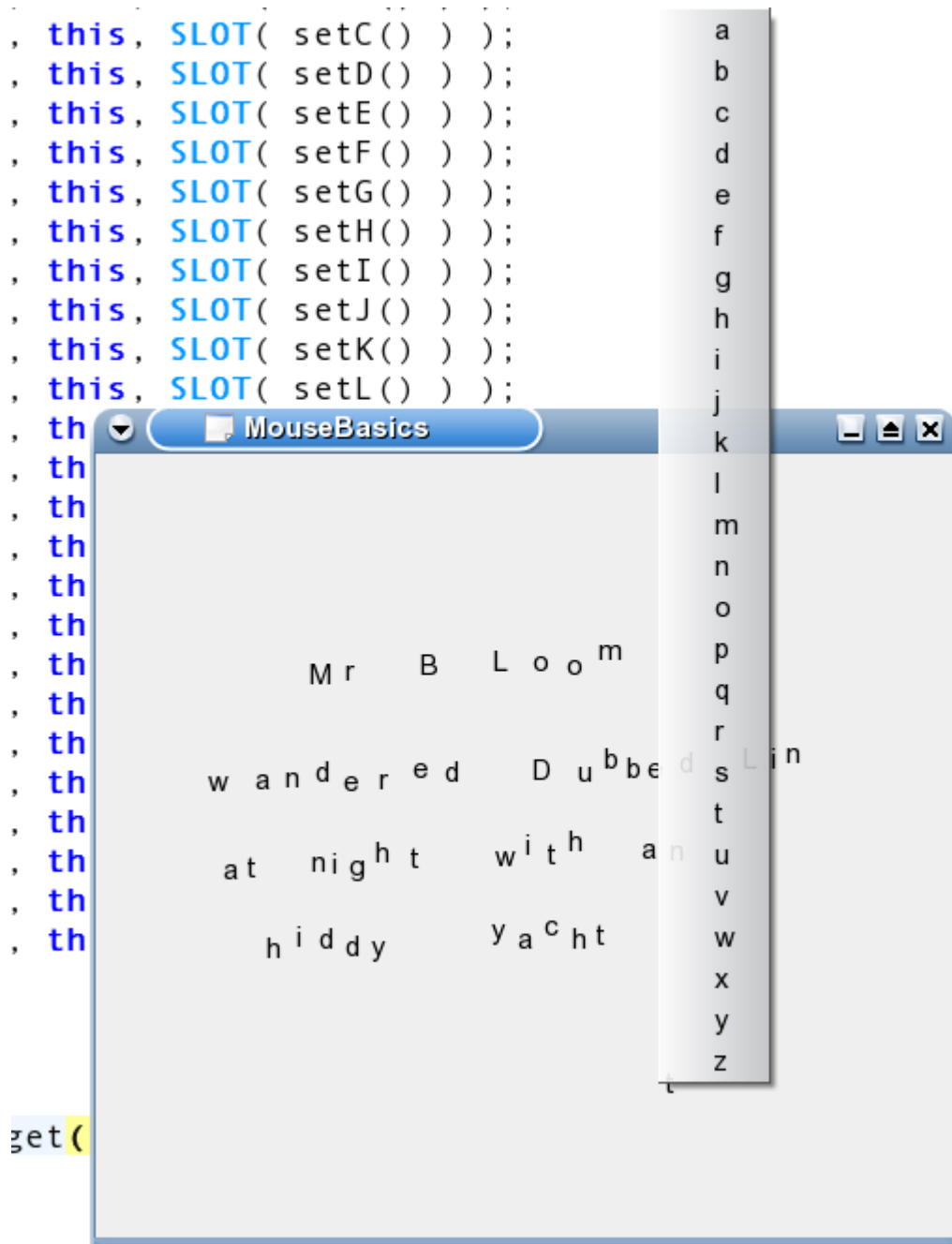
    if( mouseEvent->button() == Qt::RightButton )
    {
        if( capsLocked() == true )
            capsPopupMenu->exec( mouseEvent->globalPos() );
        else
            popupMenu->exec( mouseEvent->globalPos() );

        return true;
    }
}
```

Chapter 11 Events

```
}  
}
```

When the event comes through we check first of all that it is a mouse button press event and then convert the QEvent passed to the filter to a QMouseEvent before checking which button was actually pressed. Don't worry about the left button press for now we'll get to that in a minute. When the right button is pressed we execute either the normal small character popup menu or the capitals popup menu which looks like,



To set the menu to capitals we, check for the wheel movement, if the wheel is rotated forwards then we use the capital popup menu and if it is rotated backwards then we use the standard lower case characters. This is done in the eventFilter function with the code.

Chapter 11 Events

```
if( e->type() == QEvent::Wheel )
{
    QWheelEvent *wheelEvent = static_cast< QWheelEvent * >( e );
    if( wheelEvent->delta() > 0 )
        setCapsLocked( true );
    else
        setCapsLocked( false );

    return true;
}
```

Collection Basics and Drawing

Later on we will look at using Collection classes in your projects in some detail but for now we will just use a basic collection class that will be needed when we draw the text. The reason for this is that when the left button is clicked with the code,

```
if( mouseEvent->button() == Qt::LeftButton )
{
    drawText( mouseEvent->x(), mouseEvent->y(), QString( QChar( text() ) ) );
    TextData textData;
    textData.setText( QChar( text() ) );
    textData.setXpos( mouseEvent->x() );
    textData.setYpos( mouseEvent->y() );
    textList.append( textData );

    return true;
}
```

We initially draw the text using the drawText function. This is fine for a start but if we then use the popup menu again and it covers the characters that we have already placed on the widget or if the widget is minimised and then resized the characters will not be drawn again. The simple reason for this being that we have effectively drawn over them and not put in any commands to tell the program that when this happens and they become visible again they should be redrawn.

To do this we need to use the paintEvent function again with the code.

```
void MouseBasicsWidget::paintEvent( QPaintEvent *paintEvent )
{
    QPainter painter( this );

    TextList::iterator it;

    for( it=textList.begin(); it != textList.end(); it++ )
    {
        drawText( ( *it ).xpos(), ( *it ).ypos(), QString( QChar( ( *it ).text() ) ) );
    }
}
```

What is happening here is that we have a TextList that we move/iterate through and then draw all the characters in the list with the drawText function. The list itself is made up of the class TextData which is declared as,

```
class TextData
{
public:
    TextData();
    virtual ~TextData();

    inline void setYpos( int pos ){ nYpos = pos; };
    inline int ypos(){ return nYpos; };
    inline void setXpos( int pos ){ nXpos = pos; };
    inline int xpos(){ return nXpos; };
};
```

Chapter 11 Events

```
inline void setText( char text ){ cText = text; };
inline char text(){ return cText; };

private:
    int nYpos;
    int nXpos;
    char cText;
};
```

Literally that is all it is the cpp file merely contains an empty constructor and destructor. As you can see it contains everything we need for drawing a single character to the screen. The y position, the x position and the character itself. So if we look at what happens when we click the left button again,

```
if( mouseEvent->button() == Qt::LeftButton )
{
    drawText( mouseEvent->x(), mouseEvent->y(), QString( QChar( text() ) ) );
    TextData textData;
    textData.setText( QChar( text() ) );
    textData.setXpos( mouseEvent->x() );
    textData.setYpos( mouseEvent->y() );
    textList.append( textData );

    return true;
}
```

You can see that we create an object of the type TextData fill it out and append it to the textList which if you have any experience of programming you should have used before the only difference here being that Qt provides its own collection classes that take value types as well as classes that take references this means that when using a value class we don't have to worry about memory management as we aren't allocating anything, we are ineffect telling the list to deal with all that for us until we either remove the item or let the list go out of scope.

The type of list we are using is a QList which is effectively a template class so in C++ we must define the type we need before we use it. This is done in the header with the code,

```
typedef QList< TextData > TextList;
```

This declares a type of object called a TextList that is defined as a QList taking a data type of TextData, to use this type we create an object as normal in the MouseBasicsWidget class

```
private:
    QPopupMenu *popupMenu;
    QPopupMenu *capsPopupMenu;
    QPoint mousePoint;
    char cText;
    bool bCapsLock;
    TextList textList;
```

Note that it is a value type so that we don't allocate any memory for it in the constructor. Once the list is created and data is added through adding characters to the widget by left clicking on the place where we want the characters to appear we can then return to the paintEvent function which draws it with the code,

```
TextList::iterator it;

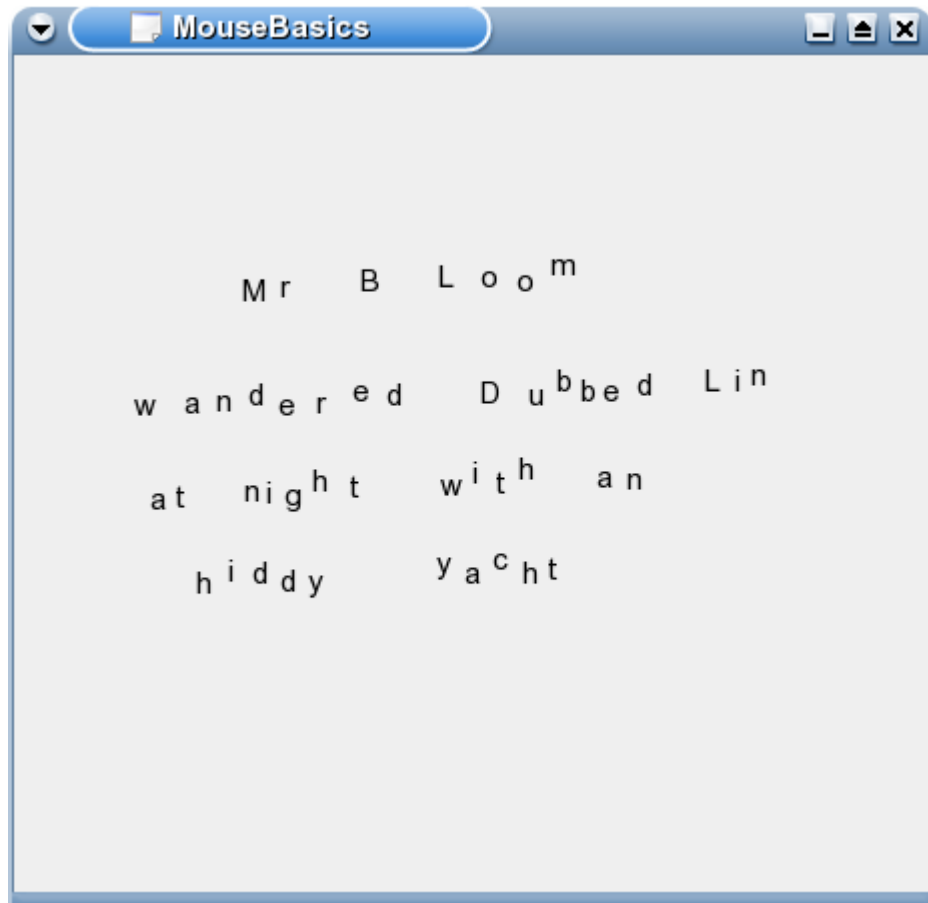
for( it=textList.begin(); it != textList.end(); it++ )
{
    drawText( ( *it ).xpos(), ( *it ).ypos(), QString( QChar( ( *it ).text() ) ) );
}
```

An iterator is effectively a pointer that is used to move through the list one item at a time. If it helps

Chapter 11 Events

think of it a TextData pointer or as a pointer of the type contained within the list and and we move it through the list using a for loop, getting the values that we require and passing them to the drawText function with each iteration through the loop.

Ultimately giving us a program that looks like,



Summary

In this chapter we have looked at the basics of keyboard and mouse input and also been introduced to popup menus and container classes. In the next chapter we take a closer look at files and directories in KDE.

Chapter 12 Drawing

In this chapter we take a closer look at drawing as it is kind of a compulsory subject for one of these things but one that most people hardly use in their careers. Drawing code can often be very long and look very complicated but as you've seen in various chapters in this book it isn't really as hard as it looks. By now you have already seen how to draw your own button and how to draw text to directly onto a form and in this chapter we will expand on what we have learned so far and add a few details about KDE while we do it.

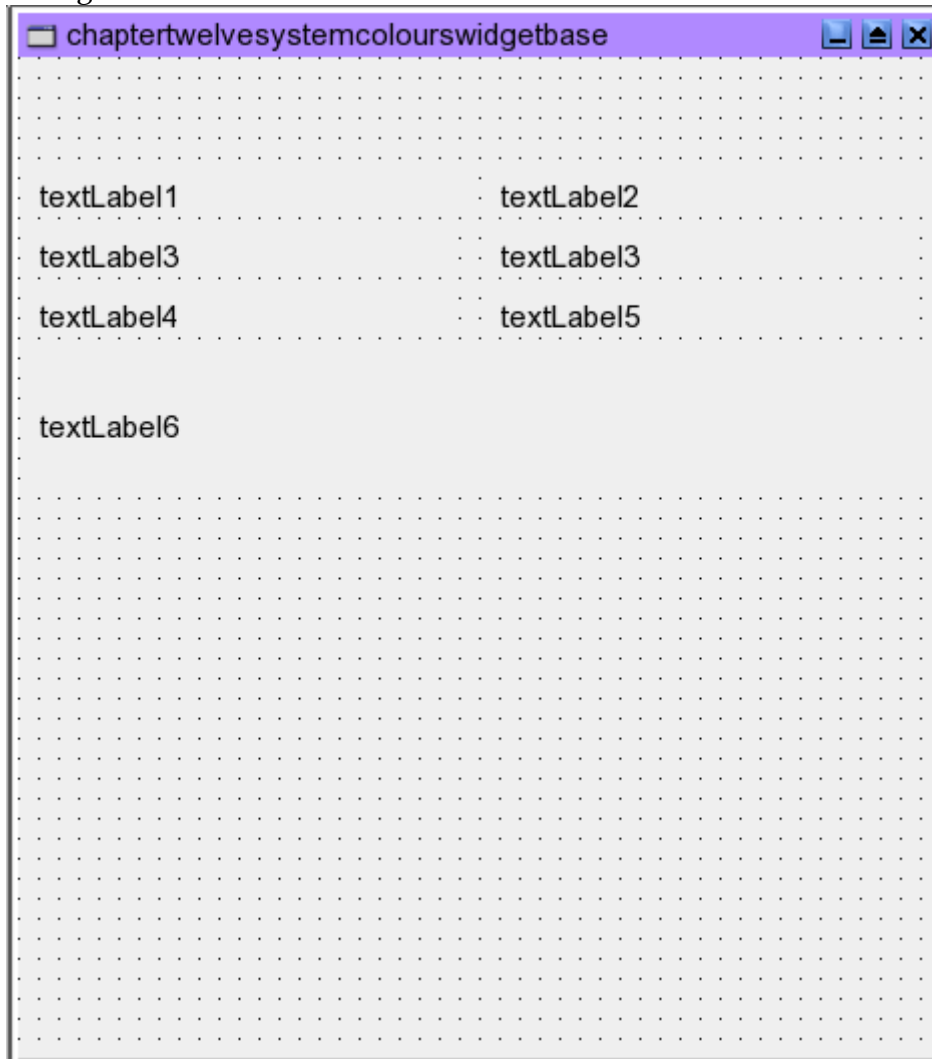
KDE Colours

To start with we are going to look at how the KDE colour scheme works, as you'll see this is largely a function of the Qt libraries that KDE is built from. The main point of interest to start with is the class KGlobalSettings which you will find in the help. It works from the idea that you don't specify a specific colour to do your drawing but you select a system colour that then draws the in the selected colour. You saw this in the KSquareButton class with the code,

```
painter->setBrush( backgroundColor().dark() );
```

Which returns the background colour and then calls the QColor dark() function. With the KGlobalSettings class we can access the colours specified by the KDE system.

Chapter 12 Drawing



KDEChapterTwelveMenusRunning1.png The project starts with just six text labels on a form, it is a drawing example after all. These text labels are for the KDE system fonts so let's look at them first.

The code for the fonts is set up in the constructor so if you change the system fonts you will need to restart the application for the change. It is a pretty basic procedure to display the fonts. We assign a few strings to display and then get the fonts

```
QFont currentFont = KGlobalSettings::generalFont();
QFont fixedFont = KGlobalSettings::fixedFont();
QFont toolBarFont = KGlobalSettings::toolBarFont();
QFont menuFont = KGlobalSettings::menuFont();
QFont windowTitleFont = KGlobalSettings::windowTitleFont();
QFont taskBarFont = KGlobalSettings::taskbarFont();
QFont largeFont = KGlobalSettings::largeFont();
```

then set the fonts on the labels with the code,

```
currentFontLabel->setFont( currentFont );
fixedFontLabel->setFont( fixedFont );
toolBarFontLabel->setFont( toolBarFont );
menuFontLabel->setFont( menuFont );
windowTitleFontLabel->setFont( windowTitleFont );
taskBarFontLabel->setFont( taskBarFont );
largeFontLabel->setFont( largeFont );
```

by calling setFont on each label we get,

Chapter 12 Drawing



This is the current font This is the fixed font

This is the tool bar font This is the menu font

This is the window title font This is the task bar font

This is the large font

on my system. The system colours on the other hand are all shown during the `paintEvent` function and this at first looks quite complex. There are two aspects to this complexity in drawing code the first is the calculation code which at first glance can be quite confusing and the second is the wealth of unfamiliar functions and the number of parameters they take. So for this example there is precisely one drawing function that you haven't seen before and two font related functions which means that in about 150 lines of code there are about three lines that should be completely new, the rest is simple a case of wash, rinse and repeat.

We'll start with the basic maths which calculates where we are going to place the text and the rectangles that we are using to display the colours.

```
int nCenter = width() / 2;
int nQuarter = nCenter / 2;
int nThreeQuarters = nCenter + nQuarter;
int nStartHeight = 240;
int nStartWidth = 10;
int nDefaultHeight = 20;
int nRectWidth = nQuarter - 20;
int nRectHeight = height() / 10;
int nFirstRectLine = nStartHeight + ( nDefaultHeight * 2 );
int nSecondTextHeight = nStartHeight + ( nDefaultHeight * 2 ) + nRectHeight + 20;
int nSecondRectLine = nSecondTextHeight + nDefaultHeight;
int nThirdTextHeight = nSecondTextHeight + ( nDefaultHeight * 2 ) + nRectHeight + 20;
int nThirdRectLine = nThirdTextHeight + nDefaultHeight;
int nFourthTextHeight = nThirdTextHeight + ( nDefaultHeight * 2 ) + nRectHeight + 20;
int nFourthRectLine = nFourthTextHeight + nDefaultHeight;
```

Frightening huh? First we find a few basic positions on the widget.

```
int nCenter = width() / 2;
int nQuarter = nCenter / 2;
int nThreeQuarters = nCenter + nQuarter;
```

The `nCenter` variable is the exact center of the widget, the `nQuarter` is exactly half that and the `nThreeQuarters` is three quarters of the way across the widget. Now we have a few basic points lets define some defaults.

```
int nStartHeight = 240;
int nStartWidth = 10;
int nDefaultHeight = 20;
```

The `nStartHeight` is where we start drawing text below the text labels for the fonts. The `nStartWidth` gives us a little space from the left hand edge of the widget and the `nDefaultHeight` is the value we will use as spacer between the text and the rectangles we are going to draw.

The final variables we set with are original values are

```
int nRectWidth = nQuarter - 20;
int nRectHeight = height() / 10;
```

These define the height and the width of the rectangles that we are going to draw. The variables are

Chapter 12 Drawing

defined like this so that the rectangles that we draw are based on the size of the widget that we are drawing to, this means that if you resize the program the rectangles will resize along with the widget.

The variables,

```
int nFirstRectLine = nStartHeight + ( nDefaultHeight * 2 );
int nSecondTextHeight = nStartHeight + ( nDefaultHeight * 2 ) + nRectHeight + 20;
int nSecondRectLine = nSecondTextHeight + nDefaultHeight;
int nThirdTextHeight = nSecondTextHeight + ( nDefaultHeight * 2 ) + nRectHeight + 20;
int nThirdRectLine = nThirdTextHeight + nDefaultHeight;
int nFourthTextHeight = nThirdTextHeight + ( nDefaultHeight * 2 ) + nRectHeight + 20;
int nFourthRectLine = nFourthTextHeight + nDefaultHeight;
```

are all set to variables that we have already initialised and calculate the heights to start drawing the text and the rectangles.

Once we have set up the variables we need to set the title for this section on the widget.

```
QFontMetrics generalFontMetrics( KGlobalSettings::generalFont() );
int nGeneralFontWidth = generalFontMetrics.width( strTitle );
painter.drawText( nCenter - ( nGeneralFontWidth / 2 ), 50, strTitle );
```

We use the QFontMetrics class to get the information for the current font which is the standard or default font which is returned by calling KGlobalSettings::generalFont(). Once we have got the font information we can calculate the exact width of the string we are using for the title by passing it to the QFontMetrics width function. Then we draw the string to the center of the widget using drawText.

The rest of the function is a repetition of this code,

```
painter.drawText( nStartWidth, nStartHeight, strColours );
painter.drawText( nStartWidth, nStartHeight + nDefaultHeight,
    strToolBarHighlightColour );

painter.fillRect( nStartWidth, nFirstRectLine, nRectWidth, nRectHeight, QBrush(
    KGlobalSettings::toolBarHighlightColor() ) );

int nFirstFontWidth = generalFontMetrics.width( strToolBarHighlightColour ) + 20;
if( nFirstFontWidth > nQuarter )
{
    painter.drawText( nFirstFontWidth, nStartHeight + nDefaultHeight,
        strInactiveTitleColour );
    painter.fillRect( nFirstFontWidth, nFirstRectLine, nRectWidth, nRectHeight,
        QBrush( KGlobalSettings::inactiveTitleColor() ) );
}
else
{
    painter.drawText( nQuarter, nStartHeight + nDefaultHeight, strInactiveTitleColour );
    painter.fillRect( nQuarter, nFirstRectLine, nRectWidth, nRectHeight, QBrush(
        KGlobalSettings::inactiveTitleColor() ) );
}

int nSecondFontWidth = nFirstFontWidth + generalFontMetrics.width(
    strInactiveTextColour ) + 20;
if( nSecondFontWidth > nCenter )
{
    painter.drawText( nSecondFontWidth, nStartHeight + nDefaultHeight,
        strInactiveTextColour );
    painter.fillRect( nSecondFontWidth, nFirstRectLine, nRectWidth, nRectHeight,
        QBrush( KGlobalSettings::inactiveTextColor() ) );
}
else
{
    painter.drawText( nCenter, nStartHeight + nDefaultHeight, strInactiveTextColour );
    painter.fillRect( nCenter, nFirstRectLine, nRectWidth, nRectHeight, QBrush(
        KGlobalSettings::inactiveTextColor() ) );
}
```

Chapter 12 Drawing

```
}

int nThirdFontWidth = nSecondFontWidth + generalFontMetrics.width(
    strActiveTitleColour ) + 20;

if( nThirdFontWidth > nThreeQuarters )
{
    painter.drawText( nThirdFontWidth, nStartHeight + nDefaultHeight,
        strActiveTitleColour );
    painter.fillRect( nThirdFontWidth, nFirstRectLine, nRectWidth, nRectHeight,
        QBrush( KGlobalSettings::activeTitleColor() ) );
}
else
{
    painter.drawText( nThreeQuarters, nStartHeight + nDefaultHeight,
        strActiveTitleColour );
    painter.fillRect( nThreeQuarters, nFirstRectLine, nRectWidth, nRectHeight,
        QBrush( KGlobalSettings::activeTitleColor() ) );
}
```

This page of code is repeated four times with the only changes being the string that is output and the colour of the rectangles being drawn. Actually if you looked closely at the code above you'll notice that this page itself draws the first line of rectangles and is made up of this code,

```
painter.drawText( nStartWidth, nStartHeight + nDefaultHeight,
    strToolBarHighlightColour );

painter.fillRect( nStartWidth, nFirstRectLine, nRectWidth, nRectHeight, QBrush(
    KGlobalSettings::toolBarHighlightColor() ) );

int nFirstFontWidth = generalFontMetrics.width( strToolBarHighlightColour ) + 20;
if( nFirstFontWidth > nQuarter )
{
    painter.drawText( nFirstFontWidth, nStartHeight + nDefaultHeight,
        strInactiveTitleColour );
    painter.fillRect( nFirstFontWidth, nFirstRectLine, nRectWidth, nRectHeight,
        QBrush( KGlobalSettings::inactiveTitleColor() ) );
}
else
{
    painter.drawText( nQuarter, nStartHeight + nDefaultHeight, strInactiveTitleColour );
    painter.fillRect( nQuarter, nFirstRectLine, nRectWidth, nRectHeight, QBrush(
        KGlobalSettings::inactiveTitleColor() ) );
}
```

This code draws the label and the first rectangle, before drawing the label and second rectangle. It starts by drawing the text for the first label at the correct width and height and then draws the rectangle directly below it. The second rectangle is drawn differently because we have to first of all calculate where we are going to draw it. We do this by getting the width of the string that labels the rectangle and seeing if it is larger than a quarter of the way across the screen, the reason for this being that if the screen is too small for the text it will overlap and it looks better if we draw outside the bounds of the widget than it does if we try to squash everything in on itself. So if the width of the string is greater we draw it regardless if not we draw it at the required location which in this case is nQuarter in the next case it will be nCenter and in the final case it will be nThreeQuarters.

The end result is that when we run the program we get.



When we run it with the KDE default colours. If we go to the main menu and select Control Center/Appearance and Themes/Colors and select Blue Slate we get



Remember though that as the fonts are set during the constructor you will need to stop and restart the program before you will notice any changes to them. The colours will change whenever the colours are reset and you set the focus to the application which will force it to redraw thus changing the colours.

Tip Of the Day

Probably the most important thing to remember when painting to a widget is never call a function

Chapter 12 Drawing

that is going to issue a repaint to the widget. This is the reason why the fonts are set up in the constructor as changing the font forces the widget to repaint itself so drawing the fonts by hand on the widget and then changing the drawing font for each one just means that the widget is constantly getting told to redraw itself. Actually as the different fonts are displayed in labels it would be perfectly acceptable to move them into the paint function and have them update at the same time as the coloured rectangles because the paint messages would be issued to the labels and not to the widget itself.

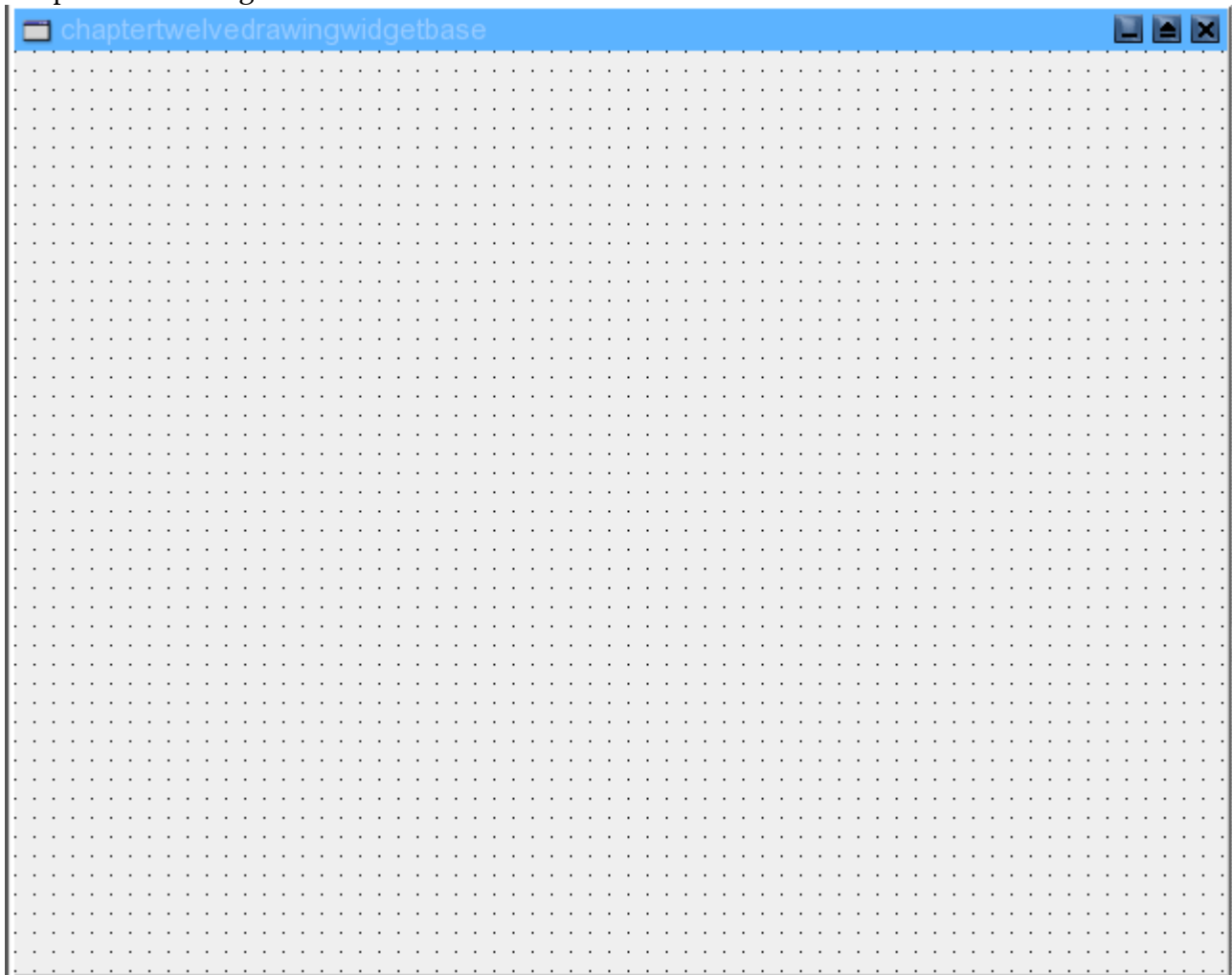
Drawing Example

In every book like this it is almost an unwritten rule that there will be some mostly pointless drawing example that you will never ever need to write in the real world. The reason these are included is because when you do need to do this sort of thing it can be a complete pain to get right and it does help to give an understanding of how the programs are working. So in the spirit of things we are going to look at some drawing examples the first being the ChapterTwelveDrawing example. This is a deliberately restricted example for two reasons, one I wanted to include owner draw menus and because I use them the design doesn't scale up into a completely workable application and two I wanted to show an example that displays the mechanics of how these work better than I have seen in some other beginners books. This is not to say that the other books are wrong it is just that the mechanics of how things work tend to get lost in the complications of getting everything working correctly.

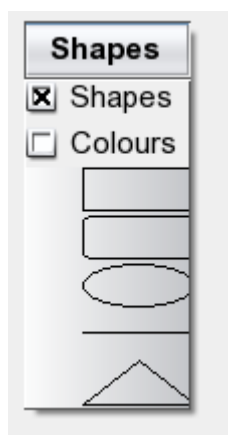
So the plan is this with the first program we will concentrate on how to get the basics working and in the next program we will aim for a more functional application.

The first program is the ChapterTwelveDrawing example and it looks like this,

Chapter 12 Drawing



It is a plain and simple dialog to start with because we are going to use the surface to draw on and don't want to clutter it up with widgets. The idea here is that we have a minimal interface and do everything through a single popup menu that is activated when we right click on the qwidget. You have seen a popup menu before but not one quite like this for a start it looks like,



and



Yes it is a single menu with the display being chosen by the check boxes on the menu. Admittedly this could have been better implemented as two separate menus but the idea is to show what can be done here, not what is necessarily best from a design point of view. Of course the first thing that should strike you about the menus is that apart from the check box widgets they are entirely owner drawn so let's have a look at how it's done.

Owner Draw Menus.

We've already seen how versatile popup menu's can be and a quick glance at the help page for KPopupMenu and/or QPopupMenu shows us that we can use pictures as menu items but what if we want to be more expressive and do something that isn't already catered for in that case we can use the QCustomMenuItem class. This class is the class object that is used as a single menu item. The QCustomMenuItem class contains all the information required to qualify as a menu item while allowing us to specify exactly what happens when the menu item is drawn. So given the two views of the menu above we need six distinct classes. Five classes are to draw the shapes and one class to draw the coloured rectangles.

You should note that the check boxes are standard check boxes added in the normal way with,

```
coloursCheckBox = new QCheckBox( popupMenu, "Colours" );
coloursCheckBox->setText( "Colours" );
coloursCheckBox->setChecked( false );
popupMenu->insertItem( coloursCheckBox, ColoursCheckBoxItem );
```

Chapter 12 Drawing

The signals are then connected using,

```
connect( coloursCheckBox, SIGNAL( clicked() ), this, SLOT( coloursClicked() ) );
```

To start with the standard rectangle we would define the class as,

```
class KRectMenuItem : public QCustomMenuItem
{
public:
    KRectMenuItem();
    virtual ~KRectMenuItem();

    virtual void paint( QPainter *p, const QColorGroup& cg, bool act, bool enabled,
        int x, int y, int w, int h );
    virtual QSize sizeHint();
};
```

As you can see the KRectMenuItem contains only a constructor, a destructor and the two functions required to control the drawing, these being the sizeHint function which reserves space in the menu and the paint function. The important one for us being the paint function,

```
void KRectMenuItem::paint( QPainter *p, const QColorGroup& /*cg*/, bool /*act*/, bool
/*enabled*/, int x, int y, int w, int h )
{
    p->drawRect( x, y, w*2, h-2 );
}
```

When the paint function is called it passes more than enough information than we need here for what we are trying to do. We are given a valid painter to the menu item that we are painting and the colours available in the QColorGroup object. The active and the enabled parameters are the menus status while the x and y are the starting point for our menu item, with the w being the width and the h being the height. In all the calculations for the owner draw menu items shown here we use w*2 as we are drawing over the space reserved for the short cut keys as well.

Given this information we can see that in order to draw an owner drawn round rect we need to declare an identical class and in the constructor use the function.

```
p->drawRoundRect( x, y, w*2, h-2 );
```

and for the ellipse,

```
p->drawEllipse( x, y, w*2, h-2 );
```

In order to draw the KLineMenuItem we have to be a little more creative,

```
p->drawLine( x, y+10, x+( w*2 ), y+10 );
```

increasing the height given in the y parameter to center the line drawing and then using x+(w*2) to specify exactly where we want the line drawing to. With the KTriangleMenuItem we have to be a lot more creative,

```
QPointArray points;
points.setPoints( 4, x + w, y, x, y + ( h-2 ), x + ( w*2 ), y + ( h-2 ), x + w, y );
p->drawPolyline( points );
```

Here we need to create a QPointArray and then declared four distinct points, these being x+w, y which is exactly halfway across the top of the drawing space at the height of y, then x, y + (h-2), which is the point at the x at the height of y + (h-2) which in English translates to the bottom left hand corner as you look at it. Then the next point is the bottom right hand corner as you look at it which translates to the code x + (w*2), y + (h-2), with the final point being back at the top in the middle of the menu item or x + w, y. once the points are set we pass the points array to the

Chapter 12 Drawing drawPolyline function.

The KColourRectMenuItem is dealt with in exactly the same way first we define the class as,

```
class KColourRectMenuItem : public QCustomMenuItem
{
public:
    KColourRectMenuItem();
    KColourRectMenuItem( QColor colour );
    virtual ~KColourRectMenuItem();

    virtual void paint( QPainter *p, const QColorGroup& cg, bool act, bool enabled,
        int x, int y, int w, int h );
    virtual QSize sizeHint();

private:
    QColor cColour;
};
```

Then the paint function contains the code,

```
p->fillRect( x, y, w*2, h-2, QBrush( cColour ) );
```

which draw a rectangle the same as the drawRect function and fills it with the passed in colour in a standard brush. The standard brush is a solid pattern to see the available brush styles see the QBrush reference documentation.

Once they are created the new custom menu items are added to the popup menu exactly as you would expect.

```
rectMenuItem = new KRectMenuItem();
popupMenu->insertItem( rectMenuItem, RectMenuItem );

roundRectMenuItem = new KRoundRectMenuItem();
popupMenu->insertItem( roundRectMenuItem, RoundRectMenuItem );

ellipseMenuItem = new KEllipseMenuItem();
popupMenu->insertItem( ellipseMenuItem, EllipseMenuItem );
```

Each menu item is created and then added to the QPopupMenu with the insertItem function which takes the newly created menu item and an identifier that is returned when a menu item is selected on the popup menu.

The identifiers are defined in the ChapterTwelveDrawingWidget.h class file as,

```
enum PopupMenuIdentifiers{ ShapesCheckBoxItem, ColoursCheckBoxItem, RectMenuItem,
RoundRectMenuItem, EllipseMenuItem, LineMenuItem, TriangleMenuItem, BlackColourMenuItem,
WhiteColourMenuItem, DarkGreyColourMenuItem, GreyColourMenuItem, LightGreyColourMenuItem,
RedColourMenuItem, GreenColourMenuItem, BlueColourMenuItem, CyanColourMenuItem,
MagentaColourMenuItem, YellowColourMenuItem, DarkRedColourMenuItem,
DarkGreenColourMenuItem, DarkBlueColourMenuItem, DarkCyanColourMenuItem,
DarkMagentaColourMenuItem, DarkYellowColourMenuItem };
```

These are used not only to identify what colour or shape has been selected but also to set what is being displayed, so initially the colour menu items are set up as.

```
blackRectMenuItem = new KColourRectMenuItem();
popupMenu->insertItem( blackRectMenuItem, BlackColourMenuItem );
popupMenu->setItemVisible( BlackColourMenuItem, false );

whiteRectMenuItem = new KColourRectMenuItem( Qt::white );
popupMenu->insertItem( whiteRectMenuItem, WhiteColourMenuItem );
popupMenu->setItemVisible( WhiteColourMenuItem, false );
```

As you can see we add the colour menu items to the QPopupMenu but set them to be hidden by

Chapter 12 Drawing

default and then we use the check boxes to set which menu items we are going to see.

```
void ChapterTwelveDrawingWidget::coloursClicked()
{
    bColours = true;
    popupMenu->changeTitle( 1, "Colours" );
    // remove the shapes
    popupMenu->setItemVisible( RectMenuItem, false );
    popupMenu->setItemVisible( RoundRectMenuItem, false );
    popupMenu->setItemVisible( EllipseMenuItem, false );
    popupMenu->setItemVisible( LineMenuItem, false );
    popupMenu->setItemVisible( TriangleMenuItem, false );
}
```

With the above code showing the start of the coloursClicked function which hides all the shape menu items and unhides all the colour menu items.

Drawing The Shapes

In order that we can draw the shapes on the widget we must first track them when they are drawn by the user of the application. This is done by installing an eventFilter,

installEventFilter(this);

and the capturing the mouse button presses and movements.

```
bool ChapterTwelveDrawingWidget::eventFilter( QObject *watched, QEvent *e )
{
    if( watched == this )
    {
        if( e->type() == QEvent::MouseButtonPress )
        {
        }
        if( e->type() == QEvent::MouseButtonRelease )
        {
        }
        if( e->type() == QEvent::MouseMove )
        {
        }
    }

    return false;
}
```

As you can see from the skeleton code above we are interested in the button press the button release and the mouse move events. The button press events are used to control the popup menu and to start the drawing code. The code is,

```
QMouseEvent *mouseEvent = static_cast< QMouseEvent * >( e );
if( mouseEvent->button() == Qt::LeftButton )
{
    bDrawOutline = true;
    startPosition = mouseEvent->pos();
}

if( mouseEvent->button() == Qt::RightButton )
{
    DisplayPopup();
}

return true;
```

Pressing the right mouse button on the widget calls the DisplayPopup function which displays the popup menu with its custom objects,

```
nPopupReturn = popupMenu->exec( QCursor::pos() );
```

Chapter 12 Drawing

```
/// selected a shape
if( bColours == false )
{
    nDrawItem = nPopupReturn;
}
else /// selected a colour
{
    switch( nPopupReturn )
    {
        case BlackColourMenuItem : cDrawingColour = Qt::black; break;
        case WhiteColourMenuItem : cDrawingColour = Qt::white; break;
        case DarkGreyColourMenuItem : cDrawingColour = Qt::darkGray; break;
        case GreyColourMenuItem : cDrawingColour = Qt::gray; break;
        case LightGreyColourMenuItem : cDrawingColour = Qt::lightGray; break;
        case RedColourMenuItem : cDrawingColour = Qt::red; break;
        case GreenColourMenuItem : cDrawingColour = Qt::green; break;
        case BlueColourMenuItem : cDrawingColour = Qt::blue; break;
        case CyanColourMenuItem : cDrawingColour = Qt::cyan; break;
        case MagentaColourMenuItem : cDrawingColour = Qt::magenta; break;
        case YellowColourMenuItem : cDrawingColour = Qt::yellow; break;
        case DarkRedColourMenuItem : cDrawingColour = Qt::darkRed; break;
        case DarkGreenColourMenuItem : cDrawingColour = Qt::darkGreen; break;
        case DarkBlueColourMenuItem : cDrawingColour = Qt::darkBlue; break;
        case DarkCyanColourMenuItem : cDrawingColour = Qt::darkCyan; break;
        case DarkMagentaColourMenuItem : cDrawingColour = Qt::darkMagenta; break;
        case DarkYellowColourMenuItem : cDrawingColour = Qt::darkYellow; break;
    };
}
```

The result is stored in the a class variable either the nDrawItem or the cDrawingColour depending on which section of the menu is being displayed.

Pressing the left mouse button doesn't call any functions it simply sets the boolean variable bDrawOutline to true and stores the current mouse position. In order to make something happen the user has to press the left mouse button and then move the mouse whilst holding the button down, thus giving us the MouseMove event which we respond to with the code,

```
if( bDrawOutline == true )
{
    QMouseEvent *mouseEvent = static_cast< QMouseEvent * >( e );
    DrawOutLine( mouseEvent->pos() );
}

return true;
```

The DrawOutLine function draws a dotted outline of the currently selected shape so the user can see the size and position,

```
QPainter painter( this );
int nWidth = currentPos.x() - startPosition.x();
int nHeight = currentPos.y() - startPosition.y();

QRect rect( startPosition.x(), startPosition.y(), nWidth, nHeight );
if( nDrawItem == LineMenuItem )
{
    painter.eraseRect( rect );
    repaint( rect );
}
else
{
    repaint( rect );
}

painter.setPen( Qt::DotLine );

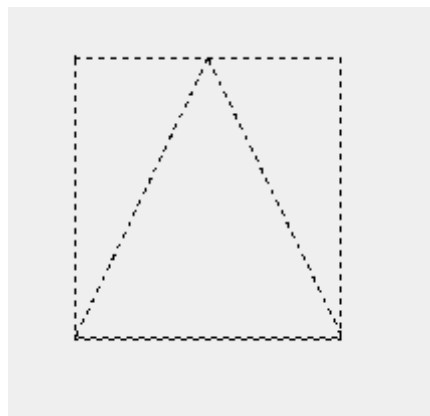
switch( nDrawItem )
```

Chapter 12 Drawing

```
{
    case RectMenuItem:
    {
        painter.drawRect( rect );
        painter.drawRect( startPosition.x(), startPosition.y(), nWidth, nHeight );
    }; break;
    case RoundRectMenuItem:
    {
        painter.drawRect( rect );
        painter.drawRoundRect( rect );
    }; break;
    case EllipseMenuItem:
    {
        painter.drawRect( rect );
        painter.drawEllipse( rect );
    }; break;
    case LineMenuItem:
    {
        painter.drawLine( startPosition.x(), startPosition.y(), currentPos.x(),
            currentPos.y() );
    }; break;
    case TriangleMenuItem:
    {
        painter.drawRect( rect );
        QPointArray points;
        points.setPoints( 4, startPosition.x() + nWidth/2, startPosition.y(),
            startPosition.x(), startPosition.y() + ( nHeight-2 ), startPosition.x() +
            ( nWidth ), startPosition.y() + ( nHeight-2 ), startPosition.x() + nWidth/2,
            startPosition.y() );
        painter.drawPolyline( points );
    }; break;
}
```

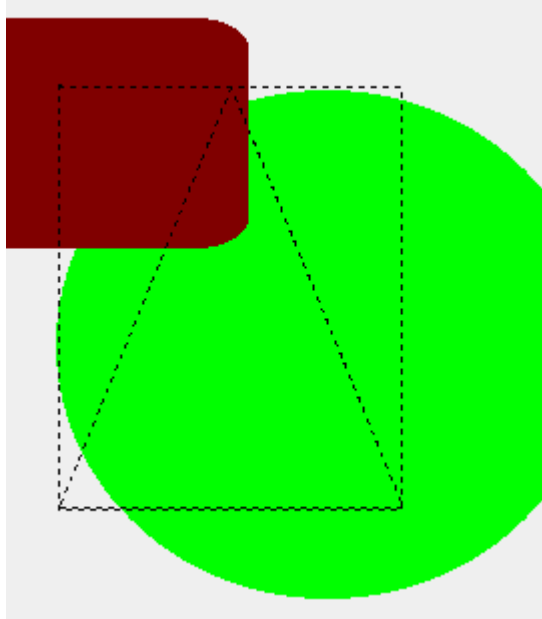
We start by calculating the current rectangle from the start position which was saved when the mouse button was first pressed down and then we erase that rectangle on the widget with a call to repaint. This erases the old dotted line drawing for the shape we are drawing so that we can then redraw it with the new size. To see how this works properly comment out the repaint and run the program then. You can see that the line drawing function is treated differently this is because of how we are trying to reduce the amount of drawing that we are doing at anyone point, in order to reduce screen flicker. There are other, better ways of reducing screen flicker that we will look at shortly.

This is what drawing a triangle will look like at the start.



And this drawing a triangle over other shapes,

Chapter 12 Drawing



The shape itself is not drawn until after the mouse button is released. The code executed on the `MouseButtonRelease` event is,

```
QMouseEvent *mouseEvent = static_cast< QMouseEvent * >( e );
if( mouseEvent->button() == Qt::LeftButton )
{
    bDrawOutline = false;
    /// save the current shape and repaint the drawing area
    KShape save;
    if( nDrawItem != LineMenuItem )
    {
        save.setX( startPosition.x() );
        save.setY( startPosition.y() );
        save.setWidth( mouseEvent->x() - startPosition.x() );
        save.setHeight( mouseEvent->y() - startPosition.y() );
        save.setShape( nDrawItem );
        save.setColour( cDrawingColour );
    }
    else
    {
        save.setX( startPosition.x() );
        save.setY( startPosition.y() );
        save.setWidth( mouseEvent->x() );
        save.setHeight( mouseEvent->y() );
        save.setShape( nDrawItem );
        save.setColour( cDrawingColour );
    }

    shapeList.append( save );
    repaint();

    return true;
}
```

All the `MouseButtonRelease` does is save the current shape and position, with the extra code being to handle the line where we are saving specific points and not the whole rectangle which is required to draw the other shapes. The shape details are saved in the `KShape` class which as we have seen previously is a simple data class that looks like,

```
class KShape
{
public:
    KShape();
    KShape( int x, int y, int width, int height, int shape, QColor colour );
    virtual ~KShape();
}
```

Chapter 12 Drawing

```
inline int x(){ return nX; };
inline void setX( int x ){ nX = x; };
inline int y(){ return nY; };
inline void setY( int y ){ nY = y; };
inline int width(){ return nWidth; };
inline void setWidth( int width ){ nWidth = width; };
inline int height(){ return nHeight; };
inline void setHeight( int height ){ nHeight = height; };
inline int shape(){ return nShape; };
inline void setShape( int shape ){ nShape = shape; };
inline QColor colour(){ return cColour; };
inline void setColour( QColor colour ){ cColour = colour; };

private:
    int nX;
    int nY;
    int nWidth;
    int nHeight;
    int nShape;
    QColor cColour;
};
```

As previously the data class is then stored in a collection which is defined as,

```
typedef QList< KShape > ShapeList;
```

The shapes are then appended to the collection and the MouseButtonRelease code then issues a repaint call which will issue a paintEvent to the widget.

The paintEvent function is then pretty much what we should expect,

```
void ChapterTwelveDrawingWidget::paintEvent( QPaintEvent /*paintEvent*/ )
{
    QPainter painter( this );

    /// draw the saved shapes

    ShapeList::iterator it;
    KShape shape;

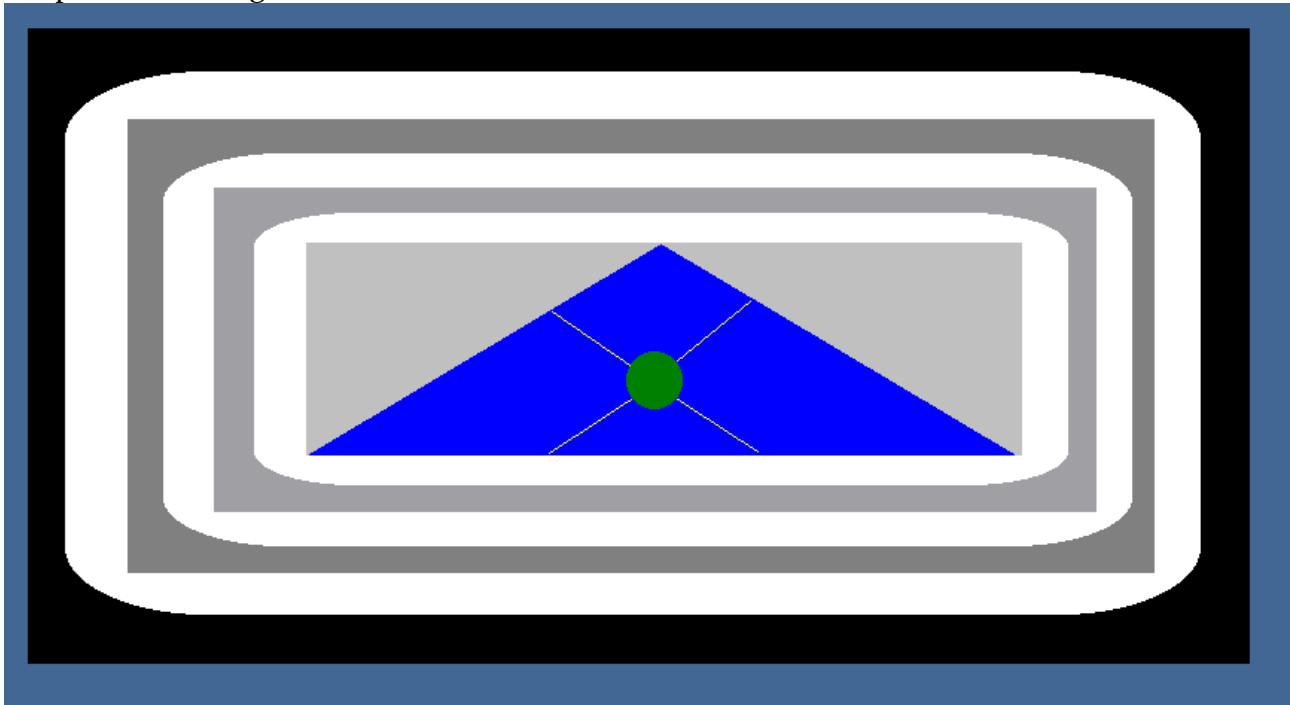
    painter.setPen( Qt::SolidLine );

    for( it = shapeList.begin(); it != shapeList.end(); it++ )
    {
        shape = ( KShape )( *it );
        painter.setBrush( shape.colour() );
        painter.setPen( shape.colour() );

        switch( shape.shape() )
        {
```

We set the current pen to a solid line and then iterate through ShapeList collection of KShapes using the switch statement to identify which shape is to be drawn and then drawing it in the currently selected colour.

We can then draw things like,



if you really want to.

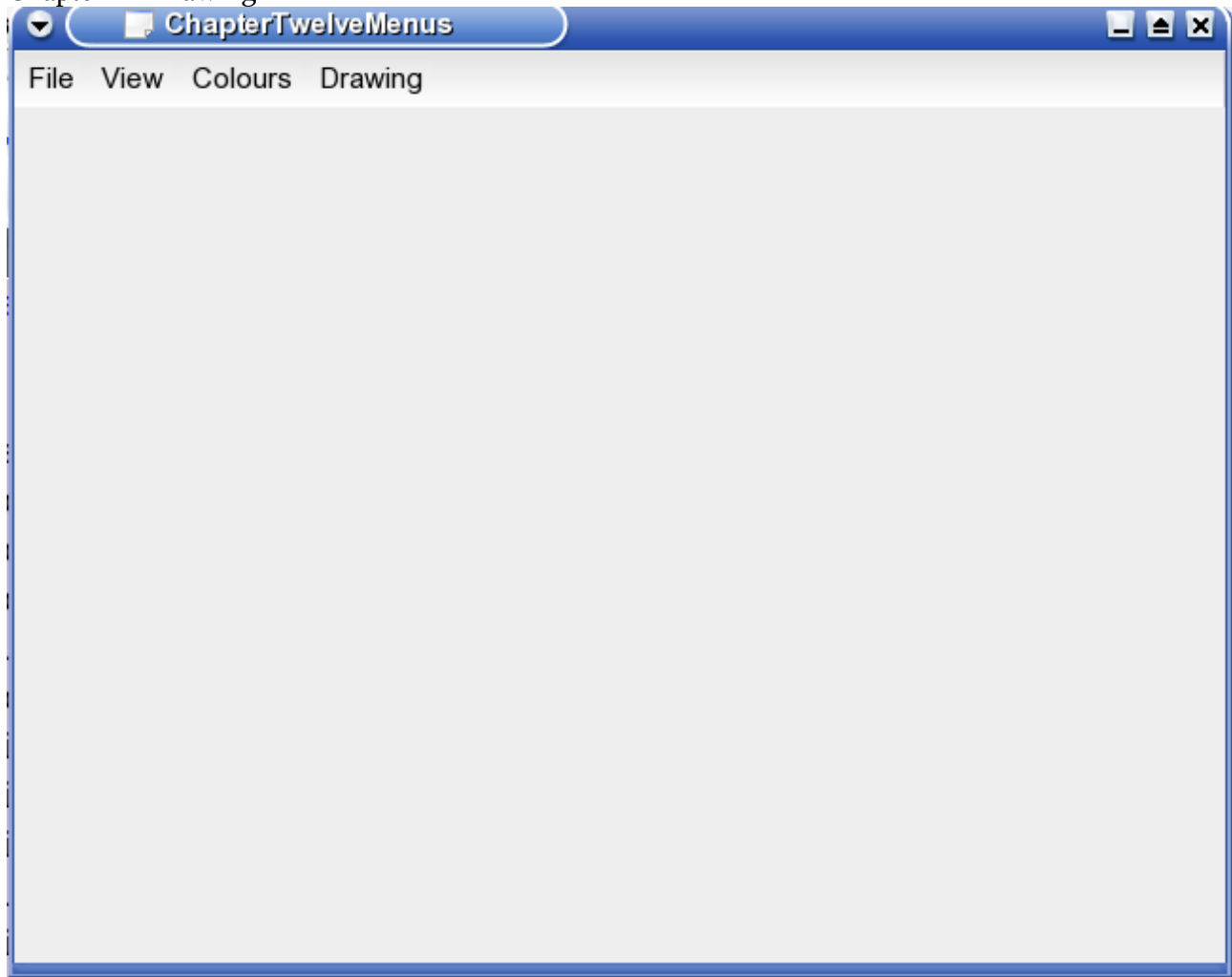
Menus And Double Buffering

In the previous example the demonstration code had obvious limitations which were largely caused by the way in which the menus for the application were chosen, using only the popup menus it was clear that trying to implement more options would have led to a more and more unmanageable menu system that would just get confusing and irritating for the user. So for the next application we will add the menus by hand to see what is required when we want to write an application that looks like a proper program. Another problem is that the more images you draw on the widget the more the application flickers when the widget is drawn, although this has been kept to a minimum it is still noticeable and basically no flicker is good. So the next program the ChapterTwelveMenus project also includes double buffering in such a way that you can run the program and turn it on and off to see the improvement for yourself. But first to the menus.

Menus

The first thing you should notice about the ChapterTwelveMenus project is that this is the point where we start to write applications that look like real applications, by having menus etc. The project looks like,

Chapter 12 Drawing



For the purposes of this project we have four menu options and we add them all to the project by hand. To do this we need these objects,

```
KMenuBar *menuBar;  
KColoursMenu *coloursMenu;  
KDrawingMenu *drawingMenu;  
KPopupMenu *viewPopup;  
KPopupMenu *filePopup;
```

This introduces three new classes that need a little explanation firstly the KColoursMenu and the KDrawingMenu classes are classes that I have written to encapsulate what we have already done before in that they control the options for the colours and the drawing options from the previous ChapterTwelveDrawing project. The header for the KColoursMenu class looks like this,

```
class KColoursMenu : public KPopupMenu  
{  
  
public:  
    KColoursMenu();  
  
    ~KColoursMenu();  
  
    inline void setDrawingColour( QColor colour ){ cDrawingColour = colour; };  
    inline QColor drawingColour(){ return cDrawingColour; };  
  
private:  
    KColourRectMenuItem *blackRectMenuItem;  
    KColourRectMenuItem *whiteRectMenuItem;
```

Chapter 12 Drawing

```
KColourRectMenuItem *darkGreyRectMenuItem;
KColourRectMenuItem *greyRectMenuItem;
KColourRectMenuItem *lightGreyRectMenuItem;
KColourRectMenuItem *redRectMenuItem;
KColourRectMenuItem *greenRectMenuItem;
KColourRectMenuItem *blueRectMenuItem;
KColourRectMenuItem *cyanRectMenuItem;
KColourRectMenuItem *magentaRectMenuItem;
KColourRectMenuItem *yellowRectMenuItem;
KColourRectMenuItem *darkRedRectMenuItem;
KColourRectMenuItem *darkGreenRectMenuItem;
KColourRectMenuItem *darkBlueRectMenuItem;
KColourRectMenuItem *darkCyanRectMenuItem;
KColourRectMenuItem *darkMagentaRectMenuItem;
KColourRectMenuItem *darkYellowRectMenuItem;

    QColor cDrawingColour;
};
```

As you can see it simply encapsulates the colour items from the previous example and the implementation is identical also with the constructor containing code along the lines of,

```
blackRectMenuItem = new KColourRectMenuItem();
insertItem( blackRectMenuItem, BlackColourMenuItem );

whiteRectMenuItem = new KColourRectMenuItem( Qt::white );
insertItem( whiteRectMenuItem, WhiteColourMenuItem );

darkGreyRectMenuItem = new KColourRectMenuItem( Qt::darkGray );
insertItem( darkGreyRectMenuItem, DarkGreyColourMenuItem );
```

As you can imagine the KDrawingMenu is likewise an implementation of the Drawing options available in the previous project.

The KMenuBar on the other hand is completely new and draws a blank menu for us across the top of the widget. We do this with the code,

```
menuBar = new KMenuBar( this );
```

The way the menus work is that you start out with a KMenuBar and then you add the popup menus to it and then add the actual menu options to the popup menu, so with the file menu we add the file KPopupMenu object filePopup with the code,

```
filePopup = new KPopupMenu();
menuBar->insertItem( strFile, filePopup );
```

Actions

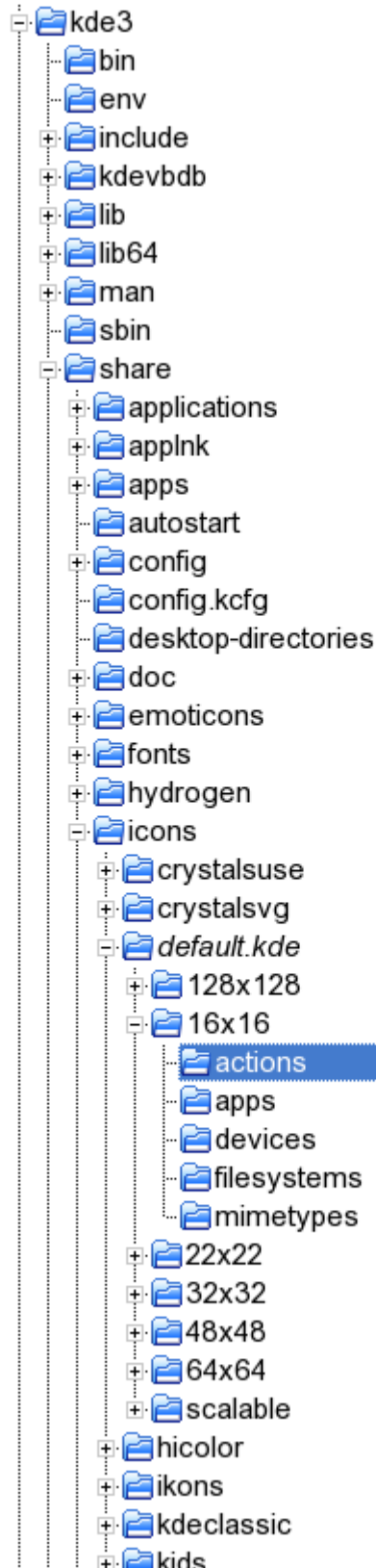
The preferred way of doing things in KDE is with actions. An action is an object set up to define a menu option and what image and text it shows. The action is then added to the menu or the toolbar and yes the same action object can be used for both. A call to connect is then used to define what happens when that action is chosen as a menu or toolbar option. We will see how they work with toolbars later but for now we will just define them to work with a menu. So taking the first one first the File New menu option is set up with the code.

```
fileNewAction = new QAction( strFileNew, 0, filePopup );
fileNewAction->setIconSet( BarIconSet( "filenew.png" ) );
fileNewAction->setStatusTip( strFileNewStatusTip );
connect( fileNewAction, SIGNAL( activated() ), this, SLOT( fileNew() ) );
fileNewAction->addTo( filePopup );
```

Chapter 12 Drawing

To start with we create the `fileNewAction` object giving it the string “File New”, we don't specify a keyboard accelerator but we do specify the `filePopup` menu as the parent.

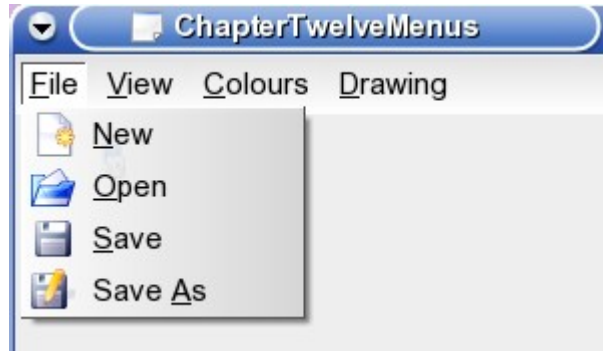
We then set the icon using the `BarIconSet` function which is a function exported by the KDE core library and is documented as a related function of `KIconLoader`. This function gets the icon file named and displays it on the menu. On Suse 10 the icons are to be found in `root/opt`



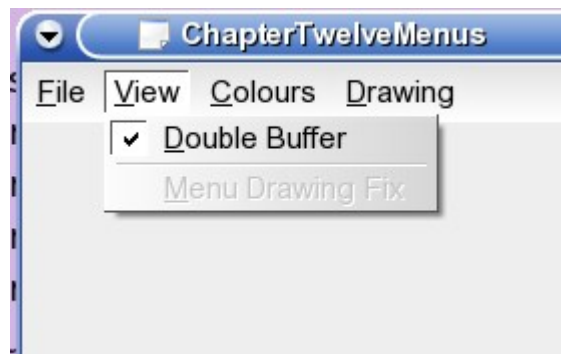
Chapter 12 Drawing

The default KDE icons are shown here but the `BarIconSet` function will get whatever files you have selected in the Control Center/Appearance And Themes/Icons section. Next we connect the actions activated signal to our `fileNew` function before adding the action to the `filePopup` menu.

The rest of the file menu is set up in exactly the same way, create the Action setting the text in the constructor and then adding the icon for the menu item before setting the tool tip and then connecting it to a function.



The view menu is slightly different in that the view menu contains two items,



These both work as check box options that are used within the code, we'll look at what they do later first of all you set up a check box menu item with the code,

```
viewPopup = new KPopupMenu();
menuBar->insertItem( strView, viewPopup );
doubleBufferAction = new QAction( strDoubleBuffer, 0, viewPopup );
doubleBufferAction->setToggleAction( true );
doubleBufferAction->setOn( true );
doubleBufferAction->setStatusTip( strDoubleBufferStatusTip );
connect( doubleBufferAction, SIGNAL( toggled( bool ) ), this, SLOT(
    toggleDoubleBuffer( bool ) ) );
doubleBufferAction->addTo( viewPopup );
```

We set up the menu the same as previously by creating a `KPopupMenu` item and adding it to the menu bar, then we call the `setToggleAction` function on the action to true and in the case of the double buffer action set it to true by default.

For the colours and the drawing menus there is a difference as we have already created the classes to hold the menus.

```
coloursMenu = new KColoursMenu();
menuBar->insertItem( i18n( "Colours" ), coloursMenu );
coloursAction = new QAction( coloursMenu, "ColoursAction" );
```

Chapter 12 Drawing

```
connect( coloursMenu, SIGNAL( activated( int ) ), this, SLOT( coloursClicked( int ) ) );
```

Here we create the entire menu as one object and as it is derived from KPopupMenu we insert it into the QMenuBar as normal. The difference is that when we create the action we create it for the whole menu not for each specific menu option which means that the activated function will be fired whenever an item on the menu is clicked which works exactly the way it would if we were using a standard popup menu so the code that is called is the repeated code,

```
switch( id )
{
    case BlackColourMenuItem : coloursMenu->setDrawingColour( Qt::black ); break;
    case WhiteColourMenuItem : coloursMenu->setDrawingColour( Qt::white ); break;
    case DarkGreyColourMenuItem : coloursMenu->setDrawingColour( Qt::darkGray );
        break;
    case GreyColourMenuItem : coloursMenu->setDrawingColour( Qt::gray ); break;
    case LightGreyColourMenuItem : coloursMenu->setDrawingColour( Qt::lightGray ); break;
    case RedColourMenuItem : coloursMenu->setDrawingColour( Qt::red ); break;
    case GreenColourMenuItem : coloursMenu->setDrawingColour( Qt::green ); break;
    case BlueColourMenuItem : coloursMenu->setDrawingColour( Qt::blue ); break;
    case CyanColourMenuItem : coloursMenu->setDrawingColour( Qt::cyan ); break;
    case MagentaColourMenuItem : coloursMenu->setDrawingColour( Qt::magenta ); break;
    case YellowColourMenuItem : coloursMenu->setDrawingColour( Qt::yellow ); break;
    case DarkRedColourMenuItem : coloursMenu->setDrawingColour( Qt::darkRed ); break;
    case DarkGreenColourMenuItem : coloursMenu->setDrawingColour( Qt::darkGreen );
        break;
    case DarkBlueColourMenuItem : coloursMenu->setDrawingColour( Qt::darkBlue );
        break;
    case DarkCyanColourMenuItem : coloursMenu->setDrawingColour( Qt::darkCyan );
        break;
    case DarkMagentaColourMenuItem : coloursMenu->setDrawingColour( Qt::darkMagenta );
        break;
    case DarkYellowColourMenuItem : coloursMenu->setDrawingColour( Qt::darkYellow );
        break;
};
```

from the ChapterTwelveDrawing project.

Saving and Loading QPixmaps

The project is capable of loading and saving files of the type .png through the file menu and it does through the KFileDialog to set the name of the file and by calling the QPixmap save and load functions, the restriction to .png files is largely laziness on my part. So the save function is,

```
setDontDrawOnSave( true );
QString strOpenFile = KFileDialog::getSaveFileName( strCurrentFile, "*.png|PNG
    Files");
qpBufferPixmap.save( strOpenFile, "PNG" ); /// NEEDS TO BE UPPERCASE;
setLoadedPixmap( true );
qpLoadedPixmap = qpBufferPixmap;
shapeList.clear();
setDontDrawOnSave( false );
issueRepaint();
```

The main lines here are the second and third lines which call the KFileDialog getSaveFileName and the QPixmap save function, that is all there is to it really the rest of the code is concerned with making sure that the drawing is done properly when the paint function is called. The setDontDrawOnSave function simply tells the program not to try and draw the picture until it is ready and the setLoadedPixmap to true tells the code to draw the saved picture as the background to the image and so clears the shapeList as we don't need it any more.

Chapter 12 Drawing

Double Buffering

The project has been specifically written to show the effects of double buffering so it allows you to turn double buffering on and off in the view menu. Double buffering is itself an attempt to reduce screen flicker when drawing by drawing as little to the screen as possible. You can get really complicated with this and draw the screen in sections but we are just going to use a single double buffer that draws to the whole screen. The idea is that when you draw to the screen what you in effect drawing is a picture, it doesn't matter if it's a picture of a mountain, a photo of someone you know or an on screen button to the computer screen it is an image to be displayed. The problem is that when you are constantly drawing to the screen the screen flickers as you draw over the same pixels again and again so what you do is draw the image off screen to a blank page, or in this case a QPixmap object, and then just draw the one image to the screen as quickly as you can. So in the header for the ChapterTwelveMenusWidget.h we have,

```
/// Double Buffering
///
QPixmap qpBufferPixmap;
bool bDoubleBuffer;
```

the idea being that when you set the double buffer to true in the view menu the shapes you draw on the screen are drawn to the qpBufferPixmap which is then drawn to the screen. This means that the paintEvent function now contains code like,

```
QPainter painter( this );
QPainter bufferPainter( &qpBufferPixmap );

...

case RoundRectMenuItem:
{
    if( doubleBuffer() == true )
    {
        bufferPainter.drawRoundRect( shape.x(), shape.y(), shape.width(), shape.height() );
    }
    else
        painter.drawRoundRect( shape.x(), shape.y(), shape.width(), shape.height() );
}; break;
```

Which sets up two painter objects one to the widget and one to the buffer pixmap and draws to whichever one is required.

The paintEvent function now ends with the code,

```
/// draw the doubleBuffer pixmap to the widget
///
if( doubleBuffer() == true )
    bitBlt( this, paintEvent->rect().topLeft(), &qpBufferPixmap, paintEvent->rect() );
```

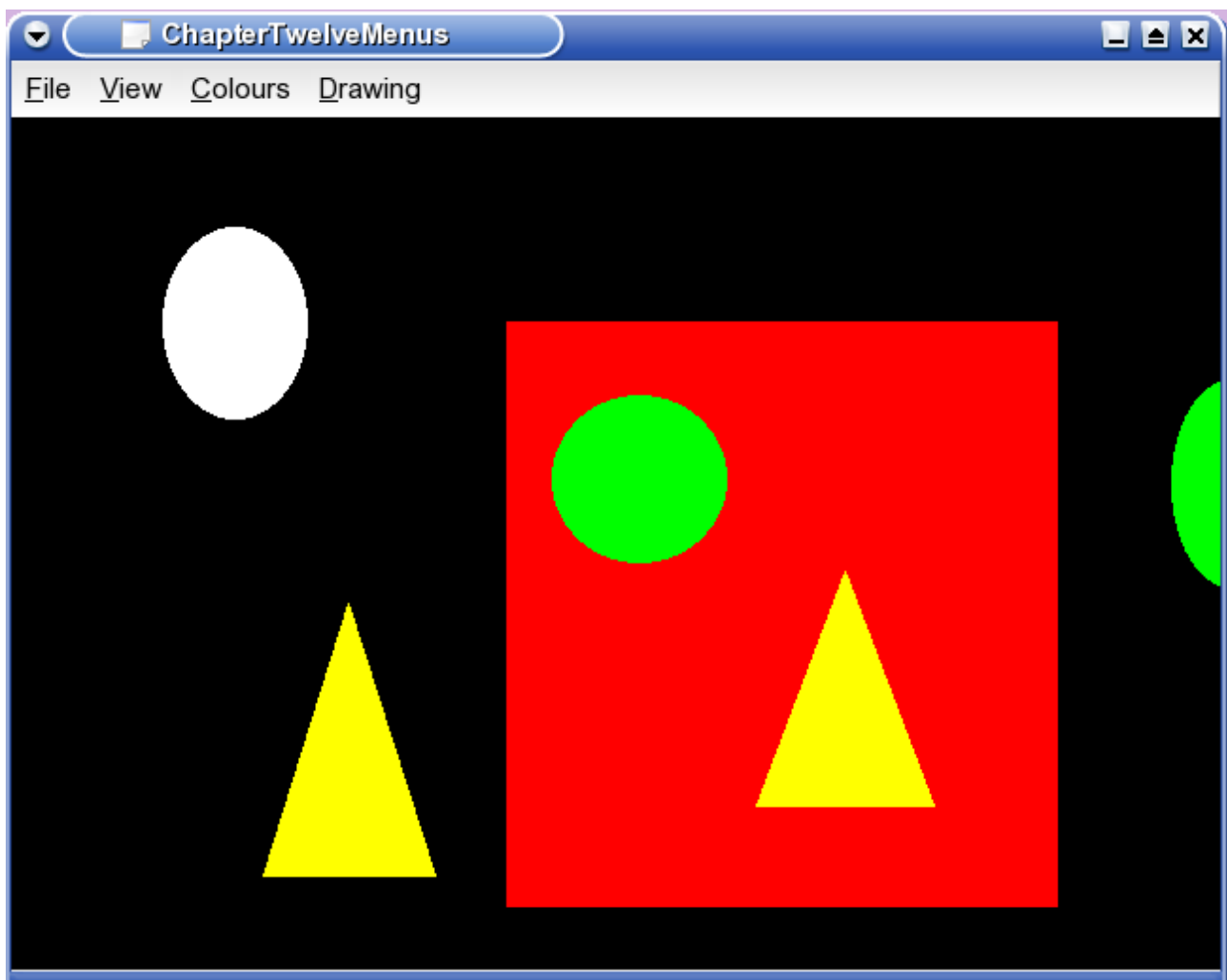
which draws the buffer pixmap to the screen if the double buffer option is set.

As the double buffering option is the default, the project starts up with it switched on yet you can use the option in the view menu to turn it off. This will allow the program to work without double buffering and enable the menu drawing fix option on the view menu. This option is a quick fix to draw over the menu images by blanking out the widget before it is redrawn, this is not a good idea to do it this way if you are trying to reduce flicker.

Enabling and disabling a menu is simply a call to,

```
menuDrawingFixAction->setEnabled( false );
```

Chapter 12 Drawing
using true to enable and false to disable.

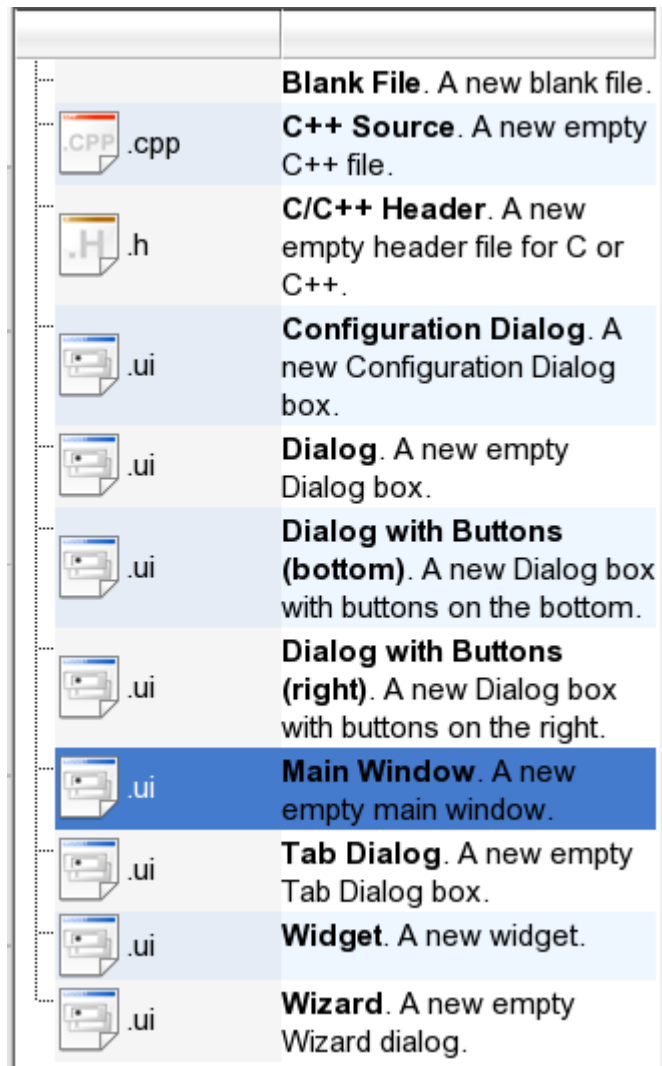


Adding A Main Window

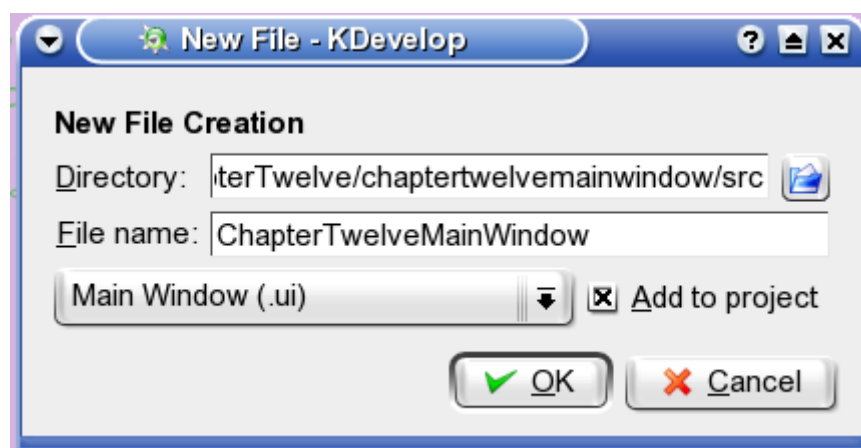
If we are ever going to consider writing so called proper applications then we need to use a proper main window that contains a widget, a toolbar and a status bar as well as a menu, sure we can get away without these things for small applications but eventually there comes a time when a small application is well, just not good enough to do what we want.

Unfortunately at the moment (Suse 10) there is no way to create an application in KDevelop from scratch that uses a main window. You could use a Simple KDE Application project but there is no .ui file to edit with it so if you don't want to write all the gui code by hand then you will need to add a main window to the project in exactly the same way that we are going to add a main window to the standard Simple Designer Based KDE Application that we have been using all along and will continue to use for the time being.

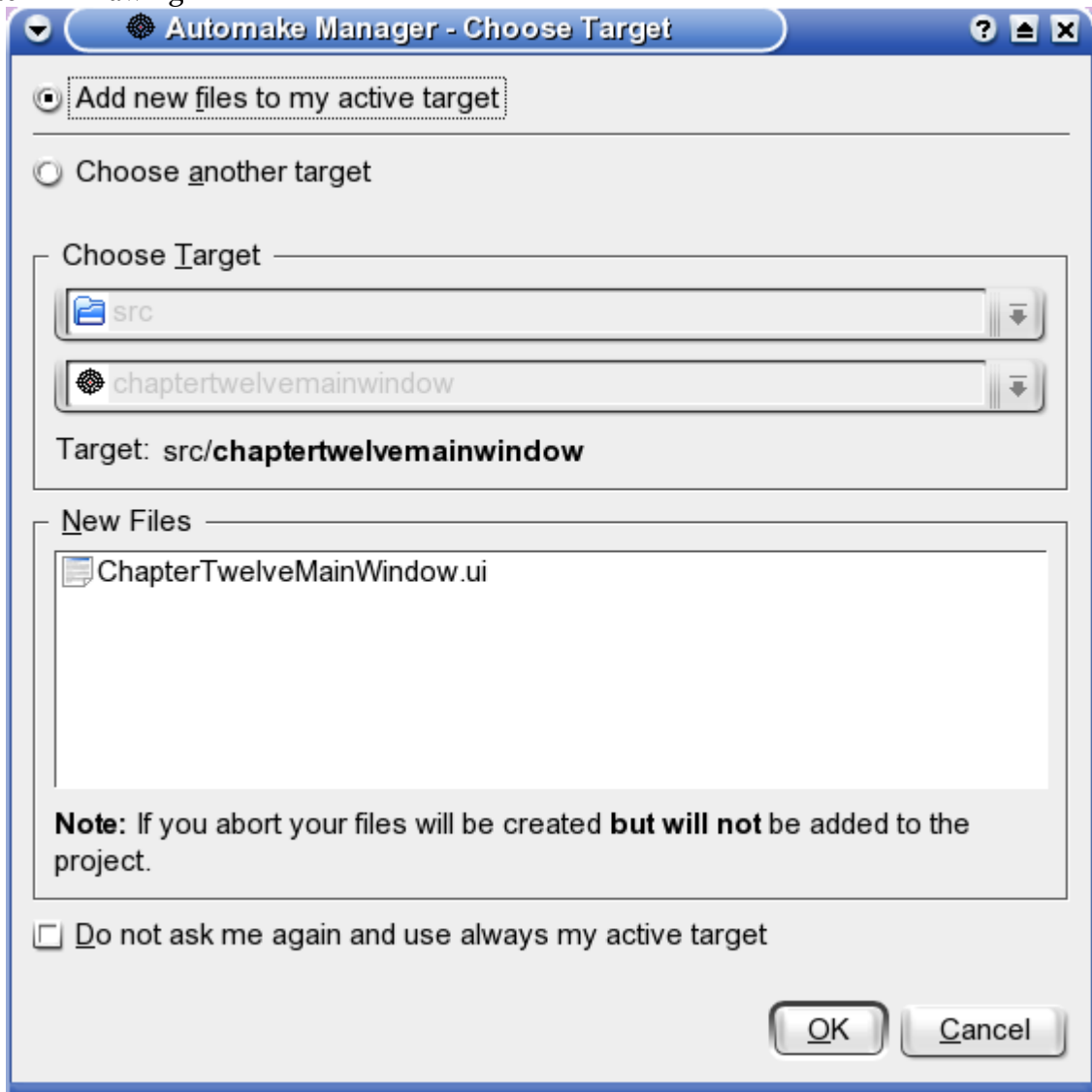
First of all we create the ChapterTwelveMainWindow application the same way as before, only this time we don't need to delete the generated hello world code as we won't be using it anyway. Open the New File tab,



Like so, and click on the Main Window option which will give you this dialog,

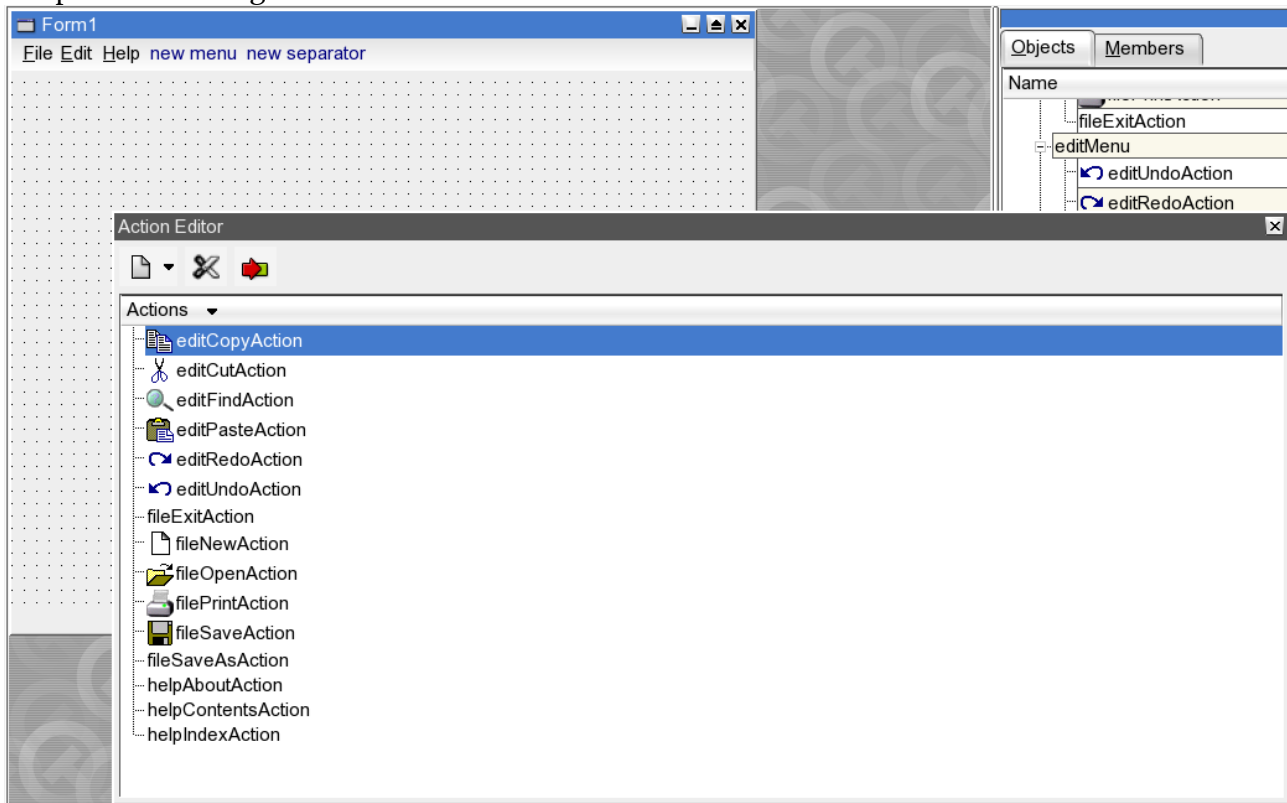


So we call the File the ChapterTwelveMainWindow, which when we click O.K. opens the automake manager dialog,

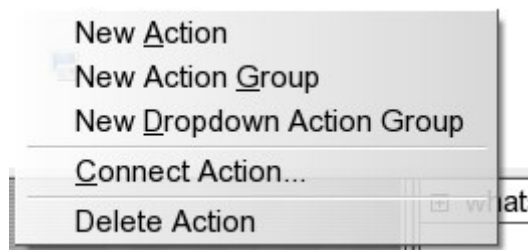


Just click O.K. for this dialog. Now if you click on the ChapterTwelvemainWindow.ui file that is added to the Automake Manager then you will see,

Chapter 12 Drawing



the action editor. We looked at actions recently and this is just a visual way of implementing them. Right clicking on the Action Editor gives to the options dialog,



What happens here is that you create menus in the traditional windows style way by clicking on the “new menu” option on the menu itself and then typing in the name of the menu,

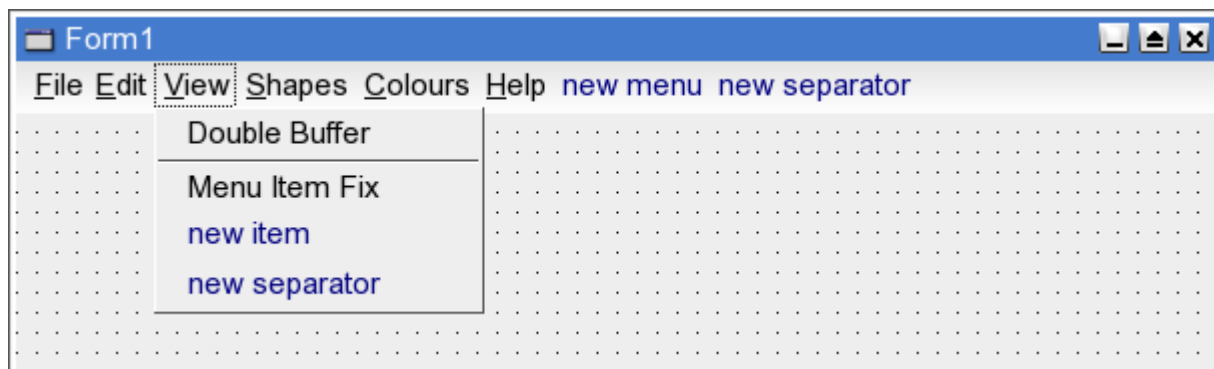


and to add menu items you do exactly the same thing on the menu.

Chapter 12 Drawing

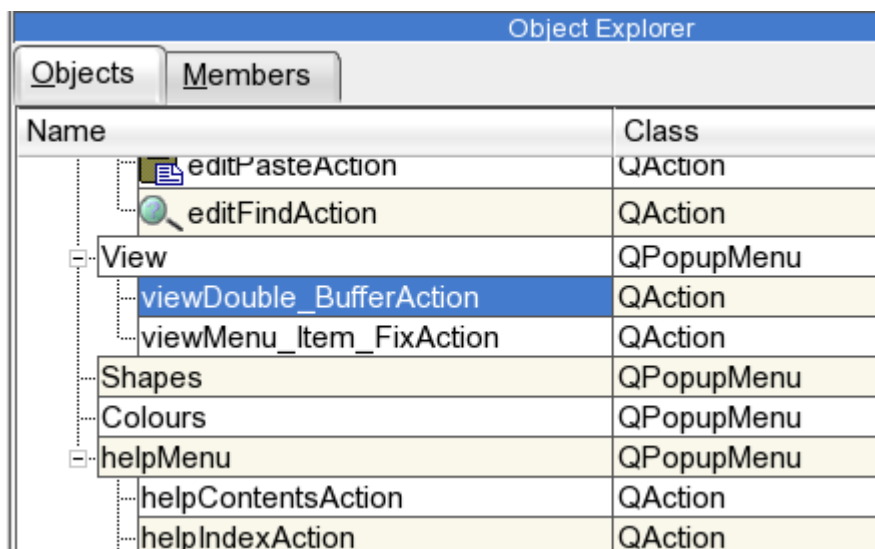


Positioning of the menus and the items on the menu is done by dragging them around the menu or menubar. So with a bit of practice we get,



It should be noted that the toggle action property shows up in the object editor so there is no problem setting the Double buffer and the Menu Item Fix options as check boxes the way they were in the previous program. The Shapes and the Colours Menu will have to be added to the menus by hand as they were in the ChapterTwelveDrawing example so that we can get the owner draw working correctly. This is done solely so that we can position the menu where we want them without having them showing up on the wrong side of the help menu.

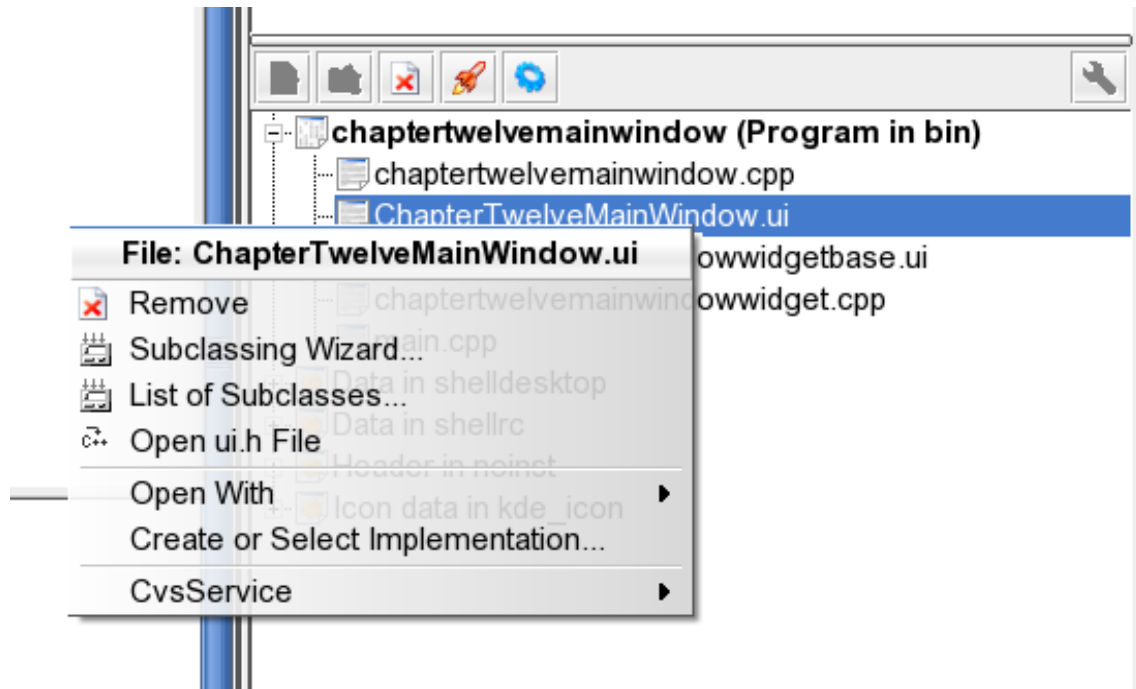
If we look at the Object Explorer we can see,



That the Actions have already been created for us, so all we have to do is connect them to an

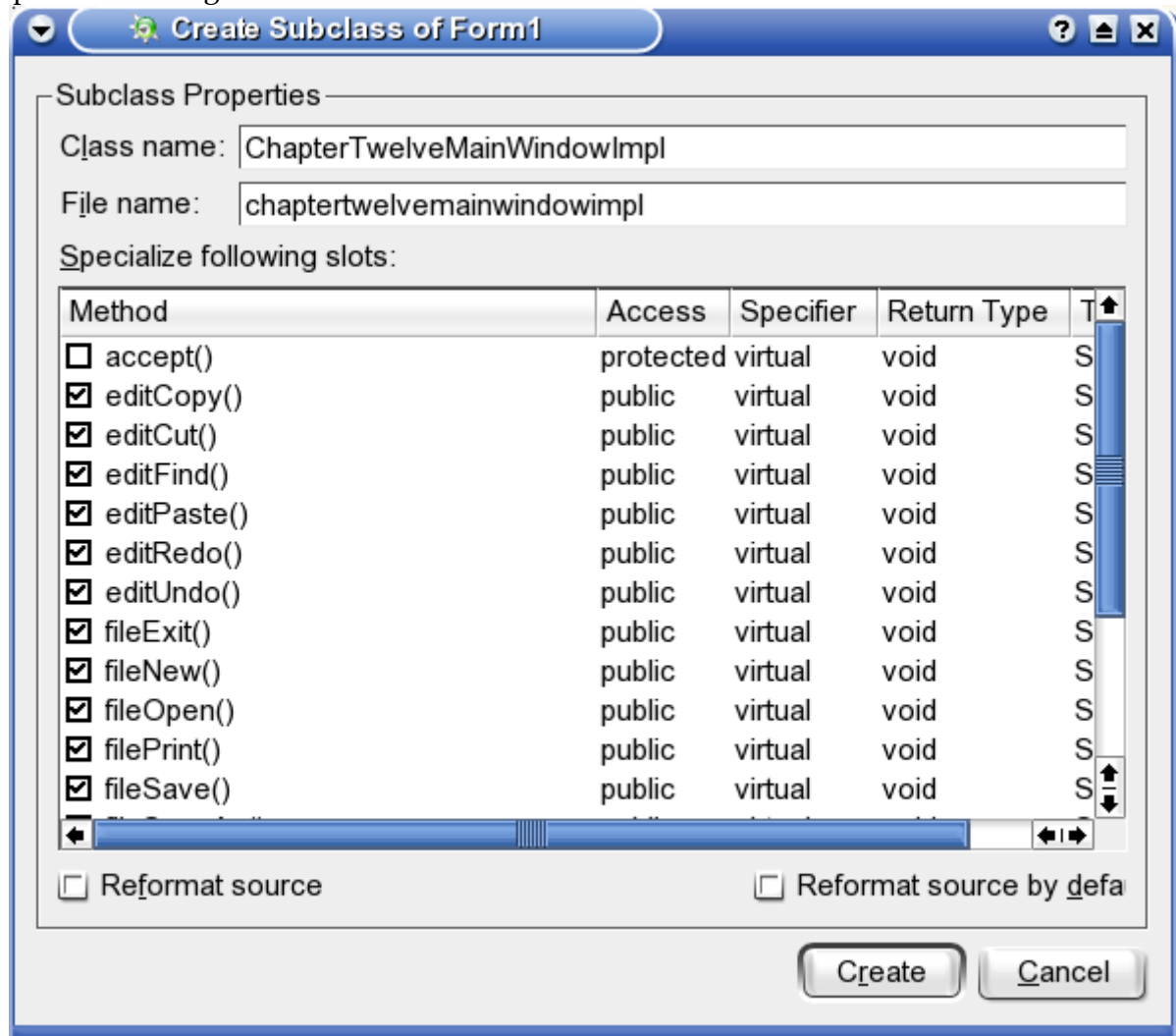
Chapter 12 Drawing
implemetation function.

At least that's the theory in practice if we try it we will notice that we haven't got an implementation class for the ChapterTwelveMainWindow.ui file so we shall have to create one. Open the Automake Manager and right click on the ChapterTwelveMainWindow.ui file,

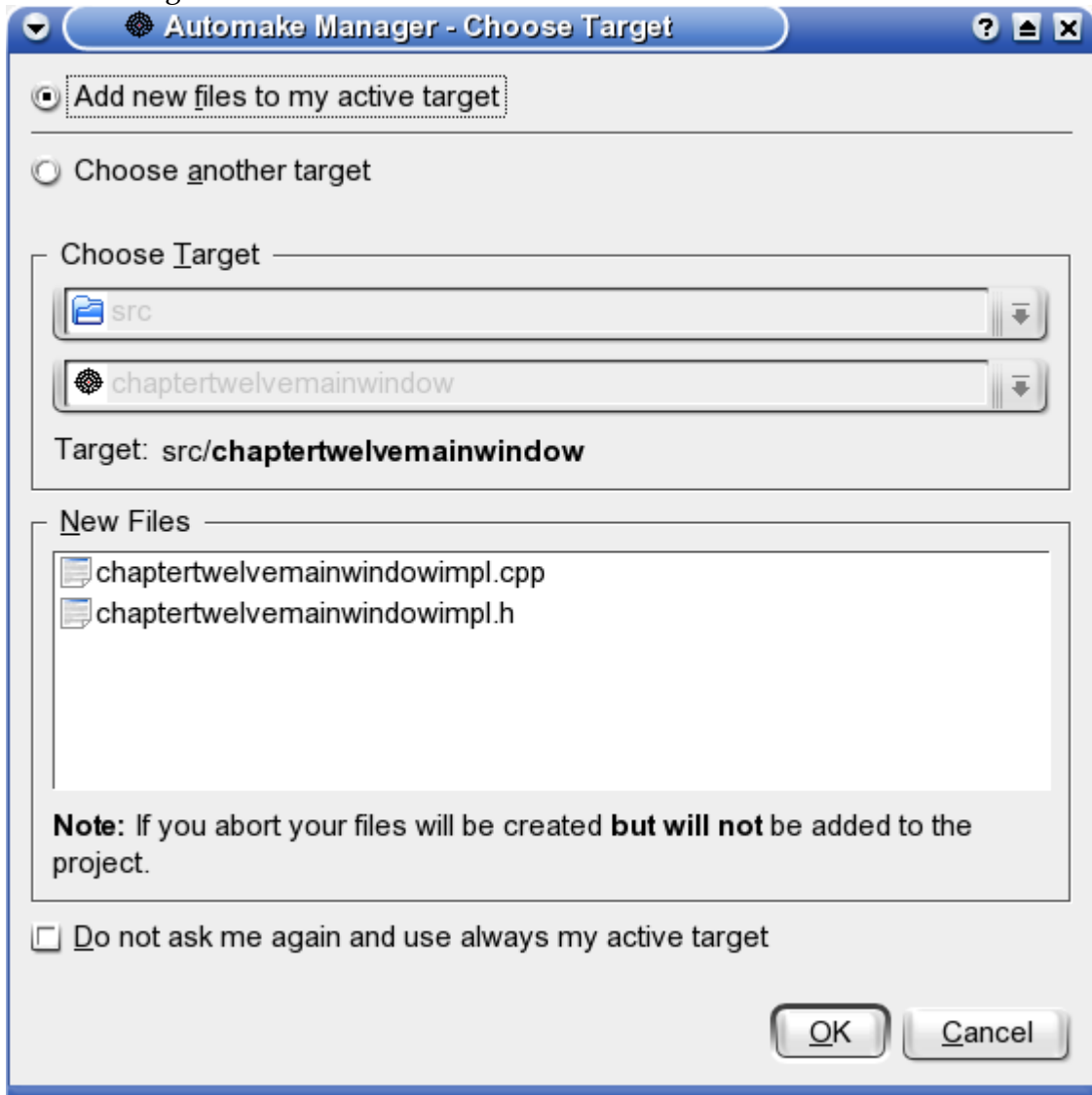


and select the Subclassing Wizard (I know we've been through this before but I figured it would be best to have the main window process all in one place as it will be more convenient.)

Chapter 12 Drawing



I've added "Impl", for implementation to the end of the file name just so that we don't end up with two files and classes named identically. Once we fill out the class name and hit create the confirmation dialog pops up,



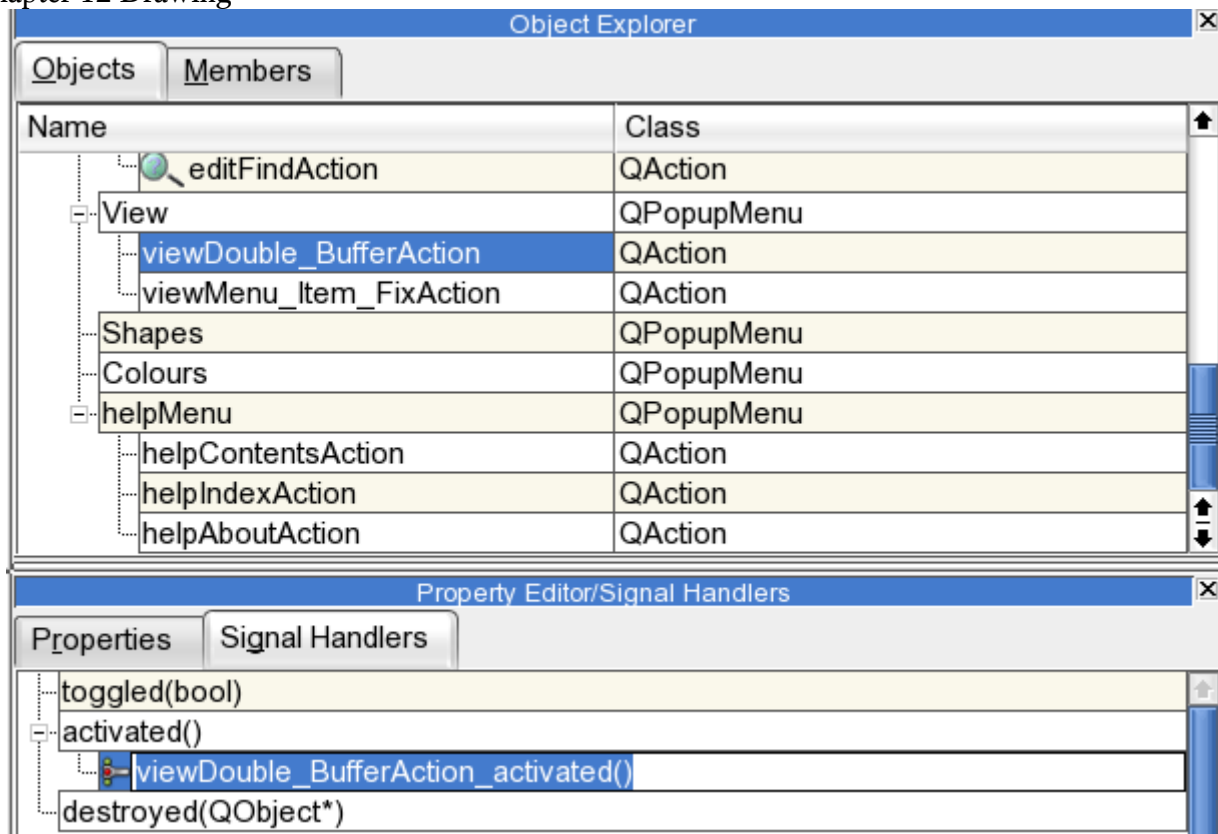
As we want to add this to the current project we can just click on O.K. and continue.

The generated class will be a standard implementation class that will also contain the headers and the empty implementation functions for the menu items that are already set up for us when we add the main window,

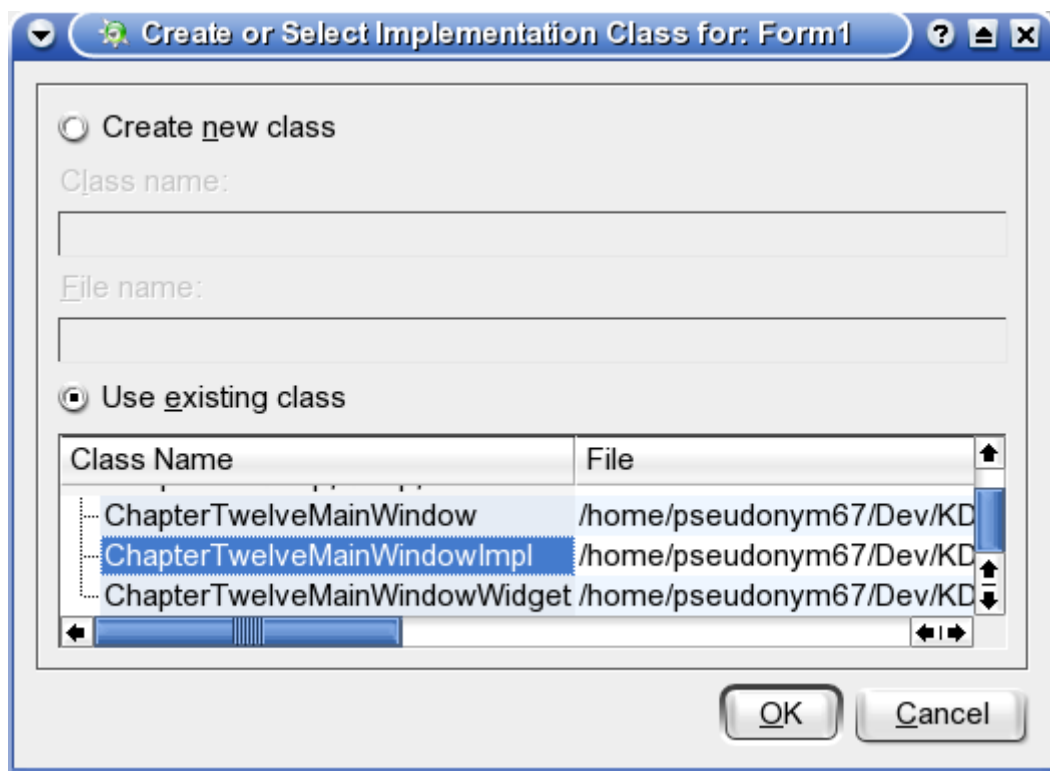
```
virtual void helpAbout();
virtual void helpContents();
virtual void helpIndex();
virtual void editFind();
virtual void editPaste();
virtual void editCopy();
virtual void editCut();
virtual void editRedo();
virtual void editUndo();
virtual void fileExit();
virtual void filePrint();
virtual void fileSaveAs();
virtual void fileSave();
virtual void fileOpen();
virtual void fileNew();
```

So all we have to do is implement the functions and connect up any menu items that we have added. We do this as always by selecting the menu item action we want to connect,

Chapter 12 Drawing



and then selecting the implementation file,



The main thing we need to do now is replace the standard widget with the main window ui that we have added to the project.

Chapter 12 Drawing

We do this in the main.cpp file, the part we are interested in reads,

```
KApplication app;
ChapterTwelveMainWindow *mainWin = 0;

if (app.isRestored())
{
    RESTORE(ChapterTwelveMainWindow);
}
else
{
    // no session.. just start up normally
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    /// @todo do something with the command line args here

    mainWin = new ChapterTwelveMainWindow();
    app.setMainWidget( mainWin );
    mainWin->show();

    args->clear();
}
```

All we do is change all the references for the ChapterTwelveMainWindow class to the ChapterTwelveMainWindowImpl class, so it reads,

```
KApplication app;
ChapterTwelveMainWindowImpl *mainWin = 0;

if (app.isRestored())
{
    RESTORE(ChapterTwelveMainWindow);
}
else
{
    // no session.. just start up normally
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    /// @todo do something with the command line args here

    mainWin = new ChapterTwelveMainWindowImpl();
    app.setMainWidget( mainWin );
    mainWin->show();

    args->clear();
}
```

You may notice that we haven't changed the text in the restore macro, there is a reason for this. To start with the restore macro is or was defined as,

```
#define RESTORE(type) { int n = 1;\
    while (KMainWindow::canBeRestored(n)){\
        (new type)->restore(n);\
        n++;}}
```

This functionality is entirely for KDE session management and has been replaced by templated functions with the one we would use defined as,

```
template <typename T>
inline void kRestoreMainWindows() {
    for ( int n = 1 ; KMainWindow::canBeRestored( n ) ; ++n ) {
        const QString className = KMainWindow::classNameOfToplevel( n );
        if ( className == QString::fromLatin1( T::staticMetaObject()->className() ) )
            (new T)->restore( n );
    }
}
```

Chapter 12 Drawing

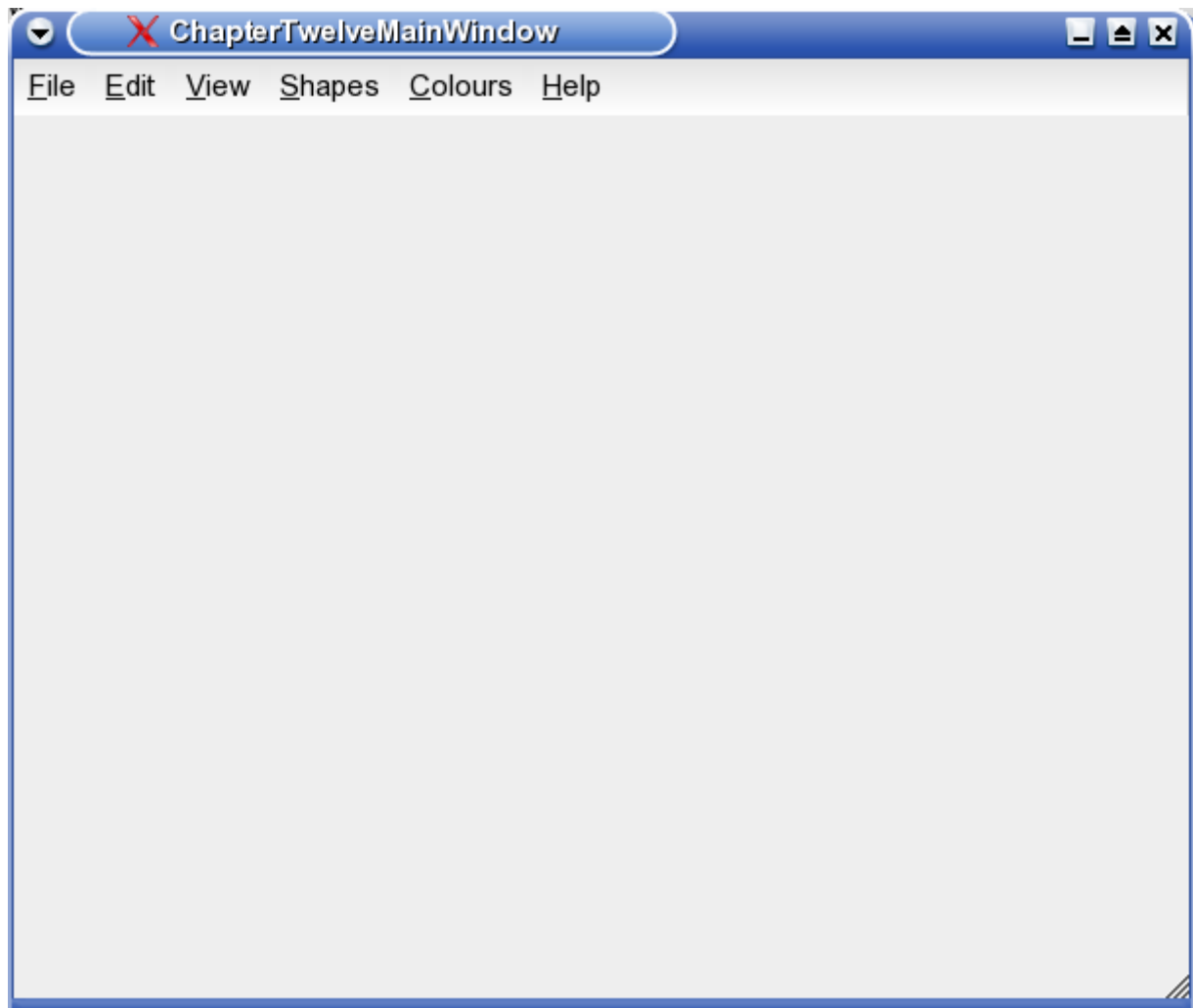
The first problem here is the one that is reported by the compiler if you actually try to change the `ChapterTwelveMainWindow` to `ChapterTwelveMainWindowImpl` in the call to the `restore` macro. That is that the `QMainWindow` class doesn't have a `restore` function. This is because the `ChapterTwelveMainWindowImpl` class in this project is derived from the `Form1` class which in turn is derived from `QMainWindow`. The `QMainWindow` class doesn't have the `restore` function as it is KDE functionality.

The `KMainWindow` class does have a `restore` function and if we were using a class derived from `KMainWindow` then there would be no problem, there are two ways of dealing with this. One we could wait until the coding for the project is finished and then when we know there is going to be no more work done we can edit the generated files so that the class derives from `KMainWindow`. Or alternatively we could just wait and ignore it until `KDevelop` gives us a way to specify the derived class for the generated files.

Then we include the header file.

```
#include "chaptertwelvemainwindowimpl.h"
```

and we're ready to go



All we need to do now is implement the program. the code itself is a mixture of the previous two programs in that we are using the menu code from the drawing example and the drawing code, complete with the double buffer option from the menu's example.

The only thing that we need to bear in mind is that some of the code is already written for use so if

Chapter 12 Drawing

we look in the ChapterTwelveMainWindow.h file we see that all the QActions have already been defined.

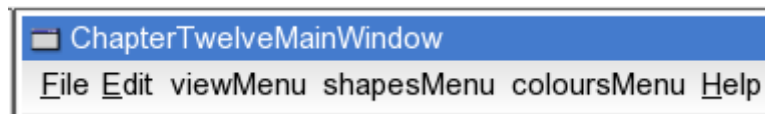
```
QAction* fileNewAction;  
QAction* fileOpenAction;  
QAction* fileSaveAction;  
QAction* fileSaveAsAction;  
QAction* filePrintAction;
```

Along with the required popups and the function connections are made in the ChapterTwelveMainWindow.cpp file.

```
// signals and slots connections  
connect( fileNewAction, SIGNAL( activated() ), this, SLOT( fileNew() ) );  
connect( fileOpenAction, SIGNAL( activated() ), this, SLOT( fileOpen() ) );  
connect( fileSaveAction, SIGNAL( activated() ), this, SLOT( fileSave() ) );
```

Setting The Menu Text

Unfortunately there is a minor problem with the gui editor for the menus in that it doesn't allow you specify the name for the QPopupMenu that will implement the menu you add. If you add a label for a Popup Menu as "Shapes" then the QPopupMenu object that is created will be called Shapes. A name which totally messes up our naming conventions so to fix it call the the menus by the names you want the variables to be.



Like so, and then in the constructor for the class use the menuBar to find the menu and change the name,

```
/// Set the text correctly for the menus  
if( menubar->findItem( 3 ) != 0 )  
    menubar->findItem( 3 )->setText( "&View" );  
if( menubar->findItem( 4 ) != 0 )  
    menubar->findItem( 4 )->setText( "&Shapes" );  
if( menubar->findItem( 5 ) != 0 )  
    menubar->findItem( 5 )->setText( "&Colours" );
```

now our variables will be listed in the generated ChapterTwelvemainWindow.h file as,

```
QPopupMenu *viewMenu;  
QPopupMenu *shapesMenu;  
QPopupMenu *coloursMenu;
```

Adding The Status Bar

Adding the status bar to a main window application is really simple the QMainWindow class contains a function called statusBar(). This function returns the status bar for the main window. If the status bar doesn't exist it creates it and any subsequent calls to statusBar() reference the newly created QStatusBar. The QStatusBar has one main function that you will use, this is the message function. The message function takes a string for the text to be displayed and if you want a number of milliseconds for the amount of time for the message to be displayed.

```
statusBar()->message( "Initialising", 5000 );
```

Chapter 12 Drawing

will display the message initialising for five seconds. it could be remembered that the status bar is to tell the user of the application what is going on so they have no interest in what function you have called or how fancy the code is. It should only display messages that are relevant to the user of the application. In this example application the status bar is only used to confirm selections that the user has made, giving a running comentary if you like on what the user is doing.

One thing you will notice on the status bar is that as you scroll through a menu the status bar text changes to reflect the currently highlighted item. So the when you highlight the File menu option New the status bar will show the text “New” and the short cut for this menu option. As we are using custom menus for the shapes and the colours menu this doesn;t work for us so we are going to have to fake it.

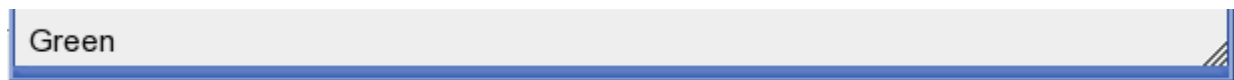
Simply do a connection to the QPopupMenu highlighted signal

```
connect( shapesMenu, SIGNAL( highlighted( int ) ), this, SLOT( shapesHighLighted( int ) ) );
```

and the implement the status bar text manually,

```
switch( id )
{
    case RectMenuItem: statusBar()->message( "Rect", 5000 ); break;
    case RoundRectMenuItem: statusBar()->message( "Round Rect", 5000 ); break;
    case EllipseMenuItem: statusBar()->message( "Ellipse", 5000 ); break;
    case LineMenuItem: statusBar()->message( "Line", 5000 ); break;
    case TriangleMenuItem: statusBar()->message( "Triangle", 5000 ); break;
};
```

In the code this is done for both the Shapes menu and the Colours menu.



Adding ToolBars

Back in the old days of programming a toolbar was a single item and it was considered design if you had a seperator between the different sections, nowadays each tool bar is made up of mulitple toolbars each of which contains the icons for it's own section and can be displayed or not at will. As we are using a main window we don;t have to worry about most of the details about how a toolbar works we can focus primarily on setting it up and making sure that it does and looks the way we want.

For a quick bit of background though a toolbar works as a docking window which can be slotted into place when required, think of it as a widget that can be moved around programatically, with the tool bar we just tell it to stick to the top section of the main window, and let the QMainWindow take care of the details of maintaining the main window space. i.e. making sure that something appearing at $x = 3$ $y = 3$ actually appears below the toolbar rather than behind or over it.

To start with then we have to declare some toolbars

```
KToolBar *fileToolBar;
KToolBar *editToolBar;
KToolBar *shapesToolBar;
KToolBar *coloursToolBar;
```

We define one for the file menu operations, one for the edit menu operations, one for the shape menu operations and one for the colours menu operations. Creating the toolbar should be what

Chapter 12 Drawing you'd expect by now.

```
fileToolBar = new KToolBar( this, "File Toolbar" )
```

Setting the options on the toolbar is actually really simple at this stage because we did all the required set up when defining the menu's well to be more accurate the options for the file and edit menu where all predefined for us with the main window but you can see the principal for your own menus. All you need to do to add buttons to the toolbar is add the QAction items that you set up for the menus so for the file tool bar we add the following actions,

```
fileNewAction->addTo( fileToolBar );  
fileOpenAction->addTo( fileToolBar );  
fileSaveAction->addTo( fileToolBar );
```

The more observant may notice that these aren't all the default options provided by the default implementation of a QMainWindow. The reason for this is that I've removed the items that don't really make sense for the current application, or as in the case of the print that I just couldn't be bothered with.

That is all you need to do as the setup for the action should contain all the information that is needed for the display and the function call from the menu setup.

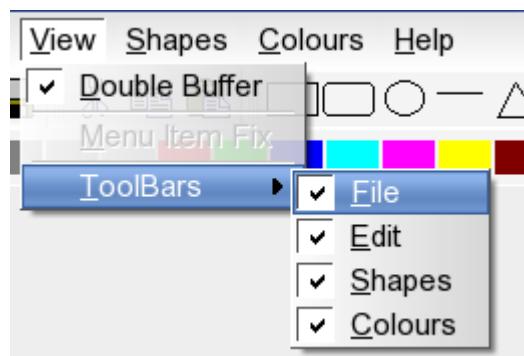
Adding the toolbars to the main window is done with,

```
addDockWindow( fileToolBar );  
addDockWindow( editToolBar );
```

which will give us,



We then add a ToolBars option to the view menu,



By default the four toolbars are turned on but the code to turn them off or back on again when the menu option is clicked is,

```
if( toggle == true )  
    coloursToolBar->show();  
else  
    coloursToolBar->hide();
```

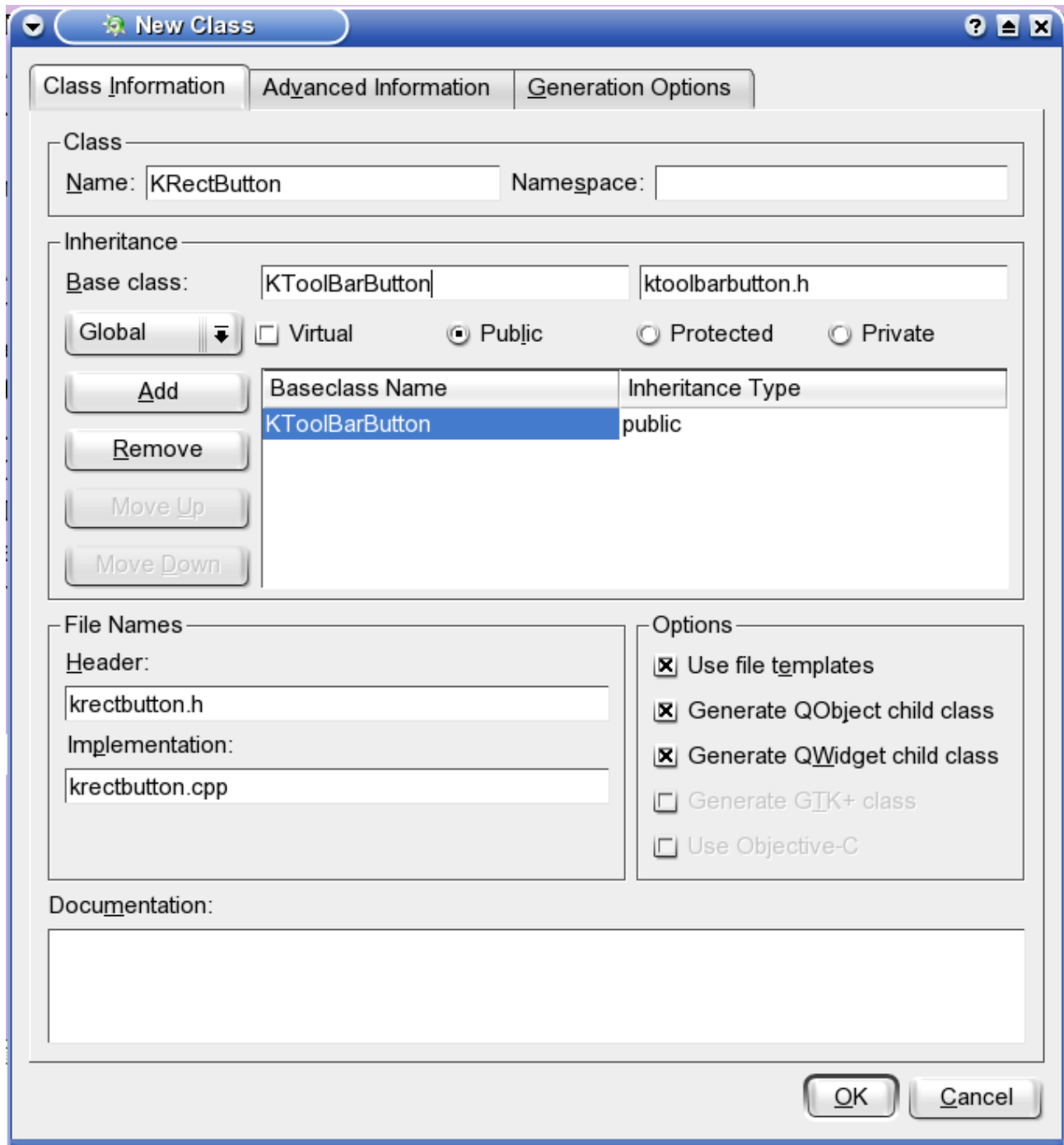
which is all really straight forward if we are just using standard menu items. Unfortunately we aren't using custom drawn menu items and being as it's a demonstration program we aren't smart

Chapter 12 Drawing

enough to just have the images drawn to a jpg file but are drawing them ourselves the hard way. So.

Custom ToolBar Buttons

We have to create our own custom toolbar buttons pretty much in the same way that we created the menu items in the first place. We start by adding a new class



As you can see we use the class wizard to do this which makes things much simpler than adding things by hand as it puts the files in the correct place and uses the correct naming conventions. In the above example we are defining a class called `KRectButton` which will be our custom implementation for the rect shape in the same way that we previously implemented the drawing of

Chapter 12 Drawing the rect on the menu.

When using the class wizard if you select the Generate QWidget child class option then you can add the base class information as shown in the dialog. Once you have the class set up correctly then you hit OK and accept the confirmation dialog that follows, the class will look something like this.

```
#ifndef KRECTBUTTON_H
#define KRECTBUTTON_H

#include <ktoolbarbutton.h>

/**
 * @author pseudonym67
 */
class KRectButton : public KToolBarButton
{
    Q_OBJECT
public:
    KRectButton(QWidget *parent = 0, const char *name = 0);

    ~KRectButton();

protected:
    virtual void drawButton( QPainter *painter );
};

#endif
```

All I have added is the drawButton function override so that we can draw the required shape when required. The code for the function as you should expect by now is,

```
painter->drawRect( 2, 5, width()-2, height()-10 );
```

One thing that you should notice is the constructor which reads,

```
KRectButton::KRectButton(QWidget *parent, const char *name)
    : KToolBarButton(parent, name)
{
    setText( "Rect" );
}
```

There is an important reason for drawing your attention to this and that is that if you derive the class from QToolButton and dont put any text here then the button when displayed will be about three pixels wide. If you derive the class from KToolBarButton as we have and don't put any text here then the button will not display at all.

The text itself is displayed if you hold the mouse over the button, like so.



As with the custom menu items the rest is just a matter of repeating the same set up for the different drawing and then using a single class to draw the filled rectangles for the colours.

As you'd expect though setting up the buttons properly is a bit more long winded than it is with the standard implementations.

Chapter 12 Drawing

First we create the button objects using,

```
shapesToolBar = new KToolBar( this, "Shapes Toolbar" );
rectButton = new KRectButton( shapesToolBar );
roundRectButton = new KRoundRectButton( shapesToolBar );
ellipseButton = new KEllipseButton( shapesToolBar );
lineButton = new KLineButton( shapesToolBar );
triangleButton = new KTriangleButton( shapesToolBar );
```

Note here that while we set the parent of the toolbar as the main window we set the parent of the buttons as the toolbar itself.

Next we add the Buttons to the toolbar with,

```
shapesToolBar->insertWidget( RectWidgetItem, 20, rectButton );
shapesToolBar->insertWidget( RoundRectWidgetItem, 20, roundRectButton );
shapesToolBar->insertWidget( EllipseWidgetItem, 20, ellipseButton );
shapesToolBar->insertWidget( LineWidgetItem, 20, lineButton );
shapesToolBar->insertWidget( TriangleWidgetItem, 20, triangleButton );
```

Using the insertWidget function which takes an id the suggested size and the widget itself. This function allows you to add just about any widget you want to the toolbar though obviously you should only add widgets that make sense to be used in a toolbar context.

Once the Buttons are added to the toolbar we add the toolbars to the window.

```
addDockWindow( shapesToolBar );
addDockWindow( coloursToolBar );
```



Of course at the moment they don't do anything so we have to write some connections.

First we want to display a text message to the status bar when the button is highlighted and this is where we hit a minor snag in that the value used as the id in the call to insertWidget isn't the one that we get when we connect to the highlight function,

```
connect( blackButton, SIGNAL( highlighted( int, bool ) ), this,
        SLOT( blackButtonHighLighted() ) );
connect( whiteButton, SIGNAL( highlighted( int, bool ) ), this,
        SLOT( whiteButtonHighLighted() ) );
```

So we just add a new function and add the message to the status bar ourselves,

```
void ChapterTwelveMainWindowImpl::blackButtonHighLighted()
{
    statusBar()->message( "Black", 5000 );
}
```

Once this is done all we need to do is connect up the buttons with the correct functionality.

```
connect( blackButton, SIGNAL( buttonClicked( int, Qt::ButtonState ) ), this, SLOT(
        blackButtonClicked() ) );
connect( whiteButton, SIGNAL( buttonClicked( int, Qt::ButtonState ) ), this,
        SLOT( whiteButtonClicked() ) );
```

As with the previous call to connect we just add a function and then call the required function ourselves with the correct parameter.

```
void ChapterTwelveMainWindowImpl::blackButtonClicked()
{
}
```


Chapter 12 Drawing

```
coloursMenuSelected( BlackColourMenuItem );  
}
```

And now everything is wired up the only thing left to do is implement the code for the pregenerated file and edit menu options.

The Generated Functions

One of the first thing you are going to want to do when adding menu items is remove one, because it isn't required for the program or because you didn't add it in the right place or duplicated functionality. The problem is that through the gui you can add and edit menu items but you can't delete them so the easiest way to do it is to close KDevelop and open the .ui file in Kxml Editor.

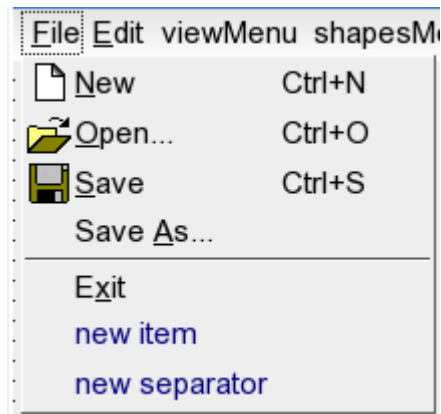
The screenshot displays the KXmlEdit interface. On the left, a hierarchical tree view shows the structure of a Qt Designer .ui file. The tree includes elements like 'height', 'property', 'string', 'menubar', 'property', 'cstring', 'item', 'action', and 'separator'. One 'action' element is highlighted in blue. On the right, a table lists the properties of the selected element. The table has three columns: 'Namespace', 'Name', and 'Value'. The first row shows '1' in the 'Namespace' column, an empty cell in the 'Name' column, and 'filePrintAction' in the 'Value' column. Below the table, the XML code for the selected element is shown: `<action name="filePrintAction" />`.

| | Namespace | Name | Value |
|---|-----------|------|-----------------|
| 1 | | name | filePrintAction |

```
<action name="filePrintAction" />
```

Select the item you want and then simply delete it. Open up KDevelop again and you'll see,

Chapter 12 Drawing



At this point you should delete the filePrint function from the header file and the source file although they won't cause any trouble if you don't there's no way to activate them now. It's just neater to get rid of them.

The edit menu is deleted in its entirety from the final menu which just gives us the file menu and the help menu as the menu's that we haven't set up ourselves. The file menu we have set up previously and the code is almost identical to earlier versions.

Help

One thing that anyone moving from Windows to Linux will miss almost instantly is the unified help system on windows. On Windows there is only one way to do help and anything that doesn't follow this way is an anomaly rather than the rule. On Linux the help is basically all over the place. KDE goes some way to fix this with the DocBook idea but this is not exactly user friendly and there are no real tools for generating the files, there are plenty of tutorials to be found but essentially it boils down to hand coding the xml tags yourself.

This is far from ideal from the point of view of a new developer writing their latest and greatest idea and then finding they get bogged down in writing the help, so we'll sort of come to a compromise. Basically a DocBook is a html file that has some xml style tags in it to set up formatting etc but and this is the important bit you need to write the html file first. So to start with we'll use a html viewer to display help pages that are written and then gently suggest that if the writer wants the application to be included as part of KDE then they should look up DocBooks or get someone who knows how to do it to edit the help pages.

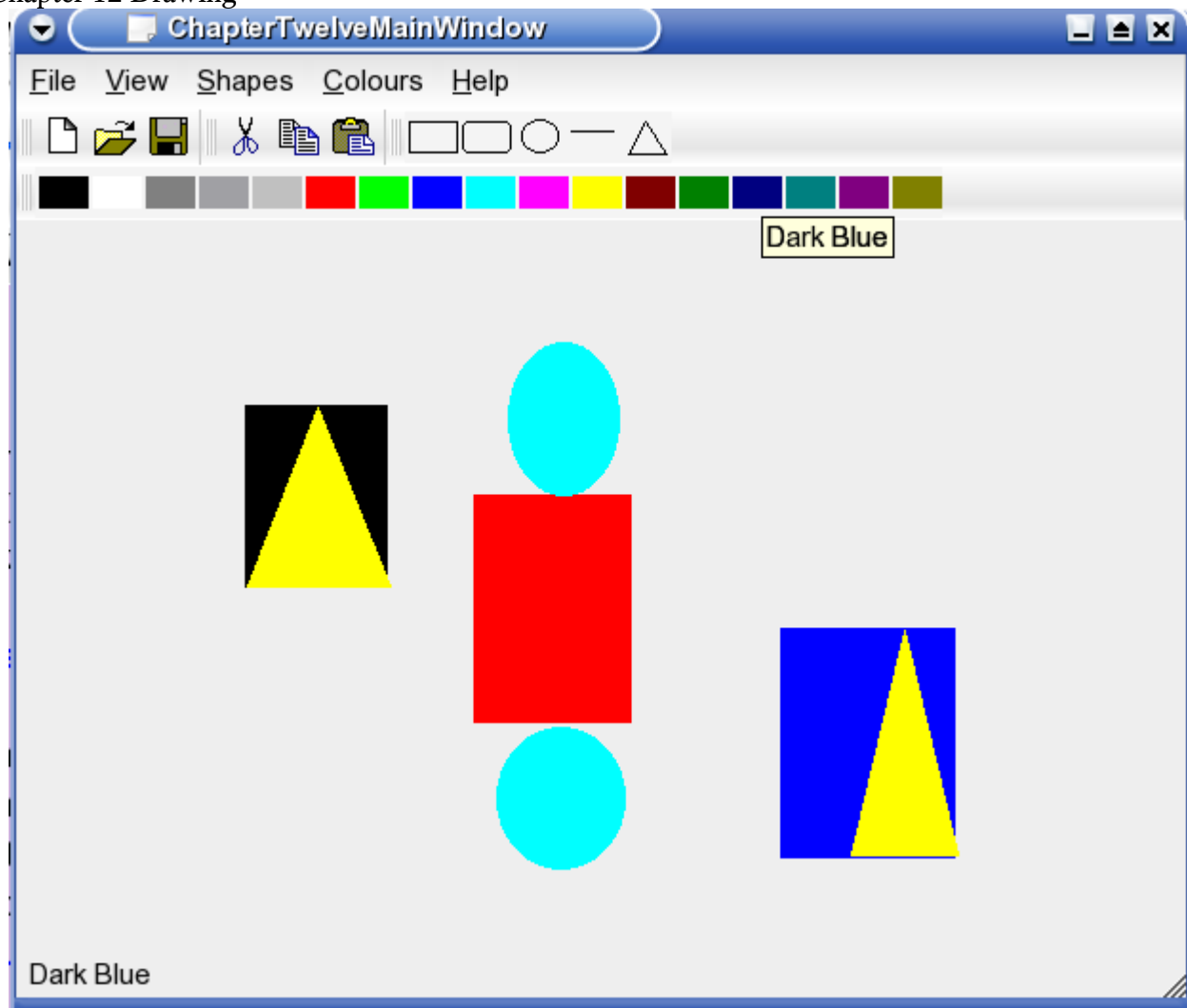
If you do a search for "Simple HTML Help Browser" in the help you'll find a Qt class that the help window with this project is based on. It is a very basic html browser that uses the QTextBrowser class to display the html so it is limited in what it can do.

This version of the class has been changed to use KDE classes more than the default Qt classes and some of the minor functionality has been changed. For example it no longer allows you to open new help windows and closing the help window doesn't shut down the application anymore. If you wish to check the nature of the changes the source is available for both the original project in the help and for the KDE version in the ChapterTwelveMainWindow source code.

Chapter 12 Drawing



The help window will display pretty much any basic html files the examples shown were simply written in open office and saved as html files.



Summary

In this chapter we have started with the basics required for a simple drawing application and built upon it so that we ended up using standard and custom menus, standard and custom toolbars and in the process have gone from using only dialog based applications to having all the the tools we need to create full working programs, complete with help.

Chapter 13 : Global Information and Configuration Files

In the previous chapter we used a function called BarIconSet which loads the standard icons from a program based on what KDE already knows about the system and where to find things on that system. In this chapter we will look deeper into just what KDE knows about the system and how we can use it as well as looking at some files types in more detail.

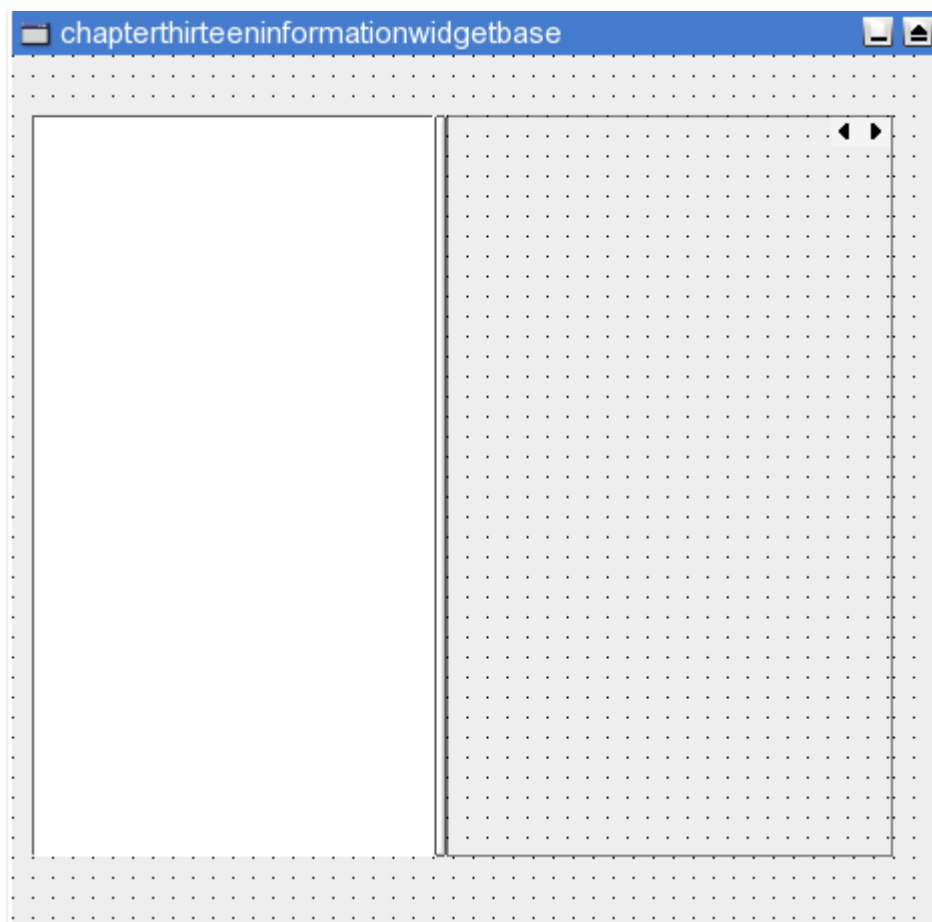
To start with though if we're going to be looking in more detail about KDE specific things that it's only right that we use a proper KDE application with which to do it.

KDE Information

The first project in this chapter is going to look at what information is available to us about the KDE environment in which we are working. So first of all we set up a Simple Designer Based KDE application as before and then completely ignore the widget.

Then main reason for this is because we want to use a splitter to seperate the two parts of the application. Another reason for this is that we want to use the KMainWindow directly, this gives us access to toolbars and status bars and the menus if we need them. Although for this project we will basically just be using the status bar.

Splitter Windows

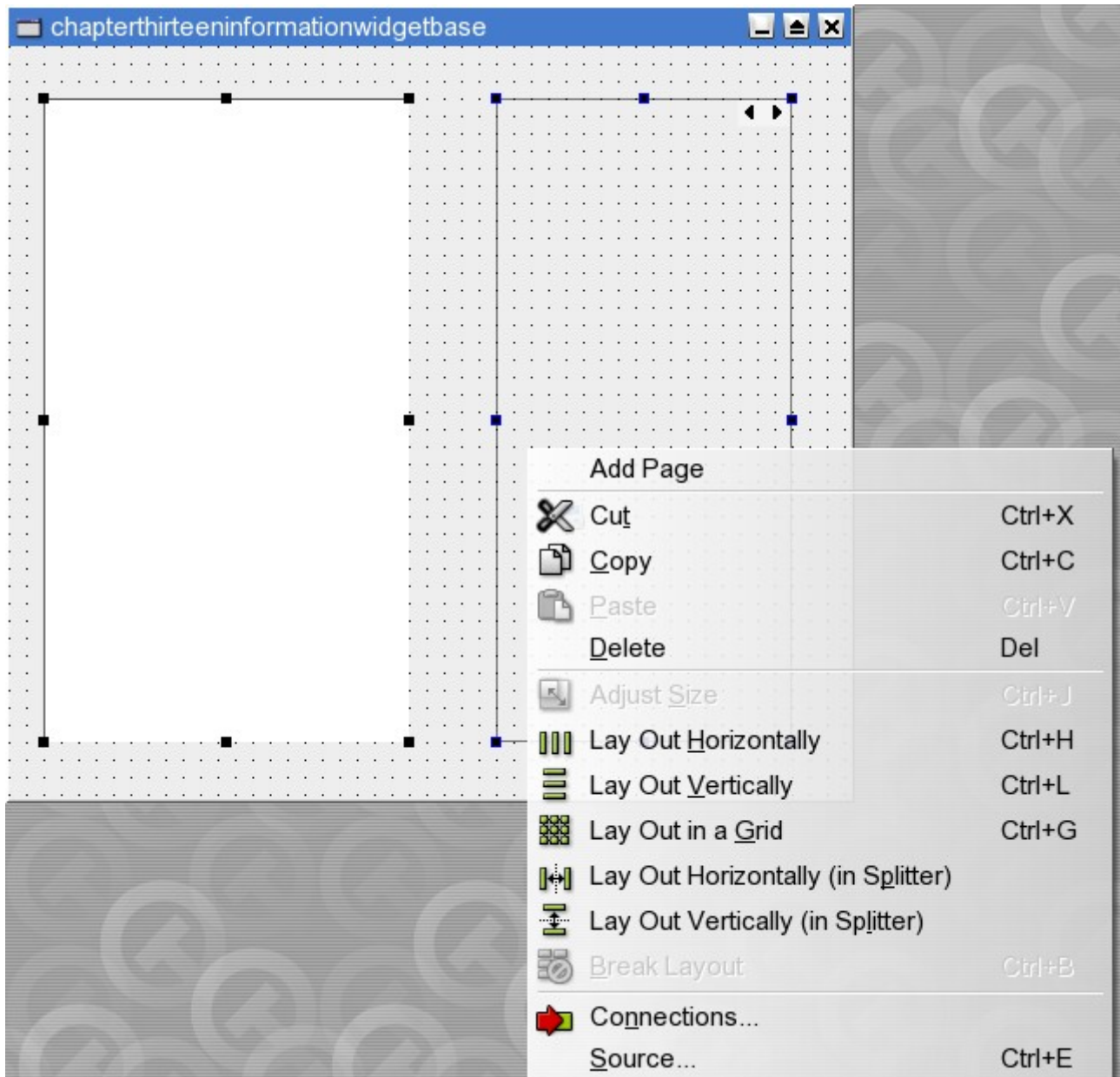


A Splitter Window will divide two or sections of the widget from each other as you can see above.

Chapter 13 : Global Information and Configuration Files

It will also maintain any sizes for widgets that are placed within it which is the main reason why we are ignoring the widget for this application and writing all the code in the generated KMainWindow file.

To add a splitter you hold down the left mouse and drag it over the widgets and then right click to get,



If you add a Splitter to the widget as shown above you get the following code in the generated ChapterThirteenInformationWidgetBase.cpp file

```
splitter1 = new QSplitter( this, "splitter1" );  
splitter1->setGeometry( QRect( 10, 30, 430, 370 ) );  
splitter1->setOrientation( QSplitter::Horizontal );
```

This is fine up to a point and the point is that in the original ChapterThirteenInformation.cpp file

Chapter 13 : Global Information and Configuration Files
you have the line.

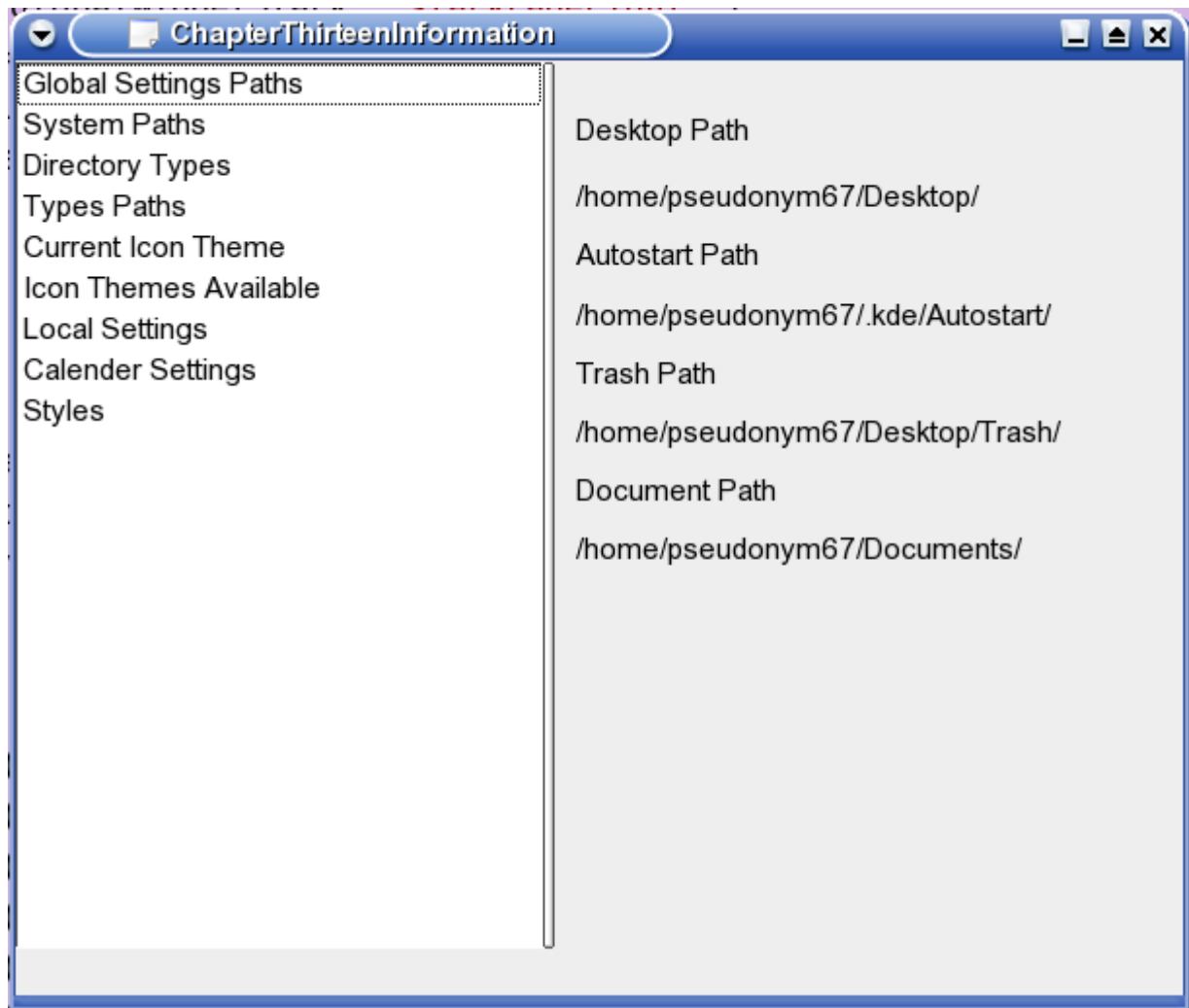
```
setCentralWidget( new ChapterThirteenInformationWidget( this ) );
```

What is happening here is that the Widget is set as the central widget and the splitter is a child of the widget. This means that when the widget is resized we have to mess around with layouts to get the splitter window to do its work properly as it is dependant on it's parent for it's sizing instructions. If instead we use the code.

```
splitter = new QSplitter( this, "splitter1" );  
splitter->setGeometry( QRect( 0, 15, 580, 410 ) );  
splitter->setOrientation( QSplitter::Horizontal );  
setCentralWidget( splitter );
```

In the ChapterThirteenInformation.cpp file what we are doing is making the splitter the central widget which means the splitter will then automatically take care of any resizing issues leaving us to concentrate on making the code work.

This means that the widget will look exactly the same as you would expect the above to look,



Because a splitter window is used as the main widget the application doesn't really require a menu bar and a toolbar as all the available options are made available through the KListBox on the left

Chapter 13 : Global Information and Configuration Files

hand side and the options selected are displayed through the use of a QWidgetStack on the right hand side. Essentially the application is nothing more than a displaying of the various paths and options that are readily available to KDE. Naturally the application is very selective and should only be taken as a starting point for system information rather than as the definitive guide.

The application essentially uses KGlobal's static functions to get its information.

```
static KInstance * instance ()
static KStandardDirs * dirs ()
static KConfig * config ()
static KSharedConfig * sharedConfig ()
static KIconLoader * iconLoader ()
static KLocale * locale ()
static KCharsets * charsets ()
static const QString & staticQString (const char *str)
static const QString & staticQString (const QString &str)
static void registerStaticDeleter (KStaticDeleterBase *d)
static void unregisterStaticDeleter (KStaticDeleterBase *d)
static void deleteStaticDeleters ()
static void setActiveInstance (KInstance *d)
static KInstance * activeInstance ()
```

We mostly use the instance which gives us the current KInstance, the dirs function which gives us the standard directories the iconLoader function which gives us the information about the current icon theme and the KLocale which gives us time and date and currency information. So the application is just a matter of setting up the gui and then displaying the information for the separate parts. A standard piece of the setup code would be,

```
stackPageFour = new QWidget( globalWidgetStack, "stackPageFour" );
```

```
typesPathList = new KListBox( stackPageFour, "typesPathList" );
typesPathList->setGeometry( QRect( 10, 15, 300, 420 ) );
QHBoxLayout *pageFourLayout = new QHBoxLayout( stackPageFour );
pageFourLayout->addWidget( typesPathList );
globalWidgetStack->addWidget( stackPageFour, 3 );
```

This sets up the stack page and KListBox to display the data. The list box is then placed in a layout so that the layout will take care of any resizing of the application and the stack page is added to the QWidgetStack. That's it the icon theme gets a little more complicated with adding labels but there's nothing that should be hard by now.

When an item is selected in the list on the left the correct page is shown and the information is filled in from scratch every time. So a sample would be,

```
globalWidgetStack->raiseWidget( 5 );
iconThemesList->clear();
```

```
QStringList list = KIconTheme::list();
```

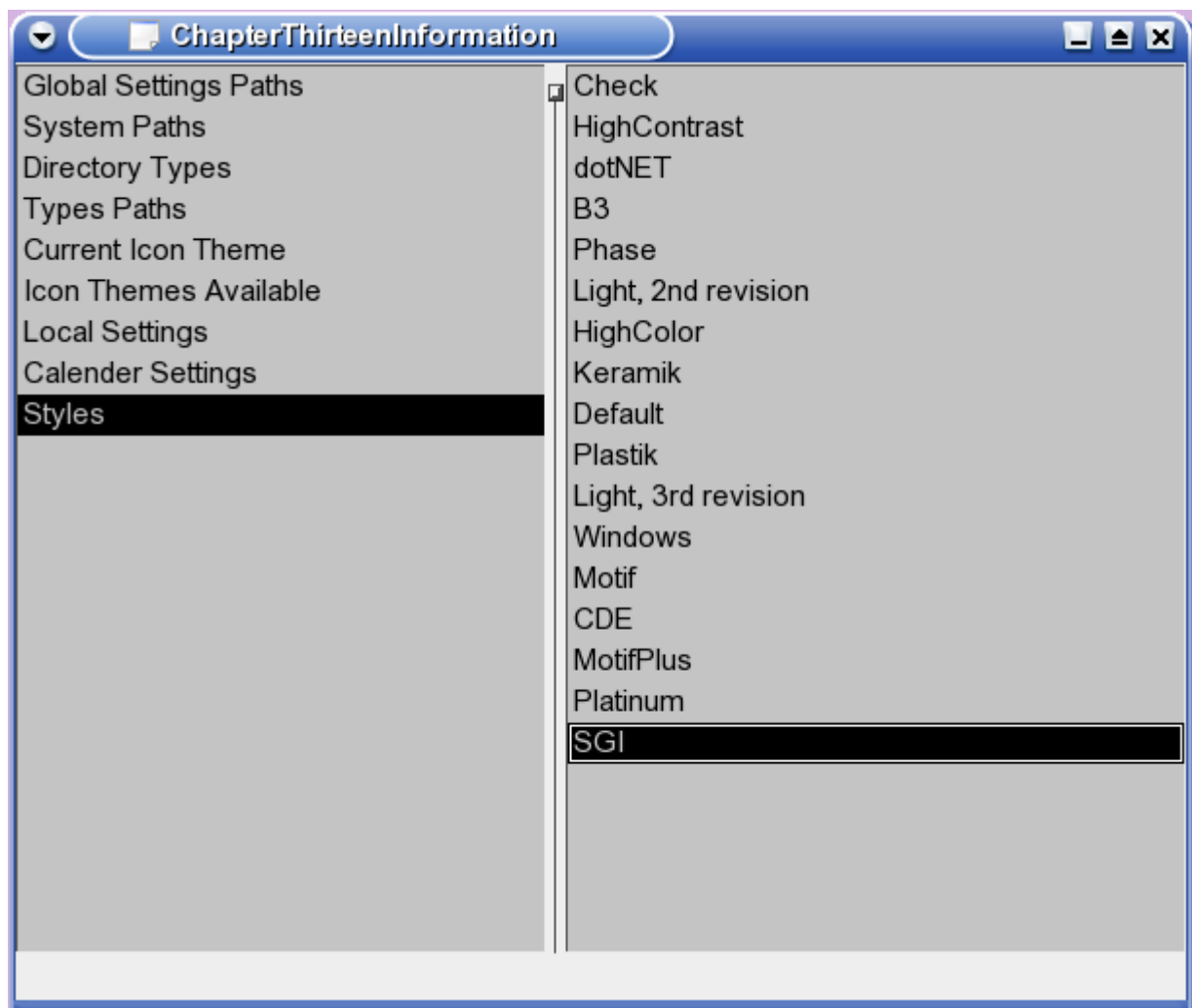
```
for( QStringList::iterator it = list.begin(); it != list.end(); it++ )
{
    iconThemesList->insertItem( *it );
}
```

In this section we raise the widget to display the current icon themes on the system and then use KIconTheme to get the list, for the rest we just iterate through the list adding the items to the list box.

Chapter 13 : Global Information and Configuration Files

I've also included styles in the list as when looking at it even I was thinking it was needed because the application looked half done without them, so even though we are technically within the Application domain they are included here.

When you click on Styles in the left list box you will get a list just like some of the other options on the difference here is that when you click on the style in the right list box that style will be set as the application style. Don't expect miracles though there are a number of items that affect the look of the gui and this just sets how the individual widgets items are drawn, so for most of them there will only be a slight change to the drawing of the splitter bar in the center (or wherever you have move it) of the application. One of the more noticable changes is if you select the SGI style, as shown below.



Tip Of The Day

When using layouts programatically rather than through the gui you may be tempted to try something like this,

```
QVBoxLayout *pageFiveBasicLayout = new QVBoxLayout( stackPageFive );
QHBoxLayout *pageFiveChildLayout = new QHBoxLayout( stackPageFive );
```

Basically what you are trying to say is that you want to use the two layouts on the current widget in this case stackPageFive. This is completely and utterly wrong. The general effect of this when you

Chapter 13 : Global Information and Configuration Files

run the program is to create two layout areas, both of the current size of the parent widget in this case `stackPageFive`, which expands your widget to twice it's size.

The correct way is,

```
QVBoxLayout *pageFiveBasicLayout = new QVBoxLayout( stackPageFive );
QHBoxLayout *pageFiveChildLayout = new QHBoxLayout( pageFiveBasicLayout );
```

Create one layout which is the parent for any other layout's on the widget, you can have as many child layout's as you wish you can even use the `pageFiveChildLayout` as a parent for another layout. But if you stick to the one main layout for the widget and the rest as children of the layout you should save yourself some headaches.

Application Settings With KDE

Configuration files are usually stored as matched pair settings, by matched pair I mean something like,

MenuColour Red

The traditional way to save an applications settings in KDE was to simply add an rc to the name of the application and save the settings in that file. So A project that was called `ChapterThirteenTest` would save it's config file as `chapterthirteentestrc`. This of course isn't entirely adhered to and you will find that some newer applications use the extension `.rc` and even some files don't use the rc identifier at all. `kdeglobals` being one example. As most applications use the rc extnesion we'll take that as the default. On a KDE 3.x system these are stored in `opt/kde3/share/config` a selection from the `kdeveloprc` file reads,

```
MinConditional=-1
PadOperators=false
PadParentheses=false
Style=UserDefined
```

[Editor]

```
EmbeddedKTextEditor=Embedded KDE Advanced Text Editor Component
```

[General Options]

```
Read Last Project On Startup=true
```

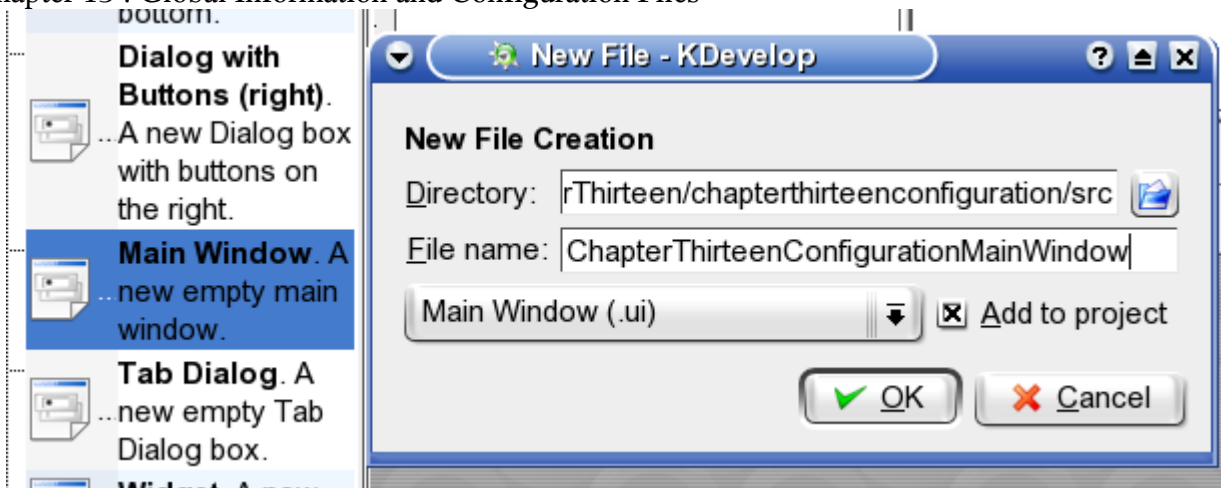
[Mainwindow]

```
Height 768=548
```

```
Width 1024=735
```

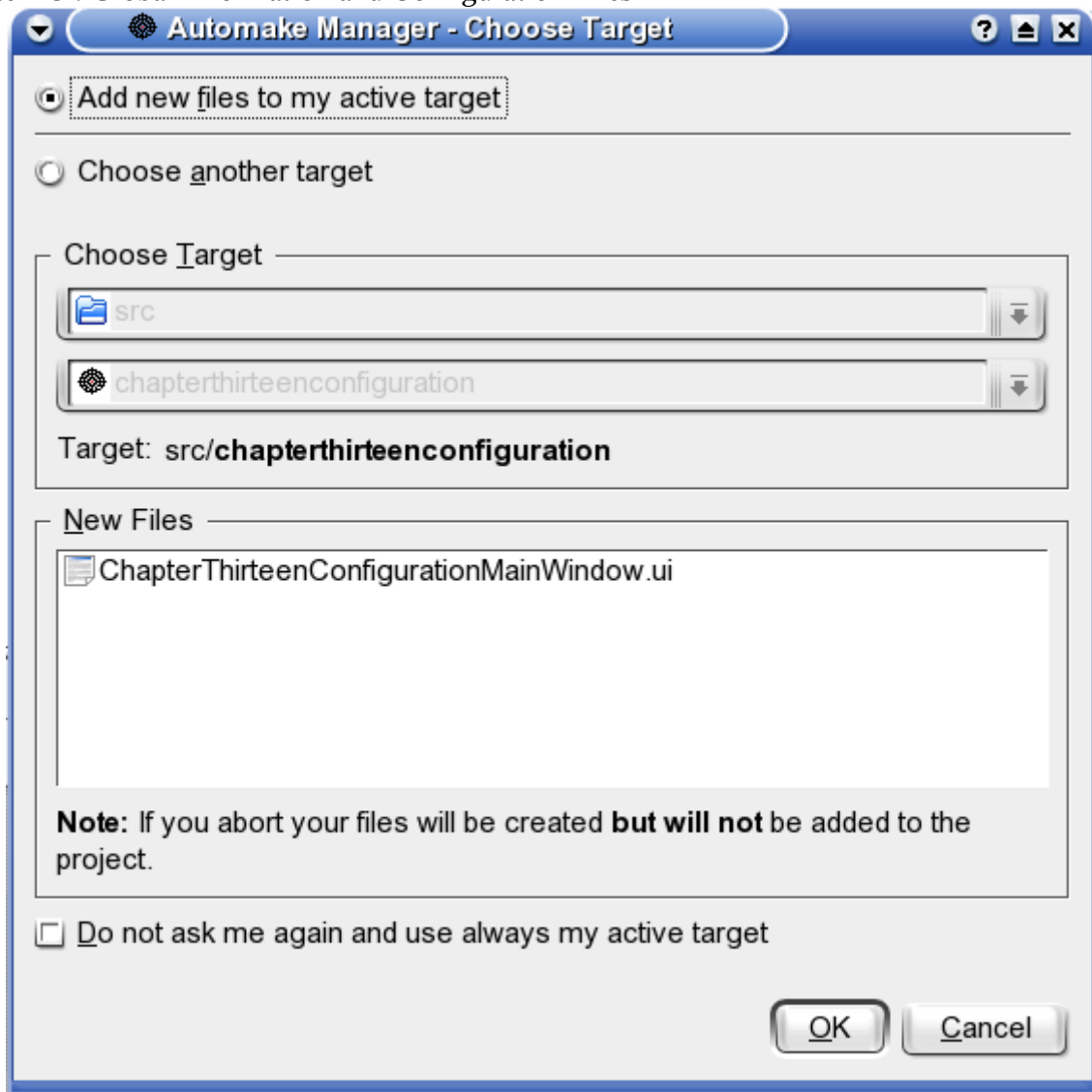
So to get us going with configuration files we'll write a little program that can edit and save these files. First of all though we need to create a project in the usual way called `ChapterThirteenConfiguration` and add a main window to it

Chapter 13 : Global Information and Configuration Files

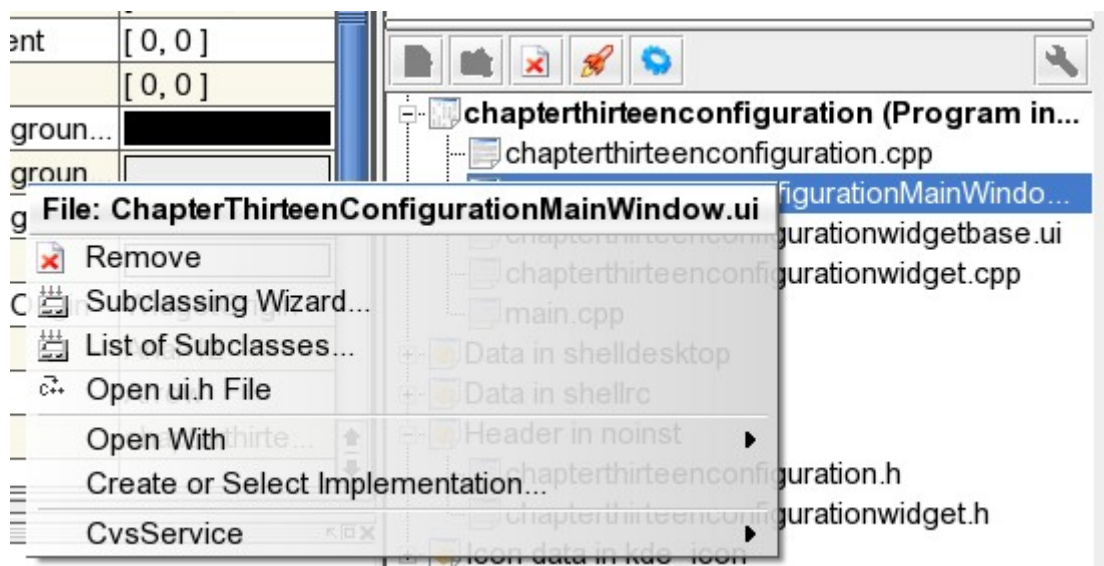


There is a simple reason for adding a new main window to the project when we already have one created for us in the setup and that is that by creating a new window we get all the file menu options and the associated images code generated for free. You could always, of course, generate a main window in another temporary project and simply copy the generated code across to the project's main window.

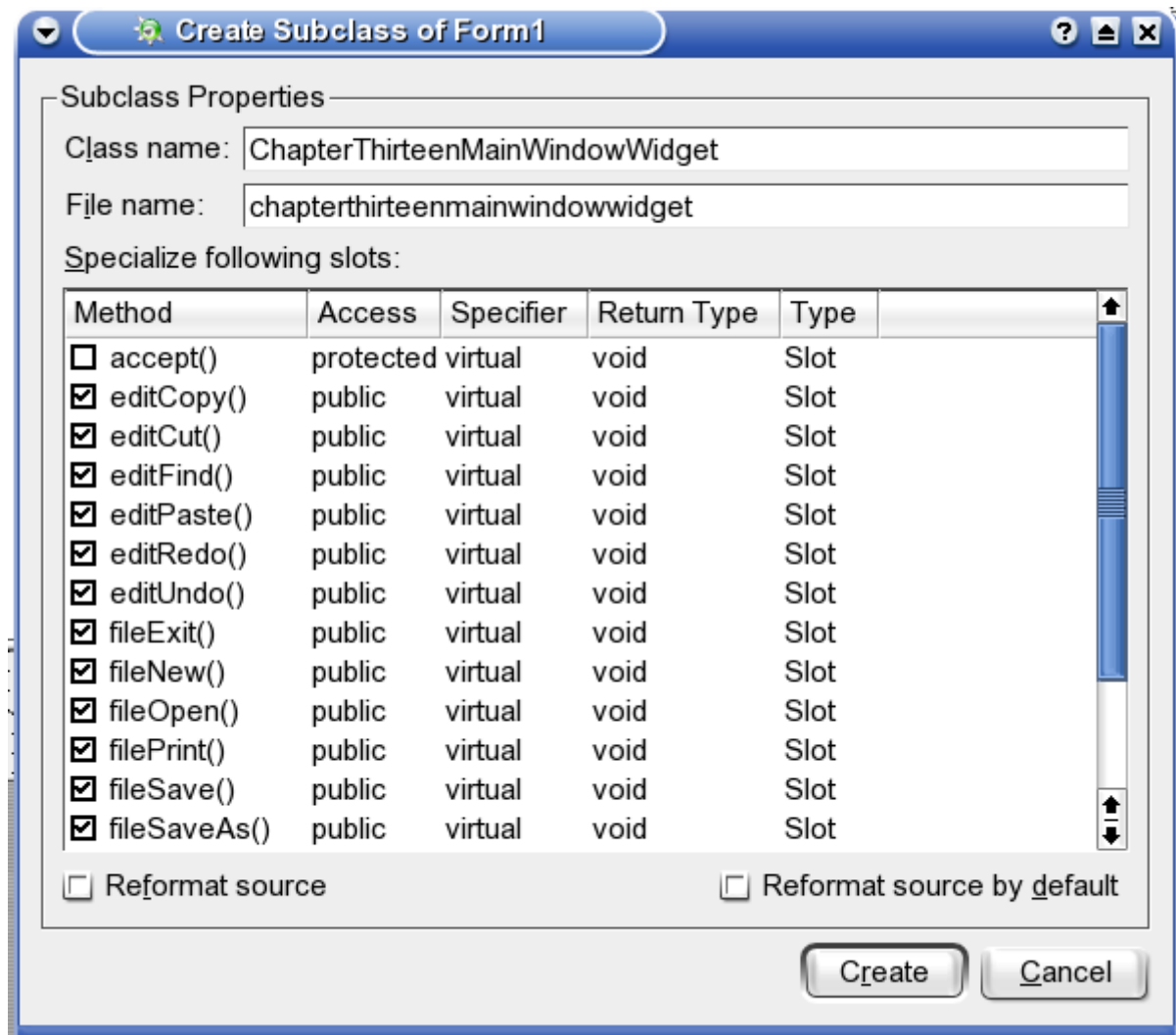
Once we have filled in the name as above we get the add to target dialog.



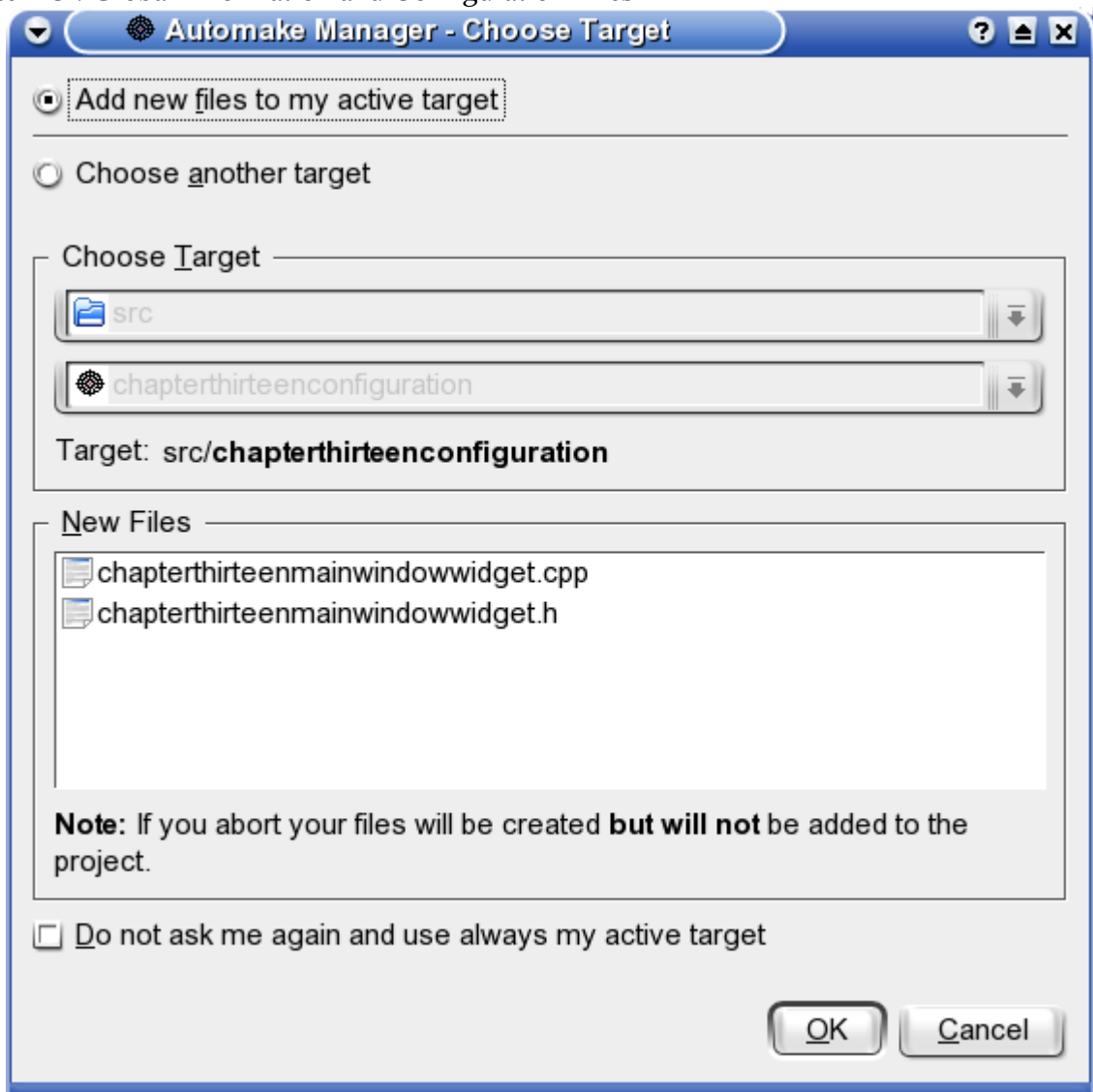
which we can just click through and then right click on the ChapterThirteenMainWindow.ui file in the Automake manager and choose Subclassing Wizard.



Chapter 13 : Global Information and Configuration Files
Which will then ask us to name the class,



We will then be asked to confirm our choices,



This will create the class and the header files for the main window which we can treat largely as if it was just any other widget with more functionality provided by the menu's and toolbars than we would get with a standard widget.

Now in a perfect world we would just change the references in main.cpp from using the ChapterThirteenConfiguration class to using the ChapterThirteenMainWindowWidget class, this won't work though because when the generated files for the ChapterThirteenMainWindowWidget class will derive the class from QMainWindow which doesn't have the restore function and not KMainWindow which does, so we leave main.cpp exactly as it is.

```
ChapterThirteenConfiguration *mainWin = 0;
```

```
if (app.isRestored())
{
    RESTORE(ChapterThirteenConfiguration);
}
else
{
    // no session.. just start up normally
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    /// @todo do something with the command line args here

    mainWin = new ChapterThirteenConfiguration();
```

```
app.setMainWidget( mainWin );
mainWin->show();
```

Now the problem is that there is stuff in the `ChapterThirteenMainWindowWidget` class that we want to use in the `ChapterThirteenConfiguration` class and there is nothing else for it but to flex the old cut and paste muscles.

```
#include <qvariant.h>
#include <qpixmap.h>
#include <qmainwindow.h>
```

Then complete the header by copying all the variables and function definitions declared in the class. We can add or remove things as required later for now just do a complete copy.

[illegible]

Chapter 13 : Global Information and Configuration Files

```
"...#aaaaaaaaaaaaa#...",
"...#aaaaaaaaaaaaa#...",
"...#aaaaaaaaaaaaa#...",
"...#####...",
".....",
"....."};
```

Now add the QPixmap constructors to the constructor, so it looks like,

```
ChapterThirteenConfiguration::ChapterThirteenConfiguration()
: QMainWindow( 0, "ChapterThirteenConfiguration" ),
  image0( (const char **) img0_ChapterThirteenConfigurationMainWindow ),
  image1( (const char **) img1_ChapterThirteenConfigurationMainWindow ),
  image2( (const char **) img2_ChapterThirteenConfigurationMainWindow ),
  image3( (const char **) img3_ChapterThirteenConfigurationMainWindow ),
  image4( (const char **) img4_ChapterThirteenConfigurationMainWindow ),
  image5( (const char **) img5_ChapterThirteenConfigurationMainWindow ),
  image6( (const char **) img6_ChapterThirteenConfigurationMainWindow ),
  image7( (const char **) img7_ChapterThirteenConfigurationMainWindow ),
  image8( (const char **) img8_ChapterThirteenConfigurationMainWindow ),
  image9( (const char **) img9_ChapterThirteenConfigurationMainWindow )
```

This will just create the images that appear on the menus. Next cut and paste everything in ChapterThirteenconfigurationMainWindow.cpp file constructor to the ChapterThirteenConfiguration constructor, you should remove the lines,

```
if ( !name )
  setName( "Form1" );
```

You can remove the line,

```
setCentralWidget( new ChapterThirteenConfigurationWidget( this ) );
```

that was in the ChapterThirteenConfiguration constructor as we won't be needing it. Then cut and paste the languagechange function from the ChapterThirteenConfigurationMainWindow.cpp file into the ChapterThirteenConfiguration.cpp file remembering to change the class name,

Finally you'll need to add

```
#include <klocale.h>
```

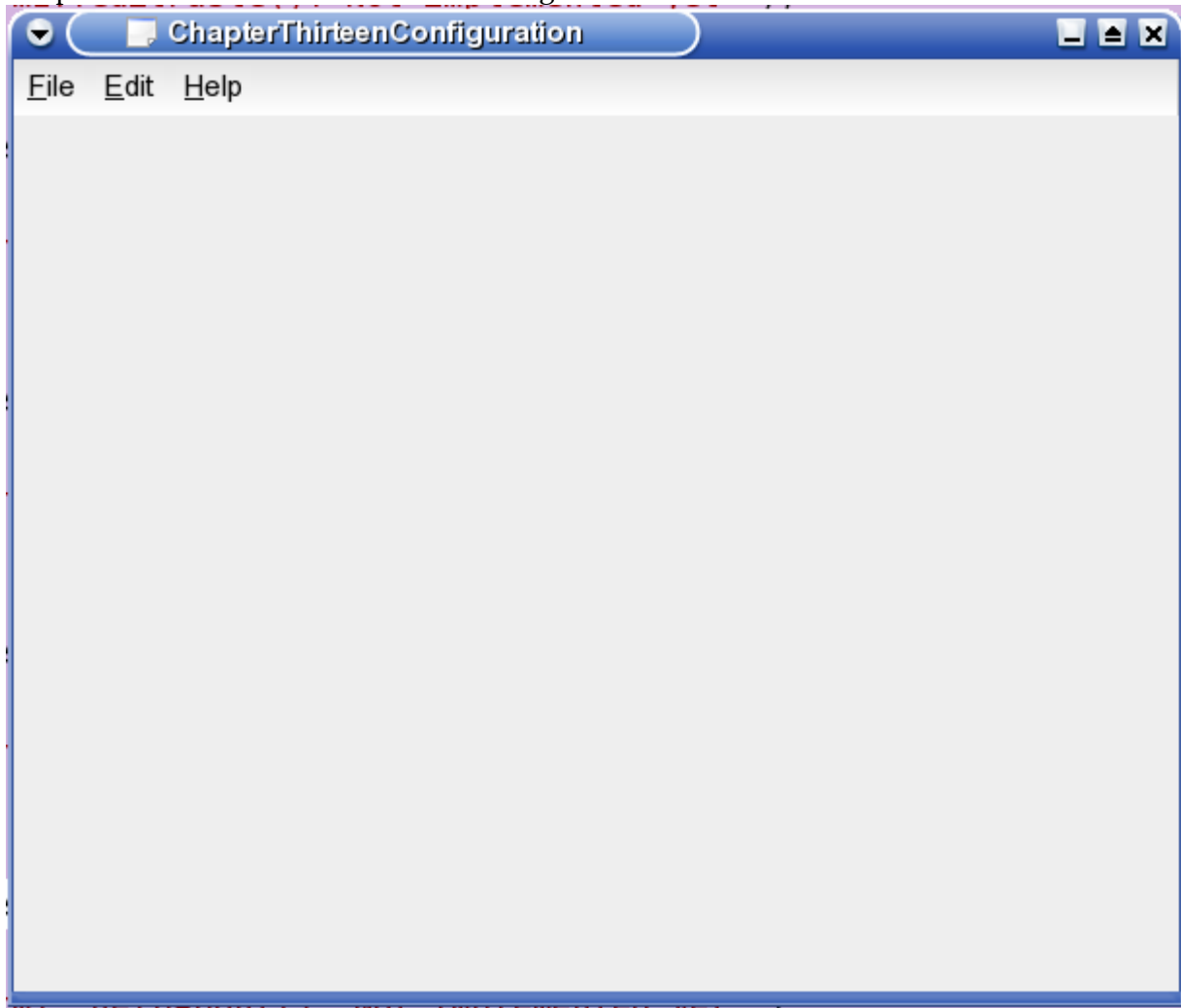
Oh and you should probably blank out the line that reads

```
setCaption( tr2i18n( "Form1" ) );
```

in the languagechange function. Or not that one's entirely optional.

to the ChapterThirteenConfiguration.cpp file or the languagechange calls to tr2i18n won't compile. Finally copy all the empty function definitions for the menus remembering to change the class names. Hit Build and Run and you should get,

Chapter 13 : Global Information and Configuration Files



All your going to need to do now is remember that any changes you make to the gui are not going to be reflected here until you cut and paste the code from the generated files.

Now back to the application to edit config files. To start with we are going to have to set up the basic gui for the application. We do this by adding a couple of splitter windows with the code,

```
mainSplitter = new QSplitter( this, "mainSplitter" );
mainSplitter->setGeometry( QRect( 10, 20, 510, 470 ) );
mainSplitter->setOrientation( QSplitter::Horizontal );

mainGroupList = new KListBox( mainSplitter, "mainGroupList" );

childSplitter = new QSplitter( mainSplitter, "childSplitter" );
childSplitter->setOrientation( QSplitter::Vertical );
```

The standard addition of the mainSplitter is as you would expect from the last program. The new thing here is that where we would have added a text box or something last time we now add a vertical splitter which divides the right hand side of the splitter into two. As with most gui systems Qt places importance on where and when you add things so if you change the ordering of the additions to the mainSplitter above so that the childSplitter is added before the listBox then the List box will still be in the same place but the splitter would be on the left. This is the reason why layouts are so much easier to implement through code than through the gui because you can add them when you require them and more importantly you can add a layout to the widget at the start and then add the widgets to it rather than adding the layout as an after thought which is the way the

Chapter 13 : Global Information and Configuration Files

gui forces you to do it and then you have to mess with the layout because layout doesn't always leave things where you put them.

Next we set up the right hand side (as you look at it) of our widget,

```
groupContentsList = new KListBox( childSplitter, "groupContentsList" );
```

```
editStack = new QWidgetStack( childSplitter, "editStack" );  
editStack->setMinimumSize( QSize( 200, 200 ) );
```

```
editPageOne = new QWidget( editStack, "editPageOne" );  
editStack->addWidget( editPageOne, 0 );
```

```
nameLabelOne = new QLabel( editPageOne, "nameLabelOne" );  
nameLabelOne->setGeometry( QRect( 12, 20, 72, 20 ) );  
nameLabelOne->setText( i18n( "Name" ) );
```

```
nameEditOne = new KLineEdit( editPageOne, "nameEditOne" );  
nameEditOne->setGeometry( QRect( 12, 40, 138, 23 ) );  
nameEditOne->setEnabled( false );
```

```
valueLabelOne = new QLabel( editPageOne, "valueLabelOne" );  
valueLabelOne->setGeometry( QRect( 10, 90, 71, 20 ) );  
valueLabelOne->setText( i18n( "Value" ) );
```

```
valueEditOne = new KLineEdit( editPageOne, "valueEditOne" );  
valueEditOne->setGeometry( QRect( 10, 110, 138, 23 ) );  
valueEditOne->setEnabled( false );
```

```
editButtonOne = new KPushButton( editPageOne, "editButtonOne" );  
editButtonOne->setGeometry( QRect( 120, 160, 124, 31 ) );  
editButtonOne->setText( i18n( "Edit" ) );  
editButtonOne->setEnabled( false );
```

```
setCentralWidget( mainSplitter );
```

We start by adding a KListBox which will be in the top right and a widget stack to the bottom right. The original idea here was that there might be a need for multiple pages at the bottom as a cursory glance at KConfigBase will show that there are multiple read write options. These proved to be more than what is required since as we are dealing with text files we can just edit the strings in the files. So we only have a one page widget stack, with a simple display and edit option. Once this code is added the running gui should look like.

Chapter 13 : Global Information and Configuration Files



The way the resource files work is that they are divided into sections called groups, each of which deals with a certain area so say one group deals with with one menu and another with another menu and another with the popup menu then all three groups could have a variable called menuColour and because each is in it's own group then there's no confusion about which menuColour is for which menu. So the idea here is that the groups are displayed on the left and the variables contained within each group are displayed on the right when a group is selected. As long as we are not viewing the global variables which is the default, when you select a variable in the right hand list box it will become available for editing in the bottom right panel. Once a variable is edited the file will automatically be saved and then reloaded. Basically the program works on the assumption that you wouldn't have edited it if you didn't mean to. And the very fact that I'm explaining that shows just how crappy using copmputers is getting these days.

Using KConfig

The KConfig class looks extremely complicated, especially if you start looking at the base class KConfigBase, with all it's different options for reading and writing just about every type imaginable. In fact if your just saving strings to a configuration file, using KConfig is really quite easy. In this project we use two versions of KConfig, the first being the default use of the global configuration options and the second being a local KConfig class object that is used to read and edit files that you either create or open. So the code itself contains lines like,

```
if( globalContents() == true )
{
    map = KGlobal::config()->entryMap( text );
}
else
{
    if( localConfig == 01 )
```

Chapter 13 : Global Information and Configuration Files

```
{
    qWarning( "local config == 0, returning" );
    return;
}

map = localConfig->entryMap( text );
}
```

which checks to see if we are using the global configuration or a local file. It then uses the `entryMap` function to return the contents.

QMaps

QMap is a Qt template class that holds a key value pair which in the context of these configuration files is a string for the name and a string for the value so if the definition of QMap is,

```
template<class Key, class T>
```

```
class QMap
```

in `qmap.h` then our definition to use it is,

```
QMap< QString, QString > map;
```

For our purposes in this project all we really need to know is that we are dealing with what is essentially a list of objects that have both a key and a value. (where any sorting and searching etc would be done using the key string) So our adding a map to the list box is almost identical to our use of `QValueLists` earlier.

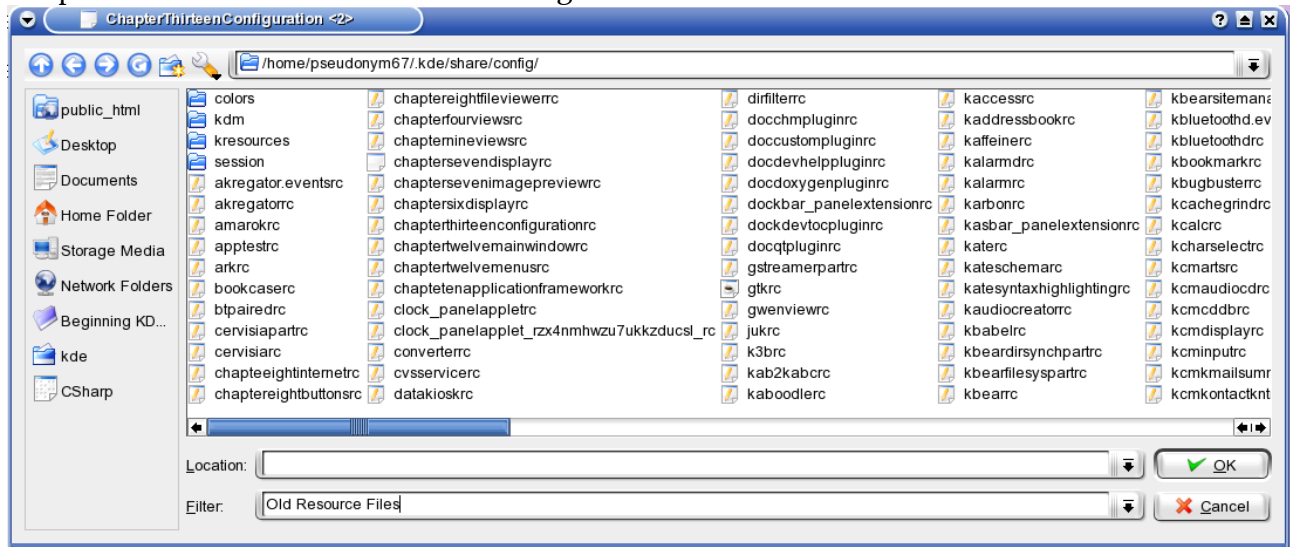
```
for( QMap< QString, QString >::iterator it = map.begin(); it != map.end(); it++ )
{
    groupContentsList->insertItem( it.key() + " " + it.data() );
}
```

with the main difference being that we declare the types in the iterator declaration rather than have a predefined definition in the header of our class.

Opening A Config File

As you can see above when dealing with the KDE global settings we just use the static `KGlobal::config()` call but when we use the local configuration files for a particular application we have to open the file ourselves,

Chapter 13 : Global Information and Configuration Files



which is just a case of opening a KFileDialog at the directory provided by,

```
QStringList list = KGlobal::dirs()->resourceDirs( "config" );
QStringList::iterator it = list.begin();
KFileDialog *dlg = new KFileDialog( *it, "*rc |Old Resource Files|*.*rc |New Resource Files", this, "Open Resource File", true );
```

Here we get the resource directories for the configuration files by using the `KGlobal::dirs()` call and then we use the first directory in the returned list which as you can see is the default to our local `.kde/share/config` folder.

When the dialog returns we pass the file name to `KConfig` and let `KConfig` take care of all that for us. So creating a new `KConfig` to open a local file goes something like,

```
localConfig = new KConfig( file, false, false );
setGlobalContents( false );

QStringList listLocal = localConfig->groupList();

for( QStringList::iterator it = listLocal.begin(); it != listLocal.end(); it++ )
{
    mainGroupList->insertItem( *it );
}
```

which as you can see creates the `KConfig` object and passes it the filename. The groups from the file are then added to the list box on the left.

Saving A Config File

The code for saving a file is a little trickier in that we have to make sure that we save the variable in the right place for example if you just saved the information

Colour red

to the configuration file it would be saved as a global variable in the file that is to say that it wouldn't be saved to any particular group. If you want to save a variable to a specific group then you have to tell the `KConfig` file object which group you want to save it to.

// If you don't set the group before a write KConfig

// assumes a default variable

```
localConfig->setGroup( currentGroup() );
localConfig->writeEntry( nameEditOne->text(), valueEditOne->text() );
```

Chapter 13 : Global Information and Configuration Files

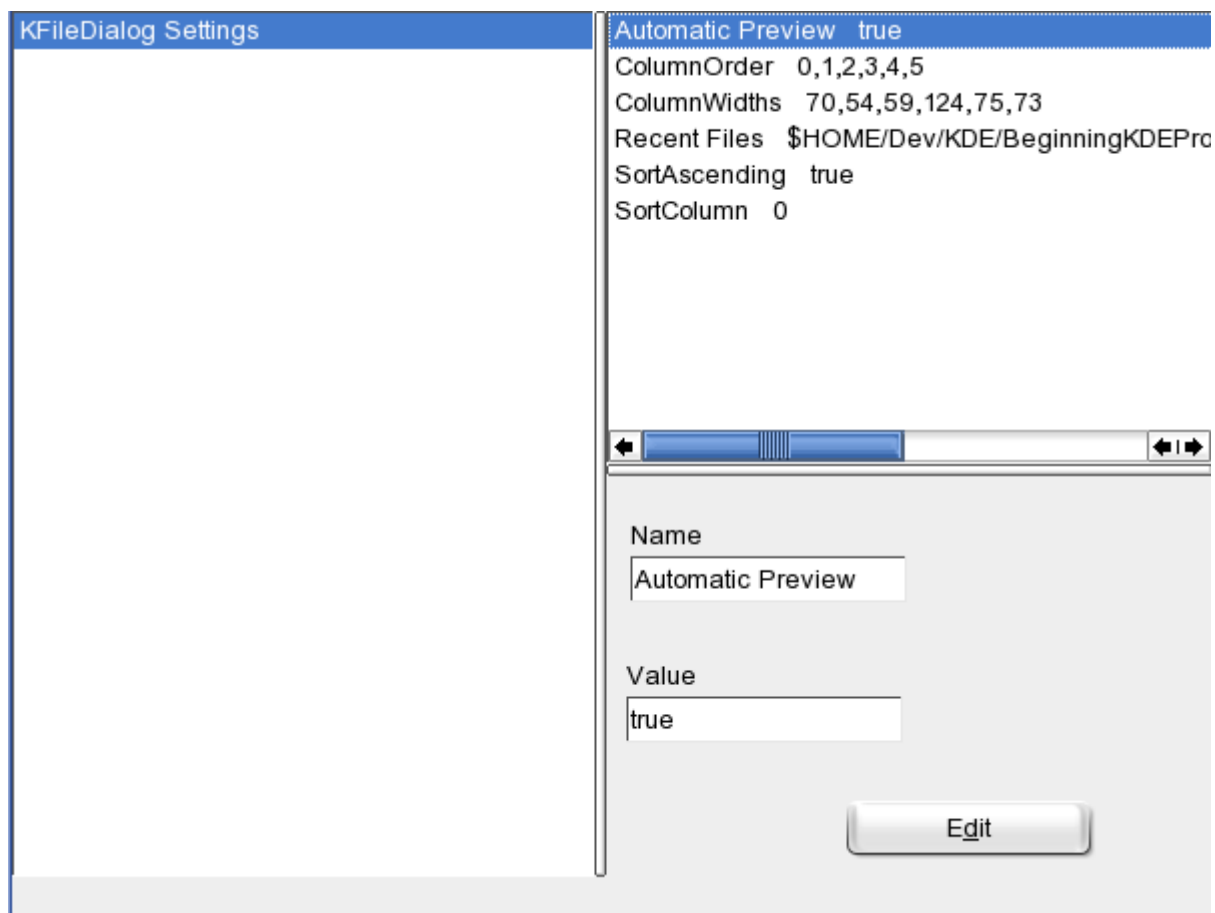
```
// call sync straight away as this is an editor it's not unreasonable to
// assume that more than one value will be edited at a time.
localConfig->sync();
// reload the file with the change
openFile( currentFile() );
```

Here we set the group to `currentGroup`. `currentGroup` is merely a `QString` class variable that I use to keep track of the current group by setting it whenever a new group is created or when one is selected in the left list box. The information from the edit boxes in the bottom right section of the window is then written to the file under the current group heading.

`sync` is then called to flush the file and write our changes to the disk before the file is reloaded. (This is required as if we don't the display will continue to show the old, now out of date information.)

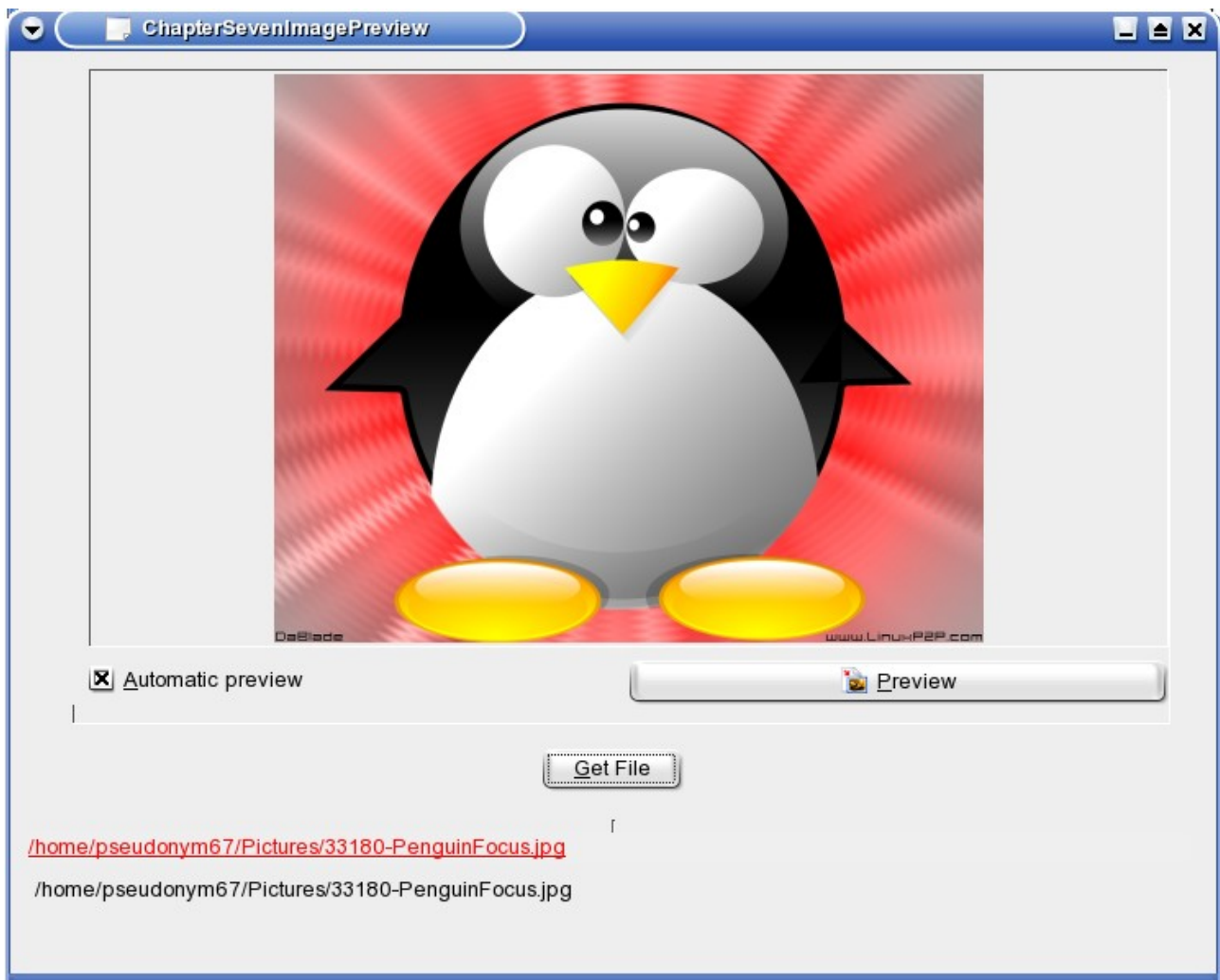
Editing A Config File

To edit a config file select `File/Open` and you'll be shown the dialog above that defaults to `.kde/share/config` in your home folder. If you've been running the demo programs you'll see that certain of the test programs have saved information in a config file without you knowing about it. If we open up the `chaptersevenimagepreviewsrc` file we see that what has been saved is the `KFileDialog` settings.



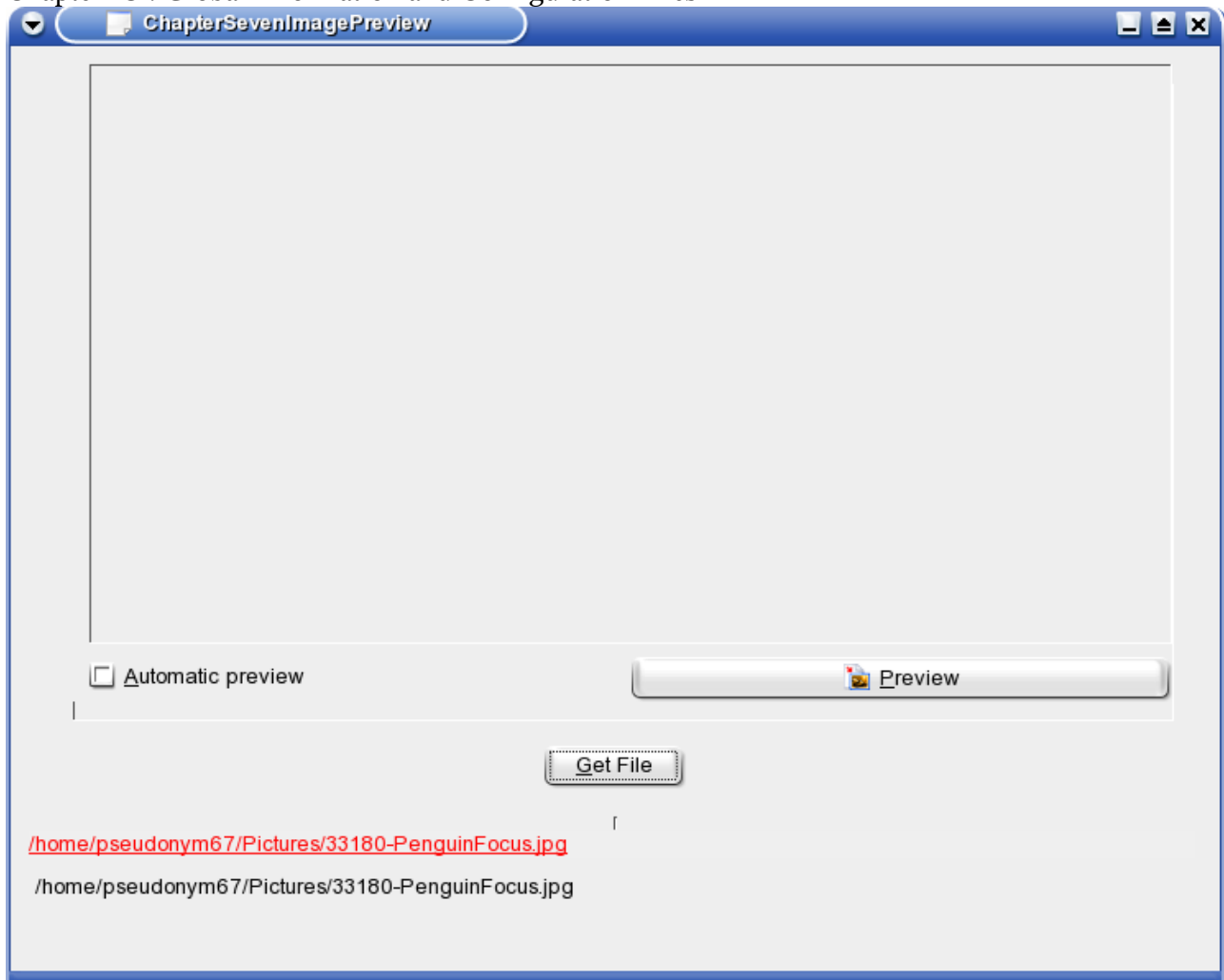
Chapter 13 : Global Information and Configuration Files

If we run the program before changing anything we get,



and if we change the Automatic Preview option to false and hit edit and then run the program again we get,

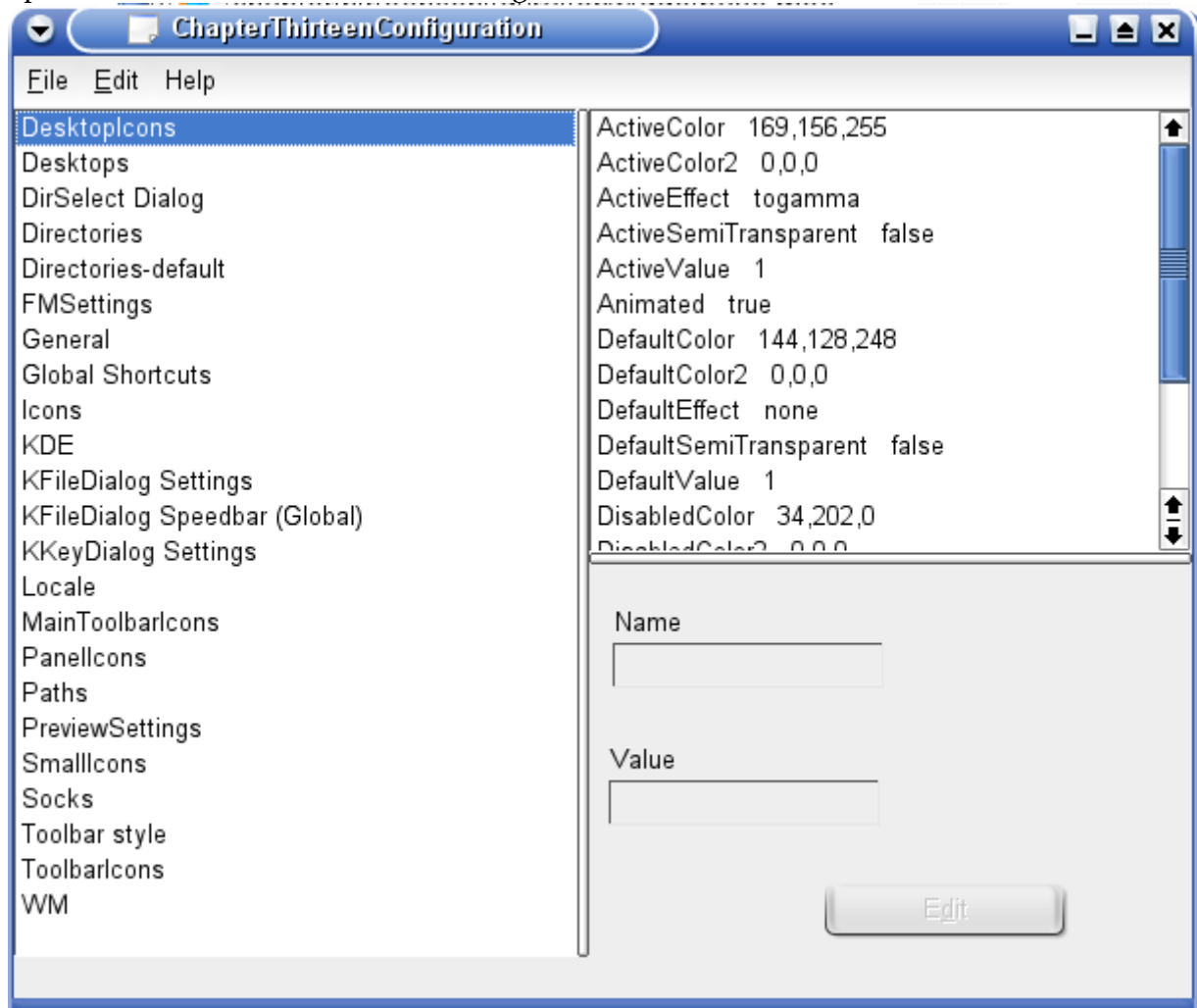
Chapter 13 : Global Information and Configuration Files



Finishing Up

The rest of the code is a matter of enabling and disabling things at the correct time, adding the help files and an about box which you should be able to work out just by looking at the code in `ChapterThirteenConfiguration.cpp`.

Chapter 13 : Global Information and Configuration Files



Summary

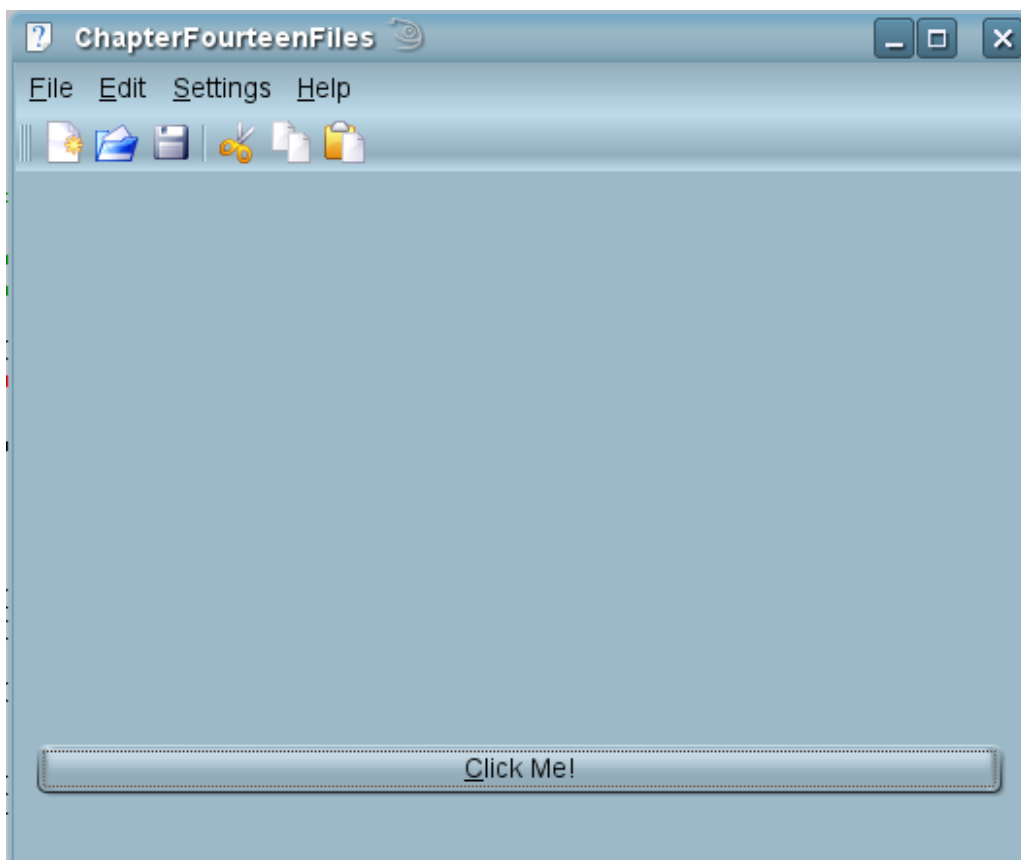
In this chapter we have taken a look at the global settings available to us and taken a rather long winded way to create an application that could view edit and create KDE application settings.

Chapter 14 A Simple Editor Application

In this chapter we are going to be looking at the standard way of setting up a KDE application. This method is used to put together a simple text editor application that can spell check the text it's displaying.

The KDE Way

So far in order to show how things actually work we have been doing menus and toolbars the hard way. In fact it could be said that we've been going out of our way to make life difficult for ourselves. So now we'll do things properly, the KDE way and start off with this,



using just this code,

```
setCentralWidget( new ChapterFourteenFilesWidget( this ) );

statusBar();

KStdAction::openNew( this, SLOT( fileNew() ), actionCollection() );
KStdAction::open( this, SLOT( fileOpen() ), actionCollection() );
KStdAction::save( this, SLOT( fileSave() ), actionCollection() );
KStdAction::saveAs( this, SLOT( fileSaveAs() ), actionCollection() );
KStdAction::quit( this, SLOT( fileQuit() ), actionCollection() );

KStdAction::cut( this, SLOT( editCut() ), actionCollection() );
KStdAction::copy( this, SLOT( editCopy() ), actionCollection() );
KStdAction::paste( this, SLOT( editPaste() ), actionCollection() );
```

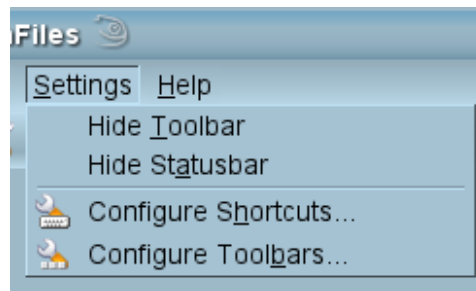
Chapter 14 A Simple Editor Application

setupGUI();

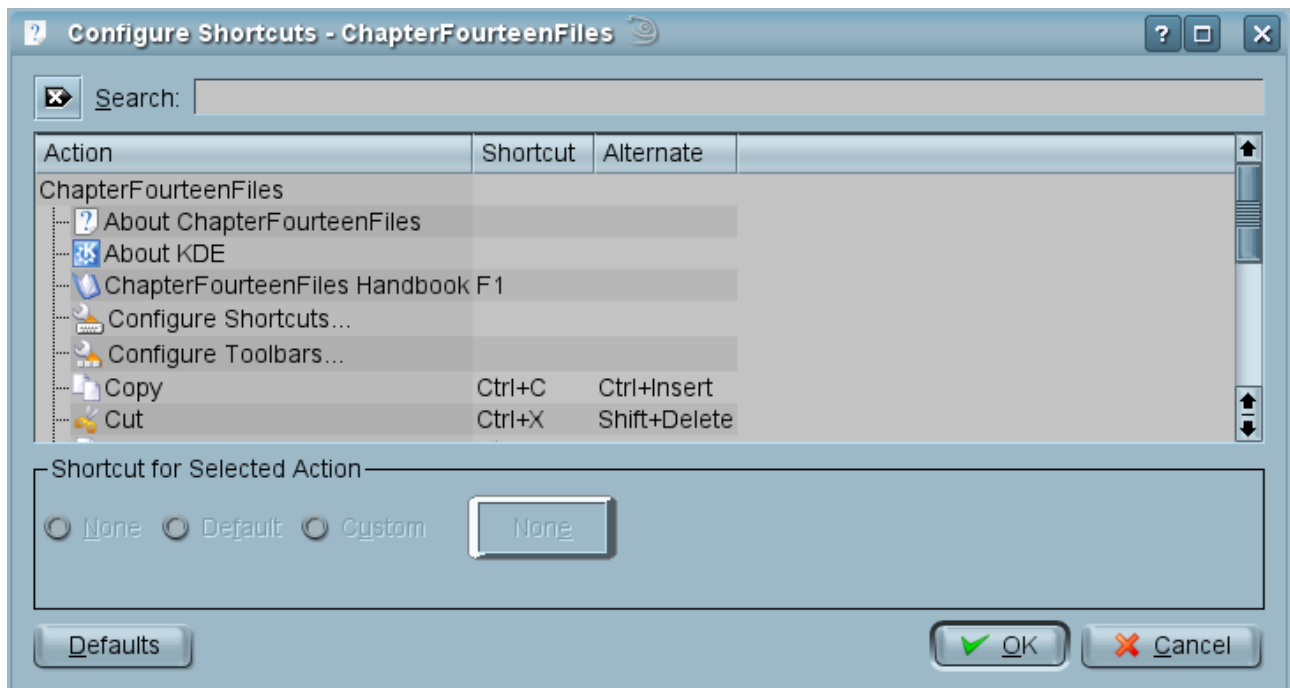
That is it, of course we have to write the function implementations ourselves but I guess we can live with that. The way it works is that by using the KStdAction to create the action it will create it with the current KDE icon settings and save it in the action collection which will also take care connecting the signal with the slot that we have provided.

The call to the setupGUI function not only creates the menus and toolbar icons as we have specified them in the constructor but it also throws in the Settings menu and the help menu for free.

The settings menu has these options,

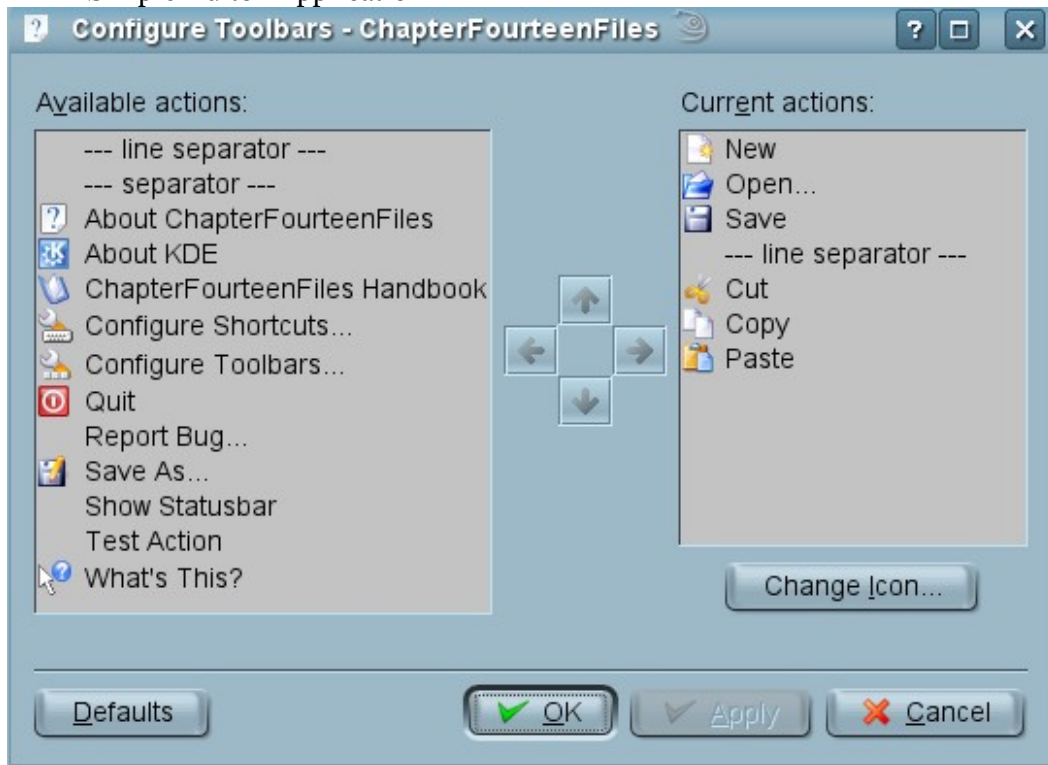


The Hide Toolbar and Hide StatusBar items should be self explanatory, the Configure Shortcuts item opens the dialog,



and the Configure Toolbars options opens the dialog,

Chapter 14 A Simple Editor Application



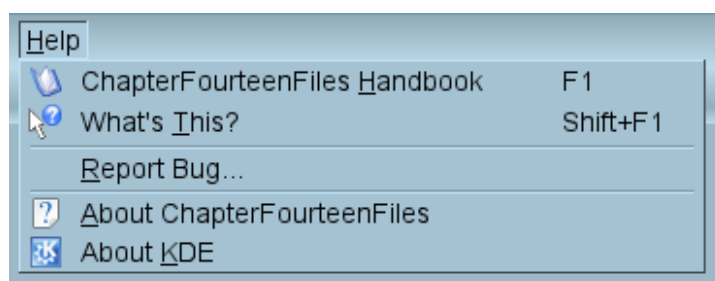
Unfortunately both these configuration dialogs have little actual effect on the program at the moment as the configuration file for the application reads,

```
Height 900=410
Width 1440=607
```

```
[General]
ShowAlternativeShortcutConfig=false
```

```
[MainWindow]
Height 900=410
Width 1440=607
```

even after you have changed items in both dialogs. The only things that are being saved at the moment are the options to display or hide the toolbar and the status bar. The help menu on the other hand,



The first item on the list is the ChapterFourteenFiles Handbook this launches the help for the program in the KDE Help Center, note though that it will only do this if you install the application so if you are just building and testing the application you will still need to install the application if you wish to see how the help files are progressing. See DocBooks The Simple Guide below for

Chapter 14 A Simple Editor Application
more information about docbooks.

The second item which you should know by now is the What's This? item which is present in every widget and provides a more detailed description of a program item than the simple tooltip.

The third item is the Report Bug item and for any programs I've written using this system it will look like this,

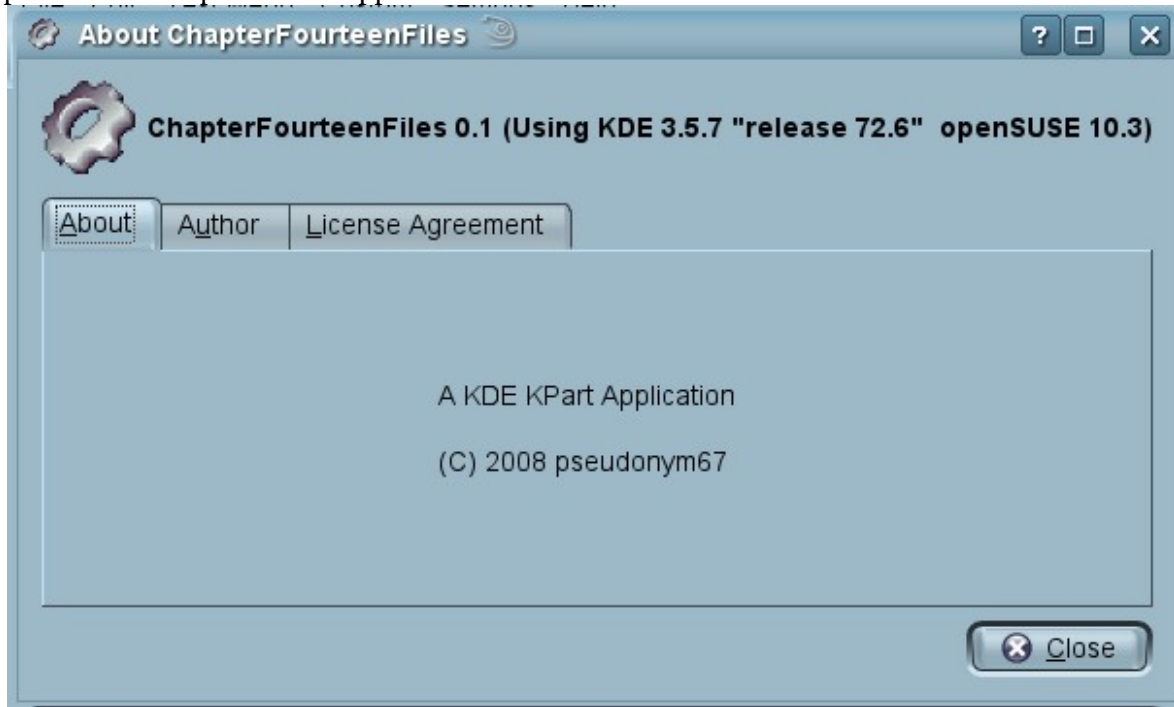
The screenshot shows a window titled "Submit Bug Report - ChapterFourteenFiles". It contains the following fields and controls:

- From:** pseudonym67
- To:** pseudonym67@hotmail.com
- Application:** chapterfourteenfiles (selected from a dropdown menu)
- Version:** 0.1 3.5.7 "release 72.6" , openSUSE 10.3
- OS:** Linux (i686) release 2.6.22.17-0.1-default
- Compiler:** Target: i586-suse-linux
- Severity:** A group of radio buttons with options: Critical, Grave, Normal (selected), Wishlist, and Translation.
- Subject:** An empty text input field.
- Instructions:** "Enter the text (in English if possible) that you wish to submit for the bug report. If you press 'Send', a mail message will be sent to the maintainer of this program."
- Text Area:** A large empty text area for the bug report details.
- Buttons:** "Send" (with an envelope icon) and "Cancel" (with a red X icon).
- Additional Button:** "Configure Email..." located next to the "To:" field.

As you can see this gives you a simple no fuss dialog that allows you to type in your problem with the current program that you are using and send that problem straight to the registered developer of the program.

The final two options are the about boxes for the program one for the program itself,

Chapter 14 A Simple Editor Application



and the second being for KDE,

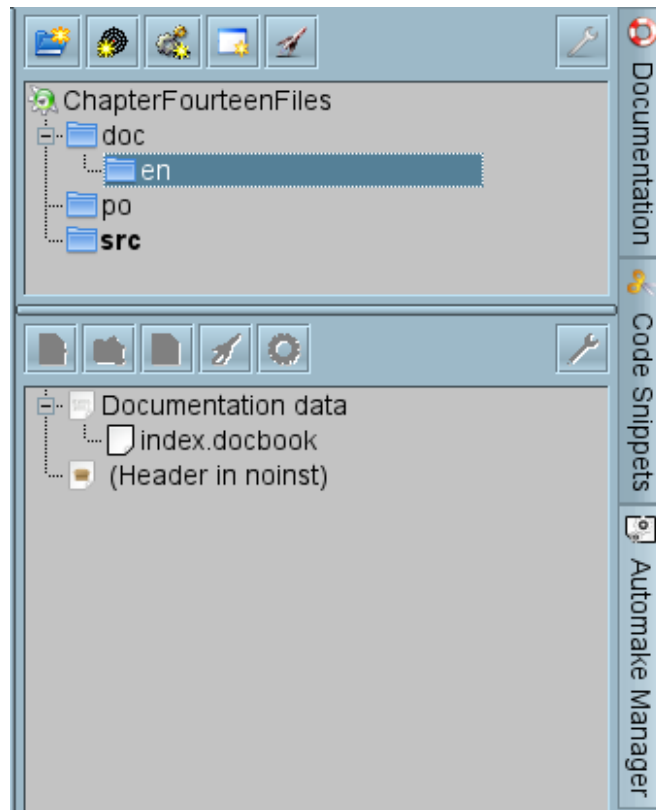


Now though we need to learn somethings about docbooks.

DocBooks The Simple Guide

If you look in the Automake Manager at the project section there is a documents (doc) section and

Chapter 14 A Simple Editor Application
in this there is a English (en) section,



If you open it as shown above you will see that there is a file called index.docbook. The naming here is a little confusing as this is not just an index file it is going to be the help file itself.

This file is a basic layout of what you can do with a docbook but to start with we are going to strip it down to the bare essentials of what we need then it will be up to each individual to enhance their docbooks as they wish.

If you are familiar with html or xml then things are pretty similar in that everything is an element and every element starts with an opening `<element>` definition and ends with a closing `</element>` definition.

The essential things that we need in a docbook are,

Chapters

The start of a chapter in a docbook is defined as

```
<chapter id="introduction">
```

This is standard xml syntax in that we start with an opening `<` and the word chapter we then add the id parameter and call it introduction, with the `>` signalling the end of the declaration.

A chapter is ended with

```
</chapter>
```

Chapter 14 A Simple Editor Application

Paragraphs

If you don't want all your text to run together in a docbook you need to wrap each of your paragraphs with

```
<para>
</para>
```

which in html would be `<p>` and `</p>`

Links

You can link to another chapter in your document by using the id name for the link

```
<link linkend="chapterorsectionid">Print Name</link>
```

Screenshots

All screenshots in the docbook should be in .png format which shouldn't be an issue as most image manipulation programs will save as .png files. The following is a modified version of what we see in the index.docbook file,

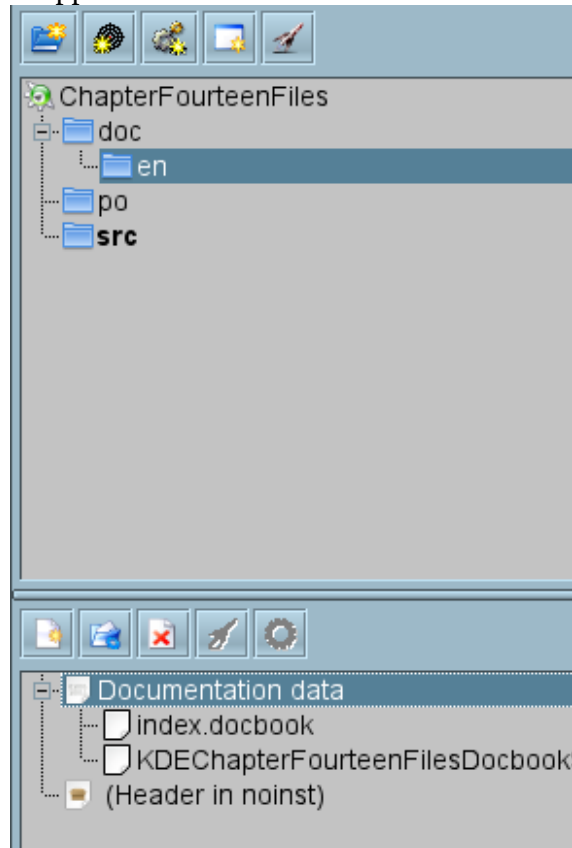
```
<screenshot>
<screeninfo>Here's a screenshot of &chapterfourteenfiles;</screeninfo>
  <mediaobject>
    <imageobject>
      <imagedata fileref="KDEChapterFourteenFilesDocbookScreenshot.png" format="PNG"/>
    </imageobject>
    <textobject>
      <phrase>A Screenshot of ChapterFourteenFiles</phrase>
    </textobject>
  </mediaobject>
</screenshot>
```

The sections `<screeninfo>` and `<textobject>` are entirely optional. The main required lines here are

```
<screenshot>
  <mediaobject>
    <imageobject>
      <imagedata> file info </imagedata>
    </imageobject>
  </mediaobject>
</screenshot>
```

There is one important thing that you need to remember when adding screenshots and that is that you need to add the image file to your project in the Document Data section. You do this by right clicking on Documentation data and selecting “Add Existing Files”.

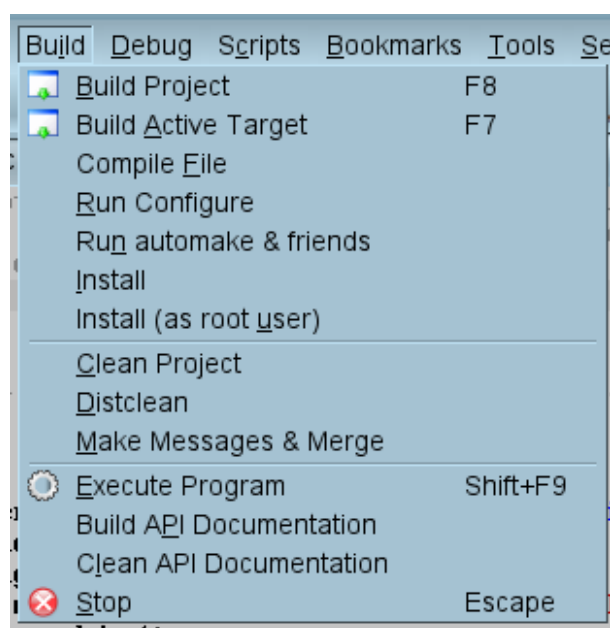
Chapter 14 A Simple Editor Application



A complete reference for docbook elements is available for download at www.docbook.org.

Installing A Docbook

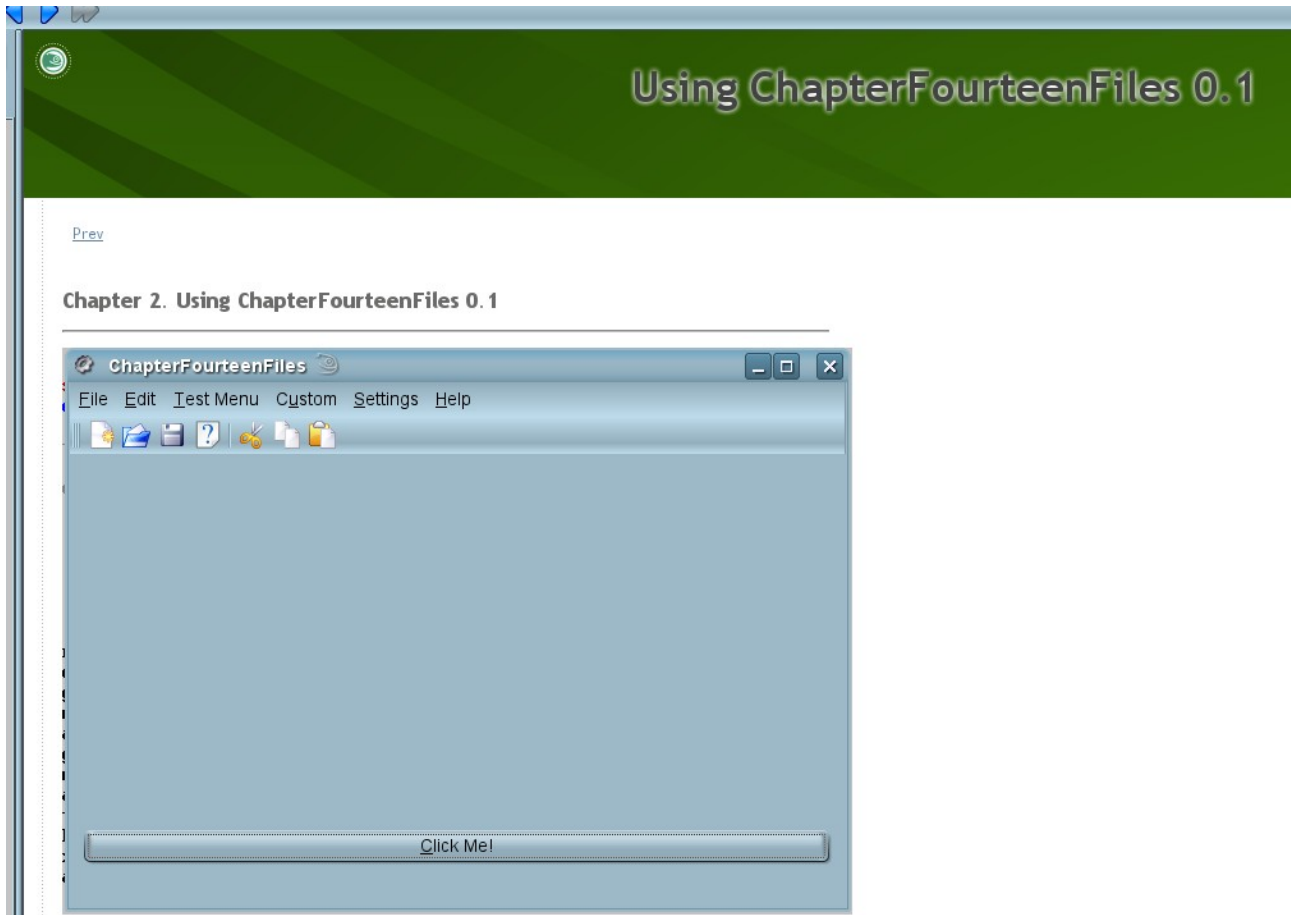
You can't just open a docbook to view it. You have to install your program first.



If you go to the build menu you can see that there are two Install options the first being the standard

Chapter 14 A Simple Editor Application

install and the second being Install (as root user). As you are going to be installing things into the root/opt directory you will need to install as root user. Once the project is installed, (the program file will go to root/opt/kde3/bin and the help file will be saved in root/opt/kde3/share/doc/HTML/en/chapterfourteenfiles.) you can run the program in the debugger and open the help correctly but remember that the help wont be updated if changed unless you install again.



Of course this is just a test to see how things work, at a later date when the program is working we will update the image and rebuild everything properly.

The index.docbook file provided is meant as a general guide of how to use docbooks. While the guide itself is quite amusing in places it cannot hide the fact that docbooks are the most ridiculously overcomplicated way to display a html page imaginable. In fact the very presence of docbooks turns this whole method into a double edged sword. It is true that the use of KStdAction will allow you to save a whole lot of time typing in generic code for the menus but the time saved and more will be lost fiddling with the docbook syntax.

The file will open in KDevelop if you click on it and it starts with,

```
<?xml version="1.0" ?>
<!DOCTYPE book PUBLIC "-//KDE//DTD DocBook XML V4.1.2-Based Variant V1.1//EN" "dtd/kdex.dtd" [
  <!ENTITY chapterfourteenfiles "<application>ChapterFourteenFiles 0.1</application>">
  <!ENTITY kappname "&chapterfourteenfiles;"><!-- Do *not* replace kappname-->
  <!ENTITY package "kde-module"><!-- kdebase, kadmin, etc -->
  <!ENTITY % addindex "IGNORE">
  <!ENTITY % English "INCLUDE"><!-- change language only here -->
```

Chapter 14 A Simple Editor Application

```
<!-- Do not define any other entities; instead, use the entities
      from kde-genent.entities and $LANG/user.entities. -->
]>
```

This is the text that defines exactly what the document type is until you are more experienced with Docbooks ignore this.

Then there are some comments before the next section which is the book info section.

```
<bookinfo>
<title>The &chapterfourteenfiles; Handbook</title>

<authorgroup>
<author>
<firstname></firstname>
<othername></othername>
<surname>pseudonym67</surname>
<affiliation>
<address><email>pseudonym67@hotmail.com</email></address>
</affiliation>
</author>
</authorgroup>

<!-- TRANS:ROLES_OF_TRANSLATORS -->

<copyright>
<year>1999</year>
<year>2008</year>
<holder>pseudonym67</holder>
</copyright>
<!-- Translators: put here the copyright notice of the translation -->
<!-- Put here the FDL notice. Read the explanation in fdl-notice.docbook
      and in the FDL itself on how to use it. -->
<legalnotice>&FDLNotice;</legalnotice>

<!-- Date and version information of the documentation
Don't forget to include this last date and this last revision number, we
need them for translation coordination !
Please respect the format of the date (YYYY-MM-DD) and of the version
(V.MM.LL), it could be used by automation scripts.
Do NOT change these in the translation. -->

<date>2001-10-18</date>
<releaseinfo>0.1</releaseinfo>

<!-- Abstract about this handbook -->

<abstract>
<para>
&chapterfourteenfiles; is a small notepad type file editor program
</para>
</abstract>

<!-- This is a set of Keywords for indexing by search engines.
Please at least include KDE, the KDE package it is in, the name
of your application, and a few relevant keywords. -->

<keywordset>
<keyword>KDE</keyword>
```

Chapter 14 A Simple Editor Application

```
<keyword>ChapterFourteenFiles</keyword>
<keyword>nothing</keyword>
<keyword>nothing else</keyword>
</keywordset>
```

```
</bookinfo>
```

This is setup information for the docbook and only the abstract section need concern us.

```
<!-- Abstract about this handbook -->
```

```
<abstract>
<para>
&chapterfourteenfiles; is a small notepad type file editor program
</para>
</abstract>
```

This section is where we add a small description of what our program does.

This is followed by the introduction which expands on what we have typed above.

```
<chapter id="introduction">
<title>Introduction</title>
```

```
<!-- The introduction chapter contains a brief introduction for the
application that explains what it does and where to report
problems. Basically a long version of the abstract. Don't include a
revision history. (see installation appendix comment) -->
```

```
<para>
&chapterfourteenfiles; is a demonstration project that is used to give examples of how to load and save files as well
as how to cut and copy items to the clipboard as well as paste them back out again.
</para>
<para>
Any problems or feature requests to the &kde; mailing lists.
</para>
</chapter>
```

Note the use of the chapter keyword here. In a docbook each chapter is a page in the help So while I would suggest you use the generated file as a rough guide I would suggest that at first you keep it simple just use each chapter as a page in the help and use paragraphs and screenshots to say what you need to say. Then use the generated file as a guide to building more featured docbooks as you become more comfortable with the format.

One picky little detail here would be that as the program now as an automatic bug reporting feature as part of the help menu the text,

```
<para>
Any problems or feature requests to the &kde; mailing lists.
</para>
```

should probably refer you to that instead of the default mailing lists.

Continuing with the index.docbook we see that we can split the chapters into sections giving each section it's own subheader, the example code for this is,

```
<sect1 id="chapterfourteenfiles-features">
<title>More &chapterfourteenfiles; features</title>

<para>It slices! It dices! and it comes with a free toaster!</para>
<para>
The Squiggle Tool <guiicon><inlinemediaobject>
```

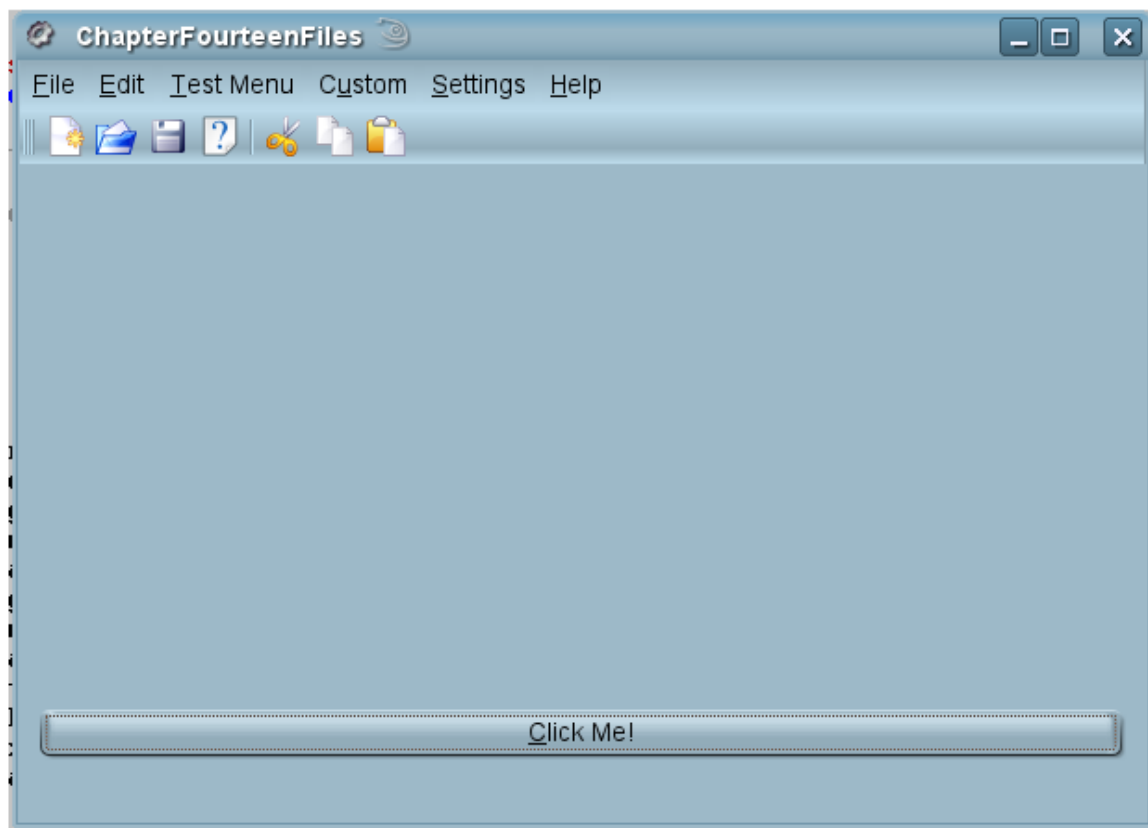
Chapter 14 A Simple Editor Application

```
<imageobject>
  <imagedata fileref="squiggle.png" format="PNG"/>
</imageobject>
<imageobject>
  <imagedata fileref="squiggle.eps" format="EPS"/>
</imageobject>
<textobject>
  <phrase>Squiggle</phrase>
</textobject>
</inlinemediaobject></guiicon> is used to draw squiggly lines all over
the &chapterfourteenfiles; main window. It's not a bug, it's a feature!
</para>

</sect1>
```


Note that the squiggle.png file is not added which is why we dont see it in the image,

Chapter 2. Using ChapterFourteenFiles 0.1



More ChapterFourteenFiles 0.1 features

It slices! It dices! and it comes with a free toaster!

The Squiggle Tool  is used to draw squiggly lines all over the ChapterFourteenFiles 0.1 main window. It's not a bug, it's a feature!

[Prev](#)

[Home](#)

[Introduction](#)

[Up](#)

In the index for your docbook this will look like,

Table of Contents

[1. Introduction](#)

[2. Using ChapterFourteenFiles 0.1](#)

[More ChapterFourteenFiles 0.1 features](#)

Chapter 14 A Simple Editor Application

The next section in the index.docbook file is the commands section which lists the menu options and keyboard shortcuts for your application. As with the above this is a single chapter.

```
<chapter id="commands">
<title>Command Reference</title>

<!-- (OPTIONAL, BUT RECOMMENDED) This chapter should list all of the
application windows and their menubar and toolbar commands for easy reference.
Also include any keys that have a special function but have no equivalent in the
menus or toolbars. This may not be necessary for small apps or apps with no tool
or menu bars. -->

<sect1 id="chapterfourteenfiles-mainwindow">
<title>The main &chapterfourteenfiles; window</title>

<sect2>
<title>The File Menu</title>
<para>
<variablelist>
<varlistentry>
<term><menuchoice>
<shortcut>
<keycombo action="simul">&Ctrl;<keycap>N</keycap></keycombo>
</shortcut>
<guimenu>File</guimenu>
<guimenuitem>New</guimenuitem>
</menuchoice></term>
<listitem><para><action>Creates a new document</action></para></listitem>
</varlistentry>
<varlistentry>
<term><menuchoice>
<shortcut>
<keycombo action="simul">&Ctrl;<keycap>S</keycap></keycombo>
</shortcut>
<guimenu>File</guimenu>
<guimenuitem>Save</guimenuitem>
</menuchoice></term>
<listitem><para><action>Saves the document</action></para></listitem>
</varlistentry>
<varlistentry>
<term><menuchoice>
<shortcut>
<keycombo action="simul">&Ctrl;<keycap>Q</keycap></keycombo>
</shortcut>
<guimenu>File</guimenu>
<guimenuitem>Quit</guimenuitem>
</menuchoice></term>
<listitem><para><action>Quits</action> &chapterfourteenfiles;</para></listitem>
</varlistentry>
</variablelist>
</para>

</sect2>

<sect2>
<title>The <guimenu>Help</guimenu> Menu</title>

<!-- Assuming you have a standard help menu (help, what's this, about -->
<!-- &chapterfourteenfiles;, about KDE) then the documentation is already written. -->
<!-- The following entity is valid anywhere that a variablelist is -->
```

Chapter 14 A Simple Editor Application

<!-- valid. -->

&help.menu.documentation;

</sect2>

</sect1>

</chapter>

As with the above it starts with the chapter definition and title

<chapter id="commands">

<title>Command Reference</title>

It then defines a section one,

<sect1 id="chapterfourteenfiles-mainwindow">

<title>The main &chapterfourteenfiles; window</title>

which will give us an indented subheading as we saw above

Then it defines the two menu items that are listed under the id section 2 or <sect2>. The contents menu which displays only the sections and chapters will show,

```
..... 3. Command Reference
          The main ChapterFourteenFiles 0.1 window
              The File Menu
              The Help Menu
```

while the page itself will display as,

Chapter 14 A Simple Editor Application

Chapter 3. Command Reference

The main ChapterFourteenFiles 0.1 window

The File Menu

File->New (**Ctrl+N**)

Creates a new document

File->Save (**Ctrl+S**)

Saves the document

File->Quit (**Ctrl+Q**)

Quits ChapterFourteenFiles 0.1

The Help Menu

Help->ChapterFourteenFiles 0.1 Handbook (**F1**)

Invokes the KDE Help system starting at the ChapterFourteenFiles 0.1 help pages. (this document).

Help->What's This? (**Shift+F1**)

Changes the mouse cursor to a combination arrow and question mark. Clicking on items within ChapterFourteenFiles 0.1 will open a help window (if one exists for the particular item) explaining the item's function.

Help->Report Bug...

Opens the Bug report dialog where you can report a bug or request a "wishlist" feature.

Help->About ChapterFourteenFiles 0.1

This will display version and author information.

Help->About KDE

This displays the KDE version and other basic information.

[Prev](#)

[Home](#)

Using ChapterFourteenFiles 0.1

[Up](#)

To display the individual options you first need to start a variable list element using,

```
<para>
<variablelist>
```

The items themselves are then defined as variablelistentries or

```
<varlistentry>
<term>
<menuchoice>
<shortcut>
  <keycombo action="simul">&Ctrl;<keycap>N</keycap></keycombo>
</shortcut>
<guimenu>File</guimenu>
<guimenuitem>New</guimenuitem>
</menuchoice>
```

Chapter 14 A Simple Editor Application

```
</term>
<listitem>
  <para>
    <action>Creates a new document</action>
  </para>
</listitem>
</varlistentry>
```

We start the varlistentry which is an item of the variable list and then begin a term which is the definition of the actual item. The item is then defined as a menuchoice with a short cut defines as CTRL + N. The guimenu that the item exists on is the File menu and the guimenuitem is the New option. Finally the listitem which gives the explanation of the action taken when the menu item is selected.

```

      File->New (Ctrl+N)
      Creates a new document
```

When we get to the next section that details the help options there is a difference in that,

```
<sect2>
<title>The <guimenu>Help</guimenu> Menu</title>

<!-- Assuming you have a standard help menu (help, what's this, about -->
<!-- &chapterfourteenfiles;, about KDE) then the documentation is already written. -->
<!-- The following entity is valid anywhere that a variablelist is -->
<!-- valid. -->

&help.menu.documentation;

</sect2>
```

Here the main body of text is the explanation that if we are using the standard help menu as generated by KDevelop then we just need to tell it to include the standard documentation which is done with,

```
&help.menu.documentation
```

The next section in the demonstration is the developer guide as with the demonstration guide if you are writing a program that requires a programming guide or plugins and scripting then you will be dealing with more advanced docbook topics and should read the manual at www.docbook.org.

Note we will not be using programming guides or scripting for any applications in this book. (version 3.x) so this section of the docbook.index file will be removed.

The next chapter is the F.A.Q. chapter which reads,

```
<chapter id="faq">
<title>Questions and Answers</title>

<!-- (OPTIONAL but recommended) This chapter should include all of the silly
(and not-so-silly) newbie questions that fill up your mailbox. This chapter
should be reserved for BRIEF questions and answers! If one question uses more
than a page or so then it should probably be part of the
"Using this Application" chapter instead. You should use links to
cross-reference questions to the parts of your documentation that answer them.
```

Chapter 14 A Simple Editor Application

This is also a great place to provide pointers to other FAQ's if your users must do some complicated configuration on other programs in order for your application work. -->

&reporting.bugs;
&updating.documentation;

```
<qandaset id="faqlist">
<qandaentry>
<question>
<para>My Mouse doesn't work. How do I quit &chapterfourteenfiles?</para>
</question>
<answer>
<para>You silly goose! Check out the <link linkend="commands">Commands
Section</link> for the answer.</para>
</answer>
</qandaentry>
<qandaentry>
<question>
<para>Why can't I twiddle my documents?</para>
</question>
<answer>
<para>You can only twiddle your documents if you have the foobar.lib
installed.</para>
</answer>
</qandaentry>
</qandaset>
</chapter>
```

You might have noticed by now that docbooks have a specialist tag for everything, though this shouldn't be too hard to follow from what you've seen already,

We start with some standard text and then get straight in with the Q and A set with the id "Faqlist", each item is then divided into a qandaentry with a question and an answer. One thing this does show is how to put a link to a different chapter by using,

```
<link linkend="commands">Commands Section</link>
```

To link back to the command section.

The Credits chapter is,

```
<chapter id="credits">
```

```
<!-- Include credits for the programmers, documentation writers, and
contributors here. The license for your software should then be included below
the credits with a reference to the appropriate license file included in the KDE
distribution. -->
```

```
<title>Credits and License</title>
```

```
<para>
&chapterfourteenfiles;
</para>
<para>
Program copyright 2008 pseudonym67 <email>pseudonym67@hotmail.com</email>
</para>
<para>
Contributors:
<itemizedlist>
```

Chapter 14 A Simple Editor Application

```
<listitem><para>Konqui the KDE Dragon <email>konqui@kde.org</email></para>
</listitem>
<listitem><para>Tux the Linux Penguin <email>tux@linux.org</email></para>
</listitem>
</itemizedlist>
</para>

<para>
Documentation copyright 2008 pseudonym67 <email>pseudonym67@hotmail.com</email>
</para>

<!-- TRANS:CREDIT_FOR_TRANSLATORS -->

&underFDL;          <!-- FDL: do not remove -->

<!-- Determine which license your application is licensed under,
and delete all the remaining licenses below:

(NOTE: All documentation are licensed under the FDL,
regardless of what license the application uses) -->

&underGPL;          <!-- GPL License -->
&underBSDLicense;   <!-- BSD License -->
&underArtisticLicense; <!-- BSD Artistic License -->
&underX11License;   <!-- X11 License -->

</chapter>
```

This contains the credits for the writing/testing etc for the application and the licensing information.

Finally we get to the appendix which lists all the technical information about requirements and installation etc for the application.

That completes the quick guide to the docbook file, we will come back to it later when the application is finished so that we can update it correctly.

Menus

So far we have seen that when we add a KStdAction to our program all we need to define is the slot to deal with the action and once we call setupGui the linking will be done for us. For most applications however, there will be menu items required to do specific actions that are not a part of those included in KStdAction. So you need to know how to add your own menus. The initial response when faced with this situation is to add something like this,

```
KAction *nonStdAction = new KAction( i18n( "Test Action" ),          BarIconSet( "kbugbuster" ), KShortcut::null(),
this, SLOT( testAction() ),          actionCollection(), "TestAction" );

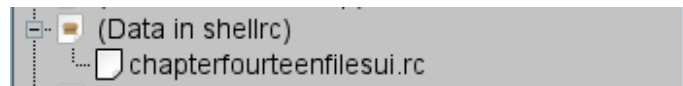
nonStdAction->plug( toolBar(), 3 );

KPopupMenu *testMenu = new KPopupMenu( this, "testmenu" );
nonStdAction->plug( testMenu );
menuBar()->insertItem( i18n( "Test Menu" ), testMenu, -1, 2 );
```

This is the standard way of adding a menu item and connecting it to a slot function and as you can see from the program it works fine as long as we add the code after the call to the setupGui function. Note that if it is added before the call to setup gui the menu item simply will not appear. There is however, another way and that is by using the resource file which is in your project and is

Chapter 14 A Simple Editor Application

found in the Data In shellrc section of the automake manager.



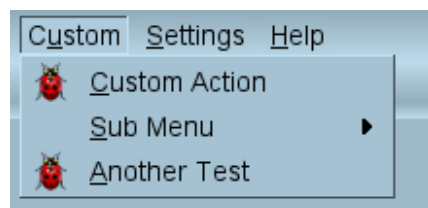
If you open it you'll get,

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartgui name="chapterfourteenfiles" version="1">
<MenuBar>
  <Menu name="custom"><text>C&ustom</text>
    <Action name="custom_action" />
  </Menu>
</MenuBar>
</kpartgui>
```

Basically the idea here is to create a system where you can add gui items to you program on the fly using xml files. Here as you can see we add an item to the menu bar called custom and it contains one action called custom_action. The Menu name is the actual menu that will appear on you menu bar and the Action is the name of a menu item, as you can see there is only one Action defined here but you can add as many as you like. As an example I've changed the rc file for this project to,

```
<!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
<kpartgui name="chapterfourteenfiles" version="1">
<MenuBar>
  <Menu name="custom"><text>C&ustom</text>
    <Action name="custom_action" />
    <Menu name="customSubMenu"><text>Sub Menu</text>
      <Action name="subMenu_action" />
    </Menu>
    <Action name="AnotherTest" />
  </Menu>
</MenuBar>
</kpartgui>
```

This shows the original menu item and then adds a submenu with a single item and then adds a new action below that it looks like,



I've used the kbugbuster icon for all the demonstration menu for no other reason than that I like it, although I may have over done it. It should be noted that unless you add actions to the code no menu options will be shown but the custom menu item will be. if you don't want to use this way of doing the menus you must remember to remove the <MenuBar> </MenuBar> section from the rc file as if you don't when the application is installed the Custom menu will show up with no menu items.

The code for the menu items shown above is.

```
KAction *testCustomAction = new KAction( i18n( "Custom Action" ), BarIconSet( "kbugbuster" ),
KShortcut::null(), this, SLOT( testAction() ), actionCollection(), "custom_action" );
KAction *testCustomAction2 = new KAction( i18n( "Sub Menu Option" ), BarIconSet( "kbugbuster" ),
```

Chapter 14 A Simple Editor Application

```
KShortcut::null(), this, SLOT( testAction() ),      actionCollection(), "subMenu_action" );
KAction *testCustomAction3 = new KAction( i18n( "Another Test" ),      BarIconSet( "kbugbuster" ),
KShortcut::null(), this, SLOT( testAction() ),      actionCollection(), "AnotherTest" );
```

Chapterfourteenfiles

You may have noticed by now that the idea behind the Chapterfourteenfiles project is to give an example of saving and loading files and use the spell check features of KDE. The idea is that we will have a simple notepad type application which can save and load files and use all the cut copy and paste options that should come with this type of application. Fortunatly the cut copy and paste functionality is already a part of the KTextEdit that we are using, so all we need to do is link up the items with.

```
void ChapterFourteenFiles::editCut()
{
    mainWidget->editBox->cut();
}

void ChapterFourteenFiles::editCopy()
{
    mainWidget->editBox->copy();
}

void ChapterFourteenFiles::editPaste()
{
    mainWidget->editBox->paste();
}
```

As you may remember before the docbook sidetrack we declared our actions using KStdAction,

```
KStdAction::cut( this, SLOT( editCut() ), actionCollection() );
KStdAction::copy( this, SLOT( editCopy() ), actionCollection() );
KStdAction::paste( this, SLOT( editPaste() ), actionCollection() );
```

then it was just a matter of calling the KEditBox functions. We do this by changing the way that our KApplication class works slightly. If you remember the standard way of setting the main widget is with the code,

```
setCentralWidget( new WidgetClass( this ) );
```

The change we make is to add the main widget to the class header with,

```
ChapterFourteenFilesWidget *mainWidget;
```

and then in the constructor,

```
mainWidget = new ChapterFourteenFilesWidget( this );
setCentralWidget( mainWidget );
```

This sets up the main widget for the application in exactly the way that we are used to but allows us to separate the functions that are application specific from the functions that are widget specific.

Loading And Saving

We've already seen how to load a file before using the QTextStream class and as we are dealing with plain text files here then we can use identical code,

Chapter 14 A Simple Editor Application

```
KURL urlOpenFile = KFileDialog::getOpenURL( 0, "*..*|All Files" );
if( urlOpenFile.pathOrURL() == QString::null )
{
    /// on cancel reload current file
    QFile file( currentFile() );
    if( file.open( IO_ReadOnly ) == true )
    {
        QTextStream stream( &file );

        mainWindow->editBox->append( stream.read() );

        file.close();

        mainWindow->editBox->setModified( false );
    }

    return;
}
```

Here we use a standard dialog to get the name of the file to be opened and then open the `QFile` object and pass it to a `QTextStream` before reading the entire contents into the edit box. Saving is very similar although for some reason the `QTextStream` doesn't have a write function so we have to use the C++ output operator

```
QFile file( currentFile() );
if( file.open( IO_WriteOnly | IO_Truncate ) == true )
{
    QTextStream stream( &file );

    stream << mainWindow->editBox->text();

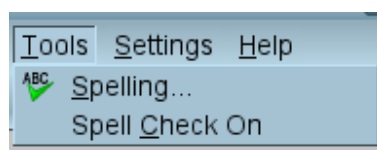
    file.close();

    statusBar()->message( currentFile() + " saved", 3000 );
}
else
{
    KMessageBox::error( this, "Unable to open the file for saving " + currentFile() );
}
```

Here we write the contents of the `KTextEdit` out to the file, opening the file with `IO_WriteOnly` and `IO_Truncate` so that if we are saving to a current file that contains text the contents are replaced with the contents of the edit box.

Using The Spell Check

If you look at the tools menu you will see two options,



The first is `Spelling` which is the main spell checker which will bring up the spell check dialog the second is the background spell checker which doesn't show a dialog but indicates an incorrectly

Chapter 14 A Simple Editor Application

spelled word by highlighting it in red.

Tip Of The Day

There is one oddity about the tools menu here and that is that the Tools menu is not declared by the programmer as such. It is declared by the code,

```
KStdAction::spelling( this, SLOT( spellCheck() ), actionCollection() );
```

While the code for the Spell Check On option is more what we are used to in that it reads,

```
KToggleAction *spellCheckerAction = new KToggleAction( i18n( "Spell Check On" ), 0, KShortcut::null(), this,
SLOT( spellCheckerOn() ), actionCollection(), "enableSpellCheck" );
```

The Tools menu is created by default when we call the KStdAction::spelling function, adding our action to the action collection and then call setupGui. So how do we get the Spell Check On item to appear on the menu at all. The answer is in the resource file for the menus,

```
<Menu name="tools"><text>&Tools</text>
  <Action name="enableSpellCheck" />
</Menu>
```

As you can see the enableSpellCheck action is added to the “tools” menu. When the setupGui function sets up the menu it names the menus with the menu name in lower case and removes any ampersand characters so if you wanted to add a custom option to the file menu you would add it to the resource file with the menu name “file”.

Setting Up The Spell Checker

The set up for the spell checker is a bit technical. If you want to get it working properly there are a couple of things you must do in the right order. Initially we declare,

```
KSpell *spellingCheck;
```

in our class definition and then initialise it in our constructor.

```
spellingCheck = new KSpell( this, "Check Spelling", this,    SLOT( spellCheckReceiver( KSpell * ) ) );
```

Now you see here that there is a slot in the constructor called spellCheckReceiver which is important because what happens is that unlike other objects the KSpell class is going to take a while to initialise and when it is done it will send a ready(KSpell *) signal. You must receive this signal or no spell checking for you. So our spell check receiver function reads,

```
void ChapterFourteenFiles::spellCheckReceiver( KSpell *spellChecker )
{
    connect( spellChecker, SIGNAL( done( const QString& ) ), this,    SLOT( spellCheckerDone( const QString & ) ) );
}
```

Here once the KSpell object is initialised then and only then do we set up a slot on it to respond to the signal done. (You should note that the KSpell * passed to this function is the KSpell * defined as spellingCheck as you would see if you set a debug breakpoint in this function.) The done signal is sent when the KSpell object has finished checking the spelling. The way it works is that when you call the KSpell check function you pass it a string and when it has finished checking the string it passes the corrected string to your done slot which in this case is spellCheckerDone,

```
void ChapterFourteenFiles::spellCheckerDone( const QString &buffer )
{
    mainWidget->editBox->clear();
}
```


Chapter 14 A Simple Editor Application

```
mainWidget->editBox->append( buffer );  
}
```

When the text is returned we remove all text from the edit box and then replace it with the text sent to us from the spell checker.

One final thing you need to do when using the spell checker is call the cleanup function so that any resources allocated by it can be cleaned up. You should do this in the destructor with,

```
spellingCheck->cleanUp();
```

Be warned though if you put this function call anywhere else in your code the spell checker will stop working, if it works at all.

Using The Spell Checker

To start with if you select the Spell Check on option it triggers this code,

```
if( mainWidget->editBox->checkSpellingEnabled() == true )  
{  
    mainWidget->editBox->setCheckSpellingEnabled( false );  
}  
else  
{  
    mainWidget->editBox->setCheckSpellingEnabled( true );  
}
```

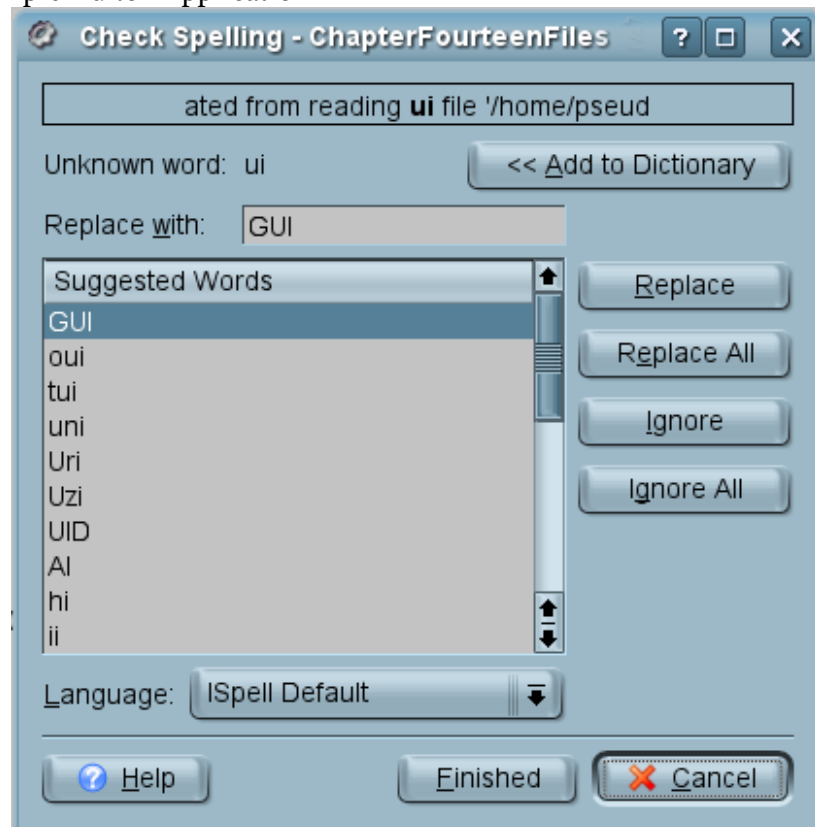
which turns on and off the background spell checker for the KTextEdit which looks like this,

```
void* ChapterFourteenFiles::qt_cast( const char* cname )  
{  
    if ( !qstrcmp( cname, "ChapterFourteenFiles" ) )  
        return this;  
    return KMainWindow::qt_cast( cname );  
}
```

and that's all there is to the background spell checker in default mode, However, because we have added the KSpell class as shown above we have two further options the first being to start the spell checker to check the spelling of the entire document using the menu option, “Spelling” or we can individually check each word by double clicking on it.

The dialog when you start the spelling check through the menu is,

Chapter 14 A Simple Editor Application



This is what you get when you start the spelling checker with one of our header files loaded the first word it picks up on is the word `ui` and then gives you the options you would expect of any spelling check software. The code for this is,

```
spellingCheck = new KSpell( this, "Check Spelling", this,      SLOT( spellCheckReceiver( KSpell * ) ) );
```

```
setDoubleClicked( false );  
QString string = mainWindow->editBox->text();  
spellingCheck->check( string );
```

We initialise the `spellingCheck` object passing it the `SLOT` for the receiver function which is called when the object is ready as discussed above and then get all the text from the edit box and pass it to the `spellingCheck` object with the `check` function.

The `spellCheckReceiver` function simply sets up the slot that is to be called when we have finished with the spelling check.

```
connect( spellChecker, SIGNAL( done( const QString& ) ), this,      SLOT( spellCheckerDone( const QString & ) ) );
```

When the spelling check is done we check first of all to see if the value returned from the spelling check dialog was `KS_CANCEL` which would mean we do nothing. Then we check if we double clicked on a single word and if not we simply replace all the text,

```
if( doubleClicked() == false )  
{  
    mainWindow->editBox->clear();  
    mainWindow->editBox->append( buffer );  
}
```

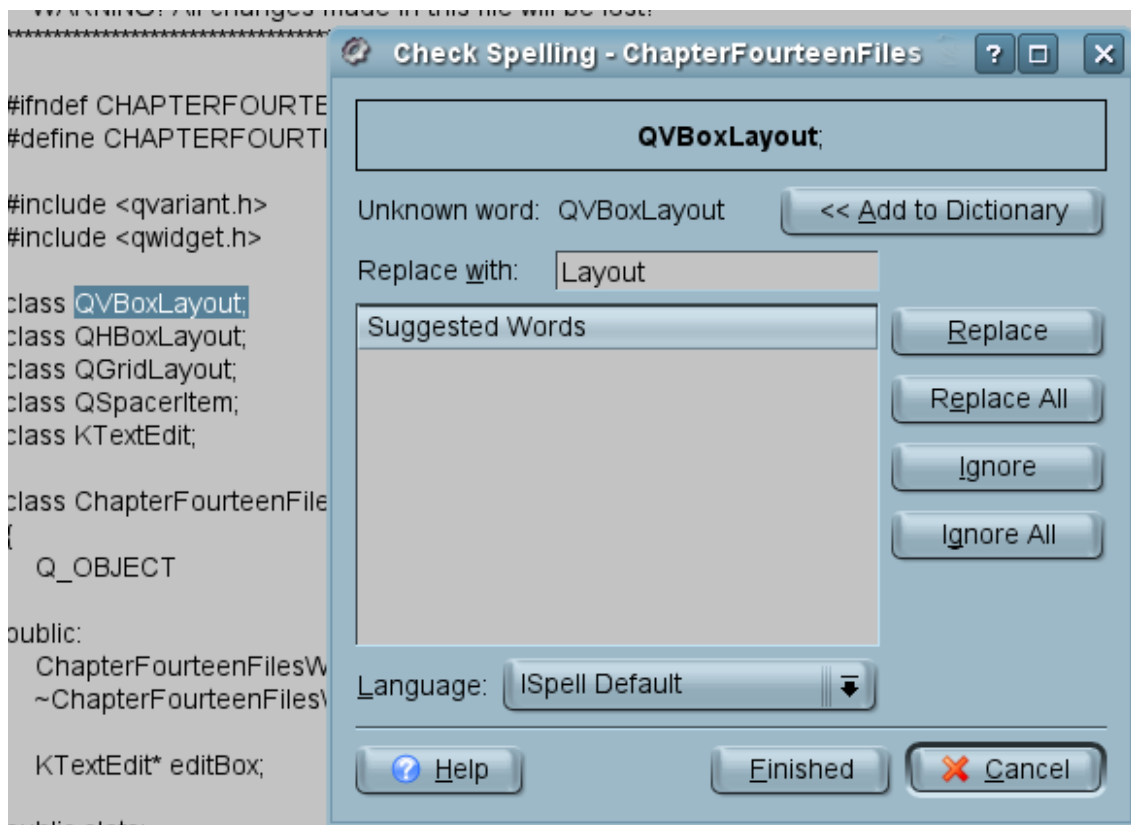
If we are doing a spelling check on a single word things are more complicated because we can't simply replace all the text. When we double click on a word we get the `textEditDoubleClicked`

Chapter 14 A Simple Editor Application

function,

```
spellingCheck = new KSpell( this, "Check Spelling", this, SLOT( spellCheckReciever( KSpell * ) ) );
if( mainWidget->editBox->hasSelectedText() == true )
{
    setDoubleClicked( true );
    setPara( para );
    setPos( pos );
    QString strHighlighted = mainWidget->editBox->selectedText();
    setOriginalWord( strHighlighted );
    spellingCheck->check( strHighlighted );
}
```

We start by creating a KSpell object as before and then saving all the information we need about the current word and it's position within the document, before passing the word to the spell checker.



As you can see here I've opened one of the header files and double clicked the word QVBoxLayout which you can see highlighted and chosen to replace it with the word layout.

When we hit either Replace or Replace All the spellCheckerDone function will be called with the return values KS_REPLACE or KS_REPLACEALL, KS_REPLACE which is used to replace a single word and KS_REPLACEALL which replaces all the spellings of a given word in the text.

There are two ways we can replace words we can work directly with the edit box and replace each individual word by highlighting the word and then cutting it out of the edit box and adding the spelling corrected word at its current position, like so,

```
if( spellingCheck->dlgResult() == KS_REPLACE )
{
    // replace a single word
```

Chapter 14 A Simple Editor Application

```
mainWidget->editBox->setCursorPosition( para(), pos() );
mainWidget->editBox->removeSelection();
mainWidget->editBox->moveCursor( QTextEdit::MoveWordBackward, true );
mainWidget->editBox->removeSelectedText();
mainWidget->editBox->removeSelection();
mainWidget->editBox->moveCursor( QTextEdit::MoveWordForward, true );
mainWidget->editBox->removeSelectedText();
mainWidget->editBox->insert( buffer + " " );

statusBar()->message( i18n( "Replaced " + originalWord() + " with " +      buffer ), 3000 );
}
```

or you could just do it the easy way and use the QString functionality,

```
if( spellingCheck->dlgResult() == KS_REPLACEALL )
{
    QString strText = mainWidget->editBox->text();

    strText.replace( originalWord(), buffer, true );

    mainWidget->editBox->clear();

    mainWidget->editBox->append( strText );

    statusBar()->message( i18n( "Replaced all occurrences of " + originalWord() +      " with " + buffer ), 3000 );
}
```

Finishing The Docbook

The rest of the docbook is mostly just editing the content that is already there so it all points to the correct places and removing things that don't apply. The only code that's added is the menu items for the Test Menu, Custom and Tools menu. There is nothing we didn't look at earlier in these menus with the Tools menu looking like,

```
<sect2>
<title>The <guimenu>Tools</guimenu> Menu</title>
<variablelist>
  <varlistentry>
    <term>
      <menuchoice>
        <guimenu>Tools</guimenu>
        <guimenuitem>Spell Checker</guimenuitem>
      </menuchoice>
    </term>
    <listitem>
      <para>
        <action>Launch the KDE spell check</action>
      </para>
    </listitem>
  </varlistentry>
  <varlistentry>
    <term>
      <menuchoice>
        <guimenu>Tools</guimenu>
        <guimenuitem>Spell Checker On</guimenuitem>
      </menuchoice>
    </term>
  </varlistentry>
</variablelist>
```

Chapter 14 A Simple Editor Application

```
<listitem>
  <para>
    <action>Background spell checking ( misspelt words highlighted in red )</action>
  </para>
</listitem>
</varlistentry>
</variablelist>
</sect2>
```

and the finished project,

The ChapterFourteenFiles 0.1 Handbook

pseudonym67 <pseudonym67@hotmail.com>

Revision 0.1 (2001-10-18)

Copyright © 1999, 2008 pseudonym67

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in [the section entitled "GNU Free Documentation License"](#).

ChapterFourteenFiles 0.1 is a small notepad type file editor program

Table of Contents

[1. Introduction](#)

[2. Using ChapterFourteenFiles 0.1](#)

[More ChapterFourteenFiles 0.1 features](#)

[3. Command Reference](#)

[The main ChapterFourteenFiles 0.1 window](#)

[The File Menu](#)

[The Test Menu](#)

[The Custom Menu](#)

[The Tools Menu](#)

[The Help Menu](#)

[4. Credits and License](#)

[A. Installation](#)

[How to obtain ChapterFourteenFiles 0.1](#)

[Requirements](#)

[Compilation and Installation](#)

Chapter 14 A Simple Editor Application

Summary

In this chapter we have built a simple editor application and shown how to get the program running correctly and how to get the spell checker working, we have also been given a preview of how all the projects for KDE will be written for the version 4.x edition of this book.

Chapter Appendix A Upgrading KDevelop

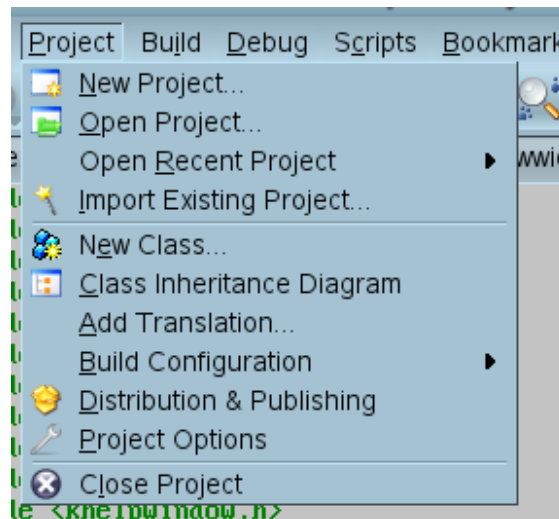
The initial project files for this book were written using Suse 10 and this was used as the development environment for the simple reason that I'm not prepared to re test every bit of code for every single point release of Suse. However as my development machine has just had a complete hardware meltdown, there have been some changes. The main one being a shiny Samsung R700 to use as a dedicated Linux programming environment. So the current development is taking place on openSuse 10.3 until openSuse 11 is released and then there will be another upgrade.

As some people have noticed the later versions of openSuse are not compitable, in fact projects written in Suse 10 will not compile on later versions of openSuse.

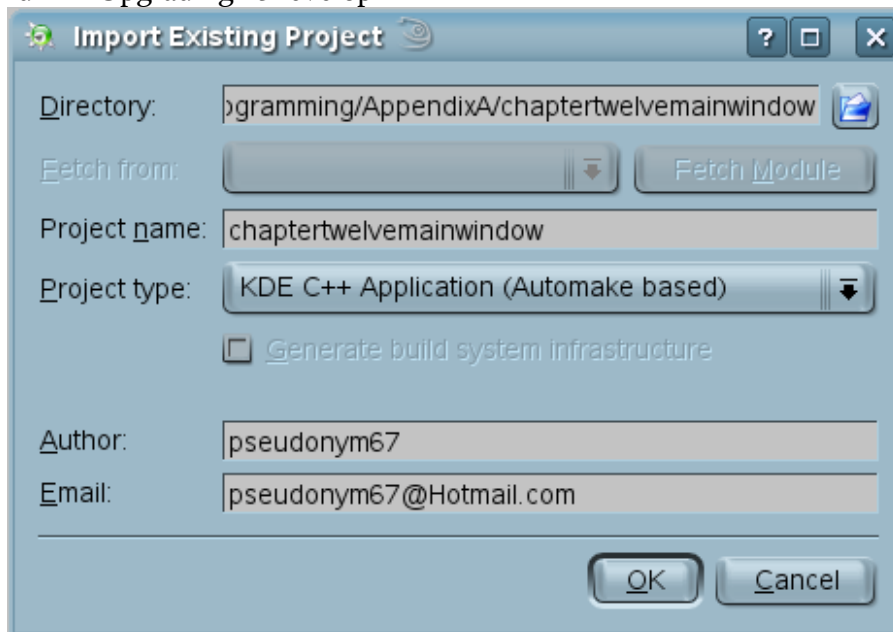
If you try you get something like this.

```
gmake: *** No rule to make target `configure.in', needed by `/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/ChapterTwelve/chaptertwelvemainwindow/configure.in'.
cd /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/ChapterTwelve/chaptertwelvemainwindow && gmake -f admin/Makefile.common subdirs
/bin/sh: line 0: cd: /home/pseudonym67/Dev/KDE/BeginningKDEProgramming/ChapterTwelve/chaptertwelvemainwindow: No such file or directory
gmake: *** [/home/pseudonym67/Dev/KDE/BeginningKDEProgramming/ChapterTwelve/chaptertwelvemainwindow/subdirs] Error 1
gmake: Failed to remake makefile `Makefile'.
gmake: Target `all' not remade because of errors.
```

There is a simple way around this, it's just a matter of getting KDevelop to recognise that you are using an older project file. As an example I am using a copy of the ChapterTwelveMainWindow project. To update the project select the project menu,.

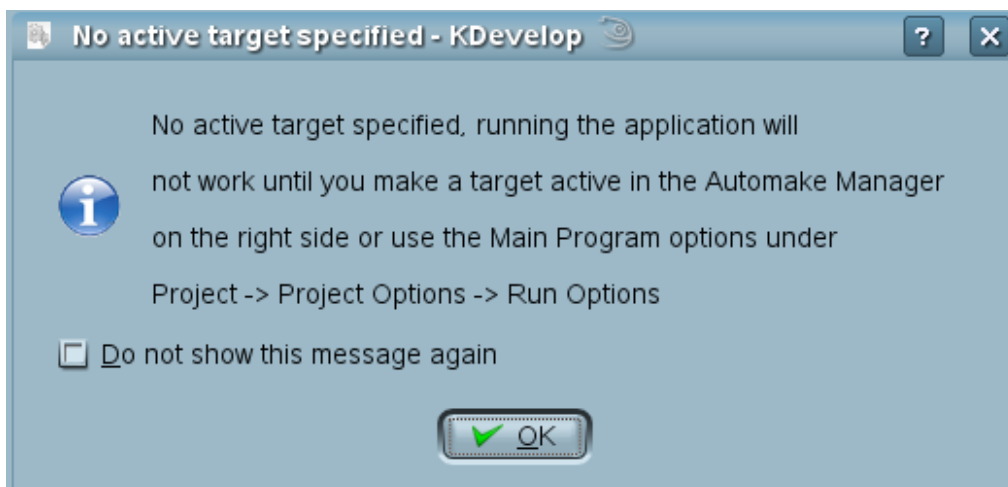


and you'll notice that the fourth item down on the list is the Import Existing Project wizard. Select this and you'll get a dialog.



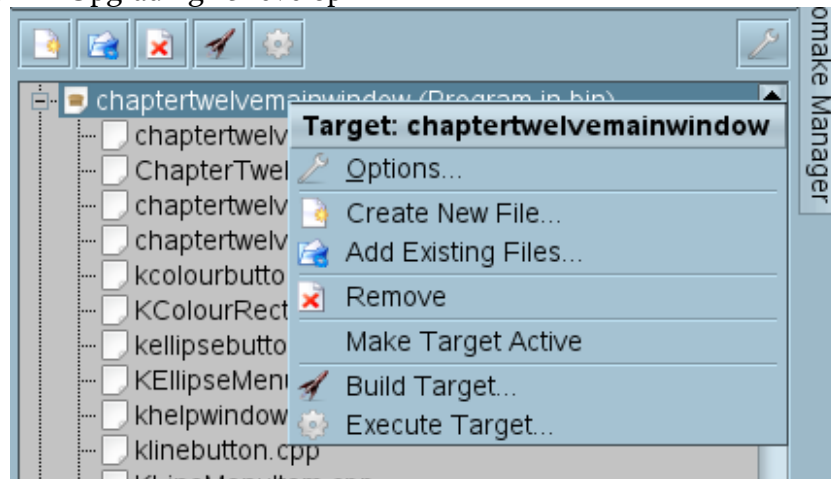
The dialog for the wizard is quite straight forward you choose the directory for the project, which in this case is AppendixA/chaptertwelvemainwindow, once you select the directory the wizard will search it for project files. Then choose the project type. All the projects for this manual have been KDE C++ Application's.

Hit O.K. and you'll get the dialog,

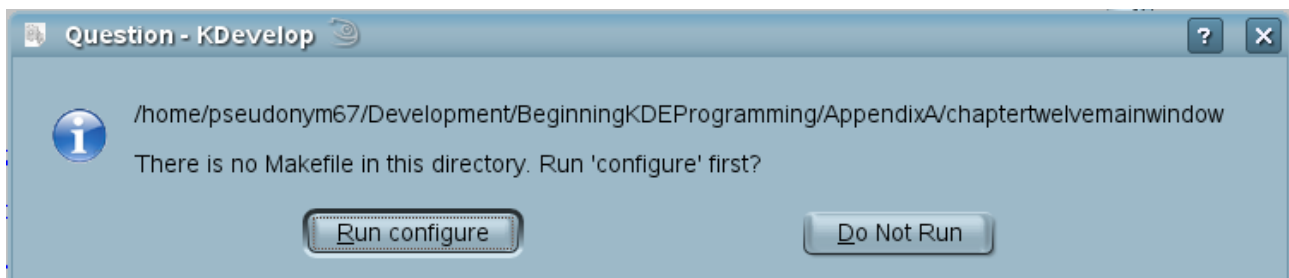


Just click O.K. and open the Automake Manager for the newly opened project. Right click on the project name and select Make Target Active.

Chapter Appendix A Upgrading KDevelop



Now before you do anything else delete the debug directory for the project. This may not always be strictly necessary but it will ensure a complete new completely clean build of the project. If you hit build you'll get



this dialog just hit Run configure and let it go. At this point it should build fine and then if you run you'll get,