

# The KDevelop Programming Handbook

Ralf Nolden and Caleb Tennis



# The KDevelop Programming Handbook

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What you should know already . . . . .	1
1.2	About this Handbook . . . . .	1
1.2.1	In the next chapter . . . . .	2
1.2.2	In the following chapters . . . . .	2
1.3	Additional Information . . . . .	2
<b>2</b>	<b>The KDE and Qt Libraries</b>	<b>4</b>
2.1	The Qt GUI Toolkit . . . . .	5
2.1.1	The first Qt Application . . . . .	5
2.1.2	The Reference Documentation for Qt . . . . .	6
2.1.2.1	Interpretation of the Sample . . . . .	6
2.1.3	User Interaction . . . . .	8
2.1.4	Object Interaction by Signals and Slots . . . . .	9
2.1.4.1	Sample Usage . . . . .	10
2.2	What KDE provides . . . . .	11
2.2.1	The KDE 3.x libraries . . . . .	11
2.2.2	Example KDE Application . . . . .	11
<b>3</b>	<b>Creating New Applications</b>	<b>13</b>
3.1	The Application Wizard . . . . .	13
3.2	Invoking the Application Wizard and Project Generation . . . . .	14
3.2.1	Starting the Application Wizard and the First Page . . . . .	14
3.2.2	Version control information . . . . .	14
3.2.3	Header and Source Templates . . . . .	15
3.2.4	Finishing Up . . . . .	15

# The KDevelop Programming Handbook

3.3	The First Build . . . . .	15
3.4	The source skeleton . . . . .	20
3.4.1	The main() function . . . . .	20
3.4.2	User Application Start . . . . .	21
3.4.3	The Constructor . . . . .	21
<b>4</b>	<b>Application View Design</b>	<b>23</b>
4.1	Introduction . . . . .	23
4.2	Using Library Views . . . . .	24
4.2.1	Qt Views . . . . .	25
4.2.2	KDE Views . . . . .	26
4.3	Creating your own Views . . . . .	26
<b>5</b>	<b>Configuring Menubars and Toolbars</b>	<b>27</b>
5.1	Introduction . . . . .	27
5.2	How does it work? . . . . .	27
5.3	Keyboard Accelerator Configuration . . . . .	28
<b>6</b>	<b>Help Functions</b>	<b>29</b>
6.1	Introduction . . . . .	29
6.2	Tool-Tips . . . . .	30
6.3	Extending the Statusbar . . . . .	30
6.4	The What's This...? Button . . . . .	30
<b>7</b>	<b>Documentation</b>	<b>31</b>
7.1	Introduction . . . . .	31
7.2	User Documentation . . . . .	31
7.3	Programmer Documentation . . . . .	31
<b>8</b>	<b>Internationalization</b>	<b>32</b>
8.1	Introdction . . . . .	32
<b>9</b>	<b>Credits</b>	<b>33</b>
<b>A</b>	<b>Bibliography</b>	<b>34</b>
A.0.0.0.0.1	Bibliography . . . . .	34

## **Abstract**

The User Guide to C++ Application Design for the K Desktop Environment (KDE) with the KDevelop IDE

# Chapter 1

## Introduction

As Unix Systems are becoming more and more popular to even beginners working with computer machines due to its advantages in regards of stability and functionality, most are somehow disappointed, because those applications don't have a consistent look and each one behaves different from another. With KDE, developers have an almost perfect way to create first-class applications for Unix desktop systems to get a wider user community by the mere quality their applications have to offer. Therefore, KDE becomes more and more popular as a base for programming design, and developers want to take advantage of the possibilities that the system has to offer.

### 1.1 What you should know already

For making the best use of this programming handbook, we assume that you already know about the C++ programming language; if not, you should make yourself familiar with that first. Information about C++ is available through various sources either in printed form at your local bookstore or by tutorials found on the Internet. Knowledge about the design of Graphical User Interfaces is not required, as this handbook tries to cover the application design for KDE programs, which also includes an introduction into the Qt toolkit as well as the KDE libraries and the design of User Interfaces. Also, you should have made yourself comfortable with KDevelop by reading The User Manual to KDevelop, which contains a descriptive review of the functionality provided by the IDE.

### 1.2 About this Handbook

This handbook has been written to give developers an introduction into KDE application development by using the KDevelop Integrated Development Environment.

## The KDevelop Programming Handbook

The following chapters therefore give an introduction on how to create projects, explains the sourcecode already generated and shows how to extend the given sources on various topics such as toolbars, menu bars and view areas.

Then the dialogeditor is discussed in detail, explaining how widgets are created and covers widget properties settings in detail for all provided widgets.

Finally, you will learn about several topics that will complete your knowledge in regards of project design and helps you work out additional issues besides coding such as adding API documentation and extending online-manuals.

### 1.2.1 In the next chapter

We'll take a look at the Qt and KDE libraries, showing basic concepts and why things are the way they are. Also, we will discuss how to create the tutorial applications provided with the Qt toolkit by using `kdevelop;`, so beginners can already see first results with a few steps, and thereby will learn how to make use of some of KDevelop's best features.

### 1.2.2 In the following chapters

You will learn how to:

- create an application with the KAppWizard
- What the project skeleton already provides
- What the code already create means
- How to create your own views
- How to extend your application's functionality by dialog, menu bars, and toolbars
- How to make your application user friendly by providing help functions
- How to write online documentation

## 1.3 Additional Information

Additional information about Qt/KDE programming is available by various sources:

- [Programming with Qt](#) by Matthias Kalle Dalheimer
- [The User Manual to KDevelop](#), provided with the KDevelop IDE
- [The Online Reference to the Qt library](#)

## The KDevelop Programming Handbook

- [The KDE Developer web site](#)

Additionally, you should look for help by subscribing to the various mailing lists, whose addresses are available on the mentioned web sites, and on the Usenet newsgroups dedicated to users of KDE and Unix Systems as well as about the C and C++ programming language.

For obtaining help about the KDevelop IDE, you should send requests to our mailinglist at [kdevelop@kdevelop.org](mailto:kdevelop@kdevelop.org). Mind that the KDevelop team is dedicated to provide the means to enable you to program applications and therefore is not intended as a technical support team in cases where the applications you're developing don't work due to implementation errors or misconfigurations of your operating system. By this, we ask all users to take advantage of the mailinglist in any case you're running into problems with the use of the IDE itself, as well as for bug reports and suggestions for improving the functionality of the development environment.

## Chapter 2

# The KDE and Qt Libraries

The Norwegian company TrollTech (<http://www.trolltech.com>) provides a so-called GUI toolkit, named Qt. GUI means "Graphical User Interface", and therefore, Qt-based applications represent themselves with buttons, windows etc, allowing user input by visualizing the functions an application provides. Such a toolkit is needed for developing graphical applications that run on the X-Window interface on Unix Systems, because X does not contain a pre-defined user interface itself. Although other toolkits are also available to create User Interfaces, Qt offers some technical advantages that make application design very easy. Additionally, the Qt toolkit is also available for Microsoft Windows systems, which allows developers to provide their applications for both platforms.

The KDE Team (<http://www.kde.org>) joined together with the goal to make using Unix Systems more friendly, and decided to use the Qt toolkit for the development of a window manager on X-Windows, plus a variety of tools included with the KDE packages. The K Desktop Environment therefore contains the window manager kwm, the file manager kfm and the launch panel kpanel as the main components plus a variety of first-class utilities and applications. After KDE was out, a lot of developers turned their eyes towards the new environment and what it has to offer them. The KDE libraries are providing essential methods and classes that make all applications designed with them look similar and consistent, so the user has the great advantage that he only has to get accustomed with an application's specific usage, not with handling dialogs or buttons. Also, KDE programs integrate themselves into the desktop and are able to interact with the file manager via drag'n drop, offer session management and many more, if all features offered by the KDE libraries are used. Both, the Qt toolkit and the KDE libraries, are implemented in the C++ programming language; therefore applications that make use of these libraries are also mostly written in C++. In the following chapter, we'll make a short trip through the libraries to see what already is provided and how Qt and KDE applications are created in general.

Both, the Qt toolkit and the KDE libraries, are implemented in the C++ programming language; therefore applications that make use of these libraries are

also mostly written in C++. In the following chapter, we'll make a short trip through the libraries to see what already is provided and how Qt and KDE applications are created in general.

## 2.1 The Qt GUI Toolkit

As said, the Qt library is a toolkit that offers graphical elements that are used for creating GUI applications and are needed for X-Window programming. Additionally, the toolkit offers:

- A complete set of classes and methods ready to use even for non-graphical programming issues
- A good solution towards user interaction by virtual methods and the signal/slot mechanism
- A set of predefined GUI-elements, called "widgets", that can be used easily for creating the visible elements
- Additional completely pre-defined dialogs that are often used in applications such as progress and file dialogs

Therefore knowing the Qt classes is very essential, even if you only want to program KDE-applications. To have an impression on the basic concept how GUI-applications are constructed and compiled, we'll first have a look at a sample Qt-only program; then we'll extend it to a KDE program.

### 2.1.1 The first Qt Application

As usual, programs in C++ have to contain a `main()` function, which is the starting point for application execution. As we want them to be graphically visible in windows and offering user interaction, we first have to know, how they can show themselves to the user. For an example, we'll have a look at the first tutorial included with the Qt Online Reference Documentation and explain the basic execution steps; also why and how the application window appears:

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    QPushButton hello( "Hello world!", 0 );
    hello.resize( 100, 30 );
}
```

```
a.setMainWidget ( &hello );  
hello.show ();  
return a.exec ();  
}
```

This application merely paints a window containing a button with "Hello world" as its text. As for all Qt-based applications, you first have to create an instance of the class `QApplication`, represented by variable `a`.

Next, the program creates an instance of the class `QPushButton` called `hello`, this will be the button. The constructor of `hello` gets a string as a parameter, which is the contents of the widget visible as the buttons text.

Then the `resize()` method is called on the `hello` button. This changes the default size a widget (which is in this case the `QPushButton`) has when created to the length of 100 pixels and the height of 30 pixels. Finally, the `setMainWidget()` method is called for `a` and the `show()` method for `hello`. The `QApplication` is finally executed by `a.exec()`, enters the main event loop and waits until it has to return an integer value to the overlaying Operating System signaling that the application is exited.

## 2.1.2 The Reference Documentation for Qt

Now, let's have a quick look at the reference documentation of the Qt library. To do this, start KDevelop and select "Qt" from the tree in the Documentation tab. The documentation browser opens and shows you the start page of the Qt reference. This will be your first place to get information about Qt, it's classes and the available functions they provide. Also, the above program is the first that is included in the tutorials section. To get to the classes we want to have a look at, `QApplication` and `QPushButton`, select "Alphabetical Class List" and search for the according names. Follow either of them to have a look at the class documentation.

Alternatively, you can use the online documentation from Trolltech's [Qt Documentation](#)

For `QApplication`, you will see the constructor and all other methods that this class provides. If you follow a link, you will get more information about the usage and meaning of the methods, which is very useful when you sometimes can't detect the correct use or want to have an example. This also counts for the KDE library documentation, which uses a similar documentation type; therefore this is almost all you have to know about using the class-references with the documentation browser.

### 2.1.2.1 Interpretation of the Sample

Starting with `QApplication`, you will find all the methods used in our first example:

- the constructor `QApplication()`

## The KDevelop Programming Handbook

- the `setMainWidget()` method
- the `exec()` method

The interpretation why we use these methods is very simple:

1. Create an instance of the class `QApplication` with the constructor, so we can make use of the GUI elements provided by Qt
2. Create a widget which will be the contents of our program window
3. Set the widget as the main widget for a
4. Execute the a instance of `QApplication`

The second object of our program is the pushbutton, an instance of the class `QPushButton`. From the two constructors given to create an instance, we used the second: this accepts a text, which is the label contents of the button; here, it is the string "Hello world!". Then we called the `resize()` method to change the size of the button according to it's contents - the button has to be larger to make the string completely visible.

But what about the `show()` method? Now, you see that like most other widgets, `QPushButton` is based on a single inheritance, the documentation says, Inherits `QPushButton`. Follow the link to the `QPushButton` class. This shows you a lot of other widgets that are inherited by `QPushButton`, which we'll use later to explain the signal/slot mechanism. Anyway, the `show()` method is not listed, therefore, it must be a method that is provided by inheritance as well. The class that `QPushButton` inherits is `QWidget`. Just follow the link again, and you will see a whole bunch of methods that the `QWidget` class provides; including the `show()` method. Now we understand what was done in the sample with the button:

1. Create an instance of `QPushButton`, use the second constructor to set the button text
2. Resize the widget to its contents
3. Set the widget as the main widget of the `QApplication` instance a
4. Tell the widget to display itself on the screen by calling `show()`, an inherited method from `QWidget`

After calling the `exec()` method, the application is visible to the user, showing a window with the button showing "Hello world!". Note: GUI programs behave somewhat differently than procedural applications. The main thing here is that the application enters a so-called "main event loop". This means that the program has to wait for user actions and then react to it, also that for a Qt application, the program has to be in the main event loop to start the event handling. The next section tells you in short what this means to the programmer and what Qt offers to process user events.

**NOTE**

For already advanced users: The button has no parent declared in the constructor, therefore it is a top-level widget alone and runs in a local event loop which doesn't need to wait for the main event loop. See the QWidget class documentation and The KDE Library Reference Guide

### 2.1.3 User Interaction

After reading the last sections, you should already know:

- What the Qt-library provides in terms of GUI applications
- How a program using Qt is created and
- Where and how to find information about classes that you want to use with the documentation browser

Now we'll turn to give the application "life" by processing user events. Generally, the user has two ways to interact with a program: the mouse and the keyboard. For both ways, a graphical user interface has to provide methods that detect actions and methods that do something as a reaction to these actions.

The Window system therefore sends all interaction events to the according application. The `QApplication` then sends them to the active window as a `QEvent` and the widgets themselves have to decide what to do with them. A widget receives the event and processes `QWidget::event(QEvent*)`, which then decides which event has been executed and how to react; `event()` is therefore the main event handler. Then, the `event()` method passes the event to so-called event filters that determine what happened and what to do with the event. If no filter signs responsible for the event, the specialized event handlers are called. Thereby we can decide between:

- Keyboard events -- TAB and Shift-TAB keys:
  - `virtual void focusInEvent(QFocusEvent *)`
  - `virtual void focusOutEvent(QFocusEvent *)`
- All other keyboard input:
  - `virtual void keyPressEvent(QKeyEvent *)`
  - `virtual void keyReleaseEvent(QKeyEvent *)`
- Mouse movements:
  - `virtual void mouseMoveEvent(QMouseEvent *)`
  - `virtual void enterEvent(QEvent *)`
  - `virtual void leaveEvent(QEvent *)`
- Mouse button actions

- virtual void mousePressEvent (QMouseEvent \*)
- virtual void mouseReleaseEvent (QMouseEvent \*)
- virtual void mouseDoubleClickEvent (QMouseEvent \*)
- Window events containing the widget
  - virtual void moveEvent (QMoveEvent \*)
  - virtual void resizeEvent (QResizeEvent \*)
  - virtual void closeEvent (QCloseEvent \*)

Note that all event functions are virtual and protected; therefore you can re-implement the events that you need in your own widgets and specify how your widget has to react. `QWidget` also contains some other virtual methods that can be useful in your programs; anyway, it is sufficient to know about `QWidget` very well.

### 2.1.4 Object Interaction by Signals and Slots

Now we're coming to the most obvious advantages of the Qt toolkit: the signal/slot mechanism. This offers a very handy and useful solution to object interaction, which usually is solved by callback functions for X-Window toolkits. As this communication requires a strict programming and sometimes makes user interface creation very difficult (as referred by the Qt documentation and explained in *Programming with Qt* by K.Dalheimer), Troll Tech invented a new system where objects can emit signals that can be connected to methods declared as slots. For the C++ part of the programmer, he only has to know some things about this mechanism:

- the class declaration of a class using signals/slots has to contain the `Q_OBJECT` macro at the beginning (without a semicolon); and have to be derived from the `QObject` class
- a signal can be emitted by the keyword `emit`, e.g. `emit signal(parameters);` from within any member function of a class that allows signals/slots
- all signals used by the classes that are not inherited have to be added to the class declaration by a signals section
- all methods that can be connected with a signal are declared in sections with the additional keyword `slot`, e.g. `public slots:` within the class declaration
- the meta-object compiler `moc` has to run over the header file to expand the macros and to produce the implementation (which is not necessary to know). The output files of `moc` are compiled also by the C++ compiler.

Another way to use signals without deriving from `QObject` is to use the `QSignal` class- see the reference documentation for more information and example usage. In the following, we assume you're deriving from `QObject`.

This way, your class is able to send signals anywhere and to provide slots that signals can connect to. By using the signals, you don't have to care about who's receiving it- you just have to emit the signal and whatever slot you want to connect to it can react to the emission. Also the slots can be used as normal methods during implementation.

Now, to connect a signal to a slot, you have to use the `connect()` methods that are provided by `QObject` or, where available, special methods that objects provide to set the connection for a certain signal.

### 2.1.4.1 Sample Usage

To explain the way how to set up object-interaction, we'll take our first example again and extend it by a simple connection:

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    QPushButton hello( "Hello world!" , 0);
    hello.resize( 100, 30 );

    a.setMainWidget( &hello );

    QObject::connect( &hello, SIGNAL( clicked() ), &a, SLOT( quit ←
        () ));

    hello.show();
    return a.exec();
}
```

You see, the only addition to give the button more interaction is to use a `connect()` method: `connect(&hello, SIGNAL( clicked() ), &a, SLOT( quit() ))`; is all you have to add. What is the meaning now? The class declaration of `QObject` says about the `connect()` method:

```
bool connect ( const QObject * sender, const char * signal, const Q-
Object * receiver, const char * member )
```

This means you have to specify a `QObject` instance pointer that is the sender of the signal, meaning that it can emit this signal as first parameter; then you have to specify the signal that you want to connect to. The last two parameters are the receiver object that provides a slot, followed by the member function which actually is the slot that will be executed on signal emission.

By using signals and slots, your program's objects can interact with each other easily without explicitly depending on the type of the receiver object. You will learn more about using this mechanism for productive usage later in this

handbook. More information about the Signals/Slot mechanism can also be found in [The KDE Library Reference Guide](#) and the [Qt online reference](#).

## 2.2 What KDE provides

### 2.2.1 The KDE 3.x libraries

The main KDE libraries you'll be using for creating your own KDE applications are:

- the kdecore library, containing all classes that are non-visible elements to provide application functionality
- the kdeui library, containing user interface elements like menubars, toolbars, etc.
- the kfile library, containing the file selection dialogs

Additionally, for specific solutions KDE offers the following libraries:

- the kdefx library, containing pixmaps, image effects the KStyle extension to QStyle
- the khtml library, containing KDE's html component
- the kjs library, containing KDE's Javascript support
- the kio library, containing low level access to network files
- the kparts library, containing support for re-usable embeddable extendable applications

Next we'll have a look at what is needed to turn out first Qt Application into a KDE one.

### 2.2.2 Example KDE Application

In the following, you will see that writing a KDE application is not much more difficult than a Qt application. For the use of KDE's features, you just have to use some other classes, and you're almost done. As an example, we'll discuss the changed version of the Qt example from above:

```
#include <kapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
    KApplication a( argc, argv );
```

## The KDevelop Programming Handbook

```
QPushButton hello( "Hello world!", 0 );
hello.resize( 100, 30 );

a.setTopWidget( &hello );

QObject::connect( &hello, SIGNAL( clicked() ), &a, SLOT( quit ←
    () ) );

hello.show();
return a.exec();
}
```

You see that first we have changed from `QApplication` to `KApplication`. Further, we had to change the previously used `setMainWidget()` method to `setTopWidget`, which `KApplication` uses to set the main widget. That's it! Your first KDE application is ready - you only have to tell the compiler the KDE include path and the linker to link in the `kdecore` library with `-lkdecore`.

As you now know what at least the `main()` function provides generally and how an application gets visible and allows user and object interaction, we'll go on with the next chapter, where our first application is made with KDevelop. There you can also test everything which was mentioned before and see the effects.

What you should have looked into additionally until now is the reference documentation for Qt, especially the `QApplication`, `QWidget` and `QObject` class and the `kdecore` library documentation for the `KApplication` class. The [KDE Library Reference handbook](#) also covers a complete description about the invocation of the `QApplication` and `KApplication` constructors including command-line argument processing.

## Chapter 3

# Creating New Applications

### 3.1 The Application Wizard

KDevelop's Application Wizard is intended to let you start working on new project with KDevelop. Therefore all of your projects are first created by the wizard, and then you can start building them and extend what is already provided by the source skeleton. You can choose from several project types according to your project goals:

- KDE Application Framework: includes source code for a complete framework structure of a standard KDE application
- QMake Project: Creates an application framework based around Trolltech's qmake configuration system
- Simple hello world program: Creates a C++ terminal based program with no GUI support
- A multitude of other program skeletons

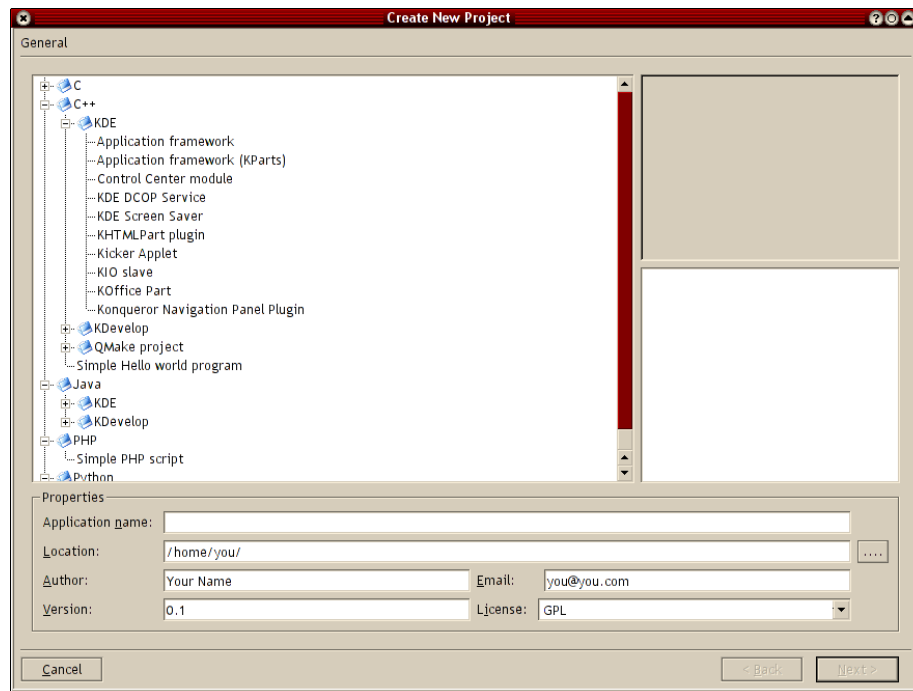
In this chapter we'll see how the Application Wizard can be invoked and what has to be done to generate a KDE application project. This will also be the initial step of our coverage, where we will create the initial version of a sample project. For all other project types the steps are usually the same, but you may not have as many options available.

## 3.2 Invoking the Application Wizard and Project Generation

### 3.2.1 Starting the Application Wizard and the First Page

To start with your KDE application, open KDevelop. From the Project menu, selection New Project. The Application Wizard starts, and you'll see the selection tree on the first page containing available project types that can be created. Choose the C++ subtree, then KDE, then Application Framework.

For our sample project, we are going to create the application KScrubble. Enter this as the application name, and change any other information at the bottom of this screen that may need it. Then, select Next.



### 3.2.2 Version control information

On this screen you have the ability to decide if your project will use a version control system like CVS. For our sample project we will not use source control, so make sure the selection box reads None and select Next.

### 3.2.3 Header and Source Templates

The next two pages show example headers that will go at the top of each of the header and source files that you create using KDevelop. For now, just leave these as the default, and select Next, then Finish. If the Finish button is not activated, you haven't set all of the options correct. Use the Back button to return to earlier menus and correct any mistakes.

### 3.2.4 Finishing Up

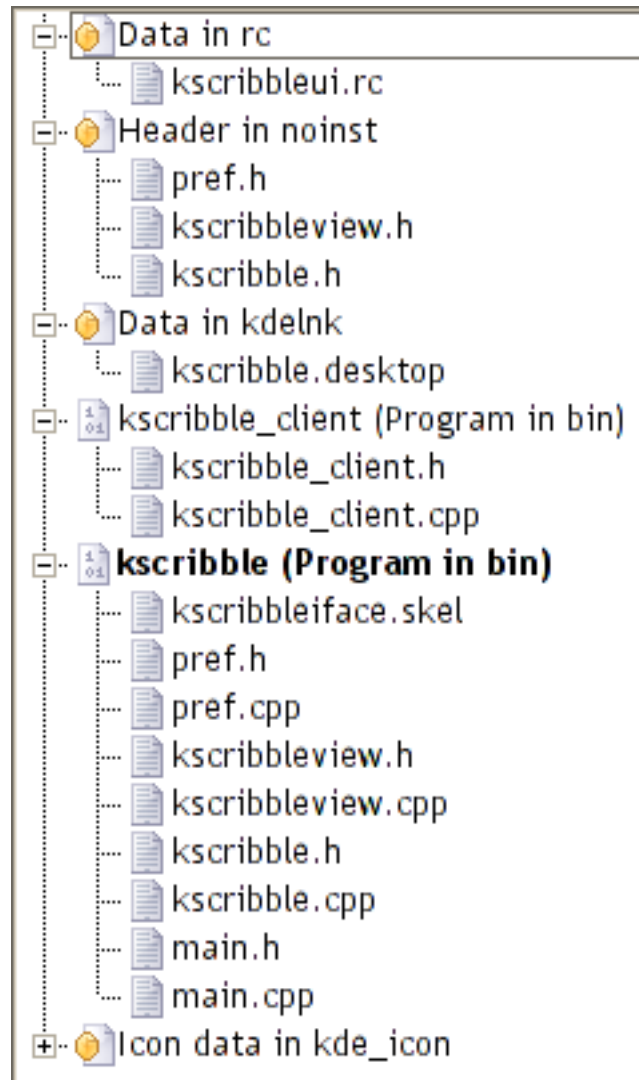
Upon completion, the Application Wizard should close and the messages window should popup displaying information about the tasks that KDevelop is currently doing. At the end of all of the tasks, you should see `*** Success ***`. This means the application framework was successfully loaded.

## 3.3 The First Build

After our project is generated, we'll first make a trip through the source code to get a general understanding of how the application framework looks. This won't only help us get started, but we'll know where to change what in later steps.

This chapter makes the assumption that you understand the basic navigation of KDevelop. Consult the KDevelop User Manual for information if you need it.

The Automake manager shows the project files as follows:



Before diving into the sources, we'll let KDevelop build and run our new application. To do this, select Build Project from the Build menu, or press F8. The output window opens and displays output messages during the compilation phase.

```

1 cd /home/caleb/kscribble && WANT_AUTOCONF_2_5=1 ↵
  WANT_AUTOMAKE_1_6=1 gmake k
2 gmake all-recursive
3 gmake[1]: Entering directory `/home/caleb/kscribble'
4 Making all in doc
5 gmake[2]: Entering directory `/home/caleb/kscribble/doc'
6 Making all in .
7 gmake[3]: Entering directory `/home/caleb/kscribble/doc'

```

## The KDevelop Programming Handbook

```
8 gmake[3]: Nothing to be done for 'all-am'.
9 gmake[3]: Leaving directory '/home/caleb/kscribble/doc'
10 Making all in en
11 gmake[3]: Entering directory '/home/caleb/kscribble/doc/en' ←
12 /usr/local/kde3/bin/meinproc --check --cache index.cache. ←
    bz2 /home/caleb/kscribble/doc/en/index.docbook
13 gmake[3]: Leaving directory '/home/caleb/kscribble/doc/en'
14 gmake[2]: Leaving directory '/home/caleb/kscribble/doc'
15 Making all in po
16 gmake[2]: Entering directory '/home/caleb/kscribble/po'
17 gmake[2]: Nothing to be done for 'all'.
18 gmake[2]: Leaving directory '/home/caleb/kscribble/po'
19 Making all in src
20 gmake[2]: Entering directory '/home/caleb/kscribble/src'
21 source='main.cpp' object='main.o' libtool=no \
22 depfile='.deps/main.Po' tmpdepfile='.deps/main.TPo' \
23 depmode=gcc3 /bin/sh /home/caleb/kscribble/admin/depcomp \
24 g++ -DHAVE_CONFIG_H -I. -I/home/caleb/kscribble/src -I.. - ←
    I/usr/local/kde3/include
    -I/usr/lib/qt/include -I/usr/X11R6/include - ←
        DQT_THREAD_SUPPORT -D_REENTRANT -Wnon-virtual-dtor
    -Wno-long-long -Wundef -Wall -pedantic -W -Wpointer-arith ←
        -Wmissing-prototypes -Wwrite-strings
    -ansi -D_XOPEN_SOURCE=500 -D_BSD_SOURCE -Wcast-align - ←
        Wconversion -O2 -fno-exceptions -fno-check-new
    -c -o main.o `test -f 'main.cpp' || echo '/home/caleb/ ←
        kscribble/src/' `main.cpp
25 /usr/lib/qt/bin/moc /home/caleb/kscribble/src/kscribble.h ←
    -o kscribble.moc
26 source='kscribble.cpp' object='kscribble.o' libtool=no \
27 depfile='.deps/kscribble.Po' tmpdepfile='.deps/kscribble. ←
    TPo' \
28 depmode=gcc3 /bin/sh /home/caleb/kscribble/admin/depcomp \
29 g++ -DHAVE_CONFIG_H -I. -I/home/caleb/kscribble/src -I.. - ←
    I/usr/local/kde3/include
    -I/usr/lib/qt/include -I/usr/X11R6/include - ←
        DQT_THREAD_SUPPORT -D_REENTRANT -Wnon-virtual-dtor
    -Wno-long-long -Wundef -Wall -pedantic -W -Wpointer-arith ←
        -Wmissing-prototypes -Wwrite-strings
    -ansi -D_XOPEN_SOURCE=500 -D_BSD_SOURCE -Wcast-align - ←
        Wconversion -O2 -fno-exceptions -fno-check-new
    -c -o kscribble.o `test -f 'kscribble.cpp' || echo '/home/ ←
        caleb/kscribble/src/' `kscribble.cpp
30 kscribble.cpp: In member function 'void K Scribble:: ←
    setupActions()'
31 kscribble.cpp:107: warning: unused variable 'KAction* ←
    custom'
32 /usr/lib/qt/bin/moc /home/caleb/kscribble/src/ ←
    kscribbleview.h -o kscribbleview.moc
```

## The KDevelop Programming Handbook

```
33 source='kscribbleview.cpp' object='kscribbleview.o' ↵
    libtool=no \
34 depfile='.deps/kscribbleview.Po' tmpdepfile='.deps/ ↵
    kscribbleview.TPo' \
35 depmode=gcc3 /bin/sh /home/caleb/kscribble/admin/depcomp \
36 g++ -DHAVE_CONFIG_H -I. -I/home/caleb/kscribble/src -I.. - ↵
    I/usr/local/kde3/include
-I/usr/lib/qt/include -I/usr/X11R6/include - ↵
    DQT_THREAD_SUPPORT -D_REENTRANT -Wnon-virtual-dtor
-Wno-long-long -Wundef -Wall -pedantic -W -Wpointer-arith ↵
    -Wmissing-prototypes -Wwrite-strings -ansi
-D_XOPEN_SOURCE=500 -D_BSD_SOURCE -Wcast-align - ↵
    Wconversion -O2 -fno-exceptions -fno-check-new -c
-o kscribbleview.o `test -f 'kscribbleview.cpp' || echo '/ ↵
    home/caleb/kscribble/src/' `kscribbleview.cpp
37 kscribbleview.cpp: In member function `void KScribbleView ↵
    ::print(QPainter*,
38 int, int)':
39 kscribbleview.cpp:79: warning: unused parameter `QPainter* ↵
    p'
40 kscribbleview.cpp:79: warning: unused parameter `int ↵
    height'
41 kscribbleview.cpp:79: warning: unused parameter `int width ↵
    ,
42 /usr/lib/qt/bin/moc /home/caleb/kscribble/src/pref.h -o ↵
    pref.moc
43 source='pref.cpp' object='pref.o' libtool=no \
44 depfile='.deps/pref.Po' tmpdepfile='.deps/pref.TPo' \
45 depmode=gcc3 /bin/sh /home/caleb/kscribble/admin/depcomp \
46 g++ -DHAVE_CONFIG_H -I. -I/home/caleb/kscribble/src -I.. - ↵
    I/usr/local/kde3/include
-I/usr/lib/qt/include -I/usr/X11R6/include - ↵
    DQT_THREAD_SUPPORT -D_REENTRANT -Wnon-virtual-dtor
-Wno-long-long -Wundef -Wall -pedantic -W -Wpointer-arith ↵
    -Wmissing-prototypes -Wwrite-strings
-ansi -D_XOPEN_SOURCE=500 -D_BSD_SOURCE -Wcast-align - ↵
    Wconversion -O2 -fno-exceptions -fno-check-new
-c -o pref.o `test -f 'pref.cpp' || echo '/home/caleb/ ↵
    kscribble/src/' `pref.cpp
47 /usr/local/kde3/bin/dcopidl /home/caleb/kscribble/src/ ↵
    kscribbleiface.h > kscribbleiface.kidl ||
( rm -f kscribbleiface.kidl ; /bin/false )
48 /usr/local/kde3/bin/dcopidl2cpp --c++-suffix cpp --no- ↵
    signals --no-stub kscribbleiface.kidl
49 source='kscribbleiface_skel.cpp' object=' ↵
    kscribbleiface_skel.o' libtool=no \
50 depfile='.deps/kscribbleiface_skel.Po' tmpdepfile='.deps/ ↵
    kscribbleiface_skel.TPo' \
51 depmode=gcc3 /bin/sh /home/caleb/kscribble/admin/depcomp \
52 g++ -DHAVE_CONFIG_H -I. -I/home/caleb/kscribble/src -I.. - ↵
```

## The KDevelop Programming Handbook

```
I/usr/local/kde3/include
-I/usr/lib/qt/include -I/usr/X11R6/include - ←
    DQT_THREAD_SUPPORT -D_REENTRANT -Wnon-virtual-dtor
-Wno-long-long -Wundef -Wall -pedantic -W -Wpointer-arith ←
    -Wmissing-prototypes -Wwrite-strings
-ansi -D_XOPEN_SOURCE=500 -D_BSD_SOURCE -Wcast-align - ←
    Wconversion -O2 -fno-exceptions -fno-check-new
-c -o kscribbleiface_skel.o `test -f `kscribbleiface_skel. ←
    cpp` ||
echo `/home/caleb/kscribble/src/`kscribbleiface_skel.cpp
53 /bin/sh ../libtool --silent --mode=link --tag=CXX g++ - ←
    Wnon-virtual-dtor -Wno-long-long -Wundef -Wall
-pedantic -W -Wpointer-arith -Wmissing-prototypes -Wwrite- ←
    strings -ansi -D_XOPEN_SOURCE=500
-D_BSD_SOURCE -Wcast-align -Wconversion -O2 -fno- ←
    exceptions -fno-check-new -o kscribble -R
/usr/local/kde3/lib -R /usr/lib/qt/lib -R /usr/X11R6/lib - ←
    L/usr/X11R6/lib -L/usr/lib/qt/lib
-L/usr/local/kde3/lib main.o kscribble.o kscribbleview.o ←
    pref.o kscribbleiface_skel.o -lkio
54 source=`kscribble_client.cpp` object=`kscribble_client.o` ←
    libtool=no \
55 depfile=`.deps/kscribble_client.Po` tmpdepfile=`.deps/ ←
    kscribble_client.TPo` \
56 depmode=gcc3 /bin/sh /home/caleb/kscribble/admin/depcomp \
57 g++ -DHAVE_CONFIG_H -I. -I/home/caleb/kscribble/src -I.. - ←
    I/usr/local/kde3/include
-I/usr/lib/qt/include -I/usr/X11R6/include - ←
    DQT_THREAD_SUPPORT -D_REENTRANT -Wnon-virtual-dtor
-Wno-long-long -Wundef -Wall -pedantic -W -Wpointer-arith ←
    -Wmissing-prototypes -Wwrite-strings
-ansi -D_XOPEN_SOURCE=500 -D_BSD_SOURCE -Wcast-align - ←
    Wconversion -O2 -fno-exceptions -fno-check-new
-c -o kscribble_client.o `test -f `kscribble_client.cpp` ←
    || echo
/home/caleb/kscribble/src/`kscribble_client.cpp
58 /bin/sh ../libtool --silent --mode=link --tag=CXX g++ - ←
    Wnon-virtual-dtor -Wno-long-long -Wundef
-Wall -pedantic -W -Wpointer-arith -Wmissing-prototypes - ←
    Wwrite-strings -ansi -D_XOPEN_SOURCE=500
-D_BSD_SOURCE -Wcast-align -Wconversion -O2 -fno- ←
    exceptions -fno-check-new -o kscribble_client -R
/usr/local/kde3/lib -R /usr/lib/qt/lib -R /usr/X11R6/lib - ←
    L/usr/X11R6/lib -L/usr/lib/qt/lib
-L/usr/local/kde3/lib kscribble_client.o -lkdecore
59 gmake[2]: Leaving directory `/home/caleb/kscribble/src'
60 gmake[2]: Entering directory `/home/caleb/kscribble'
61 gmake[2]: Nothing to be done for `all-am'.
62 gmake[2]: Leaving directory `/home/caleb/kscribble'
63 gmake[1]: Leaving directory `/home/caleb/kscribble'
```

64 \*\*\* Success \*\*\*

As you can see, we've put line numbers in front of each line which won't appear on your output but it makes it easier to describe what is happening during the build. First of all, gmake works recursively. This means that it starts from the directory it is invoked and goes into the subdirectories first, one at a time, then returns to the directory it was started, processes it, then finishes.

Our first line of interest is 24. Notice on this line that g++, which is our C++ compiler, gets called by make to compile the first source code file in our project - in this case main.cpp. Many extra command line options are also being used with the g++ compiler; some of which are defaults and some of which can be configured via KDevelop.

Before the next file (kscribble.cpp, line 29) is compiled, the moc (meta object compiler) is first invoked on kscribble.h (line 25). This is because KScribble classes use signals/slots, so the Q\_OBJECT macro must be expanded, and the moc does this for us. The resultant file, kscribble.moc, is used by kscribble.cpp via an #include statement inside of the file.

## 3.4 The source skeleton

To conceptualize how a KDE application works, we'll first have a very close look at the source skeleton already provided by the Application Wizard. As we already saw, we're having a set of source and header files that build the initial code for the application and make it ready-to-run. Therefore, the easiest way to explain the code is to follow the implementation line by line as it is processed during executing the program until it enters the main event loop and is ready to accept user input. Then, we'll have a look at the functionality that enables user interaction and how certain things work. This is probably the best way to explain the framework and, as it is similar to almost all KDE applications, will enable you to read source codes from other projects as well; additionally, you will know where to change what part of the code to make your applications behave the way they are designed for.

### 3.4.1 The main() function

As the application begins its execution with entering the main() function, this will be the start for our code examination. The main() function of KScribble is implemented in the file main.cpp and can also be found using the Class Browser by selecting the "Global Functions" folder.

```

1 int main(int argc, char **argv)
2 {
3     KAboutData about("kscribble", I18N_NOOP("KScribble"), ←
        version, description,
4     KAboutData::License_GPL, "(C) 2002 ←
        Your Name", 0, 0, "you@you.com");
```

## The KDevelop Programming Handbook

```
5     about.addAuthor( "Your Name", 0, "you@you.com" );
6     KCmdLineArgs::init(argc, argv, &about);
7     KCmdLineArgs::addCmdLineOptions(options);
8     KApplication app;
9
10    // register ourselves as a dcop client
11    app.dcopClient()->registerAs(app.name(), false);
12
13    // see if we are starting with session management
14    if (app.isRestored())
15        RESTORE(KScribble)
16    else
17    {
18        // no session.. just start up normally
19        KCmdLineArgs *args = KCmdLineArgs::parsedArgs();
20        if (args->count() == 0)
21        {
22            KScribble *widget = new KScribble;
23            widget->show();
24        }
25        else
26        {
27            int i = 0;
28            for (; i < args->count(); i++)
29            {
30                KScribble *widget = new KScribble;
31                widget->show();
32                widget->load(args->url(i));
33            }
34        }
35        args->clear();
36    }
37
38    return app.exec();
39 }
```

Now, what happens first is the usual creation of a `KApplication` object, but we've added some KDE methods that set program and author information for this application.

### 3.4.2 User Application Start

... (not written yet)

### 3.4.3 The Constructor

Let's have a look at the constructor and see how this instance is called

## The KDevelop Programming Handbook

```
1 KScrubble::KScrubble()
2     : KMainWindow( 0, "KScrubble" ),
3       m_view(new KScrubbleView(this)),
4       m_printer(0)
5 {
6     // accept dnd
7     setAcceptDrops(true);
8
9     // tell the KMainWindow that this is indeed the main ←
  widget
10    setCentralWidget(m_view);
11
12    // then, setup our actions
13    setupActions();
14
15    // and a status bar
16    statusBar()->show();
17
18    // allow the view to change the statusbar and caption
19    connect(m_view, SIGNAL(signalChangeStatusbar(const ←
  QString&)),
20           this,    SLOT(changeStatusbar(const QString&)) ←
  );
21    connect(m_view, SIGNAL(signalChangeCaption(const ←
  QString&)),
22           this,    SLOT(changeCaption(const QString&)));
23
24 }
```

Notice that `KScrubble` inherits the `KMainWindow` class - a commonly used base class for KDE applications. We initialize a class called `KScrubbleView` as our central widget, create a `KStatusBar` via the `statusBar()` method (line 16), and connect some signals and slots together.

## Chapter 4

# Application View Design

### 4.1 Introduction

When developing an application with a graphical user interface, the main work takes place in providing a so-called "view" for the application. A view generally is a widget that displays the data of a document and provides methods to manipulate the document contents. This can be done by the user via the events he emits by the keyboard or the mouse; more complex operations are often processed by toolbars and menubars which interact with the view and the document. The statusbar then provides information about the document, view or application status. As an example, we look at how an editor is constructed and where we can find which part.

An editor generally is supposed to provide an interface to view and/or change the contents of a text document for the user. If you start Kate, you see the visual interface as the following:

- The menubar: providing complex operations as well as opening, saving and closing files and exiting the application.
- The toolbar: offers icons which allow quicker access for most needed functions,
- The statusbar: displays the status of the cursor position by the current row and column,
- The view in the center of the window, displaying a document and offering a cursor connected to the keyboard and the mouse to operate on the data.

Now it's easy to understand that a view is the most unique part of the application and the design of the view decides about the usability and acceptability of an application. This means that one of the first steps in development is to determine the purpose of the application and what kind of view design would

match best to allow any user to work with the application with a minimum of work learning how to handle the user interface.

For some purposes like text editing and displaying HTML files, views are provided by the Qt and KDE libraries; we will discuss certain aspects of these high-level widgets in the next section. But for most applications new widgets have to be designed and implemented. It is that what makes a programmer also a designer and where his abilities on creativity are asked. Nevertheless, you should watch for intuitivity first. Remember, a lot of users won't accept an application that isn't:

- graphically nice.
- offering a lot of features
- easy to handle
- fast to learn how to use it

Needless to say that stability is a major design goal. Nobody can prevent bugs, but a minimum can be reached at least by clever design goals and wide use of object-oriented design. C++ makes programming a joy if you know how to exploit it's capabilities- inheritance, information hiding and reusability of already existing code.

When creating a KDE or Qt project, you always have to have a view that inherits QWidget, either by direct inheritance or because the library widget you want to use inherits QWidget. Therefore, the Application Wizard already constructed a view that is an instance of a class yourappView, which inherits QWidget already.

This chapter therefore describes how to use library widgets for creating views of KDE or Qt applications that are generated with KDevelop, then we look at the libraries and what kind of views are already offered.

## 4.2 Using Library Views

When your application design has been set up, you first should look for already existing code that will make your life a lot easier. A part of this search is to look for a widget that can be used as a view or at least as a part of it; either directly or by inheritance. The KDE and Qt libraries already contain a set of widgets that can be used for this purpose. To use them, you have two options:

1. Remove the new view class and create an instance of a library widget; then set this as the view,
2. Change the inheritance of the provided view class to the class of the library widget to use.

In either way, it is important to know that if the application framework is currently not linked against the library that contains the widget, the linker will fail. After you decided to use a certain widget, look for the library to link to; then open "Project"->"Options" from the KDevelop menubar. Switch to the "Linker Options" page and look for the checkmarks indicating the libraries that are currently used. If the library of your view widget is already checked, you can leave the project options untouched and start doing the necessary changes due to your choice. If not, and the linker options offer to add the library by a check box, check it and press "OK" to leave the project options dialog again. In any other case, add the library in the edit line below with the `-l` option. For libraries that your application has to search for before preparing the Makefiles by the configure script on the end-user machine, add the according search macro to the `configure.in` file located at the root directory of your project and add the macro to the edit line. Mind that you have to run "Build"->"Autoconf and automake" and "Build"->"Configure" before the Makefiles contain the correct expansion for the library macro.

Also, if the include files for the library to add are not in the current include path (which can be seen by the `-I` options in the output window on "Make"), you have to add the path to the Project Options dialog -"Compiler Options" page with the `-I` option or the according automake macro at the edit line for "Additional Options".

### 4.2.1 Qt Views

Looking at the first page of the Qt online documentation, you will find a link to "Widget Screenshots" where you can have a look at how the widgets Qt contains look like. These are ready to use and can be combined together to form complex widgets to create application views or dialogs. In the following, we'll discuss some of these which are very usable for creating application views, but keep in mind that the KDE libraries sometimes contain other widgets for the same purpose; those will be reviewed in the next section.

Here are a set of hints for what purpose you could use which Qt component:

1. If your view area isn't big enough to display all your data, the user must be enabled to scroll over the document with bars on the left and bottom of the view. For this, Qt provides the class `QScrollView`, which offers a scrollable child area. As explained, you could inherit your own widget from `QScrollView` or use an instance to manage your document's view widget.
2. to create a `ScrollView` yourself, inherit the `View` widget from `QWidget` and add vertical and horizontal `QScrollBars`. (This is done by KDE's `KHTMLView` widget.)
3. For text processing, use `QTextEdit`. This class provides a complete text editor widget that is already capable to cut, copy and paste text and is managed by a scrollview.

4. Use `QTable` to display data that is arranged in a table. As `QTable` is managed by scrollbars as well, it offers a good solution for table calculation applications.
5. To display two different widgets or two widget instances at the same time, use `QSplitter`. This allows to tile views by horizontal or vertical dividers. KMail is a good example what this would look like- the main view is separated by a splitter vertically, the right window then is divided again horizontally.
6. `QListView` displays information in a list and tree. This is useful for creating file trees or any other hierarchical information you want to interact with.

You see that Qt alone offers a whole set of widgets which are ready to use so you don't have to invent new solutions if these match your needs. The sideeffect when using standard widgets is that users already know how to handle them and only have to concentrate on the displayed data.

#### 4.2.2 KDE Views

The KDE libraries were invented to make designing applications for the K Desktop Environment easier and capable of more functionality than what Qt alone is offering. The `kdeui` library offers:

1. `KListView`: a more powerful version of `QListView`
2. `KIconView`: a graphical viewer of icon files

The `khtml` library, on the other hand, offers a complete HTML-interpreting widget that is ready to use. It is scrollable already, so you don't even have to take care for that. A possible use could be to integrate it as a preview widget for an HTML editor; used by applications such as Konqueror to display HTML files.

### 4.3 Creating your own Views

Not yet written

## Chapter 5

# Configuring Menubars and Toolbars

### 5.1 Introduction

Menubars and toolbars are one of the most important parts of an application to provide methods to work with a document structure. As a general rule, you should make all functions available by the menubar. Those methods that should not be available at a current stage of the application process should be disabled.

Further, an application can only contain one menubar, but several toolbars. Toolbars on the other hand should contain only the most frequently used commands by pixmap icons or provide quick access methods like combos to select values.

### 5.2 How does it work?

Our application inherits the `KMainWindow` class, which automatically handles creating a menu bar and tool bars for us. In the `KScribble::setupActions()` method there is a call to `KMainWindow::createGUI()`. This method loads a resource file, in this case `kscribbleui.rc`, to initialize menus at startup. Note that `kscribbleui.rc` is listed as one of the project files in the Automake Manager. Opening that file up reveals this:

```
1 <!DOCTYPE kpartgui SYSTEM "kpartgui.dtd">
2 <kpartgui name="kscribble" version="1">
3 <MenuBar>
4   <Menu name="custom"><text>C&ustom</text>
5     <Action name="custom_action" />
6 </Menu>
```

```
7 </MenuBar>
8 </kpartgui>
```

#### Explanation...

Another way to modify the contents of the menu and tool bars is to directly manipulate them through the methods provided by their class. For example, the `menuBar()` method returns the `KMenuBar` widget that the menubar for our program. Looking at the documentation for `KMenuBar` and its inheritor class `QMenuBar`, you will find a large number of `insertItem()` methods which allow you to add items to the menu bar.

`KMainWindow`'s methods `statusBar()` and `toolBar()` will also provide you with applicable widgets.

### 5.3 Keyboard Accelerator Configuration

A very professional thing you should always add to your application are keyboard accelerators. Those are mainly used by experienced users that want to work fast with their applications and are willing to learn shortcuts. For this, the KDE libraries provide the class `KAction`, which provides the keyboard accelerator keys and access to global configured standard keyboard accelerators.

By default, frame applications generated by KDevelop only use standard keyboard accelerators such as F1 for accessing online-help, Ctrl+N for New File etc.

If your application contains a lot of accelerators, you should make them configurable by an Options-menu; either it could be combined with other application configuration in a `QWidget` or stand alone. The KDE library already provides a `KKeyChooser` for use in tab dialogs, whereas `KKeyDialog` provides a ready-to use key-configuration dialog.

## Chapter 6

# Help Functions

### 6.1 Introduction

A very important part of the development process is to provide help functionality to the user wherever possible. Most developers tend to delay this, but you should remember that a normal user isn't necessarily a Unix expert. He may come from the the dark side of computer software usage offering all sweets that a user may need to work himself into using an application even without ever touching the manuals. Therefore, the KDE and Qt library provide all means usually considered making an application professional in the eyes of the normal user by help functions that are ready to use. Within the application, those are:

- Tool-Tips
- Statusbar help
- What's this...? buttons

Additionally, the application should provide means to access a HTML-based online manual directly using the standard help key F1. This context based help system is provided automatically through the `KMainWindow` class, though as the author you must provide the content.

As KDevelop also offers all types of help as well as the KDE framework generated by the application wizard already contains support for this, this chapter will help you find out where and how to add your help functionality.

During the development of your application you should try to be consistent whatever you're doing; therefore you should do the necessary steps directly while extending the code. This will prevent you from diving into the code again and figuring out what your application does or what you intended by certain parts of the code.

## 6.2 Tool-Tips

A very easy means of providing help are tool-tips. Those are small help messages popping up while the user moves the mouse over a widget that provides a tool-tip and disappears when the mouse moves away. The most popular usage of tool-tips is made in toolbars where your tool-tips should be kept as small as possible because toolbars can be configured to display their contents in various ways: either displaying the button, button with text on the right, button with text below, text only. This possibility should be made configurable by the user, but isn't a must-be. The text is shown as a tool-tip anyway and a toolbar usually consists of buttons and other widgets like linedits and combo boxes. For a complete reference, see the `KToolBar` class reference located in the `kdeui` library.

As an example, we have a look at the "New File" button in a generic application:

There, the part `i18n("New File")` provides a tool-tip message. It is enclosed by the `i18n()` macro provided by `kapp.h` to translate the tool-tip towards the currently selected language.

Tool-tips can also be added to any custom widget by using the `QToolTip` provided by Qt. An example of that would be:

## 6.3 Extending the Statusbar

As the applications that inherit `KMainWindow` contain a statusbar as well, it also offers a set of statusbar messages already for all menu and toolbar items. A statusbar help message is a short message that extends the meaning of a tool-tip or can be seen as a replacement for a tool-tip over menubar items and is (as the name suggests) displayed in the statusbar when the user enters a menu and highlights the menu entry.

## 6.4 The What's This...? Button

The What's This...? button provides help windows with the intention that the user wants to get help about a certain widget within the working view or a toolbar item. It is placed in the toolbar and gets activated once the user hits the button. The cursor changes to an arrow cursor with a question mark like the button itself looks like. The the user can press on a visible widget item and gets a help window. As an exercise, you could try this behavior with the What's this...? button within KDevelop.

To add the What's This...? help to one of your widgets, use the static method `QWhatsThis::add(QWidget *widget, const QString &text)`

## Chapter 7

# Documentation

### 7.1 Introduction

Due to the fact that projects often lack a complete set of user documentation, all KDevelop projects contain a pre-build handbook that can be easily adapted; therefore fulfilling another goal of KDE: providing enough online-help to support users that are not familiar with an application. This chapter therefore introduces you on how to extend the provided documentation template and what you have to do to make it available to the user.

### 7.2 User Documentation

The documentation for your project lies in `projectdir/doc/en`, or perhaps another directory if English isn't your native language. Therein lies a file, `index.docbook`, in which the documentation is stored. The format for editing this file is explained on [KDE's documentation website](#).

### 7.3 Programmer Documentation

Another important part of the documentation is including a descriptive help for your class interfaces. This will allow you and other programmers to use your classes by reading the HTML class documentation that can be created with KDoc. KDevelop supports the use of KDoc completely by creating the KDE-library documentation, also your application frameworks are already documented. To work yourself into the provided code, it would be a good start to read the included documentation online. The following describes what to do to get the API documentation, where KDevelop helps you add it and what kind of special tags KDoc provides.

## Chapter 8

# Internationalization

### 8.1 Introduction

i18n is an internationalization system that is used to offer internationalized versions of an application or project. The difficulty with writing applications is that they only support the language they originally are composed with; visually this can be seen on labels, menu entries and the like. The goal of the internationalization is to provide applications and library functions in the language of the user; therefore enabling users that are not native speakers the original language to make use of the provided functionality and feel more comfortable.

## Chapter 9

# Credits

(... to be written ...)

## Appendix A

# Bibliography

### A.0.0.0.1 Bibliography

- [1] Richard M. Stallman and Roland McGrath, *GNU Make Manual*
- [2] David MacKenzie and Tom Tromey, *GNU Automake*
- [3] David MacKenzie and Ben Elliston, *GNU Autoconf*
- [4] Richard M. Stallman, *Using the GNU Compiler Collection*
- [5] Gordon Matzigkeit, Alexandre Oliva, Thomas Tanner, and Gary V. Vaughan, *GNU Libtool*
- [6] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor, *GNU Autoconf, Automake, and Libtool*, 1st edition, October 2000, New Riders Publishing, ISBN 1578701902.
- [7] W. Richard Stevens, *Advanced Programming in the UNIX(R) Environment*, 1st edition, June 1992, Addison-Wesley Pub Co, ISBN 0201563177.
- [8] Bruce Eckel, *Thinking in C++, Volume 1: Introduction to Standard C++*, 2nd Edition, April 15, 2000, Prentice Hall, ISBN 0139798099.
- [9] Karl Fogel and Moshe Bar, *Open Source Development with CVS*, 2nd Edition, October 12, 2001, The Coriolis Group, ISBN 158880173X.
- [10] Rasmus Lerdorf and Kevin Tatroe, *Programming PHP*, 1st edition, March 2002, O'Reilly & Associates, ISBN 1565926102.
- [11] Mark Lutz, *Programming Python*, 2nd Edition, March 2001, O'Reilly & Associates, ISBN 0596000855.

## The KDevelop Programming Handbook

- [12] Boudewijn Rempt, *Gui Programming With Python : Using the Qt Toolkit*, Bk&Cd-r edition, January 2002, Opendocs Llc, ISBN 0970033044.
- [13] Larry Wall, Tom Christiansen, and Jon Orwant, *Programming Perl*, The Camel book, 3rd Edition, July 2000, O'Reilly & Associates, ISBN 0596000278.
- [14] Randal L. Schwartz and Tom Phoenix, *Learning Perl*, The Lama book, 3rd Edition, July 15, 2001, O'Reilly & Associates, ISBN 0596001320.

This documentation is licensed under the terms of the [GNU Free Documentation License](#).