The KDE Library Reference Guide

The Reference Guide to C++ Application Design for the K Desktop Environment (KDE)

Ralf Nolden <*Ralf.Nolden@post.rwth-aachen.de*>

The KDevelop Team

Version 0.2 , Mon July 7, 1999

Contents

Chapter 1

Introduction

1.1 What KDE provides

As the K Desktop Environment offers an easy way for application designers to offer their products with an intuitive way of user interaction, it provides all means to solve common tasks by a set of library classes that extend the facilities of the Qt toolkit. This also allows a unique look to applications as well as interaction with other programs and the window manager. This handbook therefore intends to provide an introduction into the usage of the KDE libraries that ship with the KDE to enable developers to find easy solutions for common programming issues and explains why certain techniques and classes should be used when creating applications that are targeting the K Desktop Environment.

Basically, KDE offers a set of standards that allow a unique look and usage of applications that should be watched when designing programs. A lot of tasks are done automatically, such as:

- Session Management
- Standard keyboard accelerator configuration
- Font, Color and Style changing
- Theme support (KDE 1.1.2 and higher)
- Internationalization

Therefore, theses issues only have to be mentioned in their functionality for complete information. Application developers only have to care about what their program is intended to do and where KDE can help. There, KDE offers user interfaces that extend the Qt toolkit where necessary. If both libraries offer similar solutions, KDE developers should (in most cases) use the methods provided by the KDE libraries.

Here, KDE offers a set of widgets that can be used for creating application specific dialogs and views. Examples are

- KSeparator, offering a common separator line
- KColorButton, offering a push button displaying a color

Normally, applications ask the user to select various values. Here, the libraries provide easy means to get these values by complex widgets that are ready to use and are already known to the user by the KDE desktop, such as:

- File dialogs,
- Color dialogs,
- Font dialogs,
- Keyboard configuration dialogs

These should be used wherever a user setting is required as it simplifies the programmer's work, extends the application's facilities dramatically and provide a common look.

The Qt library is addressed in this handbook as far as it offers solutions not provided by KDE, but as information about event handling and the signal/slot mechanism is hard to find for developers, this turned out to be a special chapter in this handbook.

1.2 Notes about KDE 2 / Qt 2.0

As you may have guessed, this handbook **explicitely** adresses **KDE 1.x** development using the **Qt 1.4x** libraries. You may ask "Hey, Qt's 2.0 version is released already, why not talk only about that ?" - but there are several issues that result in a need for a KDE 1.x reference.

The issues that lead to this are first of all that developers should consider the KDE 1.x series the stable desktop and development environment. As long as KDE 2 (which introduces Qt 2.0 as well) is under development, programmers will have a hard time to follow the changes, which means a high time-consuming search for information and, if you experience bugs, error searching. There are projects covered by the KDE core team that ensure the KDE 2 libraries will work - the KDE itself as well as the supplying applications and the KOffice suite. Nevertheless, developers who are starting or developing projects for KDE now have the choice and those that want to rely on the stable environment will also want to port their applications to KDE 2 to make it usable on one hand and to implement improvements that are introduced by new library functions. Therefore information about where KDE and Qt differ from the 1.x series in future releases have been included as footnotes as far as changes are known to the author at the time of this writing and doesn't claim to be in any way complete. Explanations of classes that are still in the KDE 1.x API but are already removed in the current KDE 2 API are left out to avoid any trouble when porting to KDE 2.

In further versions of this handbook KDE 2 development will be addressed completely and the according chapters revisited for the changes this implies.

1.3 About this Handbook

This handbook has been written in order to give developers a guideline to the usage of the KDE 1.x libraries in general in conjunction with the Qt 1.4x library on X11 desktop systems. It cannot replace any programming knowledge that is needed for C++ programming and covers the Qt classes where necessary. You should in any case look at the page "Structure Overview" of the Qt online reference which contains a general guidance to the Qt library by class usage in general. This handbook tries to follow this structure to complete your knowledge about where KDE classes are appropriate and explains the technique of KDE application development by describing class usage on topics.

Chapter 2

Class Categories

The KDE/Qt C++ class libraries offer easy solutions to extend applications dramatically with a minimum amount of coding on the side of the application programmer. This chapter therefore sorts the classes provided towards their usage by certain categories:

- Baseclass
- Application architecture classes
  - Application objects of KDE
  - Application configuration
  - Main Windows
- User Interface Objects
  - Views
  - Dialogs
  - Control Elements
  - Menus
- General purpose classes
  - Files
  - Data objects
  - Graphics
  - Processes

2.1 Baseclass

Most of the KDE/Qt classes have `QObject` as their baseclass in their inheritance hierarchy. `QObject` can be described as a baseclass because it offers the usage of Qt's signal/slot mechanism which allows object interaction within the application and should be used as the baseclass for any self-created classes that are supposed to emit signals or can connect to signals by slots.

2.2 Application Architecture Classes

KDE applications usually consist of a set of objects that interact with each other. The programmer has to use the provided classes to create a KDE application either by creating a class instance or by inheritance. A typical application contains:

- One application object of `KApplication`
- One main window class derived from `KTMainWindow`
- A class derived from `QWidget` to create the view area

2.2.1 Application objects of KDE

The K Desktop Environment provides a set of functionality that an application can use to integrate into the KDE. The class `KApplication` therefore is the baseclass for any application that targets KDE. A KDE application only contains one object of the class `KApplication` that is created in the application's `main()` function. The `KApplication` object is responsible for providing the basic interfaces and objects towards the desktop and interprets the command-line arguments of an application. As the instance is a non-visible, but the main application object, the following rules have to be watched:

- the application is terminated by `kapp->quit()`.
- the object that is representing the graphical interface has to be set the main widget with `setTopWidget()` (for widgets not inherited by `KTMainWindow`)

The `KApplication` object provides:

- access to the KDE File System
- a session configuration object
- a configuration object
- internationalization by the locale object
- changing of the visible application objects by signals

[1]

Dependencies: -lkdecore -lqt

Includes: `#include <kapp.h>`

The `kdeui` library additionally offers two classes that inherit `KApplication` for specialized purposes:

2.2.2 KWMModuleApplication

Includes: `#include <kwmmapp.h>`

Dependencies: -lkdeui -lkdecore -lqt

The class KWMModuleApplication is the base class for KDE window-manager modules. It mainly informs a module about all currently managed windows and changes to them (via Qt signals). There are no methods to manipulate windows. These are defined in the class KWM (see kwm.h). An example for using KWMModuleApplication is kcontrol.

[1] KDE 2 accesses the according instances by static methods provided by `KGlobal`.

2.2.3 KControlApplication

Includes: `#include <kcontrol.h>`

Dependencies: -lkdeui -lkdecore -lqt

KControlApplication is the common base for setup applications. It provides a tab dialog and functionality common to most setup programs. The configuration dialogs for the KDE are examples of KControlApplications.

2.2.4 KWM

Includes: `#include <kwm.h>`

Dependencies: -lkdeui -lkdecore -lqt

The KWM class provides a set of static methods to interact with the window and session-manager. Therefore, call any member with

KWM::<method()>

depending on the purpose of the desired functionality.

2.2.5 Application Configuration

The class `KConfig` provides the usage of a configuration object which can write its entries into configuration files. Dependent of the values to read and write you have to call the methods of the class `KConfigBase`.

The `KApplication` object provides an application configuration object with a resource file by default which is stored in the user's kde-directory as well as the session management file to store information between sessions.

For internationalization, the `KApplication` object uses the class `KLocale` to translate localized entries dependent on the selected language. Instead of using the `klocale->translate()` method, KDE applications should use the `i18n()` macro that contains the string to be translated as the message extraction depends on this macro.

2.2.6 Main Windows

As the application's `KApplication` instance is non-visible, it only provides the basic means to create a KDE application. Therefore a KDE application needs to have a main window representing the application towards the user graphically. The main window usually consists of a widget which can be as simple as a pure button up to the complex `KTMainWindow` widget, offering the means to create a full-featured main window with geometry management, session management support, menu bar, toolbars and statusbar.

Generally, every main window has to be set main widget with `KApplication`'s `setTopWidget()` method. An exception is a main window that inherits `KTMainWindow`.

The main window usually takes the responsibility to terminate the application by providing a user interface that is connected to `KApplication::quit()`, easily used by `kapp->quit()`.

Most KDE applications will use `KTMainWindow` to represent the application graphically.

2.3 User Interface Objects

This section covers the user interface object the KDE libraries provide. By category, these can be divided by their purpose. A user interface can be:

- a view area widget, representing the data an application is intended to produce and allowing the methods to manipulate the contents.
- dialogs, used to retrieve user input, e.g. a file dialog
- control elements to compose application specific widgets
- menus, providing user interfaces to invoke application commands.

2.3.1 Views

View are generally the content area of an application. Therefore it can be the main widget or a part of a main widget that additionally offers a set of functionality such as `KTMainWindow`

The KDE libraries offer a set of ready to use views which can be inherited to advance the desired functionality:

- KTabListBox: offers a multi-column list box where the user can change the rows with drag'n drop.
- KEdit: the base classes for the KEdit application offered with KDE. This could be used instead of QMultiLineEdit.
- KNewPanner: manages two child widgets like QSplitter.[2]
- KHTMLView: a HTML-interpreting widget (khtmlw)

For use with `KTMainWindow`, create your view instance and call `setView(QWidget*)` to enable the management by the `KTMainWindow` instance.

Individual views are usually created by inheritance of `QWidget` or any provided widget that comes closest to the desired functionality the view should offer. For widgets that want to offer scrolling facilities, you could inherit from `QScrollView` or create a `QScrollView` instance and set the view widget as the managed area with `addChild()`.

2.3.2 Dialogs

Dialogs are a main part of the user interaction wherever the application requires parameters that have to be set by the user. Fortunately, the KDE library already offers a set of dialogs that are ready to use for standard parameters such as fonts and colors. In any case where these types of information is requested by the user, the application should make use of these standard dialogs.

In cases where the given dialogs don't fit the requirements, you have to inherit from `QWidget` or `QDialog` and create your own dialog either directly coded with geometry management or by creating it visually with KDevelop's dialogeditor.

The KDE libraries offer the following dialogs:

- KColorDialog: selects a color value

[2] Removed in KDE 2. Use QSplitter instead.

- KWizard: base dialog class to create wizards
- KEdGotoLine: Go-to-Line dialog for editors
- KEdReplace: Search and replace dialog for editors
- KEdSrch: search dialog for editors
- KFontDialog: font selection dialog
- KIconLoaderDialog: Icon selection dialog
- KKeyDialog: keyboard accelerator configuration dialog
- KMsgBox: message box dialog with up to four configurable buttons
- KFileDialog: (kfile) file dialog to open and save files
- KSpellDlg: (kspell) spell-checking dialog for use with Ispell
- DatePickerDialog: (kab) date selection dialog

Additionally, the Qt library offers:

- QFileDialog
- QMessageBox
- QPrintDialog
- QProgressDialog

[3]

2.4 Control Elements

Control elements are used within visible areas of the application and can be combined together to create a dialog or view. Beyond the control elements that the Qt library provides, KDE offers:

- KButton: active raise-lower button
- KButtonBox: manages buttons
- KColorButton: button displaying a color setting, calls `KColorDialog`
- KIconLoaderButton: button displaying a selected Icon, calls `KIconLoaderDialog`
- KDatePicker: Date selection widget
- KIntegerLine: line edit that only accepts integer values
- KLedLamp: LED lamp
- KLined: line edit
- KNewPanner: panner devider managing two widgets[4]

[3] Qt 2.0 introduces a QColorDialog for selecting colors and a QFontDialog for font-selection as well
[4] Use `QSplitter` instead, this is already removed in KDE 2.

- KPopupMenu: popup menu with title
- KRestrictedLine: line edit that only accepts certain input
- KSelector: value selector
- KSeparator: standard separator
- KSlider: slider widget
- KTabCtl: tabulator widget

2.5 General Purpose Classes

2.5.1 Files and Directories

Qt already offers a set of classes to work with files and directories. Those classes are:

- `QDir`
- `QFileInfo`
- `QFile`
- `QFileDialog`

A comparable and extended technology has been introduced by the KDE libraries and have a similar usage like the corresponding classes of Qt. Those are:

- `KDir`
- `KFileInfo`
- `KFileDialog`
- `KFilePreviewDialog`

For loading and saving files, use the class `QFile` which operates with streams.

2.5.2 Data Objects

The Qt library supports data objects by classes that offer handling of lists, arrays, streams, strings and the like. See the Qt documentation for mor information.

2.5.3 Graphics

Qt supports a set of graphics formats that can be used for drawings or image programs. The graphics device for painting is `QPainter`.

2.5.4 Processes

As applications can have different types of application communication with other programs available on the system such as standard Unix actions, developers can make use of the class `KProcess` to call another application. As the application is running independently from the one that invoked it, you can only receive the current status of the application invoked by `isRunning()`. Also, the invocation can be done with various initialization values. Finally, the process can inform the application whether it has been terminated or ended. See `KProcess` for details.

Chapter 3

Classes of KDE Applications

The KDE libraries, in conjunction with the Qt library, are providing developers a complete framework for application design. As KDE is targeted towards Unix Operating Systems running the graphical X11 System, you would think you would have to understand Unix and X11- but as the libraries are already encapsulating the complex graphics subsystem, you don't have to know about that in most cases. If you're already familiar with Qt, KDE won't make too much difference in regards of using classes and widgets.

Beginners, on the other hand, have a lot of problems in the sections of

- Application design with GUI components
- Application parts
- Widget construction

The following will help you to understand, where you will generally find classes used in KDE applications, including the according replacements of Qt classes for those developers that are familiar with Qt, but didn't make use of KDE libraries. These will be the application objects, their behavior and creation.You will learn about:

- the most needed objects of a KDE application
- the difference between your code and the application framework
- KDE application classes that encapsulate the window manager communication
- the class `QWidget` that all GUI components inherit
- graphical objects
- the system clipboard

More information about KDE applications can be found in the following sections:

The KDevelop Programming Handbook

KDE Application Tutorials Handbook

For information about Qt application design, see the Qt online reference documentation.

3.1 The Application Instance

Generally, a KDE application has to be started with creating an instance (and only one!) of the class `KApplication`, which is provided by the `kdecore` library. The class `KApplication` itself is derived from the according Qt class `QApplication`.

What happens exactly is that `QApplication` manages the application event queue, which means it processes events from the underlying window system to its child objects, such as mouse movements or keyboard events. This is also the reason, why only one instance of `KApplication` can be declared and why this has to be the first thing the `main()` function executes.

The `KApplication` class extends the Qt class in terms of providing additional functionality for a unique-looking application that integrates into other desktop components and can therefore be influenced in its behavior by control applications (in KDE generally collected in the KDE Control Center) such as keyboard accelerator configuration and GUI style.

Corresponding to the Qt class, the `KApplication` class provides a static function to access the application object within the application, `KApplication::getKApplication()`. This solves the problem to pass the pointer to the application object. Further, the include file `kapp.h` provides a macro that can be used instead of the static function, `kapp`. Whenever you need to access the application object, use this macro.[1]

The `KApplication` object itself provides a set of instances that are commonly used in KDE applications and lets the programmer access them by functions. You will make extensive use of them as they avoid creating own instances. The following objects are provided by their purpose:

KConfig

a configuration object that is used to read and store configuration settings in a resource file. Use the according methods of `KConfigBase` to read and write values. The configuration object is retrieved by `kapp->getConfig()`[2], the session managment configuration by `kapp->getSessionConfig()`

KIconLoader

an object that loads icons into `QPixmaps` by using the KDE File System. This takes away the need to search for any pixmap on the file system completely as only the filename has to be entered. Use the macro `Icon("icon.xpm")` to load an icon easily. The iconloader instance can be addressed by `kapp->getIconLoader()`[3]

KLocale

an object that returns the actual localization settings. This makes applications appear in the language on the desktop chosen by the user globally. Use the macro `klocale` to access the application locale object. The klocale instance can be addressed by using `kapp->getLocale()` (as the macro klocale does already)[4]

KCharsets

the charset object currently set globally by the user. This translates key events to the correctly set charset. Retrieve the charsets instance with `kapp->getCharsets()`.[5]

[1] In KDE 2, the application object can be retrieved with `KGlobal::kApp()`.
[2] For KDE 2 use `KGlobal::config()` to retrieve the config object
[3] KDE 2 uses `KGlobal::iconLoader()` to access the icon loader instance
[4] Again, KDE 2 changes this with a static method of `KGlobal::locale()`.
[5] Replaced by `KGlobal::charsets()` in KDE2.

Further, the KApplication class provides you with the needed methods to access the KDE File System Standard. This will prevent you from problems that will always occur when hard-coding any directories into the code (see *The KDevelop Programming Handbook* for information about the KDE-FSS). The class also provides the needed access for the application icon and mini icon, current KDE fonts and session management support.

To use the clipboard, the according `QApplication` class already provides a clipboard object, `QApplication::clipboard()`, which can be used to exchange text and image data between applications (see `QClipboard`).

Internationalization is another keyword for KDE applications. This is supported by the `KLocale` object already mentioned above and is always used with the macro `i18n()` of `kapp.h`.

It just shortens the call of the KApplication pointer to the KLocale pointer and is read by gettext to extract all macro-embraced strings into an application message file that then can be translated.

At least, the `KApplication` already constructs a help menu that can be inserted into a menu bar the application may contain with a predefined (can be turned on/off) KDE hint dialog and your application's about dialog.

You should notice that `KApplication` has to be lead with:

- creating the `KApplication` object at the beginning of the `main()` function
- executing the application with `exec()` at the end of the `main()` function
- terminating the application with calling the `quit()` slot function.

Using `KApplication`, you are able to catch the following event signals:

- kdisplayPaletteChanged()
- kdisplayStyleChanged()
- kdisplayFontChanged()
- appearanceChanged()
- saveYourself()
- shutDown()

As the widgets of your application will receive these signals, they will normally update themselves. The important signals for you will be in most cases `saveYourself()` and `shutDown()` (whereby `KTMainWindow` already catches `saveYourself()` to call `KTMainWindow::saveData()`).

You will be able to communicate with the KWM window manager as well by the according static methods of `kwm.h`.

3.2 Commandline Argument Processing

Finally, an application usually wants to process commandline options. Those are entered by the user if he started the application from a console for the reason to either start it non-graphically for any processing reasons the application may be capable to execute or to start the application with a file. This is also important to interact with filemanagers that can include your application to the list of those that open a mime-type automatically when called with a filename. The `main()` function therefore is constructed with the declaration

```
int main(int argc(), const char* argv[])
```

Thereby, `argc()` is the number of commandline options and the array `argv[]` actually contains the commandline option's texts. As the `QApplication` constructor is executed before the `KApplication`, it is sufficient to know what is processed by possible options first. The already read options are automatically removed from the array and cannot be read after the `KApplication` instance is declared:

(taken from the Qt 1.42 online reference of `QApplication`)

- **-nograb**, tells Qt to never grab the mouse or the keyboard.
- **-sync** (only under X11), switches to synchronous mode for debugging.
- **-display** display, sets the X display (default is $DISPLAY).
- **-geometry** geometry, sets the client geometry of the main widget.
- **-fn** or -font font, defines the application font.
- **-bg** or -background color, sets the default background color and an application palette (light and dark shades are calculated).
- **-fg** or -foreground color, sets the default foreground color.
- **-name** name, sets the application name.
- **-title** title, sets the application title (caption).
- **-style**= style, sets the application GUI style. Possible values are motif and windows
- **-visual** TrueColor, forces the application to use a TrueColor visual on an 8-bit display.
- **-ncols** count, limits the number of colors allocated in the color cube on a 8-bit display, if the application is using the QApplication::ManyColor color specification. If count is 216 then a 6x6x6 color cube is used (ie. 6 levels of red, 6 of green, and 6 of blue); for 108, a 3x3x3, and for other values, a cube approximately proportional to a 2x3x1 cube is used.
- **-cmap** causes the application to install a private color map on an 8-bit display.

Then, the `KApplication` processes commands whose values can be set within a `.kdelnk` file. Usually, those link files contain internationalized versions for the application description and the application name as well as some other values such as the icon and miniicon name. The commandline options to use these values are:

```
foo %i %m -caption \"%c\"
```

This will start the application with the value -icon something.xpm for %i and -miniicon for %m. The application caption can be set with the -caption value %c. The class `KApplication` also provides the according methods who return the values for these commandline arguments.

Now, when it comes to your own processing of commandlines, you can either access them directly after the application object is declared to exclude any of the above values in the `main()` function. Within the application itself (e.g. the mainwindow class), `QApplication` provides the methods `argc()` and `argv()` to process any other options given at the commandline. These can be accessed by the argument number, whereby the argument `kapp->argv()[0]` is the application name; any other following arguments can be processed with `kapp->argv()[number]`.[6]

[6] KDE 2 has an additional class, `KStartParams`, to parse additional command-line parameters.

```
The KApplication class uses different methods to change the application's style,
fonts and colors by X11 Atoms which call all open KDE applications to change their
values recursively throughout all widgets.  This is done automatically when the
user changes values through using the KDE control center which causes these X11
events.  The method x11eventFilter() emits the according signals to change all
values.  As far as I know of, the only value not changed by KDE 1.x is the double
click interval, which is set to 400 ms by default in QApplication.  Changes on
this can be made using QApplication::setDoubleClickInterval().
```

3.3 Other Application Classes

By "Other Application Classes" we would describe any replacements of the `KApplication` class. The KDE library `kdeui` offers two more classes that inherit `KApplication` for more specific use in KDE applications. Those are the class `KControlApplication` and `KWMModuleApplication`.

3.3.1 KControlApplication

The `KControlApplication` is a class for specific applications that are intended to serve as setup modules. By providing a tab dialog, control modules can easily be created. The differences to `KApplication` are:

- uses `-init` as commandline option to call the method `init()`. This one has to be overwritten to initialize the dialog settings.
- the caption of the dialog has to be set independently of the `KApplication` way with `setTitle()`
- provides a complete widget frameset already where your control widgets have to be inserted.

To overwrite the virtual methods like `init()`, you have to derive an application specific `KControlApplication` class from `KControlApplication`. The class is generally used for control applications such as used in the KDE as separate programs.

3.3.2 KWMModuleApplication

KWMModuleApplication is another class that inherits `KApplication` for a certain purpose: the class provides methods that allow interaction with the window manager. An example for a KWMModuleApplication is the `kpager`, who uses the signals that the window manager sends out to manage the windows with the static methods of KWM.

To create a KWMModuleApplication, you first have to create your application instance and then call the method `connectToKWM()`

3.3.3 Docking of Windows

Another issue to general KDE application design is the use of `kpanel` to display your running application symbolized. KDE users are probably familiar with the display settings symbol left of the clock in `kpanel`.

The way it runs is rather simple: You have to provide a widget that is the docked widget in the panel; therefore has to be a top-level window by calling the constructor with 0 as parent. Include the kwm.h header file and call

```
KWM::setDockWindow(mywidget->winID());
```

Mind that for undocking hiding the widget is not enough, you have to call `destroy()` (see `QWidget` for `destroy()` and `create()`).

3.4 The Main Window

As stated above, the first object to create for a KDE application is one instance of `KApplication`. Indeed, it doesn't provide any widgets (visible user interfaces) except the about dialog similar to the `QApplication` class as a popup menu, but that isn't seen anywhere. Therefore, any application needs a top-widget to make itself visible.

When it comes to the visible parts, the programmer generally is free to choose which widget he wants to derive from or use directly as his main window. It can be a simple `QLabel` as well as the improved `KTMainWindow` that supplies all needed objects for a usual desktop application.

3.4.1 General Rules

Generally it can be said that you probably would like to use ready components that are specialized. KDE supports this with the class `KTMainWindow` as a pendant to the Qt class `QMainWindow`. Before describing the general guideline, we have a look at the exceptional: using any other widget.

When using `QApplication` as the base application class, you would first create the application instance, then create the main widget. Now, it is safe to register the widget as the top widget with the method `setMainWidget()`, because the user can use the close button of the window to exit the window. He expects the application to be terminated, if the last window is closed, but to do so, you have to call the QApplication slot `quit()`. You could do this in a derived class by a re-implementation of `QWidget::closeEvent()`, but with the method `setMainWidget()` this is already done. Now, in any case of using widgets with `KApplication`, things are almost the same with the difference that the according method of KDE is `setTopWidget()`. The only exception is when using the class `KTMainWindow` who automatically does this in its constructor (if there is no other topwidget).

3.4.2 Using KTMainWindow

As usual desktop applications provide a complete user interface following a common design rule for GUI programs, KDE provides a class that already is capable of all needed functions that an application may make use of, `KTMainWindow`, which is located in the `kdeui` library. It is strongly connected to `KApplication` and therefore very easy to use. The class provides:

- session management support
- a main widget
- a menu bar
- as many toolbars as your application may require
- a statusbar

The elements of the Widget themselves are already managed by `KTMainWindow`'s geometry implementation, therefore you usually don't have to take care of that to re-implement an application specific instance for your program.

Chapter 4

Event Handling

This chapter has a major meaning for those that want to have a better insight into the internals of KDE and Qt programming as well as explaining details of event handling in general. Additionally, the Signals and Slots mechanism is explained in detail as resources about the meaning and usage of signals and slots are somehow hard to find in other documentation as known from users; also the translated version allows non-english beginners a better understanding than originals.

Event handling therefore covers the communication of an application here - the communication between the objects and between objects and the user. Especially beginners have a hard time to work themselves into the event processing and many misconstructions only appear because of a lack of knowledge. The signals and slots are originally not a part of event handling on themselves but are used during the event processing often and are a major reason why simple widget elements such as buttons don't have to be inherited by the programmer.

4.1 Signals and Slots

This section covers the Qt mechanism of advanced object communication. In this context, objects are the instances of classes that are created during runtime by the application. The instances normally don't know about each other, but they have to communicate to allow method calls of other object's methods. The usual way for XWindow has been using virtual methods, but this lead to a very complex and not very safe way to solve this. The Qt library offers a far more better solution to this by a mechanism that is called signals and slots.

Before going into the technical details, I'll explain the mechanism by a comparison to everyday-life. Imagine you have a bunch of people somewhere and someone looks into the sky. He sees a balloon and points with his finger to the balloon and says: "Hey, there's a balloon !". Now, what do other people do that are standing around him ? One who is interested in balloons will look up as well and have a look at it, maybe take a picture with a camera. Others won't because they were never interested in balloons and don't want to know about it; they just ignore what was said.

With Signals and Slots, things are just the same, except that instead of people objects interact. Objects are instances of classes that send out a signal in a certain situation. Other objects might be interested in that kind of signal and react to it. While humans have the choice to react interactively, class objects can't because they have no ears. But they can provide a kind of ears that listen to signals that were sent out. Then, those special kind of ears have to be connected to the signal an object emits to provide the medium to transmit the message to the receiver. Any kind of ears that could react won't if they are not connected with the signal and therefore ignore the message

transmission.

I hope this made somehow clear how the signal/slot mechanism works generally. The chart shows this a bit more appropriate:

```
Person_1                          Person_2                    Person_3

signal balloon_seen();            slot i_see_it();            slot not_interested();

watch_out(){                      i_see_it(){
  if(balloon){                       look_where();
    emit balloon_seen();          }
  }
}

connect (Person_1, SIGNAL( balloon_seen()), Person_2, SLOT(i_see_it()));
```

This would be explaining more about the functionality. The class that builds Person_1 provides a signal balloon_seen(). Also it has a method watch_out() that symbolizes that he watches for something. If this method detects that a balloon is there, it emits the signal and transmits the message to the outside of its responsibility. After the emit, Person_1 is not responsible for any actions that follow as a reaction to this signal; it just does the message invocation.

Now, the table contains the other persons Person_2 and Person_3. Both provide methods that are slots; Person_2 has one slot that is called i_see_it() and Person_3 a slot not_interested(). Those slots are just like any method with the difference that they can be connected to a signal and therefore build a receiver for the connection. They will execute the slot implementation when the object receives the signal message. In this case, we have a typical connection method at the last line. The connect() takes Person_1 as a signaler object. It connects the signal balloon_seen() that the object may send out when he sees a balloon with Person_2. Person_2 is then the receiver object. Now, the receiver has to do something with the signal; we have to tell him, which method to execute whenever Person_1 sees a balloon. The implementation of the slot i_see_it() just calls another method to make this example short. look_where can symbolize a method to localize the coordinates, the color, the size of the balloon or how many people the balloon carries.

Person_3 then is anther object. The class that builds it provides a slot as well, the method not_interested(). The implementation doesn't matter for our example here, because we don't want Person_3 to react to Person_1's detection of a balloon. We could, if we add another connect though, just with Person_3 instead of Person_2 and the according slot Person_3's class provides.

4.1.1 Requirements

This way of object communication is by default not provided by C++ - its a part of the Qt library that depends on the usage of certain classes either by inheritance of your own classes or by using the class that provides this functionality. Additionally, the signals and slots have to be declared as such in the class-declaration for two good reasons:

1. you know which signals and slots a class provides and the parameters
2. the moc (Meta Object Compiler) of Qt can create the implementation for signals and slots automatically and include it to the compile process

Now, we're going into the details of the Qt library. To make use of the signal/slot mechanism, you have to:

- inherit from `QObject` or any subclass of `QObject`
- add the macro Q_OBJECT at the beginning of the class-declaration (**without a semicolon !**)
- run **moc** over the header file to produce an implementation file to be compiled

Normally, KDE and Qt applications constructed with automake and autoconf already contain a way to run moc automatically. This is done by the program `automoc`, which also does everything needed to create the meta object implementation for signals and slots as well as incorporating the correct headers for the implementation and the inclusion into the build-process. So you don't have to take care of updating any moc output files after changing header file implementations nor about the integration of the moc source file output into the project. Those will be automatically generated by detection of the Q_OBJECT macro in the class declaration.

Further, you should read yourself into the page "Using The Meta Object Compiler" of you Qt Online Reference. It covers all restrictions on using signal and slot declarations within classes.

4.1.2 Emission of Signals

This part now actually describes the several ways of signal emission. It is important to know where signals are emitted and for what purpose you would do so.

We separate two ways of signal usage, one which is the usual way through sub-classing `QObject`, and the other to use the `QSignal` class from within classes that don't want to inherit from `QObject` but want to use the signal emission features.

So, when deriving from `QObject`, we already said that we have to add the `Q_OBJECT` macro into the class declaration. Then any signal that a class object will emit has to be inserted in the class declaration with the modifier `signals:`

Example:

```
class Foo::public QObject{
  Q_OBJECT

public:
  Foo();

signals:
  void mySignal();
  void myParameterSignal(int, int);
};
```

This shows the insertion into the class-declaration and also shows that you can use signals to emit values as well. This is one of the best features and is widely used throughout KDE and Qt.

Now, this shows only one half of the work. The other is: where does the signal get emitted ? For this, you have to use the keyword `emit` in connection with the signal name and the transmitted actual parameters. The place where to emit is usually within a method that is processed and wants to inform about the state of the object by the signal to outside objects. `The keyword`

`emit` is technically only an empty `#define`, therefore the C++ -compiler only sees a normal method call. The moc takes care to add the according meta-object creation and initialization, which finally implements the signal as a member function in the moc output.

As an example, we have a look at a snippet of code where a method of `KMyClass` cuts out a part of a visible area the user works with:

```
void KMyClass::cut(){

  int xpos=view->xPos();
  int ypos=view->yPos();
  view->cut(xpos, ypos);

  emit cutting(xpos, ypos);
}
```

This method could be called e.g. from a toolbar icon "Cut" or the according menu entry of the application's menu bar. We assume that we have a view area that we retrieve as a pointer `view`. The class providing the view area offers cutting a selection by an x and y integer value. The actual values can be found out with `xPos()` and `yPos()` and stored into xpos and ypos to avoid temporaries and to reuse the values for emitting the signal. Then, we call the cutting method via the view object by the actual parameters. Finally, we want to inform about what's being done by emitting a signal `cutting()`. In case anyone needs to know about what happened, we have also included the exact information about where we have done the action by transmitting the values with the signal.

Another way to produce a signal is, as mentioned, possible without sub-classing `QObject`. Qt provides this by the class `QSignal`. The usage is rather simple, though inheritance of `QObject` should always be preferred.

To use `QSignal`, write a normal C++ class. Then add the following:

1. #include <qsignal.h>
2. add a `QSignal` member attribute to the class declaration
3. add a method `void connect(QObject* receiver, const char* member);` to the class
4. create the signal in the constructor with **new**
5. destroy the signal in the destructor with **delete**
6. implement the `connect()` method by calling `connect(receiver, member)` on the signal to emit
7. add the emission at any place in your class code with `yoursignal->activate()`

4.1.3 Slot Implementation

After explaining the ways of how to produce signals by objects, those can only be of important use if an application's classes provide slots that get connected with signals. The slot themselves are normal C++ class member functions, therefore can be called any place any time you need to, only depending on the access rights. They just have an additional feature that they can be called automatically during runtime by their connected signals. The main difference is the declaration of the methods within the class:

```
class Foo::public QObject{
  Q_OBJECT

public:
  Foo();

public slots:
  void mySlot();
  void myParameterSlot(int, int);
};
```

Above, you see that the class `Foo` has two slots declared in the class-declaration. As the modifier is also preset, here to public, it follows that you can also restrict slot usage by access rights to public, protected and private. The only thing to watch out for is that all methods after `public slots:` are slots, so you have to start with `public:` again, if you want to add public methods behind the slots declarations. Whe connecting signals to slots, the sender can only connect to slots the receiver allows to call depending on the access rights e.g. a private slot cannot be called by an instance of another class than the own (which means only instances of the same class can connect signals to this slot).

Another restriction is the return type. As slots are most often called by signals, where should they deliver any return values ? Therefore, your slots will always have `void` as return type.[1]

4.1.4 Connections

The last section of this chapter deals with connecting signals and slots. As stated in the Signals section, there are two ways to produce signals, and in the Slots section we saw that slots are methods which have modifiers as well.

When it comes to connecting them, you generally will use the static method of `QObject` to send a signal to a method:

```
bool connect(const QObject* sender, const char* signal,
             const QObject* receiver, const char* member)
bool disconnect(const QObject* sender, const char* signal,
             const QObject * receiver, const char* member)
```

Both are static public members of `QObject` and can be called everywhere in the code if you want to connect/disconnect a sender and receiver by certain signals and slots. The signal in these methods have to be used with the `SIGNAL()` macro; the slot of the receiver has to be used with the macro `SLOT()`.

`Note: within classes that inherit QObject you don't have to use the static variant, so instead of using QObject::connect(), you can also use the overloaded methods that either the sender provides (such as QMenuData to connect activated() directly to the receiver's slot while inserting a menu entry) or just call connect() directly.`

Further, the signal and slot should have the same parameter list as parameters are translated from the signal to the slot method. Slot implementations that don't require using any transmitted parameter

[1] For all restrictions of implementing slots in classes, see the Qt online reference documentation, section *The Meta Object Compiler*.

only have to declare the type but do not need a formal parameter. This avoids the unused parameter warnings you usually get when declaring formal parameters which aren't processed in the method. The slot methods itself can also have less parameters than the signal emits.

Also, signals can be forwarded. This means, you can use the `connect()` method to connect two signals, meaning that the sender's signal will cause the receiver to emit the connected signal. A signal can furthermore be connected to several slots, where the slot execution depends on the connection order.

4.2 The Event Queue

As the previous chapter dealt with the object communication by Qt's signal/slot mechanism, we know how an application can arrange a certain functionality. But this leaves out the events the user produces. Generally, he communicates with an application by the keyboard and the mouse. When running an application under XWindow, the X11 protocol ensures that the right application is called to process the events - so only the application object receives the event and therefore has to provide means to handle them. This is called event handling. The application object therefore has to keep an event queue when initialized where events run into and get processed to the right application window. The application itself is running in a so-called `main event loop`, which indicates that it waits for user interaction until the user quits the application either via the `quit()` slot or by calling `exit()`. The `exit()` function also returns the value to the main() function's call of `exec()` to terminate. If the number `exit()` is called with is higher than 0, errors occurred. The `exec()` function call in `main()` also starts the event handling.

The event handling executes fetching the window system events it claims responsible to process. Usually this means that the event queue fills up with events the user releases and which are processed to the application object. By the `QEvent` class, all events are translated into Qt events which can therefore be handled much easier.

The translated event is then processed by `QApplication`'s `notify()` function. This sends all receivers that are derived from `QObject` and are part of the application the according event with `receiver->event(QEvent* event)`. The application objects therefore get notified about any event that happened and can process the event via the re-implemented `event()` method of `QObject` if needed. `QObject` also allows a self-created event filter functionality by installing an event filter on the class. The event filter is processed first if one is installed and then the event method returns control over the event if the event filter returned false. If the `event()` method doesn't find any event processing, it returns *false* and the application gets to know that the object didn't sign responsible for the event. If the event was successfully processed and the `event()` returns *true*, the event is deleted from the event queue.

4.2.1 Processing Events

Now, Qt and KDE applications will use a graphical interface to make themselves visible. A window of the application on the other hand has to be derived from `QWidget`, as this is the baseclass for any graphical object drawn in windows. Independent of how the widget is created, the application object notices all widgets that are created and keeps a list of these. Further, the windows can have several states dependent on how they are created.

The `QWidget` class is most important to understand because it re-implements the `event()` method already to transform the incoming event to some commonly occurring events, e.g. a mouse event, and creates appropriate filter event functions which are easier to re-implement for the special purpose

a widget may need. This is e.g. used for any widget that inherits the `QWidget` class, because those events can be used to send out signals that are avoiding any sub-classing of common widgets such as pushbuttons. The `pressed()` signal e.g. is emitted on the re-implementation of `QWidget`'s `mousePressEvent()`, showing that you don't have to subclass a simple pushbutton to find out the event and to get notified that the user pressed it.

Re-implementing these methods is one of the common tasks of a programmer writing his own widgets, therefore you will have to know about the virtual event functions of `QWidget` and the event queue processing very well.

Above, we mentioned that a widget can have several states. This predefines the behavior of the widget towards the user as well as towards the application object.

A widget can be:

1. a **main widget** when set as the main widget with `QApplication`'s `setMainWidget()` or `KApplication`'s `setTopWidget`.
2. a **top widget** when the parent of the widget is 0.
3. a **modal widget** usually a `QDialog` which has its own event loop
4. a **semimodal widget** like a `QDialog`, but without its own event loop
5. a **popup widget** when the widget flag is set to WType_Popup, is also a top widget

The specialized behavior of the widgets depending on their creation is then:

1. **main widget:** a main widget is the most important widget of the application, but the application doesn't need to have a main widget of course. If it has, and the main widget gets closed, the application terminates automatically by calling `quit()`. The `QApplication` method `mainWidget()` returns the pointer to the main widget.
2. **top widget:** a top widget is a widget which has 0 as its parent. All other widgets that have non-zero parents are sub-widgets of the parent. The list of top level widgets can be found with `QApplication::topLevelWidgets()`. If an application doesn't have a main widget but only top widgets, connect `quit()` to `QApplication::lastWindowClosed()` to terminate the application, otherwise the application object will still exist even if all windows are closed. The application finds the currently active (focus enabled) widget with `QApplication::focusWidget()`.
3. **modal widget:** a modal widget is a widget derived from `QDialog`. `QDialog` widgets have their own local event loop which is entered when calling `exec()` on the dialog object. The dialog is modal, if the third widget flag is set to *true*, meaning that the dialog has to be terminated before the event processing can return to other application windows. All events are sent to the dialog by the application object. The current modal widget is found by the application by `QApplication::activeModalWidget()`.
4. **semimodal widget:** is a widget that disables events to other widgets like a modal dialog but does not have its own event loop. The modal flag has to be set to *true* like for a `QDialog`, although the semimodal dialog is derived from `QWidget`.
5. **popup widget:** a popup widget is a popup that, when it appears, makes the application object send all events to it. The popup has to be finished before the event returns to any other widget, except for another popup. The current popup widget is found by the application object by `QApplication::activePopupWidget()` to post the events to.

The application object itself keeps track of all widgets that it is responsible for. The list of widgets can be retrieved by `QApplication::allWidgets()`.

Summary: The application object is responsible for retrieving the events that were invoked by the user from the underlying window system. Then it converts these events via `QEvent` and can sent the event to any widget that is currently active. The widget itself is responsible to process the event either by accepting the event after finding out that it has an event-handler (or to be precise: the event handler has to return true to the notifying of the application's event posting). The event is deleted from the queue if an event handler was found, if all possible event handlers return false, the application is not responsible for the event and the event is ignored (deleted from the queue as well).

What is left to explain about event processing is the installation of own event filters for widgets or any other object derived from `QObject` and the way `QWidget` contains a pre-defined event handling that has to be overwritten for processing events on custom widgets. Mind that as a guideline to define own event handling, you should reimplement `QObject::event()` for all classes that do not inherit `QWidget` and the more specialized event handlers described below for all `QWidget` inherited classes. Also, preserve the declarations as virtual protected to ensure reusability and consistency for your code.

4.2.2 Event Types

The events sent to the application are, as described, converted by `QEvent` to Qt events. The event type can be found out by using the `type()` method of `QEvent`, which can then be compared with the event that you want to know about. Now, the event type that `type()` delivers is an integer number; those are declared with #define in the file `qevent.h`.[2]After filtering events for the specialized event class, more information can be found out by explicit conversion to the event class to retrieve exact data about the event.

Example:

```
bool MyClass::event( QEvent* event ){

  if( event->type() == Event_MouseButtonPress){
    if( (QMouseEvent*)event->button() == RightButton ){
      // do something with the event, eg. pop up a contextmenu
      return true;
    }
    else{
      return false;
    }
  }
  else return false;
}
```

The event has been explicitely converted to `QMouseEvent*` here to find out the button type. You could also find out the position of the mouse pointer at the time of the event, see the following section about mouse events

[2]Qt 2.0 uses an `enum` for all available event-types whose entries are similar to the current defines but generally leave out the `Event_` prefix. The event type can be retrieved as described above, so you only have to change the comparison of the event type.

As there are so many event types that can occur, I have sorted the events defined in `qevent.h` logically according to the general event type and the subclasses that provide an event handling and offer the exact information about specific events. The sorting contains:

- 4.2.2 (Window Events)
- 4.2.2 (Focus Events)
- 4.2.2 (Mouse Events)
- 4.2.2 (Keyboard Events)
- 4.2.2 (Drag'n Drop Events)

This will allow you to logically have a look at what might be interesting to reimplement or use before having to browse the Qt online documentation in depth.

Window Events

By window events, all events that are produced by the window system in regards to handling any visible part of the application windows. This does also include the event processing in the other direction, because by methods like `QWidget::close()` or `QWidget::repaint()` events are sent to the window system to execute a synthetic events to manipulate the window behavior (either inside the window or affecting the whole window).

This is sometimes a bit hard to understand, so I will give another short example here. Assuming you have an application that has a window on the desktop. This window can be manipulated by the user through actions like:

- resizing
- moving
- obscuring with another window
- closing
- showing by execution

These are incoming events that are sent to the application. The event type is determined by `QWidget`'s `event()` re-implementation and converted to the according event class that provides methods to handle the event specifically. Now, when you have a look at the `QWidget` class, a lot of methods are provided for window manipulation, e.g. `resize()`. You're using these methods, but I guess you never thought about their way of execution. In effect, these methods work the other way round: they produce an event that is sent to the display by `qt_` functions to execute actions like simulating a user action. This way, events can also be produced to gain synthetic events (see below).

Within a window, the widgets are arranged somehow. As each widget is treated like a separate window internally (it always is a `QWidget` or inherits it), the same events can be processed randomly inside the window for incoming events as well as manipulating internal parts of a window.

The following chart shows the according event classes with the event types they process:

- **QShowEvent:** Processed by `QWidget::show()`

  - Event_Show

- **QHideEvent:** Processed by `QWidget::hide()`
  - Event_Hide
- **QCloseEvent:** Processed by `QWidget::close()`
  - Event_Close
  - event handler: `QWidget::closeEvent(QCloseEvent*)`
- **QResizeEvent:** Processed by `QWidget::resize()`
  - Event_Resize
  - event handler: `QWidget::resizeEvent(QResizeEvent*)`
- **QPaintEvent:** Processed by `QWidget::repaint()` calling the event handler directly, and `QWidget::update()` which generates a window system paint event.
  - Event_Paint
  - event handler: `QWidget::paintEvent(QPainEvent*)`
- **QChildEvent:** not included in the release version of Qt; to handle these events reimplement `QObject::event()` or install an event filter. Child events are inserting a child widget or removing it
  - Event_ChildInserted
  - Event_ChildRemoved
  - Event_LayoutHint

[3]

Focus Events

Focus events are somehow special to windows, but I have added a separate section for those due to the filtering of focus events in `QWidget`. A focus event is generally the fact that a window consists of several widgets who have a focus policy, which means that there can only be one widget at a time that can have the current input focus. The focus itself can be activated by a mouse click to activate the clicked widget or pressing the TAB key to forward the focus to the next widget in the tabring focus. Backwards focus setting can be done with SHIFT+TAB. This is a common usability and users expect windows to have this behavior so they can navigate the focus to the next widget. A good example for this is a dialog. If the dialog is a modal widget, it has to be finished first, otherwise is active when it gets the focus if it is the active window. Now, on dialogs widgets can be disabled as well to prohibit any user input. These disabled widgets don't get the focus either and are painted disabled.

The `QWidget` class defines the focus handling already when receiving an event. If the event type is `Event_FocusIn`, the widget gets the keyboard focus by `event()`'s conversion into a `QFocusEvent`.

This already catches a key event of the keys TAB and the combination SHIFT+TAB without processing these keys to `QKeyEvent` if there is a widget the focus can be forwarded to. Anyway, you can influence this filtering by setting focus policy. The focus policy can be set to:

[3] Qt 2.0 includes another event class `QWheelEvent` to handle events that occur by wheel-mice. The `QWidget` class also provides an already existing event handler for this, `wheelEvent(QWheelEvent*)`. Also all drag'n drop events have their event-handlers already in `QWidget`, see the notes for drag'n drop

- **QWidget::TabFocus** TAB-focusing
- **QWidget::ClickFocus** focus on mouse clicks
- **QWidget::StrongFocus** focus on TAB and mouse clicks
- **QWidget::NoFocus** no focus at all

The `QFocusEvent` class delivers information about the focus event by comparing the event type with `type()`. The method `gotFocus()` returns true on `Event_FocusIn` and `lostFocus()` returns true on `Event_FocusOut`. The `QWidget` predefined event handlers are:

focusInEvent(QFocusEvent*) for Event_FocusIn

focusOutEvent(QFocusEvent*) for Event_FocusOut

You have a lot of choices to influence the default focus handling by the methods provided by `QWidget`, e.g. you can forward the focus to another widget with setting another focus order. Mind that the focus is arranged in a ring and your implementation of this manipulation should take care that it doesn't break the focus handling. The default focus ring depends on the declaration of your widgets while constructing; if your tests result in a fuzzy focus order you have to recheck the declaration. The default design should always be left to right and top to bottom for forwarding the tab-focus. When using the geometry layout management you should declare your widget order first and then implement the layout.

`Hint:` if your widgets use multilineedits, the user expects the tab key to produce a tab in the text, not the forwarding of the focus. Therefore a simple method is to use `setFocusPolicy(NoFocus)` or `setFocusPolicy(ClickFocus)`on all additional widgets that are in the current window. Menubars and Toolbars do not have the tabfocus by default, so you don't have to set the focus policy there. An exception is the `QWhatsThis` button, which although mostly used in a toolbar, receives the input focus on TAB.

Mouse Events

Mouse events are, as the word says, generated by the user's handling of the mouse. As these will only be of interest if the mouse is over a widget, the best use to process mouse events is to reimplement the virtual methods `QWidget` provides for this. Now, the window system sends the following event types to the application by mouse actions:

Event_MouseButtonPress

Event_MouseButtonRelease

Event_MouseButtonDblClick

Event_MouseMove

This means, that the user can handle the mouse with moving the cursor in X and Y direction, press any button and release it. A button can also be doubleclicked, which is a special event and requires special handling. As the event message is filtered by the `event()` method of `QWidget`, these event types are converted from a `QEvent` to a `QMouseEvent`. Then, the mouse event is processed, whereby `QWidget` provides a set of event handlers already. What is interesting about a mouse event is not only the type, but the other parameters, as mentioned, to implement certain actions on specific events. One of the most recently used event types are probably a right button press over a widget

to open a context menu to allow quick access to commands that are available. This requires the exact position of the event's occurrence and a comparison of the button type. Double clicks are processed by the user as producing a mouse press event followed by a mouse release event and another mouse press event. As the time between the release and the next press cannot be easily determined, the `QApplication` class has methods to define the click time which is by default 400 milliseconds: `QApplication::setDoubleClickInterval(int ms)` is what you need.

The `QMouseEvent` class allows finding out the exact event by providing information about: **Button type:** using `button()`

NoButton

LeftButton

RightButton

MidButton

Mouse Position:

pos() : relative mouse position within the widget (x,y)

globalPos() : absolute mouse position on the desktop (x,y)

globalX() : global x position of the mouse pointer from left to right

globalY() : global y position of the mouse pointer from top to bottom

x(): relative mouse position within the widget from left to right

y(): relative mouse position within the widget from top to bottom

Additional Keyboard presses at the same time: using `state()` and OR'ed with Left,Right and MidButton

ShiftButton

ControlButton

AltButton

The provided event handlers are:

- Event_MouseButtonPress
  - virtual void mousePressEvent (QMouseEvent *)
- Event_MouseButtonRelease
  - virtual void mouseReleaseEvent (QMouseEvent *)
- Event_MouseButtonDblClick
  - virtual void mouseDoubleClickEvent (QMouseEvent *)
- Event_MouseMove
  - virtual void mouseMoveEvent (QMouseEvent *)

Thereby, the `mouseDoubleClickEvent()` by default only produces a mousePressEvent. You have to reimplement the `mouseDoubleClickEvent()` to receive the event and process it as it is produced as an hypothetic event, not produced by the window system under X11. Set the double click time with `QApplication::setDoubleClickInterval()`.

For MouseMove events, you have to watch that the mouse event is only handled if a button is pressed. This can be configured by `QWidget::setMouseTracking(true)` to receive all mouse movements as `QMouseEvents` in the event handler. The implementation therefore is on QWidget: the event is raised, `event()` asks if mousetracking is set to true. If not (default), the event is ignored, if yes, the event is converted to a `QMouseEvent` and delivered to the `mouseMoveEvent()` event handler.

Additionally, the widget can detect if the mouse enters the widget's space. This is done by filtering out the mouse movement before generating the `QMouseEvent` in `QWidget::event()`:

- Event_Enter
  - virtual void enterEvent (QEvent *)
- Event_Leave
  - virtual void leaveEvent (QEvent *)

An example for reimplementing an enter and leave event is `QToolButton`. The buttons in the toolbar have a automatic raising behavior in windows style, therefore the widget uses an enter event to raise the button in 3D and lowers it when the mouse leaves the widget area.[4]

Keyboard Events

A keyboard event is generally sent to the application if the user pressed or released a keyboard button, therefore can determine the event by:

Event_KeyPress

Event_KeyRelease

Handling Now, the `QWidget` class converts a keyboard event from `QEvent` to a `QKeyEvent` if the widget has the keyboard input focus; if the widget has tabfocus policy, the TAB and SHIFT+TAB key-presses are filtered out to produce a `QFocusEvent` instead a `QKeyEvent`. The `QKeyEvent` class provides more convenient methods to process the key event. Those have some specialties which I want to discuss.

Event Handlers:

`QWidget` provides two event handlers for the two event types the keyboard produces:

virtual void keyPressEvent(QKeyEvent*) for Event_KeyPress

virtual void keyReleaseEvent(QKeyEvent*) for Event_KeyRelease

Acceptance:

The widget that receives a `QKeyEvent` and re-implements the event handlers from `QWidget` has to determine if it wants to accept or ignore the keyevent, so the widget can sent it back to the parent

[4]Wheel mice are offering an additional functionality for scrolling by the wheel. Qt 2.0 offers solutions for handling wheel events in a separate event class `QWheelEvent`, therefore these are not handled as mouse-events.

widget. Therefore you have to know that the accept flag is set to true in the constructor of a `QKeyEvent`. You can clear this flag with calling `ignore()` if you don't want to process the key and sent it back.

Modifiers

The user can press so-called key-modifiers. Those are the ShiftButton, ControlButton and AltButton. The currently pressed modifier keys can be found out with `state()`, which returns the modifiers OR'ed together.

Key Values

The key values for all keyboard keys are defined in the include file `qkeycode.h`.[5] The key that produced the event can be retrieved with `key()` and then compared to the defined keycode. The ASCII value can be found with `ascii()`. Mind that the symbolic constants for key values are platform independent and allow the best usage as they are simple to remind.

Keyboard Accelerator Questions A question that often occurs is the implementation of keyboard accelerators. As this handbook primarily targets KDE programming, I will go into that as well.

Qt has a class `QAccel` which offers connections of key presses with actions. This is done by installing an event filter that filters out keyboard events that match any item inserted into the `QAccel` object. The keyboard accelerator itself has to be a combination of the `CTRL`, `SHIFT` or `ALT` keys with a normal keyboard key. Another value can be `ASCII_ACCEL` here to use the ASCII keyboard value for the accelerator.

An accelerator instance is then created by using the widget that it should work for as an event filter with the widget as its parent. Insert the keys with `insertItem(keycode, ID )`. Although setting the ID is not necessary, you should write yourself a logical ID table containing integer value defines that allow using the ID later to find the accelerator item and helps keeping an overview over the used numbers.

Then, the item has to be connected to the object and slot it shall work for on its signal `activated(int ID)` using the `connectItem()` method instead of the usual `QObject::connect()` variant.

Popup menus (only within menu-bars) already provide accelerator usage without explicitely creating a `QAccel` instance. You only have to use `setAccel()` there; see `QMenuData` for more details.

Now, when it comes to KDE, things will be a bit different because KDE offers some additional features. First of all, you have to use the class `KAccel` instead of `QAccel`; the usage is almost the same. The `KAccel` class (part of `kdecore`) also offers an insertion into menus and configuration of accelerator keys, which then can change the menu entry as well.

Further, KDE provides globally configured accelerators for standard keys.Those are defined in `kaccel.h` and only have to be inserted. The class documentation also shows the usage of standard accelerators and accelerators in general by examples.

Whenever an application offers keyboard accelerators, users often feel uncomfortable with the given values and want to change them themselves. Also, the programmer usually sets keyboard accelerators for those slots that he thinks are the most needed functions in his program; in fact he should in any case add accelerators to all of his available menu entries and functions. Further, KDE has two ways to offer configurating the `KAccel` object as well as saving the configuration to the application config file by providing a ready-to use dialog for configuration as well as a widget that can be used within a custom configuration dialog (most often a tab dialog) to configure the keys.

[5] Qt 2.0 has all keycodes coded into namespaces of the class `Qt` located in `qnamespace.h`, `enum keys`.

For accelerator configuration dialogs, see section 5.4 (Keyboard Accelerators).

Drag'n Drop Events

One of the most advanced techniques to allow application communication is drag'n drop. This offers users a cool and fast feature to handle the objects they work with in an application by an intuitive interface, catching it by a symbolic icon or by marking parts of a document and move the dragged object away from the current area. The area the dragged object comes from is therefore called a `dragsource`. Then the user moves the object away to another area of the application, to the desktop or into the area of another application. After releasing the mouse button over there, he expects the data dragged to be dropped into the drop area. Therefore the drop area is also called a `drop site` or a `drop sink`. The window system provides a protocol for this, the XDND protocol, which causes the emission of the according events. The application windows can support these events by providing methods to drag object out of the window and methods to accept a drop event. Qt implements this by a class `QDropSite`.[6] The widget that wants to use drag'n drop has to inherit this class additionally to the base widget class. Then, the `QDropSite` offers additional event handlers that convert the `QEvent` types for drag'n drop to one of the according specialized event classes. The programmer also has to take care in his re-implementation of the mouse event handlers by which mouse button holding a drag can occur. Also, Qt currently provides two types of data to decode, text and images, which should be the most common usage. The following chart contains the window system events, the event classes handling these events and the event handlers of `QDropSite`:

- Event_DragEnter
  - QDragEnterEvent
  - event handler:virtual void dragEnterEvent(QDragEnterEvent*)
- Event_DragMove
  - QDragMoveEvent
  - event handler:virtual void dragMoveEvent(QDragMoveEvent*)
- Event_DragLeave
  - QDragLeaveEvent
  - event handler:virtual void dragLeaveEvent(QDragLeaveEvent*)
- Event_Drop
  - QDropEvent
  - event handler:virtual void dropEvent(QDropEvent*)

The event handlers are all implemented as public and reimplementations should preserve to

Note: the system event `Event_DragResponse` is automatically handled by the application object internally through the Qt implementation. It causes a `QDragResponseEvent` that accepts/rejects the drag action.

```
KDE 1.x also contains another implementation of Drag'n Drop functionality.  The
description of using KDE 1.x Drag'n Drop has been left out because this will be
```

[6] Qt 2.0 makes this a lot easier. `QWidget` already contains all event handlers that are mentionend here for the class `QDropSite`, therefore you only have to remove the inheritance from `QDropSite` of your drag'n drop enabled widget and add a call to `setAcceptDrops(TRUE)` in the widget's constructor.

removed in KDE 2 and only the Qt 2.0 implementation is going to be used with an extended implementation of the XDND protocol. You should use the Qt drag'n drop functionality as this can be ported easily to Qt 2.0 and KDE 2 without any major problems.

4.3 QWidget Virtual Methods

As the event handling generally is implemented by virtual protected methods, especially the `event()` method provided by `QObject`, the `QWidget` class reimplements this function in order to sort out the incoming event and convert it to other event types that can be handled by more specialized classes. Further, it calls the provided additional virtual methods by default implementations. The programmer has a good advantage by this pre-selection of events as the widgets he creates are all derived from `QWidget` and therefore will need one or more special event handler implementation. The most common events that are processed are mouse events and for text input mostly keyboard events. The other events mostly deal with focus handling, which moves on the keyboard input focus from one widget to the next. Programmers need to know about focus handling well, because the user will expect a certain behavior over his widget when using the TAB key and the SHIFT+TAB combination to move the input focus forward.

Like explained in the Event Queue chapter, the `QApplication` takes care of converting window system events to objects of the `QEvent` class that are handled by the `QObject::event()` method. Therefore all classes that are derived from `QObject` can process event handling. The class `QWidget` already contains an overwritten `event()` method. It first checks for installed event filters (which are additionally created event filters by the programmer to redefine the default behavior by processing the event themselves or only the wanted events). Then it decides by the `type()` of the event which kind of event was called and converts it to one of the following event classes derived from `QEvent` who are delivered to the according virtual methods:

- QCloseEvent
  - virtual void closeEvent (QCloseEvent *)
- QFocusEvent: keyboard input focus event; widget gets the focus and looses it due to preselection of TAB and SHIFT+TAB by `event()`
  - virtual void focusInEvent (QFocusEvent *)
  - virtual void focusOutEvent (QFocusEvent *)
- QMouseEvent: mouse events
  - virtual void mousePressEvent (QMouseEvent *)
  - virtual void mouseReleaseEvent (QMouseEvent *)
  - virtual void mouseDoubleClickEvent (QMouseEvent *)
  - virtual void mouseMoveEvent (QMouseEvent *) :with pressed mouse button by default. Use `setMouseTracking(true)` to receive all movements
- QMoveEvent: window move event, position change
  - virtual void moveEvent (QMoveEvent *)
- QKeyEvent: keyboard events
  - virtual void keyPressEvent (QKeyEvent *)

  - virtual void keyReleaseEvent (QKeyEvent *)

- QResizeEvent: widget is resized
  - virtual void resizeEvent (QResizeEvent *)
- QPaintEvent: widget needs repainting
  - virtual void paintEvent (QPaintEvent *)

Additionally, two events are called that don't match any other event type but may be important sometimes:

virtual void enterEvent (QEvent *): the mouse enters the widget space

virtual void leaveEvent (QEvent *): the mouse leaves the widget space

Reimplementing is always needed if your custom widget wants to process the event and react to it. The reason why the event gets split up to other `QEvent` types is that the other event classes provide methods that are suitable to directly retrieving the needed event-specific data. This means, that e.g. a `QMouseEvent` can be asked for the button that caused the event or was active at that particular event as well as the global and relative mouse position where the event occurred. Mouse events are always used to pop up context menus over widgets which need to know the button (right mousebutton) and the position, because the user expects the context menu to pop up at the same position the mouse cursor currently is.

The paint event is often needed if a widget has to draw something. Instead of creating a synthetic event (a logical event caused by the program internally), call repaint() here.

4.4 Event Filters

In addition to the normal processing of the event queue that is provided by the application object, the programmer can influence the default behavior by installing event filters. As explained above, all `QObject` inherited classes use event processing through the `event()` method. Instead of writing a completely new event handling in situations where you only need some events processed by your own methods, you should write an event filter. The event filter gets installed where you like to and filters out the event directly when `QObject::event()` is called internally.

To write an event filter, your class has to overwrite the `QObject::eventFilter()` method and call `installEventFilter()` as well as `removeEventFilter()`. The declaration of these methods in `QObject` are:

```
bool QObject::eventFilter ( QObject *, QEvent * ) [virtual]
void QObject::installEventFilter ( const QObject * obj )
void QObject::removeEventFilter ( const QObject * obj )
```

The implementation of an event filter can be done in several ways. One that is possible is to create a new class for special event filters and create an instance of this class in the program. Then you can install the event filter on every instance you like to to achieve the same event filter on all instances independent of their class as well as redefining event processing of existing classes without inheriting them.

An example would be:

```
// Classdeclaration

class KMyAppFilter: public QObject
{
protected:
  virtual bool eventFilter(QObject* object, QEvent* event);
};

// Filterimplementation

bool KMyAppFilter::eventFilter(QObject* object, QEvent* event){

  if(event->type() == [the eventtype you like to filter])
  {
    [your filter implementation]
    return true;  // the event has been caught and processed
  }
  else
  {
    return false; // return false to continue processing the event with QObject::event()
  }
}

// installing the filter

QObject* myfilter= new KMyAppFilter();
QPushButton* mybutton= new QPushButton();

mybutton->installEventFilter(myfilter);
```

Another solution would be to reimplement the `eventFilter()` method in your inherited class as long as the base class is `QObject`, e.g. if your view area of your application wants to process a certain event that is not covered by the virtual methods `QWidget` provides. Then you have to install the event filter at the place you like to; normally this would be in the constructor of your class. With removeEventFilter() you can stop the event filter from processing the events any time.

`Note: KApplication already has a global application event filter installed to filter out CTRL+ALT+F12 for KDebug`

4.5 Synthetic Events

Before describing what synthetic events are and how they can be used by the programmer, I want to review the last sections in brief.

We saw that the application object receives the window system events, processes them and creates event objects from the classes the library provides. The converted event can then be handled by event handlers that are specialized on the event class to retrieve further information about the event. Finally, we can influence the event handling itself by installing event filters and overwriting provided event handlers.

This does the "normal" job of an application to execute actions according to user invoked events. On the other side, this system offers another possibility: the fact that the events are converted to class instances can be reversed - a so-called synthetic event can be created which fakes an original

window system event. The next advantage is that these events are independent of the underlying window system.

A good possibility where this feature could be used would be e.g. for learning programs. Those are almost non-existent for Unix but could offer a market to teach beginners how to handle programs similar to commercial products already available on other platforms. Also this could be a part of a help-system which an application can provide.

An example description how to implement this:

Provide a help window with a button that invokes a step e.g. "Show me". On pressing the button, the cursor will move to the desired location, e.g. to a pushbutton on the screen. The implementation then has to find out the exact position of the button and calculate the center coordinates the mouse pointer has to move to. Then the mouse pointer could move there by construction of a `QCursor` and using `setPos()`. The start position can be found out in the mouse event that called the function. Then, the cursor has to move visually by using `setPos()` in a loop where a `QTimer` could be used to run between positions to slow down the move so that the user can follow the mouse pointer.

4.5.1 Creating Events

Now, to come to the actual implementation of a synthetic event, you have to know the event you want to create. Therefore, you need the constructor parameters for the event classes. The following list contains the constructors including the event-classes hierarchy:

QEvent(int type)

`type` is one of the events declared in `qevent.h`[7]

QCloseEvent()

takes no parameter. Mind that the accept flag is set to false

QFocusEvent(int type)

`type` is either `Event_FocusIn` or `Event_FocusOut`.

QKeyEvent(int type, int key, int ascii, int state)

takes `Event_KeyPress` and `Event_KeyRelease` as `type`. `key` is one of the keys defined in `qkeycode.h`. `state` is `ShiftButton, ControlButton, AltButton` OR'ed.

QMouseEvent (int type, const QPoint & pos, int button, int state)

The `type` parameter must be `Event_MouseButtonPress, Event_MouseButtonRelease, Event_MouseButtonDblClick` or `Event_MouseMove`. The `button` is `LeftButton, RightButton, MidButton, NoButton`. `state` is `ShiftButton, ControlButton` and `AltButton` OR'ed for event `Event_MouseButtonRelease`, for events `Event_MouseButtonPress, Event_MouseButtonDblClick state` includes `LeftButton, RightButton, MidButton`.

QMoveEvent(const QPoint & pos, const QPoint & oldPos)

`pos` is the new position the widget shall move to, `oldPos` the old position. Retrieve the old position before creating the event with `QWidget::pos()`.

QPaintEvent(const QRect & paintRect)

raise a paint event to repaint the area `paintRect`

[7] Qt 2.0 uses all events from an `enum` instead of the #defines. See `QEvent`. The types are almost the same except they leave out the `Event_` prefix.

QResizeEvent(const QSize & size, const QSize & oldSize)

resizes the widget from `oldSize` to `size`. Retrieve the old size before creating the event with `QWidget::size()`.

An example on how to create an event would be:

```
QMouseEvent press_quit(Event_MouseButtonPress,
                       quit_button->pos(), LeftButton, LeftButton);
```

This creates a mousePressEvent() for the widget `quit_button` with the left button.

4.5.2 Sending Events

After creation, the event has to be sent to the application instance to call its execution. Thereby, two ways can be used: one that directly processes the event and one that will place the event in the event queue at the last position:

Direct execution:

```
QApplication::sendEvent(quit_button, &press_quit);
```

The `sendEvent()` waits for the result and returns true or false depending if the event has been accepted or not.

Placement into event queue:

```
QApplication::postEvent(quit_button, &press_quit);
```

`The event for postEvent() must be allocated on the heap as it gets deleted immediately after the posing.`

To turn a posted event into a send event, use `sendPostedEvents(QObject * receiver, int event_type)`. This requires the options given at the constructor. As you may see, some constructors don't need an event type, therefore the according event type can be found in 4.2.2 (Event Types) but is also simple to guess as they are only responsible for one event type. Example: `QCloseEvent` only takes `Event_Close`, `QPaintEvent` only takes `Event_Paint`.

4.6 Event Precedence

In relation to influence the event behavior of the application, the programmer often faces situations where long operations block the `Event_Paint` and lead to a scrambled look of the application windows. These situations can be solved either by using a progressdialog that indicates the operation progress or by event precedence. This means that the current event gets stopped and the event queue is processed. The class `QApplication` offers a solution for this by two methods which are identical except the parameters. One is `processEvents()`, which processes pending events for 3 seconds or until there are no more events in the event queue. The other, more likely used method is `processEvents(int maxtime)`, where `maxtime` is the time in milliseconds during which pending events can be processed.

On one hand this means stopping the current long operation which then would take even longer to get finished if pending events are in the queue, but the user cares more about the visible state of an

application than if an operation which takes some time will take a second longer (or even parts of a second).

KDE offers an additional library for I/O operations in the upcoming KDE 2, which is under development. This will allow running the long I/O operations outside the application's process as multi-threading is not supported by Qt directly.

4.7 Summary

After this long chapter about signals, slots and events, I want to append a short summary so you can recapitulate the collected knowledge about application behavior.

- An application can communicate internally by signals and slots
- Signals are sent out without caring about who will catch it
- Slots are normal methods that can connect to signals and react as well as they can be called where allowed by their access attribute
- The user communicates with the application through the window system
- The window system reports the events to the application
- The application converts window system events to `QEvent`s
- The events are proceeded through an eventual application global event filter
- The event that passes the filter gets forwarded to the according window e.g. the current modal window
- The widget receives the event and can have an event filter that comes first when the reimplemented `QObject::event()` is called.
- If the event passes the widget event filter, `event()` proceeds to convert it to the according `Q***Event` class
- The event filters for these event types are called to react on the event

Further, we saw that the programmer can influence the behavior by:

- reimplementing any stage of virtual methods filtering events
- creating synthetic events
- sending synthetic events directly or into the event queue
- the event queue can be given precedence that stop long processes to allow execution of waiting events to be processed

Finally, I hope this has given at least experienced C++ programmers a good insight and explanation on how Qt and KDE work. I have collected the information by working myself into the class structure and I hope that this collection makes it a lot easier for other programmers to get started especially in the advanced chapters of application design and programming. The information value is therefore not granted to be exact; if you may find any misconcepted or incorrect information, please contact me via email.

Chapter 5

User Control Elements

5.1 The Menubar

The menu bar is a central component of the main window. It allows the user to execute operations that the application (or to be precise: the mainview) offers in regards of manipulating the main view's contents. In opposition to Qt's `QMenuBar`, KDE offers the use of the class `KMenuBar`. Additionally, the menu bar is already constructed for the programmer when using `KTMainWindow` with the first call of `menuBar()`.

The menu bar itself, independent which class is going to be used, contains entries that the user can select with the mouse or by using keyboard accelerators with the ALT-key and the underlined character. The menus that have to pop up on a selection have to be created with the class `QPopupMenu`, which itself only provides the popups, entries have to be inserted using the methods provided by `QMenuData`.

Mind that menu-bars should always contain all functions a program has to offer except those that can be accessed by additional dialogs. The menu bar also makes use of the `KApplication` help menu already provided. Inserting the menu is just easy with

```
menuBar()->insertItem(i18n("Help"), kapp->getHelpMenu());
```

Example construction of a menu bar with using `KTMainWindow` and setting the menu bar explicitely:

```
my_menubar=new KMenuBar(this,"my_menubar");

file_menu = new QPopupMenu;
file_menu->insertItem(Icon("filenew.xpm"),i18n("&New..."),
                      this,SLOT(slotFileNew()),0,ID_FILE_NEW);
file_menu->insertItem(Icon("open.xpm"),i18n("&Open..."),
                      this, SLOT(slotFileOpen()),0 ,ID_FILE_OPEN);
file_menu->insertItem(i18n("&Close"),
                      this, SLOT(slotFileClose()),0,ID_FILE_CLOSE);
file_menu->insertSeparator();
file_menu->insertItem(Icon("save.xpm"),i18n("&Save"),
                      this, SLOT(slotFileSave()),0 ,ID_FILE_SAVE);
file_menu->insertItem(i18n("Save &As..."),
                      this, SLOT(slotFileSaveAs()),0 ,ID_FILE_SAVE_AS);
```

```
file_menu->insertItem(Icon("save_all.xpm"),i18n("Save All"),
                      this, SLOT(slotFileSaveAll()),0,ID_FILE_SAVE_ALL);
file_menu->insertSeparator();
file_menu->insertItem(Icon("fileprint.xpm"),i18n("&Print..."),
                      this, SLOT(slotFilePrint()),0 ,ID_FILE_PRINT);
file_menu->insertSeparator();
file_menu->insertItem(i18n("E&xit"),
                      this, SLOT(slotFileQuit()),0 ,ID_FILE_QUIT);

my_menubar->insertItem(i18n("&File"), file_menu);

setMenu(my_menubar);
```

The example creates a menu bar and a popup menu first. Then the popup menu is filled with entries. The used method of `QMenuData` here allows an implicit connection to the method to call when the popup menu emits SIGNAL(activated(int)). The zero parameters after the slot declaration is left out as the example assumes the program will use `KAccel` to set the according keyboard accelerators with `changeMenuAccel()`. Further you can see that the integer value ID is inserted like the method name with all uppercase letters and underscores to separate the words. The menu id's themselves are set with #define in a separate file to keep track of the used numbers. You would think that you don't need the menu id if the `activated()` signal is already connected - in fact the id can be used to forward the signal `highlighted(int)` to a method that compares the id by a switch statement and sets a statusbar help message for the menu entry. You could as well do that for the signal `activated(int)` as well to call the according method by a switch statement. Then you have to add a `connect()` for each popup menu you want to use.

You can as well insert a separator into the menu bar with `my_menubar->insertSeparator()`. This will align all entries inserted after the separator to the right in Motif style, in windows style this has no effect.

The creation of a separate menu bar allows the creation of several menubars which can be set as the actual menu with `setMenu()`. This is how we did it in KDevelop to change the menu bar when switching to the dialogeditor and back to the project editor.

Finally, the ampersand in the menu entry sets the following character as the keyboard accelerator when the user presses ALT+ the character.

5.2 The Toolbar

Toolbars are another component that enhance user interaction with symbols representing most needed functions that an application provides. The KDE libraries are again offering another class to use with KDE applications, `KToolBar`. As `KTMainWindow` already handles the geometry management for all user interface elements, it also provides methods to add toolbars. Now, the good thing is that you can use as many toolbars as your application may require and the creation is done easily with `toolBar()`. This method also takes a parameter which is the according toolbarnumber. The first toolbar has by default number 0, so the parameter can be left out there; the next has to be called with `toolBar(1)` etc. You can also make this more variable with a define for your toolbar and use a descriptive name; this avoids changing the toolbarnumber everywhere when you decide to set a toolbar to another position.

Now, when using a toolbar, you have many choices. The class `KToolBar` provides a whole set of methods to insert user elements such as buttons, which is probably the most recently used method,

delayed popups, lineedits, combos and generally a widget of your choice. Further you need to know that toolbars are created with using the full width of the parent window - but that is configurable. All following toolbars are then appended to the end of the last toolbar. Also you can set the toolbar to show at a certain position. This is often used by applications that offer painting facilities. The following example shows you how to use a toolbar with `KTMainWindow`:

```
1   // first call of toolBar() - creates the toolbar 0.
2   toolBar()->insertButton(Icon("new.xpm"), ID_FILE_NEW, true, i18n("New File"));

3   QPopupMenu* select_menu = new QPopupMenu();
4   toolBar()->insertButton(Icon("select.xpm"), ID_OPTIONS_SELECT, select_menu, true, i18n("Select O

5   connect(toolBar(), SIGNAL(clicked( int )), SLOT( slotSelected( int )) );


6   Foo::slotSelected( int id ){

7     switch (id ){

8       case ID_FILE_NEW:
9         slotFileNew();
10        break;
11    }
12  }
```

The above explains some specialities for toolbars - we will discuss these now in detail. First of all, you see that we used the `toolBar()` method. This returns a pointer to the according toolbar and creates one if the toolbar doesn't exist. We use the `insertButton()` method to add a toolbar button representing the standard "New File" action. Now, when looking at the class-documentation of `KToolBar`, you see that there is a `QPixmap` required as the first parameter. Here, we only set the name of the pixmap embraced by the `Icon()` method. This is a macro which makes inserting icons very easy, provided by `kapp.h`. In fact, it makes the application's `KIconLoader` instance load the icon with the filename `new.xpm` for you using a list of standard directories within the KDE File System. Additionally, the icon `new.xpm` is already provided - you don't have to paint it yourself. The KDE libraries come with a whole set of toolbar icons that are ready to use for insertion. This is also the reason why, when testing an application, sometimes a button looks a bit scrambled although you have painted the pixmap - it just can't be found if it isn't installed at the correct location within the KDE FSSTD; whereas standard icons are already present.

The exact execution of the `Icon()` macro is therefore:

```
KApplication::getKApplication()->getIconLoader()->loadIcon("new.xpm")
```

which implicitely uses the `kapp` macro to get the application object. You see that using this macro saves lots of code but offers a very nice way to load an icon for a toolbar button.

The second parameter, the ID of the button, is a macro that our application specifies itself to name certain actions logically by a #define. Obviously, you could think that using another method of `KToolBar` would do the same when directly specifying the receiver object and the slot to call, but this way you save a lot of code. The first place is that you only have to write one `connect()` (line 5) to connect all toolbar elements. The other is, that by this way you can use the same ID for your toolbar items as well as for your menubar items. The following code completes this example with the according menubar action:

```
QPopupMenu* file_menu = new QPopupMenu();
file_menu->insertItem(Icon("filenew.xpm"),i18n("&New..."),0,ID_FILE_NEW);

connect(file_menu, SIGNAL(activated(int)), SLOT(slotSelected(int)));

menuBar()->insertItem(file_menu, i18n("&File"));
```

By this, the `file_menu` is already connected to the `slotSelected()` method and the corresponding entry "New File" executes the same action. Just collect all your ID's as #defines into one file and you can keep a good overview over the used numbers (which naturally have to be integer values).

The next example in line 3 and 4 add a button that opens a popup menu when the user presses the button. This can be used if the button itself does not perform any action but represents a better access method for e.g. a list of entries. Just create your popup menu and insert it with the according ID and pointer as a button.

Besides the example you can do a lot of other things like making a button a toggle button. This is useful if the button executes an on/off action (which in the corresponding menubar popup is represented by a checkmark). See the complete reference of `KToolBar` for more information.

5.3 The Statusbar

KDE also provides the pendant to Qt's `QStatusBar`, `KStatusBar`. The statusbar can contain labels as well as widgets, such as progress bars (those have to have the statusbar as parent). The statusbar is used to display information about the current state of the application and gives hints about the usage of commands e.g. over toolbars and menubars.

5.4 Keyboard Accelerators

Keyboard accelerators are a good enhancement for any kind of application. GUI application designers often think that the user can access all methods with the provided graphical interface, but advanced users usually want to work as fast as possible and using the mouse to call actions doesn't make an application very attractive. The more a user will make use of your application, the more he will miss keyboard accelerators. Fortunately, the Qt and KDE libraries provide a whole set of functions and classes to support keyboard accelerators in conjunction with GUI elements. This section therefore collects all these classes and shows the possible implememtation for various situations.

5.4.1 Menu Accelerators

The first thing where keyboard accelerators are used without much effort from the programmer's side is over menuentries. The menubar, as mentioned above, consists of a set of `QPopupMenus`, which are inserted in the order they will appear later from left to right. The popup menu itself can be called by the user by a keyboard shortcut if he presses the ALT-key together with the underlined character of the desired menu of the menubar. The menuentry itself has to define the underlined character at the time you insert the popup into the menubar.

Example:

```
menuBar()->insertItem(i18n("&File"), file_menu);
```

Mind the ampersand in front of "File". This makes the "F" the key with which the user can pop up the popup menu `file_menu` when pressed together with the ALT-key. The same goes with entries within the popups, where the user, after a popup is active, only has to press the key to invoke the desired action. Selfexplaining, you should watch the usage of underlined characters very closely, because a key that is used twice either in the menubar or within the same popup makes the last inserted item the one that is activated and this makes the previously defined shortcuts useless.

Using the menu-accelerators is therefore very easy for the programmer - just select the key you want to be used and set an ampersand in front of it while inserting the entry. As the KDE applications get internationalized, translators take over the responsibility to place the keys in their translated version later. They should watch the same principles for placing the keys and shouldn't only translate one-to-one but test the application later if everything is accessible again by their keyboard shortcuts.

5.4.2 Tabulator and Button Accelerators

The keyboard accelerators with the ALT-key don't only work with the menus - they do the same over dialogs, tab-pages and on buttons. Therefore it should be used whereever possible, the principle is the same: on a dialog, you have to watch the used keys, on e.g. a `QTabDialog` you have to watch the keys for each page **plus** the used keys to activate the pages in the `addTab()` methods.

Within the user interface, buttons have an additional option - as mentionend, the keyboard input focus is forwarded in user interfaces with the TAB and SHIFT-TAB keys. When a button receives the focus, it gets a slight frame like other active elements, the user has to press the SPACE-key to execute the action connected to the button. Now, this can be changed by using `setDefault(true)` on one button or by setting `setAutoDefault(true)` on several buttons on the dialog. The difference is that if the User presses the ENTER-key, the default button will be pressed. If several buttons shall provide this behavoir, `setAutoDefault()` has to be used on those. If one of them receives the keyboard input focus, it will automatically become the default button.

KKeyDialog: keyboard accelerator configuration dialog KKeyChooser KAccel

5.5 Other Widgets

Control elements are used within visible areas of the application and can be combined together to create a dialog or view. Beyond the control elements that the Qt library provides, KDE offers:

KButton: active raise-lower button KButtonBox: manages buttons KColorButton: button displaying a color setting KIconLoaderButton: button displaying a selected Icon KCombo: similar to QComboBox KDatePicker: Date selection widget KIntegerLine: lineedit that only accepts integer values KLedLamp: LED lamp KLined: line edit KPanner: panner devider KNewPanner: panner devider managing two widgets KPopupMenu: popup menu with title KRestrictedLine: lineedit that only accepts certain input KSelector: value selector KSeparator: standard separator KSlider: slider widget KTabCtl: tabulator widget

Chapter 6

KDE Dialogs

A very useful thing of the KDE libraries is that they provide already constructed dialogs for various purposes that are common to a lot of desktop applications. This has two reasons: a) the user feels comfortable using these dialogs if he knows them already from an application and b) lessens the programmer's work a lot. In section 5.4 (Keyboard Accelerators), you already got to know one of these dialogs that KDE provides to configure keybindings. For the other dialogs that are mostly part of the `kdeui` library, the usage is mostly as simple as for the `KKeyDialog` and enhances applications within seconds of coding efforts. You should always first look for an already existing solution for general value requests from the user before starting to implement a new dialog from scratch. Further, you don't have to care about internationalization as these dialogs are part of the KDE libraries and are already translated.

6.1 KMsgBox

The `KMsgBox` class provides a whole set of message boxes that match everyday life usage in applications. Using KMsgBox has a lot of advantages: you can use one of the static methods to retrieve results on standard questions and you can still influence the behavior by setting text, window text, symbol and button text.

- message(): providing a single message box with an OK button to inform the user
- yesNo(): provides a yes/no question box.
- yesNoCancel(): provides a yes/no/cancel box with three buttons. Used e.g. to quit an application with the question: Document has been modified. Would you like to save changes ?. Then the yes-button would mean saving changes and exiting the application, no would mean exit without saving and cancel would stop any exiting and just returns.

If this doesn't match your actual need, you could as well create a new KMsgBox instance that can have up to four buttons. This can be used by applications that have multiple open files but don't want the user to ask if he would like to save changes for each file separately; therefore these will need a button "Save All" or something. Then you could program the dialog towards your needs like the static methods and will receive the correct result.

As usual with dialogs, the return value is that what a programmer usually has to process by retrieving it into a variable and then compare with `if()` for the actions to execute.

6.2 KQuickHelp

The `KQuickHelp` class provides a good way to add quick-help dialogs to widgets. The user can access the quick-help by a context-menu entry "Quick-Help" and is therefore easy to use and gives enough information where a Help-button for the manuals would be too much and a `QToolTip` would be too less. The reason I include `KQuickHelp` into the provided dialogs is that the class-documentation itself contains example usage, but doesn't cover all formatting possibilities, therefore these are listed in detail here.

6.2.1 Usage

A quick-help window therefore can always be added to a widget by using the static method `add()`, also one help message can be used for more than one widget. The example shows this by adding one message that applies to two widgets that are providing a funcitionality that depends on each other:

```
#include <kquickhelp.h>
#include <kapp.h>
#include "mydialog.h"

MyDialog::MyDialog(QWidget* parent, const char* name): QDialog(parent, name)
{

  file_lineedit= new QLineEdit(this, "file_lineedit");
  file_select_button= new QPushButton(this, "file_select_button");

  KQuickHelp::add(file_lineedit,
  KQuickHelp::add(file_select_button, i18n("Select the filename to process.\n"
                                           "You can use the lineedit or the\n"
                                           "button to select the filename.")));
}
```

6.2.2 Text Formatting

The text inside your quick-help window can also be formatted to fit various needs, even hyperlinks. Using the KDE-FSSTD, you can also access your online-documentation to provide a link for further information by just using your html filename. This is considered a nice way to give the user the best information in some cases where the purpose of certain functions is too difficult to explain in a quick-help window.

The followindg list contains the valid tags for text formatting:

Font Attributes	Tag	Short-form Tag
bold font	<bold></bold>	
italic font	<italic></italic>	
underlines	<underline></underline>	<u></u>

Font Size	Tag	Short-form Tag
increase	<FONTSIZE +>	<+>
decrease	<FONTSIZE ->	<->

Font Selection	Tag	Short-form Tag
default font	<FONT DEFAULT>	<DEFAULT>
fixed font	<FONT FIXED>	<FIXED>

Indentation	Tag	Short-form Tag
right indent	<INDENT +>	<i+>
left indent	<INDENT ->	<i->

Color		
RGB color	<COLOR #>	
red text	<COLOR RED>	><red>
green text	<COLOR GREEN>	<green>
blue text	<COLOR BLUE>	<blue>
white text	<COLOR WHITE>	<white>
yellow text	<COLOR YELLOW>	<yellow>
black text	<COLOR BLACK>	<black>
brown text	<COLOR BROWN>	<brown>
magenta text	<COLOR MAGENTA>	<magenta>
cyan text	<COLOR CYAN>	<cyan>

Newline	
Hyperlinks	<link linkname></link>

Thereby, valid `linknames` are:

- http://yourlink
- info://yourlink
- ftp://yourlink
- file://yourlink
- mailto:your_address@your_domain

These links will be opened using the `kfm` (KDE File Manager). All other linknames assume that you want to access your application's online-help documentation and therefore use the `linkname` as the file you want to access and tries to open it with the KDEHelp program.

6.3 File Dialogs

As the `kfile` library provides several dialogs for retrieving filenames as well as directories, those have to be separated towards which class and method to use for which purpose.

Generally, the `kfile` library offers:

- a `KFileDialog` class, which is a specialized `KFileBaseDialog` and provides the most needed static methods to retrieve filenames.
- the `KFileDialog` class itself, which can be subclassed but also be used for retrieving several filenames and directories.
- the `KFilePreviewDialog` class, which offers file-dialogs that can display selected files by their contents if the developer provides a preview module that is able to show a preview of the filecontents

For general file/directory services, the classes `KFileInfo` and `KDir` can be used.

The following sections will discuss the usage and handling of the according file-dialogs in applications.

6.3.1 KFileDialog

The `KFileDialog` class provides four static methods to ask the user for a filename. As the filedialog itself can handle the creation of new folders, storing bookmarks etc, the user will be thankful if you use this dialog to ask for a filename to open and a filename to save files to. The `KFileDialog` class itself is a specialized class that is based on `KFileBaseDialog`, so if the given methods don't fit your needs you can always inherit from `KFileBaseDialog` to customize the settings.

The following examples show the usage for each purpose:

```
// request a filename to open

QString open_filename;
open_filename=KFileDialog::getOpenFileName()

if(!open_filename.isEmpty())
{
  // read the file
}
```

This asks the user for a filename to open. The `KFileDialog` shows and retrieves the information. If the user cancels the filedialog, the return string will be null, therefore you have to test first if `QString::isEmpty()` doesn't return `true` before opening the file actually.

The same goes with the static methods `getSaveFileName()`, `getOpenFileURL()`, `getSaveFileURL()`, whereby each function takes parameters to set the starting directory, mime-types and, as usual, the `QWidget parent,name` parameters.

The parallel methods for `getSaveFile` and `getOpenFile` behave identically for retrieving remote and/or local files with the URL dialogs.

6.3.2 KFileBaseDialog

The class `KFileBaseDialog` provides the basic interfaces for building filedialogs; therefore filedialogs can be customized in wide ranges towards your needs and is the most flexible way to construct filedialogs. Besides that, the class provides additional functionality for other standard cases like retrieving a directory name. Further functionality can be achieved by inheritance.

Retrieving a directory name

The class provides retrieving a directory name by the static method `getDirectory()`. The following example shows the usage:

```
QString the_directory;

the_directory=KFileBaseDialog::getDirectory();

if(!the_directory.isEmpty())
{
        // do something
}
```

[1]

6.3.3 KFilePreviewDialog

The class `KFilePreviewDialog` provides another specialized, but more seldomly used filedialog. Its best feature is that it provides an area where the programmer can use a preview widget for his file format to open. The best usage is made within graphic programs that operate on pictures.

6.4 KColorDialog

The `KColorDialog` provides an easy-to use interface to receive color values from the user. Color values are always requested, if the application is a drawing or painting program to select the current brush as well as for e.g. KDevelop's dialogeditor to set color values for widgets.

The usage itself is often combined with a `KColorButton`, which is a specific `QPushButton` implemented in `kdeui` that displays a color and calls the `KColorDialog` already when the user presses the button. Anyway, you can call the color dialog from your menu bar or toolbar as well to retrieve a color value from the user.

Using the colordialog is very easy inside applications. The class provides a static method which can be called to retrieve the color value:

```
#include <kcolordlg.h>


QColor myColor;

int result = KColorDialog::getColor( myColor );
```

This creates an instance of QColor to store a color value and by calling the static method `getColor()` the color gets the selected value. The returned integer value will probably be of no interest - its the result code of `QDialog` that specifies the dialog has been exited via the OK or Cancel button.

6.5 KFontDialog

The `KFontDialog` will retrieve you a value for a font currently avaliable on the system. Therefore using the fontdialog will mostly only make sense where you will need a font; the most recent usage is made by text editors but could also be used to get a formatting for a text to draw inside a widget as well as into a picture.

[1] Of course, you can also use `KFileDialog` instead of `KFileBaseDialog` if you like your API to be more consistent.

To retrieve the font value, you probably will use the static methods of `KFontDialog`. The example shows the usage:

```
QFont myFont;
int res = KFontDialog::getFont( myFont );
```

This is it already - you only have to create a `QFont` instance to contain the font value. Then call the font dialog with the font and after the dialog was executed, your font will have the selected value. Then you have the methods of `QFont` to determine which type of font the user selected etc. to use the font within the application.

6.6 KIconLoaderDialog

For applications requiring an icon selection, KDE provides the `KIconLoaderDialog`. The main purpose is to select an icon on the system to draw it on a button for example. Usage is made by the KDE window manager to select the icons for mounted/unmounted states of device links. Then the values for the link are displayed on the according button to display the current selection; the filenames get stored in the link file and can be drawn on the desktop as a symbol by loading the icons dependent on the state of the connected device. There, an additional widget of the `kdeui` libary is used, the `KIconLoaderButton`. Like the `KColorButton`, this class will call the icon loader dialog when the user presses the button and will display the selected icon on the button.

As the `KIconLoaderDialog` class does not provide any static methods, you have to create an instance first and then call `QDialog`'s `exec()` method to display the dialog. Another possibility would be to call `selectIcon()` to execute the dialog but retrieve a `QPixmap` value instead. The selected Icon will be in your `KIconLoader` instance (depending on the used constructor which one - the standard constructor uses the application's `KIconLoader`), therefore the value can be processed with the according methods of `KIconLoader`.

6.7 KWizard

The `KWizard` class already contains a predefined dialog to construct wizards that lead the user through an input process. Thereby, the wizard dialog provides the necessary buttons and draws the according page numbers already, so that you only have to construct your widgets you want to use as the single pages for the dialog and insert them in the order you want the user to proceed while calling the wizard.

6.8 KSpellDlg

(kspell) spell checking dialog for use with Ispell

6.9 DatePickerDialog

(kab) date selection dialog

6.10 Qt Dialogs

6.10.1 QFileDialog

6.10.2 QMessageBox

6.10.3 QPrintDialog

6.10.4 QProgressDialog

Chapter 7

Provided Views

7.1 The KEdit View

7.1.1 KEdGotoLine: Go-to-Line dialog for editors

7.1.2 KEdReplace: Search and replace dialog for editors

7.1.3 KEdSrch: search dialog for editors

7.2 The KHTML View

Chapter 8

Process Handling

The KDE UI library provides the classes `KProcess` and `KShellProcess` to run external processes that are invoked within the application that needs to run another application. This has generally two advantages:

- you don't have to reinvent the wheel when commandline programs already exist
- your application's event queue is not blocked by long operations

A lot of applications already make wide use of these classes as they are very flexible and provide the necessary interface not only to start another application but to control its output and termination. As mentioned, a lot of Unix applications are already available but only work on commandline. The commandline arguments are hard to remember and most users won't ever touch them if they don't need them really. For occasional usage, the interface is too complex and therefore not very user-friendly. As KDE applications target a desktop system where even unexperienced users can feel themselves at home, this is the best way to write so-called front-ends for terminal applications.

Another possible use even for KDE programmers would be to write their target application as a commandline program and provide a user-friendly GUI interface.

The following sections will describe the `KProcess` class first, then the usage of `KShellProcess`, as this is a subclass of `KProcess`, therefore differs only in its usage.

8.1 KProcess

The `KProcess` class is based on `QObject`, therefore able to communicate by signals and slots. It can be used to start any executable binary as a child process on the local system and control it by communication and run mode. To use `KProcess`, include `kprocess.h` and create an instance of `KProcess`. If the instance has been created and used already, you have to call `clearArguments()` to ensure the arguments are empty before the next usage. The actual usage is to transmit the complete commandline argument to the process instance using the operator << as strings. Then the actual process is called with `start()`. This function has to be called with the run mode and communication.

8.1.1 Run mode

The run mode of the external application can be set when calling `start()` as the first argument. The run mode can be one of `DontCare, NotifyOnExit, Block`. Now, what does this mean to the

application that is called and to the application that calls the process ?

- DontCare: The child process is called and started with the given commandline arguments. Easy to guess, DontCare means that the caller is not interested if the child process has exited or not. The two applications run concurrently, but the invoking process (usually the GUI application) doesn't get notified and runs like without starting any process.
- NotifyOnExit: both processes run concurrently like in DontCare run mode. The difference is that the process controller can emit the signal `processExited()`, which can be caught to determine the child process has finished. The notification can be used to reset any statusbar message informing the user that the process is running, this should be used as a guideline to inform the user about the current state of the application. Mind that the invoking application is responsible for the child process as it is unlikely that non-experienced users will control and other process that they see.
- Block: the child process blocks the caller's event handling and program execution. This is not recommended to use within GUI applications as even the call `processEvents()` won't be executed; therefore the event handling cannot be called to execute even by event precedence.

The `start()` method also returns if the start has been successful or not. Therefore you should always call the method with an `if()` statement to display a message box if starting the process returns false. Reasons for `start()` to return false could be:

- the commandline argument list is empty (which is your implementations's fault)
- the process which is to be called is already running
- starting the child process failed

To inform the user why the process cannot be executed, you have to investigate these three possibilities. The first possibility, an empty commandline, depends on your GUI that provides the methods setting the commandline options. Normally, you would retrieve them by a dialog where the user sets the options on how to start the application. The GUI for these dialogs normally uses radio buttons to let the user choose one of several options (or more if the process allows this), a lineedit for filenames (with an additional file-selection button to call a `KFileDialog::getOpenFileName()`), eventually a lineedit for output locations, also with a `KFileDialog::getSaveFileName()`.

The second possibility mostly occurs in situations where the application has been started twice or the user has opened another instance of its main window where or by which he caused another process call. In this case, you could use `getPid()` to determine the current process ID which can be used in a message box to show that the process is already running.

Finally, failing a call of a child process often means that the program is not available on the system. There, you should inform the user that he has to install the program to ensure functionality. Another option would be to test the `PATH` environment variable of the user for the directories he uses to call applications. Then you could test with `QFile::exists()`, if the binary is there even before trying to start it.

8.1.2 Communication

- NoCommunication
- Stdin

- Stdout
- Stderr
- AllOutput
- All

8.1.3 Example Usage

Chapter 9

Copyright