

QT/KDE Research Paper

CS 159.3 B

27 February 2002

Submitted by:

AY-AD

CAJIPE

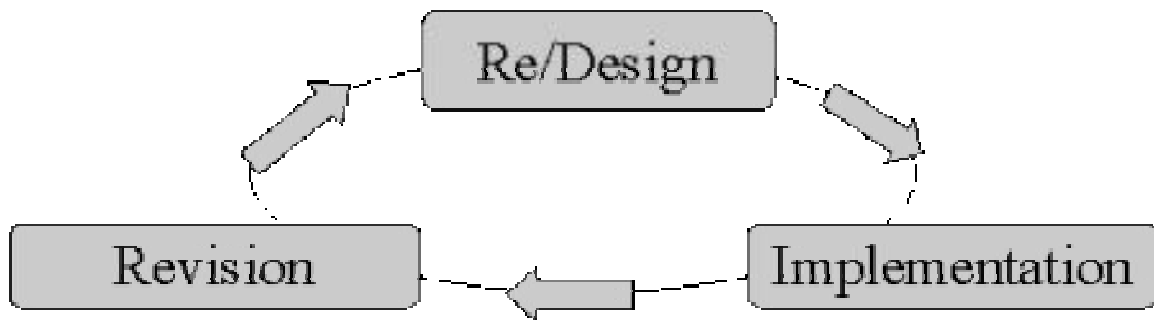
CHAN

CHENG

MAGDARAOG

KDE Facts and Figures

- KDE is a big project. While it is very hard to quantify what this means exactly, note that: The KDE CVS source code repository holds currently about 1.5 million lines of code. (To put things into perspective: The i386 version of Linux 2.0.31 consists about 600,000 lines of code.) Several hundred developers are coding for KDE.
- The translation team alone consists of 110 individuals.
- 7294 CVS commits were executed during July 2000.
- KDE has more than 24 official WWW mirrors in over 16 countries.
- KDE has more than 71 official FTP mirrors in over 30 countries.



The KDE Development Model

History

K Desktop Environment or KDE was developed by a group of dedicated programmers who wanted to create something new and useful. The KDE Project was founded in October 1996 by Matthias Ettrich in Germany. Its original purpose was to create powerful desktop environment for Linux and other Unix systems similar to the commercially available Common Desktop

Environment or CDE available for commercial Unix platforms such as Solaris. The end result has been a project much wider in scope than simply emulating CDE.

Contributors to the KDE project include hundreds of developers around the world who communicate via e-mail and the occasional in-person conference. Individuals who want to participate in the development of KDE can work on the core components such as the window manager, the desktop Panel, and the other components, or they can create their own application for KDE, using the KDE-defined Application Programming Interfaces or APIs.

Among the contributors to the KDE Project are translators, writers, user-interface designers, and multimedia specialists, in addition to systems and application programmers.

KDE doesn't replace the X Windows system on the Linux computer. Instead KDE uses the X Window system as a graphical base on which to build the desktop.

A KDE Distribution includes a large collection of applications such as standard utilities, system administration tools, and entertainment packages. It even includes KOffice, an integrated office package similar to GNOME Office and Microsoft Office.

KDE DISTRIBUTION

As with Linux, KDE uses the term "distribution" to describe a complete collection of desktop components and the default set of applications that accompany them.

A KDE distribution includes the following components:

- The graphical libraries used by all KDE applications
- The core KDE libraries, also used by all KDE applications
- A collection of support functions, used by most applications

- The base applications that define KDE, such as the window manager and the Panel/ Taskbar programs
- Optional applications that can be installed on a running KDE system.

in detail,

- KDE-Libs: Various run-time libraries
- KDE-Base: The base components
- KDE-Graphics: Graphics applications such as KDVI, KGhostview, KPaint, KFax ...
- KDE-Utilities: KEdit, KCalc, KNotes, ...
- KDE-Multimedia: KMidi, KSCD, Kaiman ...
- KDE-Games: KAsteroids, KPat, KTetris
- KDE-Admin: Various tools to aid system administration
- KDE-Network: Kppp, KNode, KMail ...

What Qt/KDE is

KDE was developed from zero into a full-featured desktop in about two years. This was possible only by using a pre-built graphical toolkit called Qt, from Troll Tech in Oslo, Norway.

KDE is a powerful Open Source graphical desktop environment for Unix workstations. It combines ease of use, contemporary functionality, and outstanding graphical design with the technological superiority of the Unix operating system.

KDE is an Internet project that is truly open in every sense. Development takes place on the Internet and is discussed on the KDE mailing lists, USENET news groups, and IRC channels to everyone is invited and welcomed.

KDE is a mature desktop suite providing a solid basis to an ever growing number of applications for Unix workstations. KDE has developed a high quality development framework for Unix, which allows for the rapid and efficient creation of applications.

By using Qt, KDE developers were able to focus on design issues for the desktop and immediately begin coding, rather than spending additional months or years creating a toolkit to standardize the interface.

Since the Qt Unix X11 version was recently re-released by Troll-Tech under the GNU General Public License, KDE is gaining increasing acceptance in the Linux community.

Both Qt and KDE itself are written in C++. You can also write KDE applications in Python, a high-level object-oriented scripting language, in Perl or in other languages as well.

About Qt

Qt is a cross-platform graphical toolkit that forms the basis of the KDE API. Any application written for KDE can use Qt functionality. The result is that a complete KDE application can be created in only a few days. It is widely regarded as being one of the fastest and most advanced GUI toolkits available today. Qt has excellent documentation: around 500 pages of postscript and fully cross-referenced online html documentation. Qt is very fast and compact because it is based directly on Xlib and uses neither Motif nor X Intrinsics. Qt's widgets (user interface objects) emulate the Motif look and feel, with slight improvements.

Qt is supported on the following platforms:

- **MS/Windows** - 95, 98, NT 4.0, ME, and 2000

- **Unix/X11** - Linux, Sun Solaris, HP-UX, Digital Unix, IBM AIX, SGI IRIX and a wide range of others
- **Macintosh** - Mac OS X
- **Embedded** - Linux platforms with framebuffer support.

The Qt/Desktop product family consists of the following products, which developers can use to target any platform:

- **Qt/Windows** is designed for MS Windows 95/98/ME, NT4, 2000 and XP.
- **Qt/X11** is designed for Linux, Solaris, HP-UX, Irix, AIX, and many other Unix variants. This is the de facto standard C++ toolkit for GUI applications on Linux.
- **Qt/Mac** is designed for Apple Mac OS X.
- **Qt/Embedded** is a version of Qt designed for resource-constrained embedded systems. It provides full GUI functionality without requiring X11 or Motif on the target system. This substantially reduces the memory and CPU demands of the embedded software.
- **Qtopia** is a window environment and application suite designed for PDAs, palmtop computers, internet appliances, and similar devices. It is fully based on Qt/Embedded.

Qt provides a platform-independent API to all central platform functionality: GUI, database access, networking, file handling, etc. The Qt library encapsulates the different APIs of different operating systems, providing the application programmer with a single, common API for all operating systems. The native C APIs are encapsulated in a set of well-designed, fully object-oriented C++ classes.

Here are some of the most important technical features of Qt:

Object Orientation

Qt has a modular design and a strong focus on reusable software components. A widget does not need to know its context and communicates with the outside world through signals and slots. All Qt widgets can be specialized through inheritance.

Component Support

Qt provides a **signals/slots** concept that is a type-safe alternative to callbacks and at the same time allow objects to cooperate without any knowledge of each other. This makes Qt very suitable for true component programming.

Superior On-Line Documentation

Qt includes on-line reference documentation in heavily cross-referenced HTML, Unix man-page and Postscript formats.

Easy To Customize and Ease of Use

Qt provides easily customizable widgets. It allows easy modification of the behavior of existing widgets and also allows easy creation of new ones.

Portability

Qt is a multi-platform GUI toolkit. It is fully portable and it includes a set of classes meant to protect the programmer from OS-dependent details in file handling, time/date handling, etc. If your main platform is MS Windows, you could probably use the standard library, the Microsoft Foundation Classes (MFC), quite happily. However, then you would miss millions of Unix users around the world. On the other hand, if your main platform is Unix, you could use any other toolkit, such as Gtk+ or XForms. But then, you would miss millions (if not billions) of MS Windows users. Therefore, the best choice would be a GUI Toolkit that's available for both MS Windows and Unix, Qt.

Rich API

Qt offers the functionality you need to create professional applications. There are around 250 C++ classes in the Qt API. Most of these classes are GUI specific; however, Qt also provides template-based **collections**, **serialization**, **file** and a general **I/O device**, **directory management**, **date/time** classes, **regular expression parsing** and more.

Full Widget Set

The basic building block in Qt programming is the **widget**. Qt contains ready-to-use widgets for creating professional-looking user interfaces. Qt introduces an innovative alternative for inter-object communication, called 'signals and slots' that replaces the old and unsafe callback technique. It also provides a conventional events model for handling mouse clicks, key presses, etc.

High-Performance Implementation

The Qt library is optimized for fast execution and conservative use of memory. Qt is able to perform many general tasks, for instance graphics rendering, much faster than platform-dependent code normally does.

GUI Emulation, Customizable Look and Feel

Qt emulates native window system widgets. Qt supports "themes", so Qt-based applications can change (even at runtime) between Motif look and feel, Windows look and feel, and any number of custom-made look and feel themes. This works independently of whether the application happens to be running under X Windows or Microsoft Windows.

Advanced Drawing Operations

Qt's drawing engine, the QPainter class, renders graphics on any paint device such as widgets, pixmaps (off-screen image), pictures (meta-file), and printer (Postscript under Unix). The code is portable on all four types of devices.

2D and 3D Graphics Support

Qt has excellent support for 2D and 3D graphics. It is the de facto standard GUI toolkit for platform-independent OpenGL programming.

Database Support

Qt makes it possible to create platform-independent database applications using standard databases. Qt includes native drivers for Oracle, Microsoft SQL Server, Sybase Adaptive Server, PostgreSQL, MySQL and ODBC-compliant databases. Qt's database functionality is fully integrated with Qt Designer, which makes possible the live preview of database data. Qt includes database-specific widgets, and any built-in or custom widget can be made data aware.

Internationalization

Qt uses Unicode throughout and has considerable support for internationalization. It includes Qt Linguist and other tools to support translators. Applications can easily use and mix text in different languages that are supported by Unicode.

Extensibility

QT applications can have their functionality extended by plug-ins and dynamic libraries. Plug-ins provide additional codecs, database drivers, image formats, styles, and widgets. Libraries can offer an unlimited range of functionality.

GUI Applications

Building modern GUI applications with Qt is fast and simple, and can be achieved by hand coding or by using **Qt Designer**, Qt's visual design tool.

Qt provides a vast collection of classes and functions that are very handy for creating modern GUI applications.

Qt Classes and Functions

Qt classes and functions can be used to create both 'main window' style applications with a menu bar, toolbars and status bar surrounding a central area, and dialog style applications that use buttons and possibly tabs to present options and information. Qt supports both SDI (single document interface) and MDI (multiple document interface). Qt also supports drag and drop and the clipboard. Tool bars can be moved around within the toolbar area (called the 'dock area'), dragged to other dock areas, or floated as tool palettes. This functionality is built in and requires no additional code, although programmers can apply constraints to toolbar behavior if they wish.

Qt simplifies programming. For example, if a menu option, a toolbar button and a keyboard accelerator all perform the same action, the action need only be coded once.

Qt also provides message boxes and a full set of standard dialogs to make it easy for applications to ask the user questions, and to get the user to choose files, folders, fonts and colors. In practice, a one-line statement using one of Qt's static convenience functions is all that is necessary to present a message box or a standard dialog.

Qt can platform-independently store application settings, such as user preferences, most recently used files, window and toolbar positions and sizes, etc.

Qt Designer

Qt Designer can be used purely as a design tool, or it can be used to create entire applications with its built-in C++ code editor. It also includes a variety of domain-specific classes. Local and remote file handling using standard protocols are provided by Qt's input/output and networking classes.

Designing a form with Qt Designer is a simple process. Developers click a toolbar button representing the widget they want, then click on a form to place the widget. The widget's properties can then be changed using the property editor. The precise positions and sizes of the widgets do not matter. Developers select widgets and apply layouts to them. For example, some button widgets could be selected and laid out side-by-side by choosing the 'lay out horizontally' option. This approach makes design very fast, and the finished forms will scale properly to fit whatever window size the end-user prefers.

Qt Designer eliminates the time-consuming compile, link and run cycle for user interface design. This makes it easy to correct or change designs. Qt Designer's preview options let developers see their forms in any style, for example, a Windows developer can preview a form in Motif style. Qt Designer provides live preview and editing of database data through its tight integration with Qt's database classes.

Developers can create both 'dialog' style applications and 'main window' style applications with menus, toolbars, balloon help, etc. Several form templates are supplied, and developers can create their own templates to ensure consistency across an application or family of applications. Qt Designer uses wizards to make creating toolbars, menus and database applications as fast and easy as possible. Programmers can create their own custom widgets that can easily be integrated with Qt Designer.

Qt Designer supports a project-based approach to application development. A project is represented by a .pro file, which qmake can use to generate Makefiles. Developers create a new project and then add forms and source files as required. Developers can completely separate the user interface from the underlying functionality by subclassing, or they can keep their source code and forms together by editing the form's source directly in Qt Designer.

Icons and other images used in the application are automatically shared across all forms in a project to reduce executable size and speed up loading.

Form designs are stored in XML format in .ui files and converted into C++ header and source files by the uic (User Interface Compiler). The qmake build tool automatically includes build rules for uic in the Makefiles it generates, so developers do not need to invoke uic themselves.

Usually forms are compiled into the executable, but in some situations customers need to modify the appearance of an application without accessing the source code. Qt supports 'dynamic dialogs': .ui files that can be loaded at run-time and dynamically converted into fully functional forms. Companies can supply application executables along with the customer-modifiable forms in .ui format, and the customer can use Qt Designer to customize the appearance of the application's forms.

Generating C++ Source Code From The Qt Designer File

Qt Designer saves files in a .ui format. To convert such files into C++ format, the user interface compiler or uic is used. Assuming that uic is in your search path, you can generate a header file as follows:

```
uic -o filename.h filename.ui
```

The option `-o` tells `uic` how to name the header file, and `filename.ui` is the name of the file you saved with Qt Designer.

To create the implementation file, you tell `uic` that you want to create an implementation file instead of a header file by specifying the option `-i` and passing the name of the already generated header file:

```
uic -i filename.h -o filename.cpp filename.ui
```

As before, the option `-o` determines the name of the generated file.

To start building the program on a Unix system with the `g++` compiler, the command lines would be:

```
moc -o moc_filename.cpp filename.h  
g++ -I$QTDIR/include -o executableName filename.cpp main.cpp \  
moc_filename.cpp -L$QTDIR/lib -lqt
```

and on a Windows system with the Microsoft Visual C++ compiler:

```
moc -o moc_filename.cpp filename.h  
cl -c -nologo -I%QTDIR%/include -Fofilename.obj filename.cpp  
cl -c -nologo -I%QTDIR%/include -Fomain.obj filename.cpp  
cl -c -nologo -I%QTDIR%/include -Fomoc_filename.obj moc_filename.cpp  
link /NOLOGO /SUBSYSTEM:windows /OUT:filename filename.obj main.obj \  
moc_filename.obj %QTDIR%/lib/qt.lib kernel32.lib user32.lib gdi32.lib \  
comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib  
imm32.lib winmm.lib wsock32.lib
```

If you use a different compiler, you might have to change the compiler command and some of the options. If you do not know what the line starting with `moc` is for, see the Qt Tutorial or Programming with Qt.

Integrating Qt Designer Files Into Your Project

Creating forms with Qt Designer gets you only halfway to completing your application. Somehow you have to integrate the created forms into your application; and not only into your application but also into your automatic build process.

To start discussing how you can achieve this, let's review the steps needed to create a form with Qt Designer:

1. Design the form interactively in Qt Designer.
2. Save the form as an XML .ui file.
3. Run uic twice from the command line on the .ui file to generate the header and implementation files. If necessary (which is usually the case), create a subclass of the generated class and implement the dialog functionality there. You can use uic to create a skeleton for this class.
4. Compile the generated source file as well as the implementation file of the subclass and all other files belonging to the application.
5. Run moc on the generated header file and the subclass header file.
6. Compile the moc-generated files.

As you can see, there are quite a number of steps involved. From the command line, the following could be the steps needed once you have saved the .ui file (let's call it myform.ui) in Qt Designer:

```
uic -o myform.h myform.ui # generate header file

uic -o myform.cpp -impl myform.h myform.ui # generate implementation file

uic -o myformimpl.h -subdecl MyFormImpl myform.h myform.ui # generate subclass
header file
```

```
uic -o myformimpl.cpp -subimpl MyFormImpl myformimpl.h myform.ui # generate
subclass implementation file

edit myformimpl.h # edit subclass header file

edit myformimpl.cpp # edit subclass implementation file

moc -o moc_myform.cpp myform.h # generate moc code for base class

moc -o moc_myformimpl.cpp myformimpl.h # generate moc code for subclass

c++ -c myform.cpp -I$(QTDIR)/include # compile base class, use similar command
on Windows

c++ -c moc_myform.cpp -I$(QTDIR)/include # compile base class moc code, use
similar command on Windows

c++ -c myformimpl.cpp -I$(QTDIR)/include # compile subclass, use similar
command on Windows

c++ -c moc_myformimpl.cpp -I$(QTDIR)/include # compile subclass moc code, use
similar command on Windows

c++ -o myform myform.o moc_myform.o myformimpl.o moc_myformimpl.o -
L$(QTDIR)/lib -lqt # link everything together, use similar command on Windows
```

But this is not enough. If you need to change anything in your dialog design, you go back to Qt Designer, make your changes interactively, save the ui file again, and go through most of the previous steps again. All this cries for automation, but before you learn how to automate these steps, we need to talk a bit about the generation of subclass files.

For two reasons, the generation of subclass files falls a bit out of the ordinary build process: First, you are required to edit the generated files, as hinted at earlier; if you do not change these files and do not add your own implementation of certain methods, there is no point in having a subclass at all.

Second, and this is a consequence of the first point, you do not regenerate the files over and over again after changes as you do with the other files. This is because you add changes to these files by hand, and if you overwrote the files, your changes would be lost.

As a result, you usually use `uic` only once per form with the options `-subdecl` and `-subimpl`.

If you later add a new slot with Qt Designer, you cannot easily use `uic` to add this slot to the subclass, because your changes would be overwritten. Of course, you can always copy your changed versions of the files to files with different names, let `uic` generate the new skeletons, and manually copy your changes back (e.g., in an editor), but this is usually more work and trouble than just adding the individual slots by hand.

But even after ruling out these steps from the ordinary build process, there are an awful lot of steps left. How you integrate these into your build process largely depends on how your build process is organized, which tools you use, etc. If you use handwritten makefiles (a solution we do not recommend) and your project is very small, you could just add the build steps by hand. For the aforementioned project, this could look like the following in Unix make syntax (Windows make syntax will differ slightly but not much):

```
all:
myform

myform: myform.o moc_myform.o myformimpl.o moc_myformimpl.o
c++ -o $@ $+ -L$(QTDIR)/lib -lqt
```

```

myform.o: myform.cpp myform.h
c++ -c $$< -I$(QTDIR)/include

myformimpl.o: myformimpl.cpp myformimpl.h myform.h
c++ -c $$< -I$(QTDIR)/include

moc_myform.o: moc_myform.cpp myform.h
c++ -c $$< -I$(QTDIR)/include

moc_myformimpl.o: moc_myformimpl.cpp myformimpl.h myform.h
c++ -c $$< -I$(QTDIR)/include

moc_myform.cpp: myform.h
moc -o $$@ $$<

moc_myformimpl.cpp: myformimpl.h
moc -o $$@ $$<

myform.h: myform.ui
uic -o $$@ $$<

myform.cpp: myform.ui myform.h
uic -o $$@ -impl myform.h $$<

```

Note that there is no rule here for generating `myformimpl.cpp` and `myformimpl.h`. As mentioned earlier, you generate them once by hand from the command line and then edit them with your text editor without regenerating them. Of course, the previous makefile with every

relation explicitly spelled out does not hold for anything but the smallest project. You can achieve a little bit more generality by using suffix rules as follows:

```
.SUFFIXES:

# define SUFFIXES

.SUFFIXES: .h .cpp .ui

# create a .h file from a .ui file

.h.ui; uic -o $@ $<

# create a .cpp file from a .ui file

.cpp.ui; uic -o $@ -impl $*.h $<

# clear SUFFIXES list
```

But depending on your project, this might not be enough sophistication.

Using tmake For Generating And Building Qt Designer Files

We heartily recommend that you use a good system for automatically generating makefiles from a project description. There are several such systems available for both Unix and Windows, but for projects involving Qt and the Qt Designer, we suggest you give tmake a try. tmake is free software that can be downloaded free of charge from <http://ftp.trolltech.com/pub/freebies/tmake/>. Just make sure that you get Version 1.5 or higher. On Unix systems, tmake requires Perl to be installed, which is usually the case these days.

If you do not know tmake yet, we suggest that you start reading the included documentation; tmake is really easy to use. We will only cover tmake's special features for use with Qt Designer here.

Actually, once you know how to use tmake for ordinary Qt projects, using it for projects that involve Qt Designer files is surprisingly easy. All you need to do is list your .ui files in the new INTERFACES tag in your tmake project file. For the aforementioned project, the following could be enough:

```
. INTERFACES = myform.ui
```

When you then generate the makefile from the tmake project file, it will contain all the clauses mentioned earlier. The only thing you need to be sure of is that uic is in your path, just as moc and your C++ compiler are.

Again, this method does not do anything with generated and edited subclass files; as earlier, you just generate them by hand and then edit them as if they were ordinary source or header files (which in fact they are).

Widgets

This part doesn't cover all widgets but it does tackle some of the more common widgets that are used.

Labels

Labels are usually used to present brief information or instructions about the widgets in a program. For example, you could use a label to define what should be entered in a text box. In Qt, you implement labels by using the `QLabel` class.

The `QLabel` class is used for displaying simple text. It's very basic yet very useful widget. Listed below is a simple example.

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

class MyMainWindow : public QWidget
{
public:
    MyMainWindow();
private:
    QLabel *text;
};

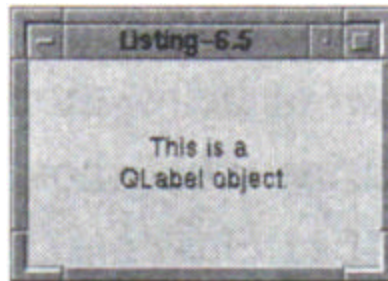
MyMainWindow::MyMainWindow()
{
    setGeometry(100, 100, 170, 100);
    text = new QLabel(this);
    text->setGeometry(10,10,150,80);
    text->setText("This is a \nQLabel object.");
    text->setAlignment(AlignHCenter | AlignVCenter);
}
```

```

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

Shown below is how the program created will look like.



There is also a standard way of setting alignments in Qt. `AlignHCenter` and `AlignVCenter` are Qt definitions. Listed below are the definitions for alignment settings.

Function	Definition
<code>AlignTop</code>	Text will be added to the top of the QLabel object.
<code>AlignBottom</code>	Text will be added to the bottom of the QLabel object.
<code>AlignLeft</code>	Text will be added along the left side of the QLabel object.
<code>AlignRight</code>	Text will be added along the right side of the QLabel object.
<code>AlignHCenter</code>	Text will be added at the horizontal center of the QLabel object.
<code>AlignVCenter</code>	Text will be added at the vertical center of the QLabel object.
<code>WordBreak</code>	If this function is set, automatic word breaking is set.
<code>ExpandTabs</code>	This function makes QLabel expand the tabulators.

The arguments to the `QLabel::setAlignment()` are separated by the “or” (`|`) operator. This method is also used when you want to add more arguments (just separate them with the `|`) operators.

Scrollbars

Scrollbars enable the user to create applications that are actually too big to fit the screen. In many cases, scrollbars are required for your application to run on all desktops. By using scrollbars, developers make programs that are easier for people to use (since the users could see all the widgets, no matter what resolution they use for their desktops).

Qt provides two classes that can be used for scrollbars. One is the `QScrollBar` class and the `QScrollView` class. The `QScrollBar` class is not suggested to be used because it involves unnecessary work – work that is done automatically by the `QScrollView` class. The `QScrollView` provides scroll-on-demand – that is, scrollbars are added when needed.

Below is an example.

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

class MyMainWindow : public QScrollView
{
public:
    MyMainWindow();
private:
    QPushButton *b1;
};

MyMainWindow::MyMainWindow()
{
```

```

//set the geometry for the scrollview
setGeometry(100,100,200,100);

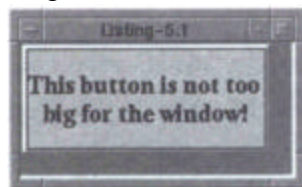
b1 = new QPushButton("This button is not too \n big for the window!",
this);
b1->setGeometry(10,10,180,180);
//the QScrollView::addChild function for each of its child widgets
addChild(b1);
}

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

Shown below is how the program will look like...

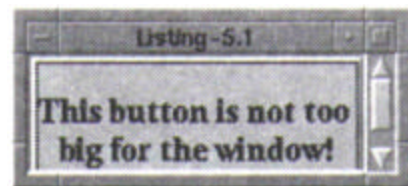
Original Form



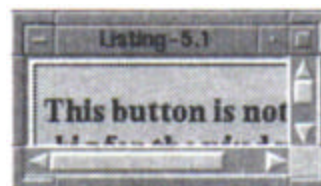
If window's width is too small to show the whole button



If window's height is too small to show the whole button



If both the window's width and height are too small to show the button



Text-Entry Fields

Text-entry fields are widgets in which you can enter text. Qt provides two classes for creating text-entry fields: `QLineEdit` and `QMultiLineEdit`. The `QLineEdit` creates a single line text-entry field and this will be the one to be shown below.

```
#include <qapplication.h>
#include <qwidget.h>
#include <qlineedit.h>

class MyMainWindow : public QWidget
{
public:
    MyMainWindow();
private:
    QLineEdit *edit;
};

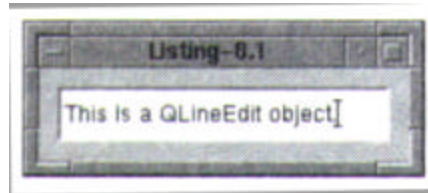
MyMainWindow::MyMainWindow()
{
    setGeometry(100, 100, 200, 50);

    edit = new QLineEdit(this);
    edit->setGeometry(10,10,180,30);
}

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}
```

One can set the text of the `QLineEdit` object manually by using the `QLineEdit::setText()` function. One can retrieve the text that's currently written in the `QLineEdit` object by making a call to the `QLineEdit::text()` function.

Shown below is how the program looks like.



Buttons

Buttons are probably the most common used GUI object. Qt provides three types of buttons:

- Push Buttons
- Radio Buttons
- Check Buttons

Push buttons are used to make certain events occur, whereas radio and check buttons are used to make some kind of selection.

Push Buttons

The QPushButton function is probably the Qt class Qt users will mostly use when creating Qt applications. As stated, push buttons are created by the QPushButton class.

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

class MyMainWindow : public QWidget
{
public:
    MyMainWindow();
private:
    QPushButton *b1;
};

MyMainWindow::MyMainWindow()
{
    setGeometry(100, 100, 200, 100);
    b1 = new QPushButton("This is a push button!", this);
    b1->setGeometry(10,10,180,80)
```

```

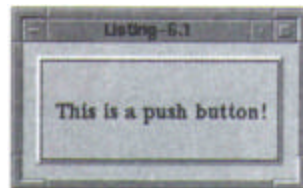
}

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

When the user wants to connect a push button to a certain event, the user connects the `QPushButton::clicked()` signal to a slot. More about signals and slots later in the paper.

Shown below is how the program looks like.



Radio Buttons

Radio buttons should be used when the developer wants one (and only one) selection out of several choices. Developers can use the `QButtonGroup` and `QRadioButton` classes to create a group of radio buttons.

```

#include <qapplication.h>
#include <qwidget.h>
#include <qbuttongroup.h>
#include <qradiobutton.h>

class MyMainWindow : public QWidget

```

```

{
public:
    MyMainWindow();
private:
    QButtonGroup * group;
    QRadioButton *b1;
    QRadioButton *b2;
    QRadioButton *b3;
};

MyMainWindow::MyMainWindow()
{
    setGeometry(100,100,150,140);

    group = new QButtonGroup("Options", this);
    group->setGeometry(10,10,130,120);

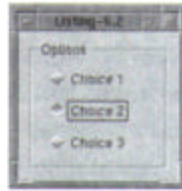
    b1 = new QRadioButton("Choice 1", group);
    b1->move(20,20);
    b2 = new QRadioButton("Choice 2", group);
    b2->move(20,50);
    b3 = new QRadioButton("Choice 3", group);
    b3->move(20,80);

    group->insert(b1);
    group->insert(b2);
    group->insert(b3);
}

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

Shown below is how the program looks like.



Checkbox

Check buttons should be used when you want the user to be able to choose more than one selection out of multiple choices. The user can create a group of check buttons using the `QButtonGroup` and `QCheckBox` classes.

```
#include <qapplication.h>
#include <qwidget.h>
#include <qbuttongroup.h>
#include <qcheckbox.h>

class MyMainWindow : public QWidget
{
public:
    MyMainWindow();
private:
    QButtonGroup * group;
    QCheckBox *b1;
    QCheckBox *b2;
    QCheckBox *b3;
};

MyMainWindow::MyMainWindow()
{
    setGeometry(100,100,150,140);

    group = new QButtonGroup("Options", this);
    group->setGeometry(10,10,130,120);

    b1 = new QCheckBox("Choice 1", group);
    b1->move(20,20);
    b2 = new QCheckBox("Choice 2", group);
    b2->move(20,50);
    b3 = new QCheckBox("Choice 3", group);
    b3->move(20,80);
```

```

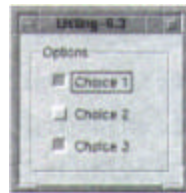
        group->insert(b1);
        group->insert(b2);
        group->insert(b3);
    }

void main(int argc, char **argv)

{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

Shown below is how the program looks like.



Layout Managers

Layout managers are a set of Qt classes that control how and when the main window is resized. By registering the widgets with a layout manager and not giving the widgets absolute positions, the layout manager will resize or move the child widgets as needed when the user resizes the parent window. Also, layout managers are a great help when the user needs to place many widgets.

Listed below are Qt's layout managers and their description.

Layout Manager	Description
QLayout	This is the layout manager class. All other layout managers inherit this class.
QGridLayout	Used when the user wants to arrange the widgets in a grid.
QBoxLayout	Base layout class for QGroupBoxLayout and QVBoxLayout. Should only be used when the user can't decide whether the widgets should be placed in rows or columns.
QHBoxLayout	Used when you want the user wants to arrange widgets in a row.
QVBoxLayout	Used when the user wants to arrange the widgets in a column.

Free

In this example, widgets are placed by hand and no layout manager is used.

```

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

class MyWidget:public QWidget
{
public:
    MyWidget();
};

MyWidget::MyWidget()
{
    setMinimumSize(200,200);

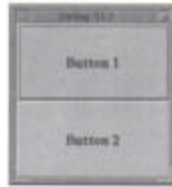
    QPushButton *b1 = new QPushButton("Button1", this);
    b1->setGeometry(0,0,200,100);
    QPushButton *b2 = new QPushButton("Button2", this);
    b2->setGeometry(0,100,200,100);
}

int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyWidget w;
    w->setGeometry(100,100,200,200)
    a.setMainWidget(&w);
    w.show();
    return a.exec();
}

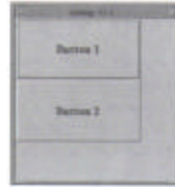
```

Shown below (a) is how this program looks like.

(a)



(b)



(b) shows what happens when you resize a window that doesn't use layout managers. Widgets don't resize and/or move suit to the main window.

QVBoxLayout

As stated in the definition, the QVBoxLayout places the widgets in a column.

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

class MyWidget:public QWidget
{
public:
    MyWidget();
};

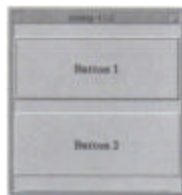
MyWidget::MyWidget()
{
    setMinimumSize(200,200);
    QPushButton *b1 = new QPushButton("Button1", this);
    b1->setMinimumSize(200,100);
    QPushButton *b2 = new QPushButton("Button2", this);
    b2->setMinimumSize(200,100);
    QVBoxLayout *vbox = new QVBoxLayout(this);
    vbox->addWidget(b1);
    vbox->addWidget(b2);
}
```

```

int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyWidget w;
    w->setGeometry(100,100,200,200)
    a.setMainWidget(&w);
    w.show();
    return a.exec();
}

```

Shown below is how the program looks like.



With this layout, the problem of widgets not adjusting to the size of the main window is solved.

QGridLayout

If you want to layout your widgets in a grid, the user should use the QGridLayout class.

```

#include <qapplication.h>
#include <qwidget.h>
#include <qpushbutton.h>
#include <qlayout.h>

class MyWidget:public QWidget
{
public:
    MyWidget();
};

MyWidget::MyWidget()
{
    setMinimumSize(200,200);
}

```

```

QPushButton *b1 = new QPushButton("Button1", this);
b1->setMinimumSize(200,100);

QPushButton *b2 = new QPushButton("Button2", this);
b2->setMinimumSize(200,100);

QPushButton *b3 = new QPushButton("Button3", this);
b3->setMinimumSize(200,100);

QPushButton *b4 = new QPushButton("Button4", this);
b4->setMinimumSize(200,100);

QPushButton *b5 = new QPushButton("Button5", this);
b5->setMinimumSize(200,100);

QPushButton *b6 = new QPushButton("Button6", this);
b6->setMinimumSize(200,100);

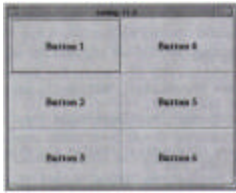
QGridLayout *grid = new QGridLayout(this,3,2);
grid->addWidget(b1,0,0);
grid->addWidget(b2,1,0);
grid->addWidget(b3,2,0);
grid->addWidget(b4,0,1);
grid->addWidget(b5,1,1);
grid->addWidget(b6,2,1);
}

int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyWidget w;
    w->setGeometry(100,100,200,200)
    a.setMainWidget(&w);
    w.show();
    return a.exec();
}

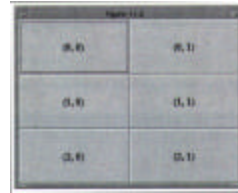
```

Shown (a) at the next page is how the program looks like.

(a)



(b)



(b) shows how the cells in the grid are defined.

Event Handling in Qt: Slots and Signals

The technology behind this slots and signals differ a bit from more traditional callback functions. After all, slots and signals technology was developed independently by Troll Tech – its not a C++ feature.

Understanding Slots

Slots are actually normal member functions (functions that are members of a class). However, they have some special features added to them that make it possible to connect signals to them. Every time a signal connected to a particular slot is emitted, the slot (function) is executed.

Many Qt classes already include a few predefined slots which will be discussed and shown later in the paper.

So, slots are actually normal functions. Therefore, they can be called just like other functions.

Understanding Signals

Signals are also member functions. However, they're implemented in a slightly different way.

When something happens inside an object (internal state changes), it can send out a signal (although it doesn't have to). If this signal is connected to a slot, that slot

(function) is then executed. Users can connect multiple slots to the same signals; the slots are then executed one by one, in an arbitrary order.

So, signals are a special type of function. They're defined to be emitted when certain events occur. Then, any connected slots are executed.

Example 1: Using QPushButton

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

class MyMainWindow:public QWidget
{
public:
    MyMainWindow();
private:
    QPushButton *b1;
    QLabel *label;
};

MyMainWindow::MyMainWindow()
{
    setGeometry(100,100,200,170);

    b1 = new QPushButton("Quit", this);
    b1->setGeometry(20,20,160,80);

    label = new QLabel(this);
    label->setGeometry(10,110,180,50);
    label->setText("If you click the button above, the whole program will
exit");
    label->setAlignment(AlignCenter);

    //The following line makes the program exit when the button b1 is clicked
    connect(b1, SIGNAL(clicked()), qApp, SLOT(quit()));
}
```

```

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

Here, the button `b1`'s signal `clicked()` is connected to the slot `quit()` of `qApp`. Now, when the button is clicked, the `QPushButton::clicked()` signal will be emitted, the `quit()` slot of `qApp` will be executed, and the program will exit.

Note: `qApp` is a built-in pointer in Qt. It is designed to always point to the `QApplication` object of the program (`a`, in this case). It works just like the *this* pointer: It points to an object that's not yet created.

Example 2: Using `QPushButton` and `QLineEdit`

```

#include <qapplication.h>
#include <qwidget.h>
#include <qpushbutton.h>
#include <qlineedit.h>
#include <qstring.h>

class MyMainWindow:public QWidget
{
public:
    MyMainWindow();
private:
    QPushButton *b1;
    QPushButton *b2;
    QPushButton *b3;
    QLineEdit *ledit;
};

MyMainWindow::MyMainWindow()
{
    setGeometry(100,100,300,200);
}

```

```

        b1 = new QPushButton("Click here to mark the text", this);
        b1->setGeometry(10,10,280,40);
        b2 = new QPushButton("Click here to unmark the text", this);
        b2->setGeometry(10,60,280,40);
        b3 = new QPushButton("Click here to remove the text", this);
        b3->setGeometry(10,110,280,40);
        ledit = new QLineEdit("This is a line of text", this);
        ledit->setGeometry(10,160,280,30);

        //The following three lines connects each button to a predefined slot on the
        // ledit object.
        connect(b1, SIGNAL(clicked()), ledit, SLOT(selectAll()));
        connect(b2, SIGNAL(clicked()), ledit, SLOT(deselect()));
        connect(b3, SIGNAL(clicked()), ledit, SLOT(clear()));
    }

void main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    a.setMainWidget(&w);
    w.show();
    a.exec();
}

```

This example uses the clicked() signal of QPushButton and connects it to three different slots included in the QLineEdit class. By doing this, the user can control the text in the QLineEdit object by clicking the buttons.

Interesting Features of Slots and Signals

Connecting Signals to Signals

Sometimes, it may seem necessary to connect a signal to another signal.

Although this may sound strange, it's completely possible to do. For example, the

user might want a certain event to occur when a button is clicked, it's a good idea to connect the button's clicked() signal to the signal that will start the event.

If the user wants to declare a signal to be connected to another signal, the third argument of the connect function must be changed, as shown in the following:

```
connect(button, SIGNAL(clicked()),this, SIGNAL(anothersignal()));
```

Now, every time the clicked() signal of button is emitted, it will appear that the another signal() signal of the currently-defined class is emitted too. If the user then connects the othersignal() signal to some slot, that slot (function) will be executed every time clicked() is emitted.

Disconnecting Slots and Signals

It is possible that the user may find it necessary to disconnect a signal from a slot (or a signal from a signal) at some point. This is good to do when the user wants to disable a function in the program for a while. To accomplish this, the user uses the QObject::disconnect() function. It expects the exact same parameters as QObject::connect(). For example, if the user wants to disconnect a clicked() signal from quit(), the user should use the following:

```
disconnect(button, SIGNAL(clicked()), qApp, SLOT(quit()));
```

If the user implements this line in (for example) the example that contains a Quit button, anyone who uses the program won't be able to terminate the program through the button anymore.

Conclusion

With so many features and really the ease of use of designing GUI's with Qt, we would easily suggest using the program. For Linux users, a drawback would be having to learn C++ and not just simply sticking to something Linux users are very knowledgeable about already, the C language. But having to learn C++ is not a really big obstacle to the benefits one will reap through its features (e.g. being able to also use it in Windows). So in the end, even if it takes time to learn a few things before designing GUI and making your programs in Qt/KDE, it surely is not a waste of time.

References

Research Material was acquired from the following sources:

Qt Website, <http://www.troll.no>

KDE Website, <http://www.kde.org>

Solin, Daniel, [Qt Programming](#).

Comments

Images have been resized.