Qt 4.0 Whitepaper

Trolltech
www.trolltech.com

Abstract

This whitepaper describes the Qt C++ framework. Qt supports the development of cross-platform GUI applications with its "write once, compile anywhere" approach. Using a single source tree and a simple recompilation, applications can be written for Windows 98 to XP, Mac OS X, Linux, Solaris, HP-UX, and many other versions of Unix with X11. Qt applications can also be compiled to run on embedded Linux platforms. Qt introduces a unique inter-object communication mechanism called "signals and slots." Qt has excellent cross-platform support for 2D and 3D graphics, internationalization, SQL, XML, as well as providing platform-specific extensions for specialized applications. Qt applications can be built visually using *Qt Designer*, a flexible user interface builder with support for IDE integration.

Contents

1. Introduction **3**
1.1. Executive Summary 3

2. Widgets **5**
2.1. Built-in Widgets 5
2.2. Custom Widgets 7

3. Signals and Slots **10**
3.1. A Signals and Slots Example 11
3.2. Meta-Object Compiler 12

4. GUI Applications **13**
4.1. Main Window Classes 14
4.1.1. The Main Window 14
4.1.2. Menus 14
4.1.3. Toolbars 15
4.1.4. Actions 15
4.1.5. Dock Windows 15
4.1.6. Dialogs 16
4.1.7. Interactive Help 17
4.1.8. Multiple Document Interface 18
4.2. Settings 18
4.3. Multithreading 19

5. Qt Designer **20**
5.1. Working with Qt Designer 20
5.2. Qt Assistant 20
5.3. GUI Application Example 22
5.4. Extending Qt Designer 25

6. 2D and 3D Graphics **26**
6.1. Painting 26
6.2. Images 27
6.3. Paint Devices 27
6.4. 3D Graphics 28

7. Item Views **29**
7.1. Standard Item Views 29
7.2. Qt's Model/View Framework 30

8. Text Handling **31**
8.1. Rich Text Editing 31
8.2. Rich Text Processing 32
8.3. Custom Text Layouts 32

9. Databases **33**
9.1. Executing SQL Commands 33
9.2. SQL Models 34

10. Internationalization **36**
10.1. Text Entry and Rendering . . . 36
10.2. Translating Applications . . . 37
10.3. Qt Linguist . . . 38

11. Layouts **39**
11.1. Built-in Layout Managers . . . 39
11.2. Nested Layouts . . . 40

12. Styles and Themes **42**
12.1. Built-in Styles . . . 42
12.2. Custom Styles . . . 43

13. Events **44**
13.1. Event Creation . . . 44
13.2. Event Delivery . . . 44

14. Input/Output and Networking **45**
14.1. Reading and Writing Files . . . 45
14.2. XML . . . 46
14.3. Inter-Process Communication . . . 46
14.4. Networking . . . 47

15. Collection Classes **49**
15.1. Containers . . . 49
15.2. Implicit Sharing . . . 49

16. Plugins and Dynamic Libraries **51**
16.1. Plugins . . . 51
16.2. Dynamic Libraries . . . 51

17. Building Qt Applications **52**
17.1. Qt's Build System . . . 52
17.2. Qt's Resource System . . . 53

18. Qt's Architecture **54**
18.1. X11 . . . 54
18.2. Microsoft Windows . . . 55
18.3. Mac OS X . . . 55

19. Platform Specific Extensions and Qt Solutions **56**
19.1. ActiveX Interoperability . . . 56
19.2. Qt Solutions . . . 57

20. The Qt Development Community **58**

1. Introduction

Qt is the de facto standard C++ framework for high performance cross-platform software development. In addition to an extensive C++ class library, Qt includes tools to make writing applications fast and straightforward. Qt's cross-platform capabilities and internationalization support ensure that Qt applications reach the widest possible market.

The Qt C++ framework has been at the heart of commercial applications since 1995. Qt is used by companies and organizations as diverse as Adobe®, Boeing®, IBM®, Motorola®, NASA, Skype®, and by numerous smaller companies and organizations. Qt 4 is designed to be easier to use than previous versions of Qt, while adding more powerful functionality. Qt's classes are fully featured and provide consistent interfaces to assist learning, reduce developer workload, and increase programmer productivity. Qt is, and always has been, fully object-oriented.

This whitepaper gives an overview of Qt's tools and functionality. Each section begins with a non-technical introduction before providing a more detailed description of relevant features. Links to online resources are also given for each subject area.

To evaluate Qt for 30 days, visit http://www.trolltech.com/.

1.1. Executive Summary

Qt includes a rich set of widgets ("controls" in Windows terminology) that provide standard GUI functionality (page 5). Qt introduces an innovative alternative for inter-object communication, called "signals and slots" (page 10), that replaces the old and unsafe callback technique used in many legacy frameworks. Qt also provides a conventional event model (page 44) for handling mouse clicks, key presses, and other user input. Qt's cross-platform GUI applications (page 13) can support all the user interface functionality required by modern applications, such as menus, context menus, drag and drop, and dockable toolbars.

Qt also includes *Qt Designer* (page 20), a tool for graphically designing user interfaces. *Qt Designer* supports Qt's powerful layout features (page 39) in addition to absolute positioning. *Qt Designer* can be used purely for GUI design, or to create entire applications with its support for integration with popular integrated development environments (IDEs).

Qt has excellent support for 2D and 3D graphics (page 26). Qt is the *de facto* standard GUI framework for platform-independent OpenGL® programming. Qt 4's painting system offers high quality rendering across all supported platforms.

Qt makes it possible to create platform-independent database applications using standard databases (page 33). Qt includes native drivers for Oracle®, Microsoft® SQL Server, Sybase® Adaptive Server, IBM DB2®, PostgreSQL™, MySQL®, Borland® Interbase, SQLite, and ODBC-compliant databases. Qt includes database-specific widgets, and any built-in or custom widget can be made data-aware.

Qt programs have native look and feel on all supported platforms using Qt's styles and themes support (page 42). From a single source tree, recompilation is all that is required to produce applications for Windows® 98 to XP®, Mac OS X®, Linux®, Solaris™, HP-UX™, and many other versions of Unix® with X11™. Qt applications can also be compiled to run

on Qtopia. Qt's `qmake`

2. Widgets

Qt provides a rich set of standard widgets that can be used to create graphical user interfaces for applications. Qt's widgets are flexible and can easily be subclassed to suit specialized requirements.

Widgets are visual elements that are combined to create user interfaces. Buttons, menus, scroll bars, message boxes, and application windows are all examples of widgets. Qt's widgets are not arbitrarily divided between "controls" and "containers"; all widgets can be used both as controls and as containers. Custom widgets can easily be created by subclassing existing Qt widgets, or created from scratch if necessary.

Standard widgets are provided by the **QWidget** class and its subclasses, and custom widgets can be created by subclassing them and reimplementing virtual functions.

A widget may contain any number of child widgets. Child widgets are shown within the parent widget's area. A widget with no parent is a top-level widget (a "window"), and usually has its own entry in the desktop environment's task bar. Qt imposes no arbitrary limitations on widgets. Any widget can be a top-level widget; any widget can be a child of any other widget. The position of child widgets within the parent's area can be set automatically using layout managers (page 39), or manually if preferred. When a parent widget is disabled, hidden, or deleted, the same action is recursively applied to all its child widgets.

Labels, message boxes, tooltips, and other textual widgets are not confined to using a single color, font, and language. Qt's text-rendering widgets can display multi-language rich text using a subset of HTML (see Text Entry and Rendering on page 36).

2.1. Built-in Widgets

The screenshots on the following page present a selection of Qt widgets used in various user interface components. The widgets were arranged using *Qt Designer* and rendered using the Plastique style, demonstrating Qt 4's standard look and feel on Linux.

The widgets shown include standard input widgets like **QLineEdit** for one-line text entry, **QCheckBox** for enabling and disabling simple independent settings, **QSpinBox** and **QSlider** for specifying quantities, **QRadioButton** for enabling and disabling exclusive settings, and **QComboBox**, which opens to present a menu of choices when clicked. Clickable buttons are provided by **QPushButton**.

Container widgets such as **QTabWidget** and **QGroupBox** are also shown. These widgets are managed specially in *Qt Designer* to allow designers to rapidly create new user interfaces while helping to keep them maintainable. More complex widgets such as **QScrollArea**, as shown in the "Create Poster" dialog (Figure 1), are often used more by developers than by user interface designers because they are often used to display specialized or dynamic content.

Qt provides many other widgets than those listed here. Many of the available widgets are shown with links to their class documentation in Qt's online Widget Gallery.

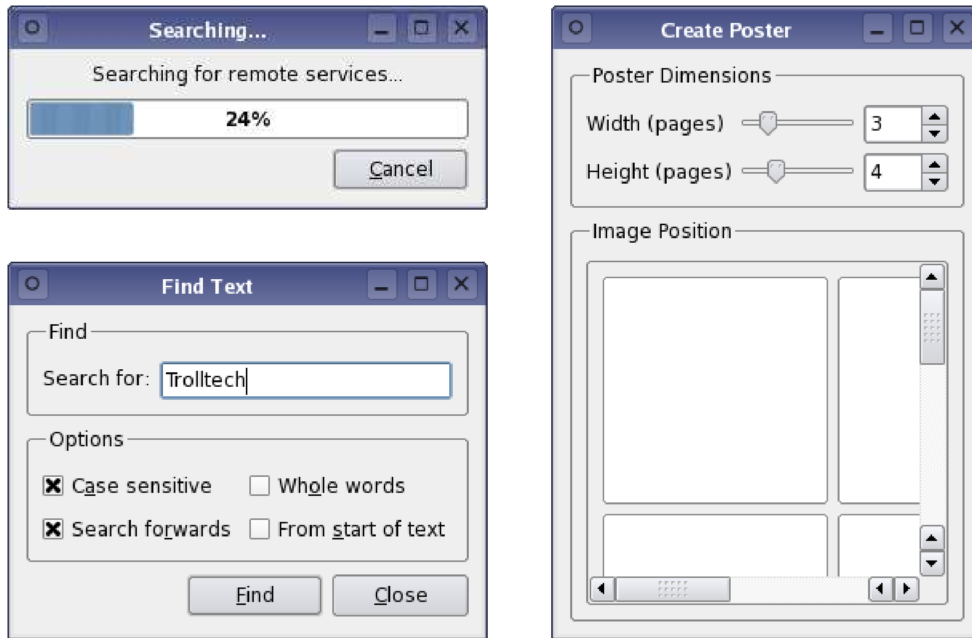

Figure 1: Dialogs created using a variety of different widgets.

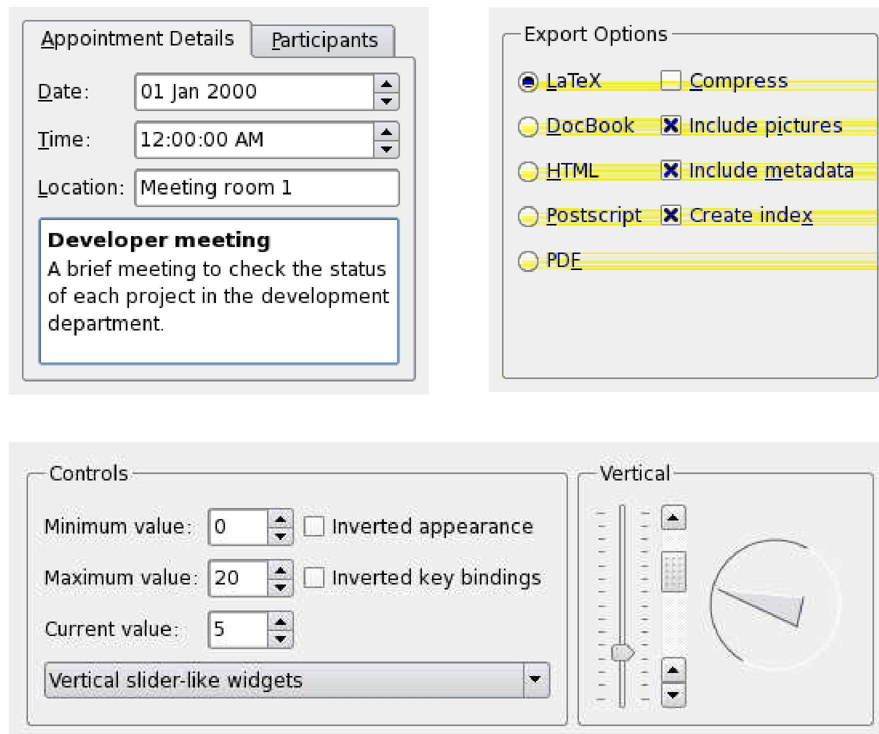

Figure 2: Qt provides a wide variety of standard widgets.

User interfaces can easily be written by hand. The following code could be used to create the options group box in the “Find Text” dialog (Figure 1):

```
QGroupBox *optionsGroupBox = new QGroupBox(tr("Options"));
QCheckBox *caseCheckBox = new QCheckBox(tr("C&ase sensitive"));
QCheckBox *directCheckBox = new QCheckBox(tr("Search fo&rwards"));
QCheckBox *wordsCheckBox = new QCheckBox(tr("Whole &words"));
QCheckBox *startCheckBox = new QCheckBox(tr("From &start of text"));

QGridLayout *optionsLayout = new QGridLayout;
optionsLayout->addWidget(caseCheckBox, 0, 0);
optionsLayout->addWidget(wordsCheckBox, 0, 1);
optionsLayout->addWidget(directCheckBox, 1, 0);
optionsLayout->addWidget(startCheckBox, 1, 1);
optionsGroupBox->setLayout(optionsLayout);
```

2.2. Custom Widgets

Developers can create their own widgets and dialogs by subclassing **QWidget** or one of its subclasses. To illustrate subclassing, we present the complete code for an analog clock widget from the Qt 4 examples directory that displays the current time and updates itself automatically.

The **AnalogClock** widget is defined in the `analogclock.h` file:

```
#include <QWidget>

class AnalogClock : public QWidget
{
    Q_OBJECT

public:
    AnalogClock(QWidget *parent = 0);

protected:
    void paintEvent(QPaintEvent *event);
};
```

The widget inherits the generic **QWidget** class and has a constructor typical of widget classes, with an optional `parent` parameter. The paintEvent() function is inherited from **QWidget** and is called whenever the widget needs to be updated.

The **AnalogClock** class is implemented in the `analogclock.cpp` file:

```
#include <QtGui>
#include "analogclock.h"

AnalogClock::AnalogClock(QWidget *parent)
    : QWidget(parent)
{
    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(update()));
    timer->start(1000);
    setWindowTitle(tr("Analog Clock"));
    resize(200, 200);
}
```

The constructor sets up a timer, gives the window a title, and ensures that it has a reasonable default size. The timer is configured to emit a signal every 1000 milliseconds. Before it is started, it is connected to the widget’s update() function using Qt’s signals and slots mechanism (page 10) to ensure that the clock is kept up to date.

The paintEvent() function simply redraws the entire widget each time it is called. It uses Qt's painting system (page 26) to render a clock face with hour and minute hands:

```
void AnalogClock::paintEvent(QPaintEvent *)
{
    static const QPoint hourHand[3] = {
        QPoint(7, 8),
        QPoint(-7, 8),
        QPoint(0, -40)
    };
    static const QPoint minuteHand[3] = {
        QPoint(7, 8),
        QPoint(-7, 8),
        QPoint(0, -70)
    };

    QColor hourColor(127, 0, 127);
    QColor minuteColor(0, 127, 127);

    int side = qMin(width(), height());
    QTime time = QTime::currentTime();
```

The first part of the function sets up the information about some simple primitives and their colors, using the shortest side of the widget as the size of the clock.

The rest of the function performs the task of painting the clock face in the center of the widget and drawing the hands in the correct positions, using anti-aliasing if available:

```
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
    painter.translate(width() / 2, height() / 2);
    painter.scale(side / 200.0, side / 200.0);

    painter.setPen(Qt::NoPen);
    painter.setBrush(hourColor);

    painter.save();
    painter.rotate(30.0 * ((time.hour() + time.minute() / 60.0)));
    painter.drawConvexPolygon(hourHand, 3);
    painter.restore();

    painter.setPen(hourColor);

    for (int i = 0; i < 12; ++i) {
        painter.drawLine(88, 0, 96, 0);
        painter.rotate(30.0);
    }

    painter.setPen(Qt::NoPen);
    painter.setBrush(minuteColor);

    painter.save();
    painter.rotate(6.0 * (time.minute() + time.second() / 60.0));
    painter.drawConvexPolygon(minuteHand, 3);
    painter.restore();

    painter.setPen(minuteColor);

    for (int j = 0; j < 60; ++j) {
        if ((j % 5) != 0)
            painter.drawLine(92, 0, 96, 0);
        painter.rotate(6.0);
    }
}
```

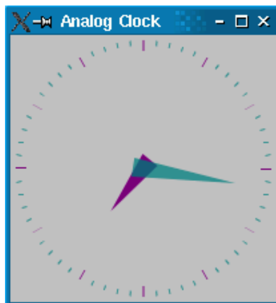

Figure 3: The Qt 4 Analog Clock example shows how to make a simple custom widget.

The main() function for this example is minimal. It simply sets up an application object, constructs the clock widget, and shows it. Finally, the application's event loop is started so that Qt can start processing events:

```
#include <QApplication>

#include "analogclock.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    AnalogClock clock;
    clock.show();
    return app.exec();
}
```

This example program contains a single top-level clock widget and no child widgets. Complex widgets are built by combining widgets in layouts.

Online References

http://doc.trolltech.com/4.0/qwidget.html
http://doc.trolltech.com/4.0/examples.html#widget-examples
http://doc.trolltech.com/4.0/tutorial.html
http://doc.trolltech.com/4.0/gallery.html

3. Signals and Slots

Signals and slots provide inter-object communication. They are easy to understand and use, and are fully supported by Qt Designer.

GUI applications respond to user actions. For example, when a user clicks a menu item or a toolbar button, the application executes some code. More generally, we want objects of any kind to be able to communicate with each other. The programmer must relate events to the relevant code. Older toolkits use mechanisms that are not type-safe (i.e., they are crash-prone), are inflexible, and are not object-oriented.

Trolltech has invented a solution called "signals and slots." The signals and slots mechanism is a powerful inter-object communication mechanism that can be used to completely replace the crude callbacks and message maps used by legacy toolkits. Signals and slots are flexible, fully object-oriented, and implemented in C++.

To associate some code with a button using the old callback mechanism, it is necessary to pass a function pointer to the button. When the button is clicked, the function is then called. Old toolkits do not ensure that arguments of the correct type are given to the function when it is called, making crashes more likely. Another problem with the callback approach is that it tightly binds the GUI element to the functionality, making it difficult to develop classes independently.

Qt's signals and slots mechanism is different. Qt widgets emit signals when events occur. For example, a button will emit a "clicked" signal when it is clicked. The programmer can choose to connect to a signal by creating a function (a "slot") and calling the connect() function to relate the signal to the slot. Qt's signals and slots mechanism does not require classes to have knowledge of each other, which makes it much easier to develop highly reusable classes. Since signals and slots are type-safe, type errors are reported as warnings and do not cause crashes to occur.

For example, if a Quit button's clicked() signal is connected to the application's quit() slot, a user's click on Quit makes the application terminate. In code, this is written as

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

Connections can be added or removed at any time during the execution of a Qt application, they can be set up so that they are executed when a signal is emitted or queued for later execution, and they can be made between objects in different threads.

The signals and slots implementation smoothly extends C++'s syntax and takes full advantage of C++'s object-oriented features. Signals and slots are type-safe, can be overloaded or reimplemented, and may appear in the public, protected, or private sections of a class.

To benefit from signals and slots, a class must inherit from **QObject** or one of its subclasses and include the `Q_OBJECT` macro in the class's definition. Signals are declared in the `signals` section of the class, while slots are declared in the `public slots`, `protected slots`, or `private slots` sections.

3.1. A Signals and Slots Example

Here is an example **QObject** subclass:

```
class BankAccount : public QObject
{
    Q_OBJECT

public:
    BankAccount() { curBalance = 0; }
    int balance() const { return curBalance; }

public slots:
    void setBalance(int newBalance);

signals:
    void balanceChanged(int newBalance);

private:
    int currentBalance;
};
```

In the style of most C++ classes, the **BankAccount** class has a constructor, a balance() “getter” function, and a setBalance() “setter” function. It also has a balanceChanged() signal which is emitted when the balance in the account is changed. When a signal is emitted, the slots it is connected to are executed.

The set function is declared in the `public slots` section, so it is a slot. Slots are member functions that can be called like any other function and that can also be connected to signals. Here's the implementation of the setBalance() slot:

```
void BankAccount::setBalance(int newBalance)
{
    if (newBalance != currentBalance) {
        currentBalance = newBalance;
        emit balanceChanged(currentBalance);
    }
}
```

The statement

```
emit balanceChanged(currentBalance);
```

causes the balanceChanged() signal to be emitted with the new current balance as its argument. The keyword `emit`, like `signals` and `slots`, is provided by Qt and is transformed into standard C++ by the C++ preprocessor.

Here's an example of how to connect two **BankAccount** objects:

```
BankAccount x, y;
connect(&x, SIGNAL(balanceChanged(int)), &y, SLOT(setBalance(int)));
x.setBalance(2450);
```

When the balance in `x` is set to 2450, the balanceChanged() signal is emitted. The signal is received by the setBalance() slot in `y`, which sets the balance in `y` to 2450.

One object's signal can be connected to many different slots, and many signals can be connected to one slot in a particular object. Connections are made between signals and slots whose parameters have the same types. A slot can have fewer parameters than the signal and ignore the extra parameters.

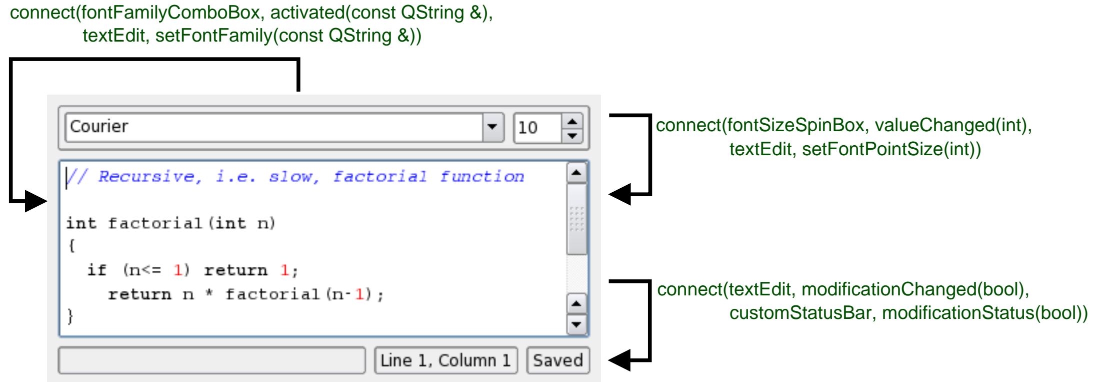

Figure 4: An example of signals and slots connections.

3.2. Meta-Object Compiler

The signals and slots mechanism is implemented in standard C++. The implementation uses the C++ preprocessor and `moc`, the Meta-Object Compiler, included with Qt.

The Meta-Object Compiler reads the application's header files and generates the necessary code to support the signals and slots mechanism. It is invoked automatically by makefiles generated by `qmake` (see Qt's Build System on page 52). Developers never have to edit or even look at the generated code.

In addition to handling signals and slots, the Meta-Object Compiler supports Qt's translation mechanism, its property system, and its extended run-time type information. It also makes run-time introspection of C++ programs possible in a way that works on all supported platforms.

Online References

http://doc.trolltech.com/4.0/object.html
http://doc.trolltech.com/4.0/signalsandslots.html
http://doc.trolltech.com/4.0/moc.html

4. 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 all the classes and functions necessary to create modern GUI applications. Qt 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.

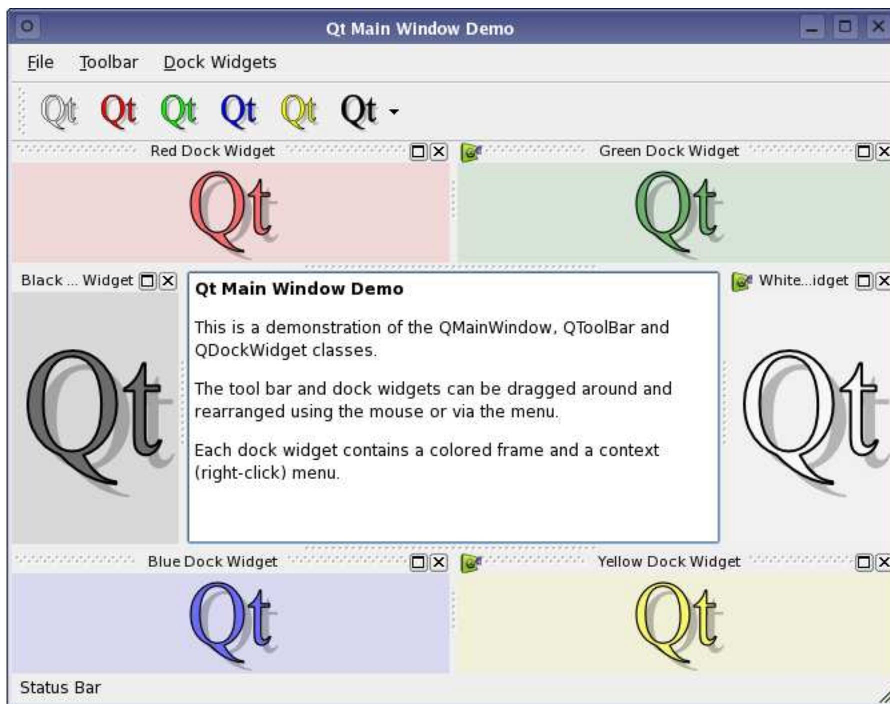

Figure 5: The Qt 4 Main Window demo shows an application main window with a main menu, a toolbar, dock windows, and a central widget.

Toolbars can be moved around within the toolbar area, dragged to other 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 required.

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 store application settings in a platform-independent way, using the system registry or text files, allowing items such as user preferences, most recently used files, window and toolbar positions and sizes to be recorded for later use.

4.1. Main Window Classes

4.1.1. The Main Window

The **QMainWindow** class provides a framework for typical application main windows. A main window contains a set of standard widgets. The top of the main window is occupied by a menu bar, beneath which toolbars are laid out in toolbar areas at the top, left, right, and bottom of the window. The area of the main window below the bottom toolbar area is occupied by a status bar. Tooltips and "What's this?" help provide balloon help for the user-interface elements.

For SDI applications, the central area of a **QMainWindow** can contain any widget. For example, a text editor could use a **QTextEdit** as its central widget:

```
QTextEdit *editor = new QTextEdit(mainWindow);
mainWindow->setCentralWidget(editor);
```

For MDI applications, the central area will usually be occupied by a **QWorkspace** widget.

4.1.2. Menus

The **QMenu** widget presents menu items to the user in a vertical list. Menus can be standalone (e.g., a context popup menu), can appear in a menu bar, or can be a sub-menu of another popup menu. Menus can have tear-off handles.

Each menu item can have an icon, a checkbox, and an accelerator. Menu items usually correspond to actions (e.g., "Save"). Separator items are displayed as a line and are used to group related actions visually. Here's an example that creates a File menu with New, Open, and Exit menu items:

```
QMenu *fileMenu = new QMenu(this);
fileMenu->addAction(tr("&New"), this, SLOT(newFile()), tr("Ctrl+N");
fileMenu->addAction(tr("&Open..."), this, SLOT(open()), tr("Ctrl+O"));
fileMenu->addSeparator();
fileMenu->addAction(tr("E&xit"), qApp, SLOT(quit()), tr("Ctrl+Q"));
```

When a menu item is chosen, the corresponding slot is executed. Note that, in this case, the tr() function is used to retrieve menu text in the user's native language (see Internationalization on page 36).

The **QMenuBar** class implements a menu bar. It is automatically laid out at the top of its parent widget (typically a **QMainWindow**), splitting its contents across multiple lines if the parent window is not wide enough. Qt's layout managers take any menu bar into consideration. On the Macintosh, the menu bar appears at the top of the screen.

Here's how to create a menu bar with File, Edit, and Help menus:

```
QMenuBar *menuBar = new QMenuBar(this);
menuBar->addMenu(tr("&File"), fileMenu);
menuBar->addMenu(tr("&Edit"), editMenu);
menuBar->addMenu(tr("&Help"), helpMenu);
```

Qt's menus are very flexible and are part of an integrated *action system* (see Actions). Actions can be enabled or disabled, dynamically added to menus, and removed again later.

4.1.3. Toolbars

Toolbars contain collections of buttons and other widgets that the user can access to perform actions. They can be moved between the areas at the top, left, right, and bottom of the central area of a main window. Any toolbar can be dragged out of its toolbar area, and floated as an independent tool palette.

The **QToolButton** class implements a toolbar button with an icon, a styled frame, and an optional label. Toggle toolbar buttons turn features on and off. Other toolbar buttons execute commands. Different icons can be provided for the active, disabled, and enabled modes, and for the on and off states. If only one icon is provided, Qt automatically distinguishes the state using visual cues, for example, graying out disabled buttons. Toolbar buttons can also trigger popup menus.

Tool buttons usually appear side by side within a toolbar. An application can have any number of toolbars, and the user is free to move them around. Toolbars can contain widgets of almost any type; for example, **QComboBox** and **QSpinBox** widgets are often used.

4.1.4. Actions

Applications usually provide the user with several different ways to perform a particular action. For example, most applications have traditionally provided a "Save" action available from the File menu, from the toolbar (a "floppy disk" toolbar button), and as an accelerator (Ctrl+S). The **QAction** class encapsulates this concept. It allows programmers to define an action in one place.

The following code implements a "Save" action with a menu item, a toolbar button, and a keyboard accelerator, all with interactive help provided by a tooltip and "What's This?" information:

```
QAction *saveAct = new QAction(tr("Save"), saveIcon, tr("&Save"), tr("Ctrl+S"),
                               this);
connect(saveAct, SIGNAL(activated()), this, SLOT(save()));
saveAct->setWhatsThis(tr("Saves the current file."));
saveAct->addTo(fileMenu);
saveAct->addTo(toolbar);
```

As well as avoiding duplication of work, using a **QAction** ensures that the state of menu items stay in sync with the state of related toolbar buttons, and that interactive help is displayed when necessary. Disabling an action will disable any corresponding menu items and toolbar buttons. Similarly, if the user clicks a toggle button in a toolbar, the corresponding menu item will also be toggled.

4.1.5. Dock Windows

Dock windows are windows that the user can move inside a toolbar area or from one toolbar area to another. The user can undock a dock window and make it float on top of the application, or minimize it. Dock windows are provided by the **QDockWidget** class.

A custom dock window can be created by instantiating **QDockWidget** and by adding widgets to it. The widgets are laid out side by side if the dock window occupies a horizontal

area (e.g., at the top of the main window) and above each other if the area is vertical (e.g., on the left of the main window).

Some applications, including *Qt Designer* (page 20) and *Qt Linguist* (page 38), use dock windows extensively. **QMainWindow** provides operators to save and restore the position of dock windows and toolbars, so that applications can easily restore the user's preferred working environment.

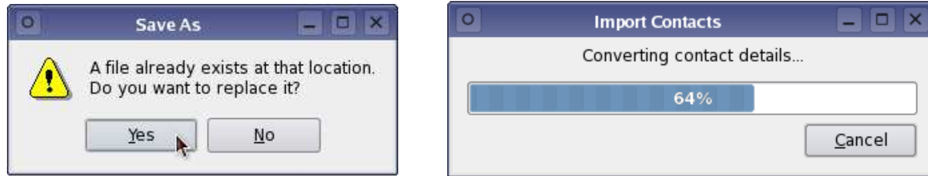

Figure 6: A **QMessageBox** and a **QProgressDialog** shown in Plastique style.

4.1.6. Dialogs

Most GUI applications use dialog boxes to interact with the user for certain operations. Qt includes ready-made dialog classes with convenience functions for the most common tasks. Screenshots of some of Qt's standard dialogs are presented below. Qt also provides standard dialogs for color selection and printing options.

Dialogs operate in one of three ways:

1. A *modal* dialog blocks input to the other visible windows in the same application. Users must close the dialog before they can access any other window in the application.
2. A *modeless* dialog operates independently of other windows.
3. A *semi-modal* dialog returns control to the caller immediately. These dialogs behave like modal dialogs from the user's point of view, but allow the application to continue processing. This is particularly useful for progress dialogs.

Modal dialogs are typically used like this:

```
OptionsDialog dialog(&optionsData);
if (dialog.exec()) {
    do_something(optionsData);
}
```

QFileDialog is a sophisticated file selection dialog. It can be used to select single or multiple local or remote files (e.g., using FTP), and includes functionality such as file renaming and directory creation. Like most Qt dialogs, **QFileDialog** is resizable, which makes it easy to view long file names and large directories. Applications can be set to automatically use the native file dialog on Windows and Macintosh.

Other common dialogs are also provided: **QMessageBox** is used to provide the user with information or to present the user with simple choices (e.g., "Yes" and "No"); **QProgressDialog** displays a progress bar and a Cancel button.

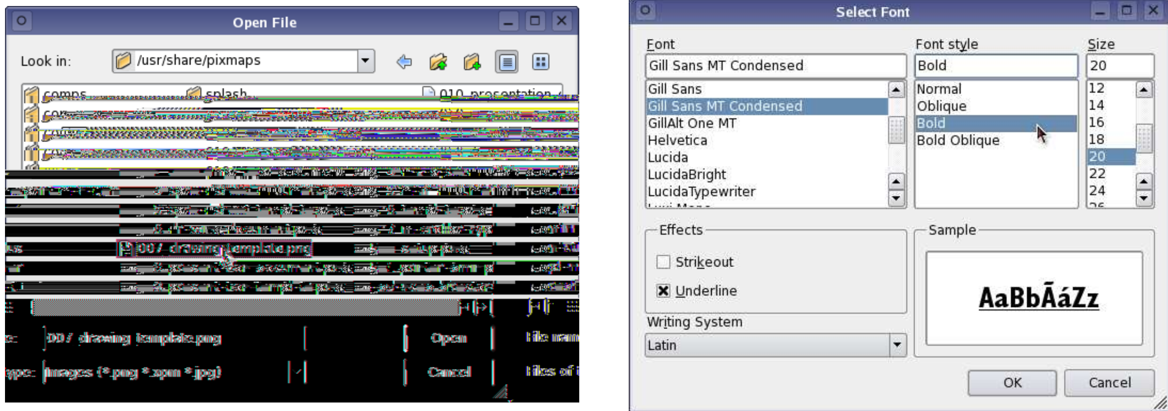

Figure 7: A **QFileDialog** and a **QFontDialog** shown in the Plastique style. On Windows and Mac OS X, native dialogs are used instead.

Programmers can create their own dialogs by subclassing **QDialog**, a subclass of **QWidget**, or use any of the standard dialogs provided. *Qt Designer* also includes dialog templates to help developers get started with new designs.

4.1.7. Interactive Help

Modern applications use various forms of interactive help to explain the purpose of user interface elements. Qt provides two mechanisms for giving brief help messages: tooltips and "What's this?" help.

Tooltips are small, usually yellow, rectangles that appear automatically when the mouse pointer hovers over a widget. Tooltips are often used to explain the purpose of toolbar buttons, since toolbar buttons are rarely displayed with text labels. Here's how to set the tooltip of a "Save" toolbar button:

```
QToolTip::add(saveButton, tr("Save"));
```

It is also possible to use longer pieces of text to be displayed in a main window's status bar when each tooltip is shown.

"What's this?" help is similar to tooltips, except that the user must request it, for example by pressing **Shift+F1** and then clicking a widget or menu item. "What's this?" help is typically longer than a tooltip. Here's how to set the "What's this?" text for a "Save" toolbar button:

```
QWhatsThis::add(saveButton, tr("Saves the current file."));
```

The **QToolTip** and **QWhatsThis** classes provide virtual functions that can be reimplemented for more specialized behavior, such as providing dynamic tooltips that display different text depending on the position of the mouse within a widget.

More detailed information about an application can be provided by an online help browser. The **QAssistantClient** class allows applications to use *Qt Assistant* (see page 20) to show relevant pages from their user manuals at the user's request.

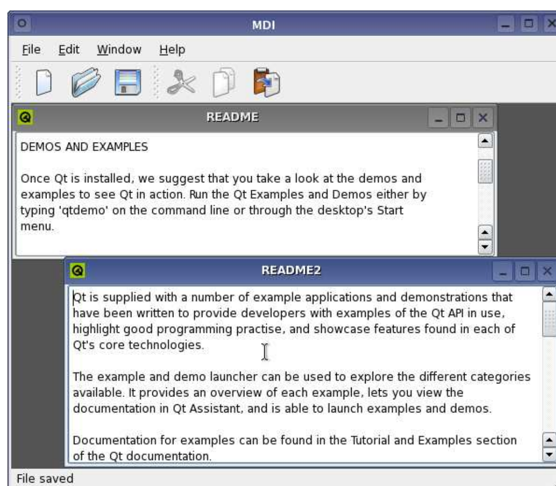

Figure 8: An application main window containing a **QWorkspace** widget to provide a Multiple Document Interface.

4.1.8. Multiple Document Interface

Multiple Document Interface (MDI) features are provided by the **QWorkspace** class, which is typically used as the central widget of a **QMainWindow**.

Child widgets of **QWorkspace** can be widgets of any type. They are rendered with a frame similar to the frame around top-level widgets, and functions such as those to show, hide, maximize, and set the window title work in the same way for child MDI widgets as for ordinary top-level widgets.

QWorkspace provides positioning strategies such as *cascade* and *tile*. If a child widget extends outside the MDI area, scroll bars can be set to appear automatically. If a child widget is maximized, the frame buttons (e.g., **Minimize**) are shown in the menu bar.

4.2. Settings

User settings and other application settings can easily be stored on disk using the **QSettings** class. On Windows, **QSettings** makes use of the system registry; on Mac OS X, it uses the system's `CFPreferences` mechanism; on other platforms, settings are stored in text files.

A particular setting is stored using a key. For example, the key

```
/SoftwareInc/Zoomer/RecentFiles
```

might contain a list of recently used files. Boolean values, numbers, Unicode strings, and lists of Unicode strings can be stored.

A variety of Qt data types can be used seamlessly with **QSettings** and will be serialized for storage and later retrieval by applications. See Reading and Writing Files on page 45 for more information about serialization of Qt's data types.

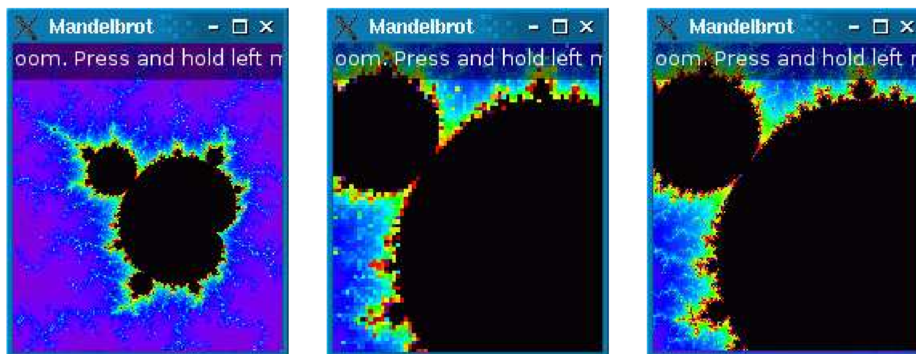

Figure 9: The Qt 4 Mandelbrot example shows how threading can be used to keep the user interface responsive while performing time-consuming tasks.

4.3. Multithreading

GUI applications often use multiple threads: one thread to keep the user interface responsive, and one or many other threads to perform time-consuming activities such as reading large files and performing complex calculations. Qt can be configured to support multithreading, and provides classes to represent threads, mutexes, semaphores, thread-global storage, and classes that support various locking mechanisms.

Qt 4's meta-object system enables objects in different threads to communicate using signals and slots, making it possible for developers to create single-threaded applications that can later be adapted for multithreading without an extensive redesign. Additionally, components can communicate across thread boundaries by posting events to one another. Certain types of object can also be moved between threads.

Online References

http://doc.trolltech.com/4.0/qmainwindow.html
http://doc.trolltech.com/4.0/qt4-mainwindow.html
http://doc.trolltech.com/4.0/threads.html

5. Qt Designer

Qt Designer is a graphical user interface design tool for Qt applications. Applications can be written entirely as source code, or using Qt Designer to speed up development. A component-based architecture makes it possible for developers to extend Qt Designer with custom widgets and extensions, and even integrate it into integrated development environments.

Designing a form with *Qt Designer* is a simple process. Developers drag widgets from a toolbox onto a form, and use standard editing tools to select, cut, paste, and resize them. Each 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. See Layouts on page 39 for information about Qt's automatic layouts.

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 other styles; for example, a Macintosh developer can preview a form in the Windows style.

Commercial licensees on Windows can enjoy *Qt Designer*'s user interface design facilities from within Microsoft Visual Studio®. On Mac OS X, developers can use *Qt Designer* from within Apple's Xcode® environment.

5.1. Working with Qt Designer

Developers can create both "dialog" style applications and "main window" style applications with menus, toolbars, balloon help, and other standard features. Several form templates are supplied, and developers can create their own templates to ensure consistency across an application or family of applications. Programmers can create their own custom widgets that can easily be integrated with *Qt Designer*.

Qt Designer supports a form-based approach to application development. A form is represented by a user interface (`.ui`) file, which can either be converted into C++ and compiled into an application, or processed at run-time by the **QFormBuilder** class to produce dynamically-generated user interfaces. Qt's build system (page 52) is able to automate the compile-time construction of user interfaces to make the design process easier.

5.2. Qt Assistant

Qt Designer's on-line help is provided by the *Qt Assistant* application. *Qt Assistant* displays Qt's entire documentation set, and works in a similar way to a web browser. But unlike web browsers, *Qt Assistant* applies a sophisticated indexing algorithm to provide fast full text searching of all the available documentation.

Qt's reference documentation consists of around 2,200 HTML pages, documenting Qt's

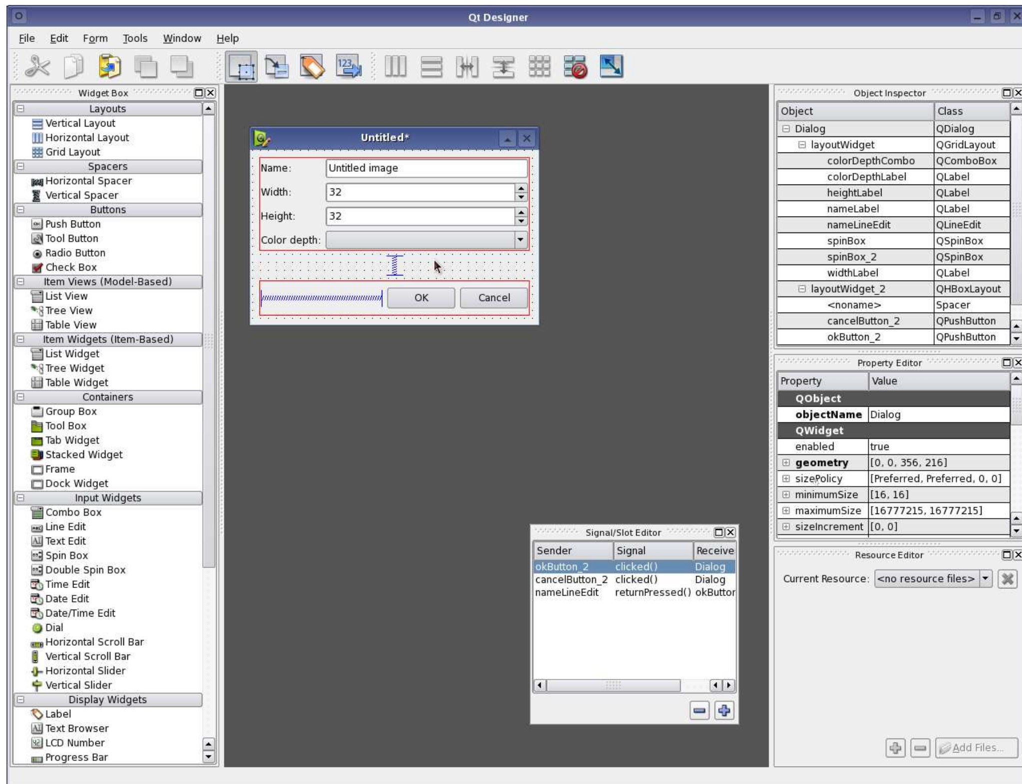

Figure 10: *Qt Designer* in "Docked Window" mode.

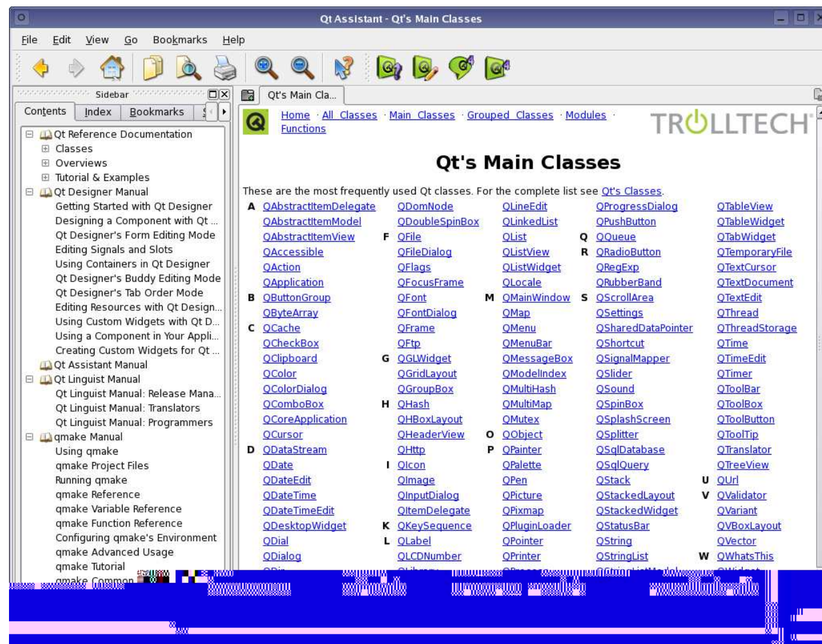

Figure 11: *Qt Assistant* displaying a page from the Qt 4 documentation.

classes and tools, and includes overviews and introductions to various aspects of Qt programming.

Developers can deploy *Qt Assistant* as the help browser for their own applications and documentation sets. *Qt Assistant* integration is achieved using the **QAssistantClient** class. *Qt Assistant* renders Qt's HTML reference documentation using **QTextEdit**; developers can use this class directly to implement their own help browsers if preferred. **QTextEdit** supports a subset of HTML 4.0, and can also be used to render other document formats when used with the rich text classes provided by Qt (see Rich Text Processing on page 32).

5.3. GUI Application Example

The "Class Hierarchy" application is a classic dialog-style application where the user chooses some options, in this case paths, and then carries out some processing based on those options.

The complete code for the application is presented below. The form was designed in *Qt Designer* and stored in a `.ui` file. The `.ui` file is converted into C++ by `uic`, leaving the developer free to focus on the application's functionality.

Although *Qt Designer* makes designing the user interface easy, additional application logic is usually required. Developers usually provide additional features by subclassing the user interface generated by `uic` and implementing new functionality.

The dialog's constructor simply sets up the user interface provided by *Qt Designer* and creates two labels for a tree widget:

```
ClassHierarchy::ClassHierarchy(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    QStringList headings;
    headings << tr("Class") << tr("Source file");
    treeWidget->setHeaderLabels(headings);
}
```

The on_<name>_clicked() functions are all slots that are automatically connected to signals from push buttons in the dialog when setupUi() is called. The following slots simply change the items in the list widget:

```
void ClassHierarchy::on_addSearchPathButton_clicked()
{
    QString path = QFileDialog::getExistingDirectory(
            this, "Select a Directory", QDir::home().path());
    if (!path.isEmpty() &&
        searchPathBox->findItems(path, Qt::MatchExactly).count() == 0)
        searchPathBox->addItem(path);
}

void ClassHierarchy::on_removeSearchPathButton_clicked()
{
    delete searchPathBox->takeItem(searchPathBox->currentRow());
}
```

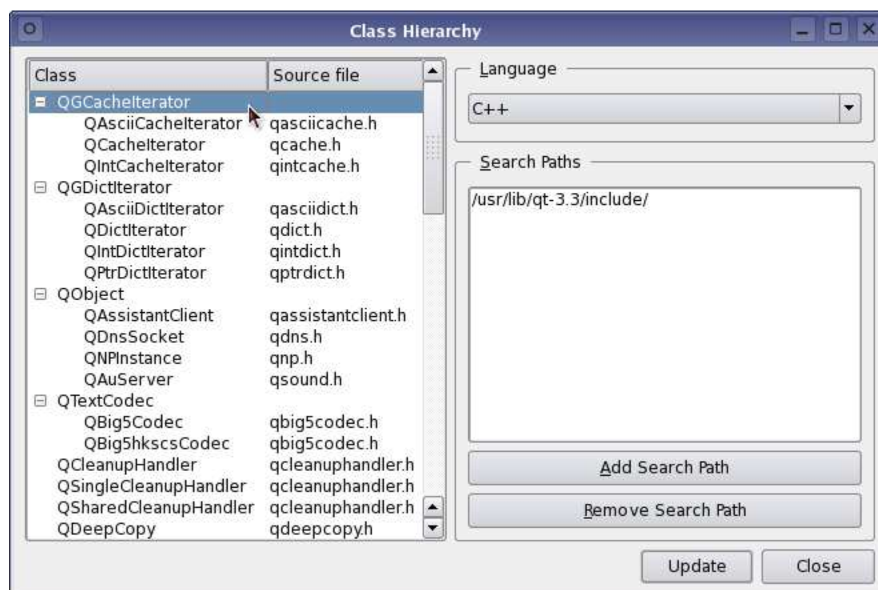

Figure 12: A simple class hierarchy application created using *Qt Designer*.

The on_updateButton_clicked() slot repopulates the tree widget with classes found in files whose names match the relevant pattern:

```
void ClassHierarchy::on_updateButton_clicked()
{
    QStringList fileNameFilter;
    QRegExp classDef;

    if (language->currentText() == "C++") {
        fileNameFilter << "*.h";
        classDef.setPattern("\\bclass\\s+([A-Z_a-z0-9]+)\\s*"
                            "(?:\\{|:\\s*public\\s+([A-Z_a-z0-9]+))");
    } else if (language->currentText() == "Java") {
        fileNameFilter << "*.java";
        classDef.setPattern("\\bclass\\s+([A-Z_a-z0-9]+)\\s+extends\\s*"
                            "([A-Z_a-z0-9]+)");
    }

    classMap.clear();
    treeWidget->clear();

    for (int i = 0; i < searchPathBox->count(); i++) {
        QDir dir = searchPathBox->item(i)->text();
        QStringList names = dir.entryList(fileNameFilter);

        for (int j = 0; j < names.count(); j++) {
            QFile file(dir.filePath(names[j]));
            if (file.open(QIODevice::ReadOnly)) {
                QString content = file.readAll();
                int k = 0;
                while ((k = classDef.indexIn(content, k)) != -1) {
                    processClassDef(classDef.cap(1), classDef.cap(2), names[j]);
                    k++;
                }
            }
        }
    }
}
```

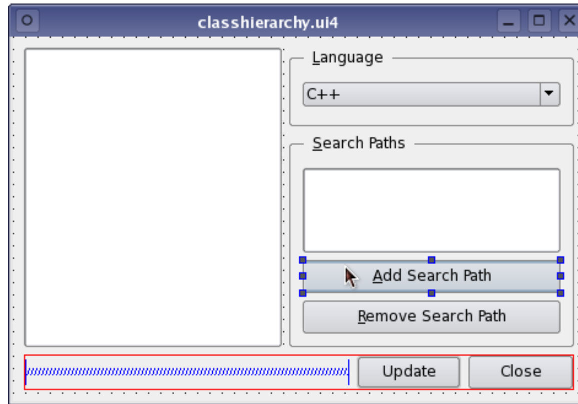

Figure 13: Editing the class hierarchy window in *Qt Designer*.

The Close button is connected to the dialog's accept() slot in *Qt Designer*.

The private utility functions for processing class definitions and inserting them into the tree widget are listed below for completeness:

```
void ClassHierarchy::processClassDef(const QString &derived,
        const QString &base, const QString &sourceFile )
{
    QTreeWidgetItem *derivedItem = insertClass(derived, sourceFile);

    if (!base.isEmpty()) {
        QTreeWidgetItem *baseItem = insertClass(base, "");
        if (derivedItem->parent() == 0) {
            treeWidget->takeTopLevelItem(
                treeWidget->indexOfTopLevelItem(derivedItem));
            baseItem->addChild(derivedItem);
            derivedItem->setText(1, sourceFile);
        }
    }
}

QTreeWidgetItem *ClassHierarchy::insertClass(const QString &name,
                                             const QString &sourceFile)
{
    if (classMap[name] == 0) {
        QTreeWidgetItem *item = new QTreeWidgetItem(treeWidget);
        item->setText(0, name);
        item->setText(1, sourceFile);
        treeWidget->setItemExpanded(item, true);
        classMap.insert(name, item);
    }
    return classMap[name];
}
```

The above example shows how user interfaces can be "compiled into" an application. User interfaces can also be dynamically generated from `.ui` files at run-time with the **QFormBuilder** class, making it possible for developers to create single-executable applications that can be customized for different uses.

The tools used to create and edit the source code for applications created with *Qt Designer* will depend on each developer's personal preferences; some will want to take advantage of the integration features provided with *Qt Designer* to develop from within Microsoft Visual Studio or Apple's Xcode environment.

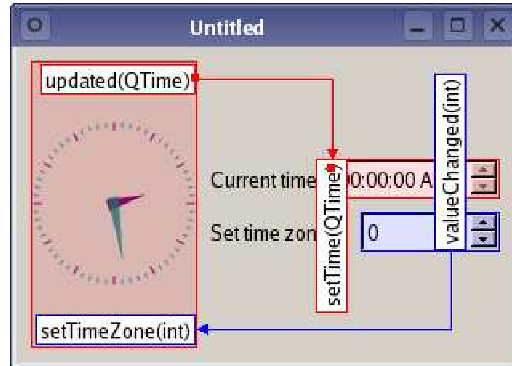

Figure 14: A World Time Clock custom widget is packaged as a plugin for *Qt Designer* and added to a form. The widget provides a custom signal and a custom slot, and these are automatically made available to other widgets on the form.

5.4. Extending Qt Designer

The component-based architecture used as a foundation for *Qt Designer* was specifically designed to allow developers to extend its user interface and editing tools with custom components. In addition, the modular nature of the application makes it possible to make *Qt Designer*'s user interface design features available from within integrated development environments such as Microsoft Visual Studio and KDevelop.

In total, the **QtDesigner** module provides 20 classes for working with `.ui` files and extending *Qt Designer*. Many of these allow third parties to customize the user interface of the application itself.

Third party and custom widgets for in-house work are easily integrated into *Qt Designer*. Adapting an existing widget for use within *Qt Designer* only requires a the widget to be compiled as a plugin, using an interface class to supply default widget properties and construct new instances of the widget. The plugin's interface is exported to *Qt Designer* using a macro similar to that described in Plugins on page 51.

Online References

http://doc.trolltech.com/4.0/designer-manual.html
http://www.trolltech.com/products/qt/vs-integration.html
http://doc.trolltech.com/4.0/qtdesigner.html
http://doc.trolltech.com/4.0/examples.html#qt-designer-examples

6. 2D and 3D Graphics

Qt provides excellent support for 2D and 3D graphics. Qt's 2D graphics classes support raster and vector graphics, and can load and save a wide and extensible range of image formats. Qt can draw Unicode rich text, rotated and sheared as required. Qt is the de facto standard GUI framework for platform-independent OpenGL programming.

Graphics are drawn using device-independent painter objects that allow the developer to reuse the same code to render graphics on different types of device, represented in Qt by paint devices (see Painting on page 26). This approach ensures that a wide range of powerful painting operations are available for each of the variety of devices supported by Qt.

Support for device-independent colors is provided by the **QColor** class. Colors are specified by ARGB, AHSV, or ACMYK values, or by common names (e.g., "skyblue"). **QColor**'s color channels are 16 bits wide, and colors can be specified with an optional level of opacity; Qt automatically allocates the requested color in the system's palette, or uses a similar color on color-limited displays.

6.1. Painting

The **QPainter** class provides a platform-independent API for painting onto widgets and other paint devices. It provides primitives as well as advanced features such as transformations and clipping. All of Qt's built-in widgets paint themselves using **QPainter**, and programmers invariably use **QPainter** when implementing their own custom widgets.

QPainter provides standard functions to draw points, lines, ellipses, arcs, Bezier curves, and other primitives. More complex painting operations include support for polygons and vector paths, allowing detailed drawings to be prepared in advance and drawn using a single function call. Text can also be painted directly with a painter or incorporated in a path for later use.

Qt's painting system also provides a number of advanced features to improve overall rendering quality:

- Alpha blending and Porter-Duff composition modes enable the developer to use sophisticated graphical effects and provide a high level of control over the output on screen.
- Anti-aliasing of graphics primitives and text can be used to mimic the appearance of a higher resolution display than the one in use.
- Linear, radial, and conical gradient fills allow more detailed graphics to be created, such as 3D bevel buttons, without much effort by the developer.

QPainter supports clipping using regions composed of rectangles, polygons, ellipses, and vector paths. Complex regions may be created by combining simple regions using standard set operations.

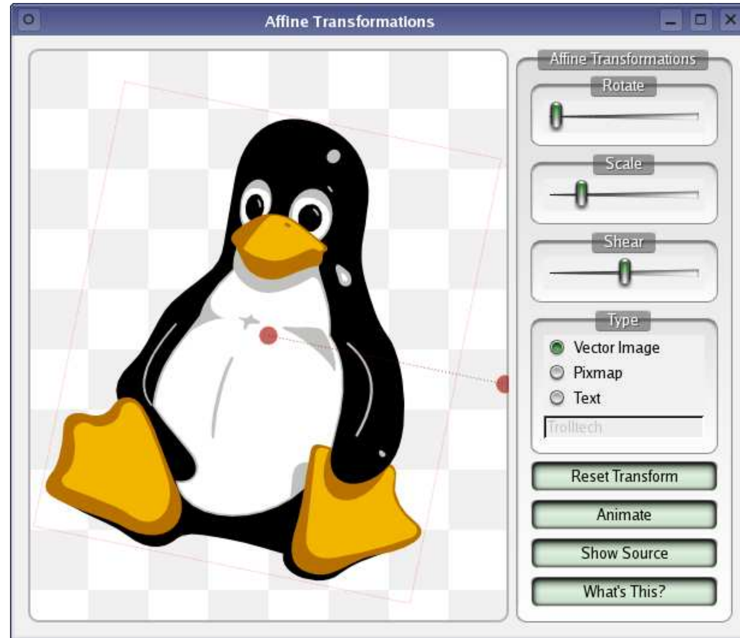

Figure 15: The Qt 4 Affine Transformations demonstration shows how transformations can be used to achieve a desired effect.

6.2. Images

The **QPixmap** and **QImage** classes supports input, output, and manipulation of images in several formats, including BMP, GIF*, JPEG, MNG, PNG, PNM, XBM, and XPM. Both classes can be used as paint devices and used in interactive graphical applications, or they can be used to preprocess images for later use in standard user interface components. Many of Qt's built-in widgets, such as buttons, labels, and menu items, are able to display images.

QPixmap is usually used when applications need to render images quickly. **QImage** is more useful for pixel manipulation, and handles images in a variety of color depths and pixel formats. Programmers can manipulate the pixel and palette data, apply transformations such as rotations and shears, and reduce the color depth with dithering if desired. Support for "alpha channel" data along with the color data enables applications to use transparency and alpha-blending for image composition and other purposes.

The range of graphics file formats that can be used with these classes can be extended through the use of an extensible plugin mechanism.

6.3. Paint Devices

QPainter can operate on any paint device. The code required to paint on any supported device is the same, regardless of the device.

*If you are in a country that recognizes software patents and where Unisys holds a patent on LZW decompression, Unisys may require you to license the technology to use GIF. We believe that this patent will have expired world-wide by the end of 2004.

Figure 16: Qt applications can use OpenGL to render 3D graphics alongside conventional GUI controls.

Qt supports the following paint devices:

- All **QWidget** subclasses are paint devices. Qt uses *double buffering* to reduce flickering during the painting process.
- A **QPixmap** is essentially a **QImage** with the same properties as a widget on the screen, and is often a system device that can be accessed quickly and efficiently.
- A **QImage** is a device-independent image with a specified color depth and pixel format. Images can be created with support for varying levels of transparency and painted onto custom widgets to achieve certain effects.
- A **QPicture** is a vector image that can be scaled, rotated, and sheared gracefully. Pictures are stored as a list of paint commands rather than as pixel data.
- A **QPrinter** represents a physical printer. On Windows, the paint commands are sent to the Windows print engine, which uses the installed printer drivers. On Unix, PostScript is written out and sent to the print daemon.

6.4. 3D Graphics

OpenGL is a standard API for rendering 3D graphics. Qt developers can use OpenGL to draw 3D graphics in their GUI applications. Qt's OpenGL module is available on Windows, X11, and Mac OS X, and uses the system's OpenGL library.

To use OpenGL-enabled widgets in a Qt application, developers only need to subclass **QGLWidget** and draw onto it with standard OpenGL functions. Qt provides functions to convert **QColor** values to OpenGL's color format to help developers provide a consistent user interface for their applications.

Online References

http://doc.trolltech.com/4.0/qt4-arthur.html
http://doc.trolltech.com/4.0/qpainter.html
http://doc.trolltech.com/4.0/opengl.html

7. Item Views

Qt's item view widgets provide standard GUI controls for displaying and modifying large quantities of data. The underlying model/view framework isolates the way data is stored from the way it is presented to the user, enabling features

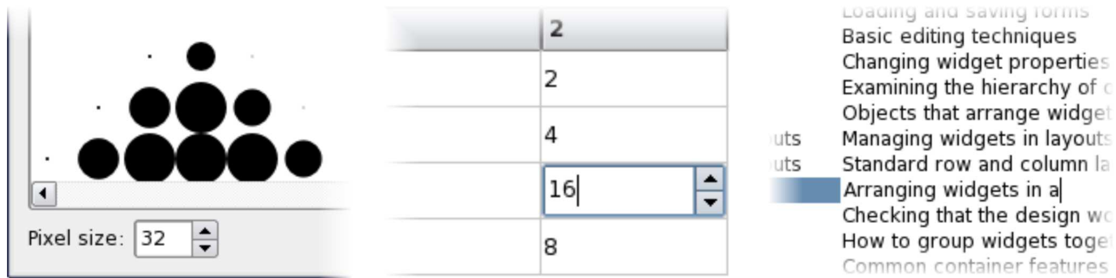

Figure 18: The component-oriented architecture of the model/view framework makes it easy to customize item views.

7.2. Qt's Model/View Framework

The model/view framework provided by Qt is a variation of the well-known *Model-View-Controller* pattern, adapted specially for Qt's item views. In this approach, models are used to supply data to other components, views display items of data to the user, and delegates handle aspects of the rendering and editing processes.

Models are wrappers around sources of data that are written to conform to a standard interface provided by **QAbstractItemModel**. This interface enables widgets derived from **QAbstractItemView** to access data supplied via the model, irrespective of the nature of the original data source.

The separation between data and its presentation which this approach enables provides a number of improvements over classic item views:

- Since models provide a standard interface for accessing data, they can be designed and written separately from other components, and replaced if necessary.
- Data obtained from models can be shared between views. This enables applications to provide multiple views onto the same data set, and potentially show different representations of data.
- Selections can be shared between views, or kept separate, depending on the user's requirements and expectations.
- For standard list, tree, and table views, most of the rendering is performed by delegates. This makes it easy to customize views for most purposes without having to write a lot of new code.

The model/view system is also used by Qt's SQL models (page 34) to make database integration simpler for non-database developers.

Online References

http://doc.trolltech.com/4.0/model-view-programming.html
http://doc.trolltech.com/4.0/examples.html#item-view-examples

8. Text Handling

Qt provides a powerful text editor widget that allows the user to create and edit rich text documents, supports HTML import and export, and can be used to prepare documents for printing. The underlying document structure used by the editor is fully accessible to developers, allowing both the structure and content of documents to be manipulated from within applications.

Rich text documents typically contain text in a variety of fonts, colors, and sizes arranged in a series of paragraphs. Text can also be organized using lists and tables, and may be visually separated from the main body of a document by using frames. The appearance of each document element can be precisely adjusted using the many properties made available to developers through the rich text API.

Qt's rich text display features can be seen at work in *Qt Assistant*, in the Qt 4 Text Edit demonstration, and in *Qt Designer*'s text editing dialogs.

8.1. Rich Text Editing

Interactive rich text display and editing is handled in Qt by the **QTextBrowser** and **QTextEdit** widgets. These widgets fully support Unicode and are built on a structured document representation provided by **QTextDocument** that removes the need to use intermediate mark up languages to create rich text. **QTextDocument** also provides support for HTML import and export, full undo/redo capabilities (including grouping of operations), and resource handling.

The layout system used to display rich text in **QTextEdit** widgets is also able to format documents according to information obtained from a **QPrintDialog** into a series of pages suitable for printing with a **QPrinter**.

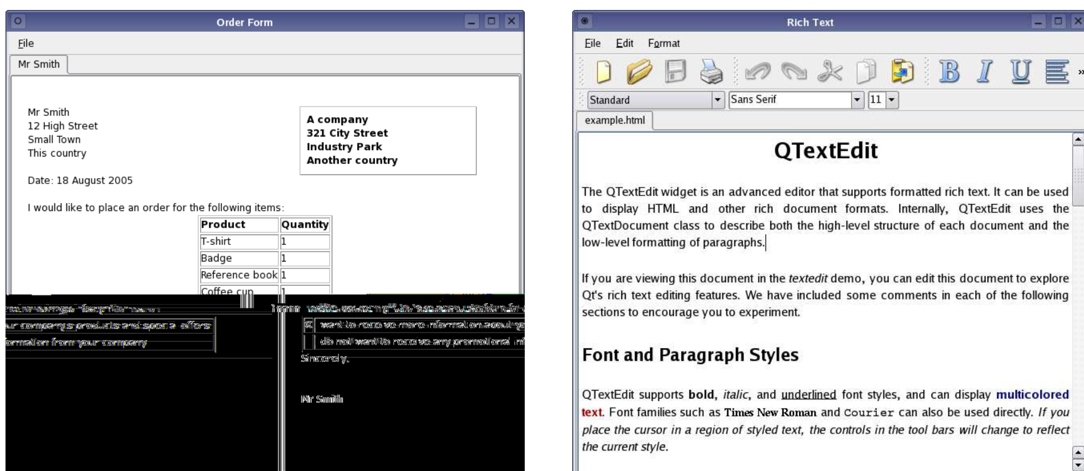

Figure 19: Qt's advanced rich text document features allow complex documents to be created and edited in **QTextEdit**.

8.2. Rich Text Processing

Rich text documents can be programatically explored using a combination of two approaches: a high-level object-based approach, and a cursor-based approach that is analogous to using a text editor. The object-based approach makes it easy to get a high-level overview of a document's structure and navigate tables, frames, and other elements. The cursor-based approach allows the document's contents to be modified and transformed, and even allows large-scale structural changes to be made.

The following code inserts a table with two rows and two columns into a document, before inserting text into the cells in the first column:

```
QTextTable *offersTable = cursor.insertTable(2, 2);

cursor = offersTable->cellAt(0, 1).firstCursorPosition();
cursor.insertText(tr("I want to receive more information about your "
                     "company's products and special offers."), textFormat);

cursor = offersTable->cellAt(1, 1).firstCursorPosition();
cursor.insertText(tr("I do not want to receive any promotional information "
                     "from your company."), textFormat);
```

In addition to the classes corresponding to structure and content, there are a number of classes which control the appearance of text and document elements. These allow the text styles used in documents to be specified precisely:

```
QTextCharFormat plainFormat(cursor.charFormat());
QTextCharFormat italicFormat = plainFormat;
italicFormat.setFontItalic(true);
QTextCharFormat boldFormat = plainFormat;
boldFormat.setFontWeight(QFont::Bold);

cursor.insertText(tr("Note: "), boldFormat);
cursor.insertText(tr("We can emphasize text by making it "), plainFormat);
cursor.insertText(tr("italic"), italicFormat);
cursor.insertText(tr("."), plainFormat);
```

Styles for tables, lists, frames, and ordinary paragraphs can also be customized to give documents the desired appearance. Documents created programatically in this way remain editable in **QTextEdit** widgets and maintain a full undo/redo history. Developers can augment the standard editing features available to let users add custom structures and content.

8.3. Custom Text Layouts

Qt 4's text handling features can also be used to provide specialized text formatting for custom widgets and rich text documents. These can be written using low-level classes such as **QTextLayout** to lay out the text line by line and integrated into the extensible text layout system provided by **QTextDocument** for use with **QTextEdit**.

Online References

http://doc.trolltech.com/4.0/qt4-scribe.html
http://doc.trolltech.com/4.0/richtext.html

9. Databases

The Qt SQL module simplifies the creation of multiplatform GUI database applications. Programmers can easily execute SQL statements, use database-specific widgets, and make any widget data-aware. Several database models are also provided that can be plugged into item views for convenient visualization of stored information.

The Qt SQL module provides a multiplatform interface for accessing SQL databases. Qt includes native drivers for Oracle, Microsoft SQL Server, Sybase Adaptive Server, IBM DB2, PostgreSQL, MySQL, Borland Interbase, SQLite, and ODBC. The drivers work on all platforms supported by Qt for which client libraries are available. Programs can access multiple databases using multiple drivers simultaneously.

Developers can easily execute any SQL statements. Qt also provides a high-level C++ interface that can be used to generate the appropriate SQL statements automatically.

Qt provides a set of SQL models for use with the other model/view components (page 30). These enable view widgets to be automatically populated with the results of database queries, and simplify the process of editing for both users and non-database developers.

Using the facilities that the Qt SQL module provides, it is straightforward to create database applications that use foreign key lookups and present master-detail relationships.

9.1. Executing SQL Commands

The **QSqlQuery** class is used to directly execute any SQL statement. It is also used to navigate the result sets produced by `SELECT` statements. In the example below, a query is executed, and the result set navigated using QSqlQuery::next():

```
while (query.next())
    cout << query.value(0);
```

Field values are indexed in the order they appear in the `SELECT` statement. **QSqlQuery** also provides the first(), prev(), last(), and seek() navigation functions.

The `INSERT`, `UPDATE`, and `DELETE` statements are equally simple. Here is an `UPDATE` example:

```
QSqlQuery query("UPDATE staff SET salary = salary * 1.10"
                " WHERE id > 1155 AND id < 8155");
if (query.isActive()) {
    cout << "Pay rise given to " << query.numRowsAffected()
         << " staff" << endl;
}
```

Qt's SQL module also supports value binding and prepared queries; for example:

```
QSqlQuery query;
query.prepare("INSERT INTO staff (id, surname, salary)"
              " VALUES (:id, :surname, :salary)");
query.bindValue(":id", 8120);
query.bindValue(":surname", "Bean");
query.bindValue(":salary", 29960.5);
query.exec();
```

Figure 20: The Qt 4 Books demonstration shows the integration between Qt's SQL classes and the model/view framework.

Value binding can be achieved using named binding and named placeholders (as above), or using positional binding with named or positional placeholders; for example:

```
QSqlQuery query;
query.prepare("INSERT INTO staff (id, surname, salary)"
              " VALUES (?, ?, ?)");
EmployeeMap::iterator it;
for (it = employeeMap.begin(); it != employeeMap.end(); ++it) {
    query.addBindValue(it.data().id());
    query.addBindValue(it.key());
    query.addBindValue(it.data().salary());
    query.exec();
}
```

Qt's binding syntax works with all supported databases, either using the underlying database support or by emulation.

9.2. SQL Models

Qt also provides a number of model classes for use with other components in the model/view framework (page 30). These allow the developer to set up SQL queries to automatically provide table views with items of data from a database. Using these database models with other components in the model/view framework requires a minimum of intervention on the part of the developer.

Setting up a query model is simply a matter of specifying a query and choosing which headers to examine:

```
QSqlQueryModel model;
model->setQuery("select * from person");
model->setHeaderData(0, Qt::Horizontal, QObject::tr("ID"));
model->setHeaderData(1, Qt::Horizontal, QObject::tr("First name"));
model->setHeaderData(2, Qt::Horizontal, QObject::tr("Last name"));
```

Setting up a table view to display the results of the query is similarly straightforward:

```
QTableView *view = new QTableView;
view->setModel(model);
view->show();
```

Models are provided for accessing SQL tables in different ways:

- **QSqlQueryModel** provides a read-only data model for SQL result sets.
- **QSqlTableModel** provides an editable data model for a single database table.
- **QSqlRelationalTableModel** acts like **QSqlTableModel**, but allows columns to be set as foreign keys into other database tables. The Qt 4 Books demonstration shown in Figure 20 uses a relational database model to find information about each of the books in a table.

The model/view framework contains a number of features that accommodate the requirements of database applications. These include support for transactions and the option to allow the contents of table to be edited on a per-row basis to avoid unnecessary round trips to a database.

Online References

http://doc.trolltech.com/4.0/sql.html
http://doc.trolltech.com/4.0/qt4-sql.html
http://doc.trolltech.com/4.0/examples.html#sql-examples

10. Internationalization

Qt fully supports Unicode, the international standard character set. Programmers can freely mix Arabic, English, Hebrew, Japanese, Russian, and other languages supported by Unicode in their applications. Qt also includes tools to support application translation to help companies reach international markets.

Qt includes tools to facilitate the translation process. Programmers can easily mark user-visible text that needs translation, and a tool extracts this text from the source code. *Qt Linguist* (page 38) is an easy-to-use GUI application that reads the extracted source texts, and provides the texts with context information ready for translation. When the translation is complete, *Qt Linguist* outputs a translation file for applications to use.

Qt uses the **QString** class to store Unicode strings, and uses it both throughout the API and internally. **QString** replaces `const char *` pointers and `std::string`, and the 16-bit **QChar** class replaces `char`. Constructors and operators are provided to automatically convert to and from 8-bit strings. Programmers can copy **QString** objects by value, since they are implicitly shared (copy on write; see page 49), which makes them fast and memory efficient.

QString is more than a 16-bit character string. Functions such as QChar::lower() and QChar::isPunct() replace tolower() and ispunct() and work over the whole Unicode range. Qt's regular expression engine, provided by the **QRegExp** class, uses Unicode strings both for the regular expression pattern and the target string.

Qt's locale support enables number-to-string and string-to-number conversions to be adapted to suit the user's geographical location and language preferences. For example:

```
QLocale iranian(QLocale::Persian, QLocale::Iran);
QString s1 = iranian.toString(195);       // s1 == "۱۹۵"
int n = iranian.toInt(s1);                // n == 195

QLocale norwegian(QLocale::Norwegian, QLocale::Norway);
QString s2 = norwegian.toString(3.14);    // s2 == "3,14" (comma)
double d = norwegian.toDouble(s2);        // d == 3.14
```

Conversion to and from different encodings and charsets is handled by **QTextCodec** subclasses. Qt uses **QTextCodec** for fonts, I/O, and input methods; developers can use it for their own purposes as well.

QTextCodec supports a wide variety of different encodings, including Big5 and GBK for Chinese, EUC-JP, JIS, and Shift-JIS for Japanese, KOI8-R for Russian, and the ISO-8859 series of standard encodings[†]. Developers can add their own encodings by providing a character map or by subclassing **QTextCodec**.

10.1. Text Entry and Rendering

Far-Eastern writing systems require many more characters than are available on a keyboard. The conversion from a sequence of key presses to actual characters is performed at the window-system level by software called *input methods*. Qt automatically supports the installed input methods.

[†] ISO is a registered trademark of the International Organization for Standardization.

Qt provides a powerful text-rendering engine for all text that is displayed on screen, from the simplest label to the most sophisticated rich text editor. The engine supports advanced features such as special line breaking behavior, bidirectional writing, and diacritical marks. It renders most of the world's writing systems, including Arabic, Chinese, Cyrillic, English, Greek, Hebrew, Japanese, Korean, Latin, and Vietnamese. Qt will automatically combine the installed fonts to render multi-language text.

10.2. Translating Applications

Qt provides tools and functions to help developers provide applications in their users' native languages. Qt itself contains about 400 user-visible strings, for which Trolltech provides French and German translations.

To make an ASCII string translatable, simply wrap it in a call to the tr() translation function; for example:

```
saveButton->setText(tr("Save"));
```

tr() attempts to replace a string literal (e.g., `"Save"`) with a translation if one is available; otherwise it uses the original text. English can be used as the source language and Chinese as the translated language, for example. The argument to tr() is converted to Unicode from the application's default encoding. Alternatively, text encoded as UTF-8 can be supplied to the translation system with the trUtf8() function.

The general syntax of tr() is

```
Context::tr("source text", "comment")
```

The "context" is the name of a **QObject** subclass. It is usually omitted, in which case the class containing the tr() call is used as the context. The "source text" is the text to translate. The "comment" is optional; along with the context, it provides additional information to human translators.

Translations are stored in **QTranslator** objects, which use disk-based `.qm` files (Qt Message files). Each `.qm` file contains the translations for a particular language. The language can be chosen at run-time, in accordance with the locale or user preferences.

Qt provides three tools for preparing `.qm` files: `lupdate`, *Qt Linguist*, and `lrelease`.

1. `lupdate` extracts a series of items, each containing a context, some source text, and a comment from the source code (including *Qt Designer* `.ui` files), then generates a `.ts` file (Translation Source file). These files are in human-readable XML format.
2. Translators use *Qt Linguist* to provide translations for the source texts in the `.ts` files.
3. Highly compressed `.qm` files are generated by running `lrelease` on the `.ts` files.

These steps are repeated as often as necessary during the lifetime of an application. It is perfectly safe to run `lupdate` frequently, as it reuses existing translations and marks translations for obsolete source texts without eliminating them. `lupdate` also detects slight changes in source texts and automatically suggests appropriate translations. These translations are marked as unfinished so that a translator can easily check them.

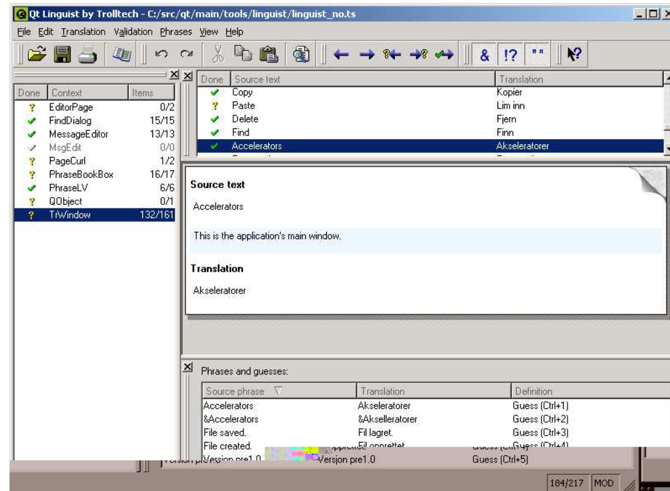

Figure 21: Working on a Norwegian translation with *Qt Linguist*.

10.3. Qt Linguist

Qt Linguist is a Qt application that helps translators translate Qt applications.

Translators can edit `.ts` files conveniently using *Qt Linguist*. The `.ts` file's contexts are listed in the left-hand side of the application's window. The list of source texts for the current context is displayed in the top-right area, along with translations. By selecting a source text, the translator can enter a translation, mark it done or unfinished, and proceed to the next unfinished translation. Keyboard shortcuts are provided for all the common navigation options, such as Done & Next and Next Unfinished. The user interface's dockable windows can be reorganized to suit the translators' preferences.

Applications often use the same phrases many times in different source texts. *Qt Linguist* automatically displays intelligent guesses based on previously translated strings and predefined translations at the bottom of the window. Guesses often serve as a good starting point that helps translators translate similar texts consistently. Common translations can also be stored in phrasebooks to make the translation of future applications more efficient. *Qt Linguist* can optionally validate translations to ensure that accelerators and ending punctuation are translated correctly.

Qt Linguist's comprehensive manual provides relevant information about the translation process for release managers, translators, and programmers.

Online References

http://doc.trolltech.com/4.0/i18n.html
http://doc.trolltech.com/4.0/unicode.html
http://doc.trolltech.com/4.0/qtextcodec.html
http://doc.trolltech.com/4.0/linguist-manual.html

11. Layouts

Layouts provide a powerful and flexible alternative to using fixed sizes and positions. Layouts free programmers from having to perform size and position calculations, and provide automatic scaling to suit the user's screen, language, and fonts.

Qt provides layout managers for organizing child widgets within their parent widget's area. They feature automatic positioning and resizing of child widgets, sensible minimum and default sizes for top-level widgets, and automatic repositioning when the contents or text font changes. *Qt Designer* (page 20) is fully able to use layout managers to position widgets.

Figure 22: The same dialog shown at different sizes.

Layouts are also useful for internationalization. With fixed sizes and positions, the translation text is often truncated (Figure 24); with layouts, the child widgets are automatically resized. Additionally, widget placement can be reversed to provide a more natural appearance for users who work with right-to-left writing systems.

11.1. Built-in Layout Managers

Qt provides layout managers to arrange widgets and other layouts horizontally, vertically, and in grids of items:

- **QHBoxLayout** organizes the managed widgets in a single horizontal row from left to right.
- **QVBoxLayout** organizes the managed widgets in a single vertical column from top to bottom.
- **QGridLayout** organizes the managed widgets in an expanding grid of cells, with the facility to let widgets span multiple cells where necessary.

Each of the built-in layout managers allow widgets to be horizontally and vertically aligned within the space allocated to them, making it possible to customize the appearance of a user interface using only simple layouts and alignment properties.

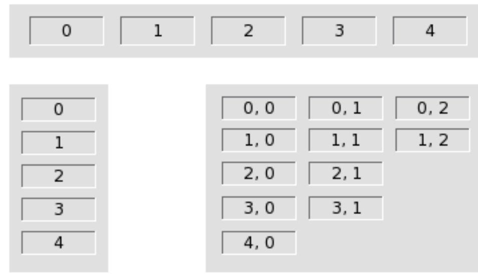

Figure 23: Widgets arranged with **QHBoxLayout**, **QVBoxLayout**, and **QGridLayout** layout managers.

In most cases, Qt's layout managers pick optimal sizes for managed widgets so that windows resize smoothly. If the defaults are insufficient, developers can refine the layout using the following mechanisms:

1. Setting a minimum size, a maximum size, or a fixed size for some child widgets.
2. Adding stretch or spacer items. These fill empty space in a layout.
3. Changing the size policies of child widgets. By calling QWidget::setSizePolicy(), programmers can fine-tune the resize behavior of a child widget. They can be set to expand, contract, or keep the same size, depending on other widgets in the layout.
4. Changing the child widgets' size hints. QWidget::sizeHint() and QWidget::minimumSizeHint() return a widget's preferred size and preferred minimum size based on its contents. Built-in widgets provide appropriate reimplementations of these functions.
5. Setting stretch factors. Stretch factors allow relative growth of child widgets; for example, allocating two thirds of any extra available space to widget A and one third to widget B.

The "spacing" between managed widgets and the "margin" around the whole layout can also be set by the programmer. By default, *Qt Designer* uses industry-standard values appropriate to the context.

Layouts can also run right-to-left and bottom-to-top. Right-to-left layouts are convenient for internationalized applications supporting right-to-left writing systems (e.g., Arabic and Hebrew). The built-in layout managers are fully integrated with Qt's style system (page 42) to provide a consistent look and feel on reversed displays.

11.2. Nested Layouts

Layouts can be nested to arbitrary levels. Figure 22 shows an example of a dialog box at two different sizes. The dialog uses three layouts: a **QVBoxLayout** groups the push buttons together, a **QHBoxLayout** groups the country list view with the push buttons, and a **QVBoxLayout** groups the "Select a country" label with the rest of the widget. A stretch item maintains the gap between the Cancel and Help buttons.

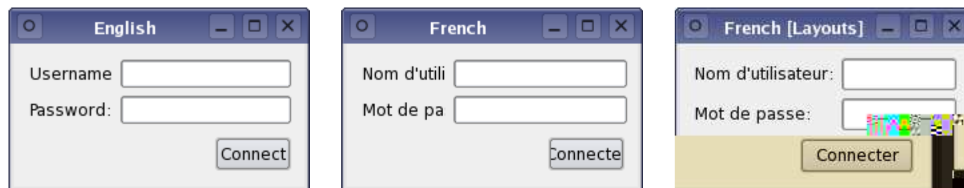

Figure 24: The effects of using layouts. In the first two pictures, the same dialog is shown containing the original English text and French text without layouts; the French text is truncated. In the right-hand picture, the labels are placed in layouts, and the French text is displayed correctly.

The dialog's widgets and layouts are created with the following code:

```
QVBoxLayout *buttonBox = new QVBoxLayout;
buttonBox->setSpacing(6);
buttonBox->addWidget(new QPushButton(tr("&OK")));
buttonBox->addWidget(new QPushButton(tr("&Cancel")));
buttonBox->addStretch(1);
buttonBox->addWidget(new QPushButton(tr("Help")));

QListWidget *countryList = new QListWidget(this);
countryList->addItem(tr("Canada"));
/* ... */
countryList->addItem(tr("United States of America"));

QHBoxLayout *middleBox = new QHBoxLayout;
middleBox->setSpacing(11);
middleBox->addWidget(countryList);
middleBox->addLayout(buttonBox);

QVBoxLayout *topLevelBox = new QVBoxLayout;
topLevelBox->setSpacing(11);
topLevelBox->setMargin(6);
topLevelBox->addWidget(new QLabel(tr("Select a country")));
topLevelBox->addLayout(middleBox);

setLayout(topLevelBox);
```

Qt makes laying out widgets so easy that programmers rarely use fixed positioning.

Developers can define custom layout managers by subclassing **QLayout**. The layout examples provided with Qt present two custom layout managers: a border layout which arranges child widgets at the points of the compass, and a flow layout which arranges widgets like words on a page. Programmers can use and modify these layouts to create new layout strategies for widgets.

Qt also includes **QSplitter**, a splitter bar that end users can manipulate. In some design situations, **QSplitter** may be preferable to a layout manager.

For complete control, it is also possible to perform layout manually in a widget by reimplementing QWidget::resizeEvent() and by calling QWidget::setGeometry() on each child widget.

Online References

http://doc.trolltech.com/4.0/layout.html

12. Styles and Themes

Qt automatically uses the native desktop style for an application's look and feel. Qt applications respect user preferences for colors, fonts, sounds, and other desktop settings. Qt programmers are free to use any of the supplied styles and can override any preferences. Programmers can modify existing styles or implement their own styles using Qt's powerful style engine.

A style implements the "look and feel" of the user interface on a particular platform. A style is a **QStyle** subclass that implements basic drawing functions such as drawing frames, buttons, and images. Qt performs all the widget drawing itself for maximum speed and flexibility.

12.1. Built-in Styles

Qt provides the following built-in styles: CDE, Motif, Mac OS X, Plastique, Windows, and Windows XP. By default, Qt uses the appropriate style for the user's platform and desktop environment. The style can also be chosen programmatically by the application developer, or by the user with the `-style` command-line option.

Figure 25: Comboboxes in the different native styles with the Windows XP style combobox selected.

A style is complemented by the user's desktop settings, which include the user's preferences for colors, fonts, sounds, etc. Qt automatically adapts to the computer's active theme. For example, Qt supports scroll and fade transition effects for menus and tooltips.

The Windows XP and Mac OS X styles are built on top of native style managers, and are available only on their respective platforms. The other styles are emulated by Qt and are available everywhere.

The default style on most modern X11 platforms is Plastique, a style inspired by the Plastik widget style for KDE that is designed to fit in on most Linux and Unix desktops.

Qt's built-in widgets are style-aware. Custom widgets and dialogs are almost always combinations of built-in widgets and layouts, and automatically adapt to the style in use. On the rare occasions when it is necessary to write a custom widget from scratch, developers can use **QStyle** to draw basic user-interface elements rather than drawing raw graphics primitives directly.

QStyle supports right-to-left languages. Based on the translation file loaded, Qt automatically uses right-to-left widget layouts rather than the default left-to-right scheme normally used. Users can run individual applications in this mode by specifying the `-reverse` command-line option. Additionally, when used in reversed mode, well-behaved styles render widgets with areas of light and shadow that are appropriate for the user's desktop environment.

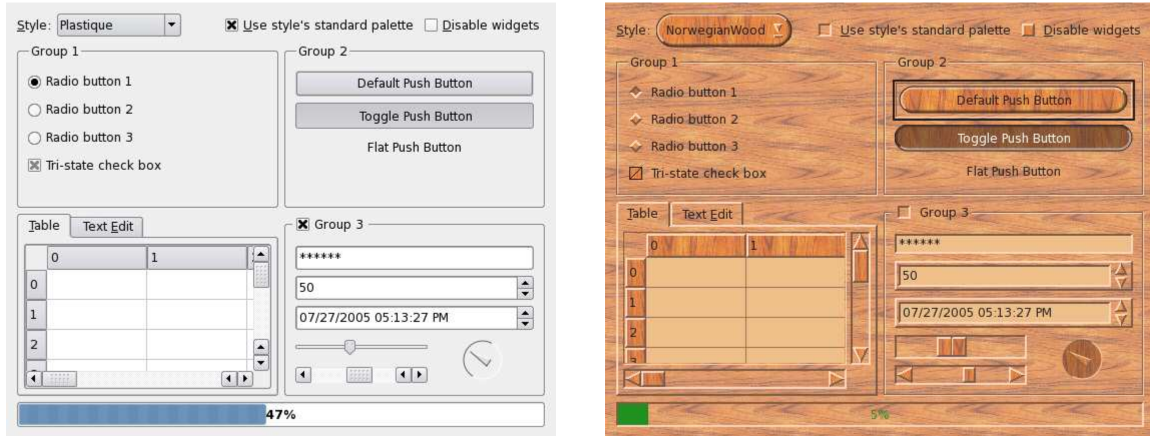

Figure 26: The Qt 4 Styles example shows the built-in styles, and includes a custom style for comparison.

12.2. Custom Styles

Custom styles are used to provide a distinct look to an application or family of applications. Custom styles can be defined by subclassing **QStyle**, **QCommonStyle**, or any of its subclasses. It is easy to make small modifications to existing styles by reimplementing one or two virtual functions from the appropriate base class.

The **QStyle** API provides information about each of the constituent components used to draw widgets, making it possible for highly customized styles to be created and fine-tuned.

Qt's platform-native styles are appropriate for most applications. However, in some cases it is necessary to override the default style in favor of a particular look and feel. Setting the application's style is as simple as this:

```
QApplication::setStyle(new MyCustomStyle);
```

A style can also be compiled as a plugin (page 51). Plugins make it possible to preview a form using a custom style in *Qt Designer* without recompiling either Qt or *Qt Designer* itself. The style of an existing Qt application can be changed using a style plugin without recompiling the application. This enables applications like the Qt 4 Styles example and the `qtconfig` tool to switch styles on-the-fly to provide previews for each of the available styles.

Online References

http://doc.trolltech.com/4.0/qt4-styles.html
http://doc.trolltech.com/4.0/widgets-styles.html
http://doc.trolltech.com/qq/qq13-styles.html

13. Events

Application objects receive system messages as Qt events. Applications can monitor, filter, and respond to events at different levels of granularity.

In Qt, an event is an object that inherits **QEvent**. Events are delivered to each **QObject** so that they can respond to them. Programmers can monitor and filter events at the application level and at the object level.

13.1. Event Creation

Most events are generated by the window system and inform an application about relevant user actions, such as key presses, mouse clicks, or when windows are resized. These events can also be simulated programmatically. There are over fifty types of event, the most common of which report mouse activity, key presses, redraw requests, and window handling operations. Developers can add their own event types that behave like the built-in events.

It is usually insufficient merely to know that a key was pressed, or that a mouse button was released. The receiver also needs to know, for example, which key was pressed, which button was released, and where the mouse was located. Each subclass of **QEvent** provides additional information relevant to the type of event, and each event handler can use this information to act accordingly.

13.2. Event Delivery

Qt delivers events by calling the virtual function QObject::event(). For convenience, QWidget::event() forwards the most common types of event to dedicated handlers, such as QWidget::mouseReleaseEvent() and QWidget::keyPressEvent(). Developers can easily reimplement these handlers when writing their own widgets, or when specializing existing widgets.

Some events are sent immediately, while others are queued, ready to be dispatched when control returns to the Qt event loop. Qt uses queueing to optimize certain types of event. For example, multiple paint events are compressed into a single event to maximize speed.

Often an object needs to look at another object's events; e.g., to respond to them or to block them. This is achieved by having a monitoring object call QObject::installEventFilter() on the object that it will monitor. The monitor's QObject::eventFilter() virtual function will be called with each event that is destined for the monitored object before the monitored object receives the event.

It's also possible to filter all the application's events by installing a filter on the unique **QApplication** instance for the application. Such filters are called before any widget-specific filters. It is even possible to reimplement QApplication::notify(), the event dispatcher, for complete control over the event delivery process.

Online References

http://doc.trolltech.com/4.0/eventsandfilters.html

14. Input/Output and Networking

Qt can load and save data in plain text, XML, and binary formats. Qt handles local files using its own classes, and remote files using the FTP and HTTP protocols. Inter-process communication and socket-based TCP and UDP networking are also fully supported.

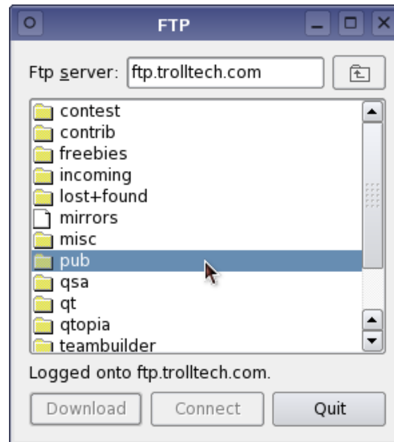

Figure 27: The Qt 4 FTP example uses the **QFtp** class to provide simple FTP browsing capabilities.

14.1. Reading and Writing Files

Qt provides classes to perform advanced I/O on multiple platforms. The **QTextStream** class has a similar interface to the standard `<iostream>` classes, and supports the encodings provided by **QTextCodec**. The **QDataStream** class is used to serialize the basic C++ types and many Qt types in a platform-independent binary format. For example, the following code writes a Unicode string, a font, and a color to the `splash.dat` file:

```
QFile outputFile("splash.dat");
if (outputFile.open(QIODevice::WriteOnly)) {
    QDataStream outputStream(&outputFile);
    outputStream << QString("SplashWidgetStyle")
                 << QFont("Times", 18, QFont::Bold)
                 << QColor("skyblue");
}
```

The data can easily be retrieved and used with the following code:

```
QString str;
QFont font;
QColor color;
QFile inputFile("splash.dat");
if (inputFile.open(QIODevice::ReadOnly)) {
    QDataStream inputStream(&inputFile);
    inputStream >> str >> font >> color;

    if (str == "SplashWidgetStyle") {
        splashWidget->setFont(font);
        splashWidget->setColor(color);
    }
}
```

The **QFile** class supports large files, long file names, and internationalized file names.

QTextStream and **QDataStream** operate on any **QIODevice** subclass. Qt also includes the **QBuffer**, **QTcpSocket**, and **QUdpSocket** subclasses, and programmers can implement their own custom devices. **QIODevice** also provides low-level functions such as readLine() and writeBlock() that can be used independently of any stream.

Directories are read and traversed using **QDir**. **QDir** can be used to manipulate path names and access the underlying file system (e.g., create a directory or delete a file). **QFileInfo** provides more detailed information about a file, such as its size, permissions, creation time, and last modification time.

The following example lists the hidden files in the user's home directory along with their size, in increasing order of size:

```
QDir dir = QDir::home();
dir.setFilter(QDir::Files | QDir::Hidden);
dir.setSorting(QDir::Size | QDir::Reversed);
QStringList names = dir.entryList();

foreach (QString name, names) {
    QFileInfo info(dir, name);
    cout << name.toLatin1().data() << " " << info.size() << endl;
}
```

Transparent access to remote files is provided by the **QHttp** and **QFtp** classes. URLs can easily be parsed and reconstructed using **QUrl**.

Some types of file can be read directly without requiring the use of a **QFile** object. For example, image files are usually read via the **QImage** class with its extensible plugin mechanism (page 27). Printing text and images is handled by **QPrinter** (page 27).

14.2. XML

Qt's XML module provides a SAX parser and a DOM parser, both of which read well-formed XML and are non-validating. The SAX (Simple API for XML) implementation follows the design of the SAX2 Java implementation, with adapted naming conventions. The DOM (Document Object Model) Level 2 implementation follows the W3C® recommendation and includes namespace support.

Many Qt applications use XML to store their persistent data. The SAX parser is used for reading data incrementally, and is especially suitable both for applications with simple parsing requirements and for those involving very large files. The DOM parser reads the entire file into a tree structure in memory that can be traversed at will.

14.3. Inter-Process Communication

The **QProcess** class is used to start external programs and to communicate with them from a Qt application in a platform-independent way. Communication is achieved by writing to the external program's standard input and potentially by reading its standard output and standard error. Since **QProcess** is a subclass of **QIODevice**, data can be streamed to and from the process with **QTextStream** and **QDataStream**.

Figure 28: A live currency convertor that uses the **QTcpSocket** class to send and receive data from a remote server.

QProcess works asynchronously, reporting the availability of data by emitting signals. Qt applications can connect to the signals to retrieve the data for processing, and optionally respond by sending data back to the external program. **QProcess** also supports a blocking mode of operation.

14.4. Networking

Qt provides a multiplatform interface for writing TCP/IP clients and servers, supporting IPv4 and IPv6. All of Qt's networking classes are reentrant and can be used from any **QThread**.

The **QTcpSocket** class provides an asynchronous buffered TCP connection. **QTcpSocket** is a **QIODevice**, making it easy to use **QTextStream** and **QDataStream** on a socket.

Both **QTcpSocket** and **QUdpSocket** are designed to work well within a GUI application, and support both blocking and non-blocking operating modes. A live currency converter application illustrates this.

The application uses the fictional protocol CCP (Currency Conversion Protocol) to access the latest exchange rates from a server. Only lines related to networking are presented.

The socket is created in the **Converter** constructor:

```
...
socket = new QTcpSocket(this);
connect(socket, SIGNAL(connected()), this, SLOT(sendSourceAmount()));
connect(socket, SIGNAL(readyRead()), this, SLOT(updateTargetAmount()));
...
```

Socket communication is asynchronous, and the socket emits the connected() signal when a connection is made, and the readyRead() signal when there is data available to read.

The convert() slot is called when the user clicks the Convert button:

```
void Converter::convert()
{
    convertButton->setEnabled(false);
    socket->connectToHost("ccp.banca-monica.nu", 1234);
}
```

This slot disables the Convert button while the conversion takes place, opens the connection, and returns immediately. When the socket connects to the server, it will emit the connected() signal.

The sendSourceAmount() slot is called when the socket has successfully connected to the server:

```
void Converter::sendSourceAmount()
{
    QString command = "CONV " + sourceAmount->text() + " " +
                      sourceCurrency->currentText() + " " +
                      targetCurrency->currentText() + "\r\n";
    socket->write(command.toLatin1().data());
}
```

This slot sends a request (e.g., `CONV 100 EUR USD`) to port 1234 on the server. **QTcpSocket** automatically resolves `ccp.banca-monica.nu` to its IP address. All these operations are non-blocking to keep the user interface responsive.

```
void Converter::updateTargetAmount() {
    if (socket->canReadLine()) {
        targetAmount->setText(socket->readLine());
        socket->close();
        convertButton->setEnabled(true);
    }
}
```

The updateTargetAmount() function is called when the server replies to the `CONV` request. It reads the reply, updates the display, closes the connection, and enables the Convert button.

Simple TCP servers can be implemented by subclassing **QTcpServer**, which works asynchronously like **QTcpSocket**. **QTcpServer** sets up a listening socket that accepts incoming connections, and calls a virtual function to serve the client. Similarly, **QUdpServer** provides a server based on **QUdpSocket**. **QTcpServer** can operate in blocking and non-blocking modes.

The **QAbstractSocket** class provides a platform-independent wrapper for native socket APIs. It provides the underlying functionality for **QTcpSocket**, **QTcpServer**, and **QUdpSocket**.

Online References

http://doc.trolltech.com/4.0/qiodevice.html
http://doc.trolltech.com/4.0/xml.html
http://doc.trolltech.com/4.0/networking.html

15. Collection Classes

Collection classes are used to store groups of items in memory. Qt provides a set of classes that are compatible with the Standard Template Library (STL), and that work regardless of whether the compiler supports STL or not. Java-style iterators are also provided for safety and convenience.

Applications often need to manage items in memory, such as groups of images, widgets, or custom objects. Many C++ compilers support the STL, which provides ready-made data structures for storing items. Qt provides lists, stacks, queues, and dictionaries with STL-syntax. Qt's collection classes even work with compilers that are not capable of supporting the STL.

Qt's rich set of portable collection classes ("containers") and associated iterators are heavily used inside Qt, and are provided as part of the Qt API. Qt's containers are optimized for speed and memory efficiency using two techniques: "private classes" and "implicit sharing". Programmers can also use STL containers on the platforms that support them, at the cost of losing Qt's optimizations.

Template classes usually increase the size of executables dramatically because the compiler generates essentially the same code for each specialized type. Qt's template collection classes are optimized for minimal code expansion since they are implemented in a thin layer over non-template private classes.

15.1. Containers

Qt provides five sequential containers that can be used to hold either values or pointers: **QList<T>**, **QLinkedList<T>**, **QVector<T>**, **QStack<T>**, and **QQueue<T>**. They have an interface very similar to the STL containers and are fully compatible with the STL algorithms. Qt provides some STL-equivalent algorithms: qCopy(), qFind(), qSort(), etc. On platforms with STL support, Qt provides automatic conversion operators between STL and Qt containers.

Additionally, Qt provides Java-style iterators for developers who are more familiar with Java containers than the STL.

Qt provides five associative containers: **QMap<Key,T>**, **QHash<Key,T>**, **QSet<T>**, **QMultiHash<Key,T>**, and **QMultiMap<Key,T>**. The "Hash" containers use a hash function to improve search performance.

Qt's sequential and associative collection classes can be used to store both value-based and pointer-based types, making them especially useful for handling **QWidget** and **QObject** pointers. When used to hold pointer-based items, convenience functions can be used to delete the contents of collections in one pass before the collection is destroyed.

15.2. Implicit Sharing

When used with Qt's value classes, the items held in these collection classes are implicitly shared ("copy on write"). Copies of these classes share the same data in memory. The

data sharing is handled automatically; if the application modifies the contents of one of the copied objects, a deep copy of the data is made so that the other objects are left unchanged. When an object is copied, only a pointer is passed and a reference count incremented, which is much faster than actually copying the data, and also saves memory.

Sharing is used wherever it makes sense: in Qt's value-based collection classes, and in other commonly-used classes such as **QBrush**, **QFont**, **QIcon**, **QPalette**, **QPen**, **QPixmap**, **QRegExp**, and **QString**. Programmers can safely and efficiently copy objects of these classes by value, avoiding the risks related to optimizing pointer-based code by hand. In particular, the implicitly shared **QString** class makes string processing easy and fast.

Qt also provides the low-level **QBitArray** and **QByteArray** classes. These classes are very efficient for handling basic data types.

Online References

http://doc.trolltech.com/4.0/containers.html
http://doc.trolltech.com/4.0/shclass.html

16. Plugins and Dynamic Libraries

Qt applications can access functions from dynamic libraries using a platform-independent API. Qt also supports plugins, allowing developers to create and distribute codecs, database drivers, image format converters, styles, and custom widgets as separate components.

16.1. Plugins

Converting a Qt component into a plugin is achieved by subclassing the appropriate plugin base class, implementing a few simple functions, and adding a macro. For example, a **QStyle** subclass called **CopperStyle** can be made available as a plugin in the following way:

```
class CopperStylePlugin : public QStylePlugin
{
public:
    QStringList keys() const {
        return QStringList() << "CopperStyle";
    }
    QStyle *create(const QString &key) {
        if (key == "CopperStyle")
            return new CopperStyle;
        return 0;
    }
};
Q_EXPORT_PLUGIN(CopperStylePlugin)
```

The new style can be set like this:

```
QApplication::setStyle(QStyleFactory::create("CopperStyle"));
```

Components supplied as plugins are detected and used by the application automatically. Many third parties provide Qt components in source form, as precompiled dynamic libraries, and as plugins.

16.2. Dynamic Libraries

The **QLibrary** class provides a cross-platform API for loading dynamic libraries. Below is an example of the most basic way to dynamically load and use a library. The example attempts to obtain a pointer to the `print_str` function from the `mylib` library (`mylib.dll` on Windows, `mylib.so` on Unix).

```
typedef void (StrFunc)(const char *str);
QLibrary lib("mylib");
StrFunc *func = (StrFunc *) lib.resolve("print_str");
if (func) func("Hello world!");
```

Calling a function this way is not type-safe, and only symbols with C linkage are supported.

Online References

http://doc.trolltech.com/4.0/plugins-howto.html

17. Building Qt Applications

Qt developers can take advantage of a suite of tools to simplify the process of building applications on all supported platforms. Applications, libraries, and plugins are described by project files that are processed to produce suitable Makefiles for each platform.

17.1. Qt's Build System

Projects are described by `.pro` files that contain terse, but readable descriptions of source and header files, *Qt Designer* forms, and other resources. These are processed by the `qmake` tool to produce suitable Makefiles for the project on each platform.

All of the Qt libraries, tools, and examples are described by project files. For example, the Qt 4 HTTP example can be described in just the following three lines:

```
HEADERS += httpwindow.h
SOURCES += httpwindow.cpp main.cpp
QT += network
```

The first two definitions inform `qmake` about the header and source files needed to build the example; the last one ensures that Qt's networking library is used. The project file syntax also lets developers fine-tune the build process with configuration options, and write conditional build rules for different deployment situations.

Project files can also be used to describe projects that are organized within a deep directory tree. For example, Qt's examples are located in a directory tree within a top-level `examples` directory. The `examples.pro` file instructs `qmake` to descend into directories for each category of examples with the following lines:

```
TEMPLATE = subdirs
SUBDIRS = dialogs draganddrop itemviews layouts linguist \
          mainwindows network painting richtext sql \
          threads tools tutorial widgets xml
```

Support for conditional builds means that the Windows-specific examples are only built when compiling a suitable edition of Qt on an appropriate supported platform:

```
win32:!contains(QT_EDITION, OpenSource|Console):SUBDIRS += activeqt
```

When `qmake` is used to build a project, all the enhanced features of Qt are automatically handled by the other tools in the build suite: `moc` (page 12) processes the header files to enable signals and slots, `rcc` compiles the specified resources, and `uic` is used to create code from user interface forms created with *Qt Designer* (page 20).

Precompiled header support, `pkg-config` integration, the ability to generate Visual Studio project files, and other advanced features allow developers to take advantage of platform-specific tools while retaining the use of a cross-platform build system for common project components.

Developers using the Desktop or Desktop Light editions of Qt can also use `qmake`'s project files from within Microsoft Visual Studio. On Mac OS X, support for project files from within Apple's Xcode is provided as standard with all Qt editions.

17.2. Qt's Resource System

Qt provides a resource system that allows data files to be stored inside executables, so that any resources required by applications can be accessed at run-time. Qt's widgets support a naming scheme that allows developers to directly refer to these packaged resources.

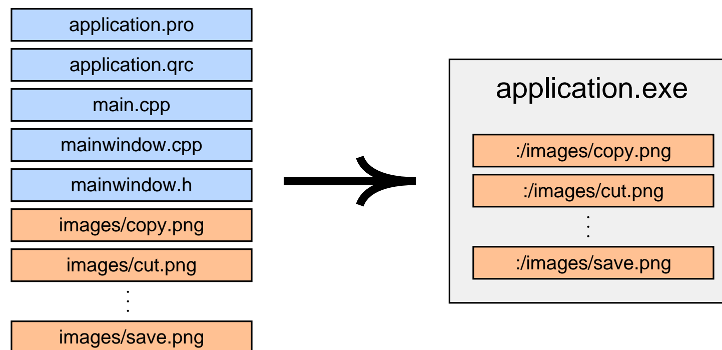

Figure 29: In this example, a set of images are packaged with the application when it is built. The application references them using the naming scheme shown.

The resources to be packaged with an application are listed in a `.qrc` (Qt Resource Collection) file, containing a list of files in the build directory along with the resource paths that are used in the application. These files are processed using `rcc` to create data that is compiled into the application. This approach ensures that certain critical resources are always available to applications, avoiding possible distribution and installation problems.

Online References

http://doc.trolltech.com/4.0/qmake-manual.html
http://doc.trolltech.com/4.0/resources.html
http://www.trolltech.com/products/qt/vs-integration.html
http://www.trolltech.com/products/qt/mac.html

18. Qt's Architecture

Qt's functionality is built on the low-level APIs of the platforms it supports. This makes Qt flexible and efficient, and enables Qt applications to fit in with single-platform applications.

Qt is a cross-platform framework which uses native style APIs to accurately follow the human interface guidelines on each supported platform. All widgets are drawn by Qt, and programmers can extend or customize them by reimplementing virtual functions. Qt's widgets accurately emulate the look and feel of the supported platforms, as described in Styles and Themes (page 42). This technique also enables developers to derive their own custom styles to provide a distinct look and feel for their applications.

Qt uses the low-level APIs of the different platforms it supports. This differs from traditional "layered" cross-platform toolkits that are thin wrappers over single-platform toolkits (e.g., MFC on Windows and Motif on X11). Layered toolkits are usually slow, since every function call to the library results in many additional calls down through the different API layers. Layered toolkits are often restricted by the features and behavior of the underlying toolkits, leading to obscure bugs in applications.

Qt is professionally supported, and takes advantage of the available platforms: Microsoft Windows, X11, Mac OS X, and Embedded Linux. Using a single source tree, a Qt application can be compiled to an executable for each target platform. Although Qt is a cross-platform framework, customers have found it to be easier to learn and more productive than many platform-specific toolkits. Many customers use Qt for single-platform development, preferring Qt's fully object-oriented approach.

18.1. X11

Qt/X11 uses Xlib to communicate with the X server directly. Qt does not use Xt (X Toolkit), Motif, Athena, or any other toolkit.

Qt supports the following versions of Unix: AIX®, FreeBSD®, HP-UX, Irix®, Linux, NetBSD, OpenBSD, and Solaris. See the Trolltech web site for an up-to-date list of supported compilers and operating system versions.

Qt applications automatically adapt to the user's window manager or desktop environment, and have a native look and feel under Motif, CDE, GNOME™, and KDE™. This contrasts with most other Unix toolkits, which lock users into their own look and feel.

Qt provides full Unicode support (page 36). Qt applications automatically support both Unicode and non-Unicode fonts. Qt combines multiple X fonts to render multi-lingual text. Qt's font handling is intelligent enough to search all the installed fonts for characters unavailable in the current font.

Qt takes advantage of X extensions where they are available. Qt supports the RENDER extension for anti-aliased and alpha-blended fonts and vector graphics. Qt provides on-the-spot editing for X Input Methods. Qt supports multiple screens both with traditional multi-head and with Xinerama.

Qt Application Source Code		
Qt API		
Qt/Windows	Qt/X11	Qt/Macintosh
GDI	X Windows	Carbon
Windows	Unix/Linux	Mac OS X

Figure 30: An overview of Qt's architecture on supported desktop platforms.

18.2. Microsoft Windows

Qt/Windows uses the Win32® API and GDI for events and drawing primitives. Qt does not use MFC or any other toolkit. In particular, Qt does not use the inflexible "common controls," but rather provides its own more powerful, customizable widgets. (For non-specialized uses, Qt uses the native Windows file and print dialogs.)

Qt/Windows customers can create Qt applications using Microsoft Visual C++® and Borland C++ that will run on Windows 95, 98, NT4, ME, 2000, and XP.

Qt performs a run-time check for the Windows version, and uses the most advanced capabilities available. For example, only Windows NT4, 2000, and XP support rotated text natively; Qt renders rotated text on all Windows versions, and uses the native support where available. Qt developers are insulated from differences in the Windows APIs.

Qt supports the Microsoft accessibility interfaces. Unlike the common controls on Windows, Qt widgets can be extended without losing the accessibility information of the base widget. Custom widgets can also provide accessibility.

Qt also supports multiple screens on Microsoft Windows.

18.3. Mac OS X

Qt supports Mac OS X using the Carbon® API.

Qt/Mac introduces layouts and straightforward internationalization support, standardized access to OpenGL, and powerful visual design with *Qt Designer*. Qt handles files and asynchronous socket input/output in the event loop. Qt provides solid database support. Developers can create Macintosh applications using a modern object-oriented API that includes comprehensive documentation and full source code.

Macintosh developers can create applications on their favorite platform and broaden their market hugely by simply recompiling on the other supported platforms.

Online References

http://www.trolltech.com/products/platforms/

19. Platform Specific Extensions and Qt Solutions

In addition to being complete in itself, Qt provides some platform-specific extensions to assist developers in certain contexts. The ActiveQt extension allows developers to use ActiveX controls within their Qt applications, and also allows them to make their Qt applications into ActiveX servers. Other platform-specific extensions are made available through Qt Solutions, a service made available to commercial Qt licensees.

In addition to the ActiveQt extension outlined below, there are additional extensions available from third party suppliers. For example, there is Tq from *froglogic*™ which provides Tcl/Tk integration, and a Microsoft Windows resource converter is available from Klarälvdalens Datakonsult.

19.1. ActiveX Interoperability

ActiveX is built on Microsoft's COM technology. It allows applications and libraries to use components provided by component servers, and to be component servers in their own right. The Qt/Windows ActiveQt module allows developers to turn their applications into ActiveX servers, and to make use of the ActiveX controls provided by other applications.

Integration with Microsoft's .NET™ technology is also possible with ActiveQt. Applications can use ActiveQt's COM support to automatically give .NET developers access to Qt widgets and data types.

ActiveQt seamlessly integrates ActiveX into Qt: ActiveX properties, methods, and events become Qt properties, slots, and signals. This makes it straightforward for Qt developers to work with ActiveX using a familiar programming paradigm, and insulates them from all the different kinds of generated code that is normally part of an ActiveX implementation.

Here's how to register Internet Explorer for use as an ActiveX component:

```
#define CLSID_InternetExplorer "{8856F961-340A-11D0-A96B-00C04FD705A2}"

QAxWidget *activeX = new QAxWidget(this);
activeX->setControl(CLSID_InternetExplorer);
```

If we want to track the user's use of the component, we could watch how its title changes:

```
connect(activeX, SIGNAL(TitleChange(const QString &)),
        this, SLOT(setWindowTitle(const QString &)));
```

ActiveQt automatically handles the conversions between ActiveX and Qt data types. ActiveQt supports the dynamicCall() function to control an ActiveX component through the control's **IDispatch** interface implementation:

```
activeX->dynamicCall("Navigate(const QString &)", "http://doc.trolltech.com");
```

Turning a Qt application into an ActiveX server is simple. If we only need to export a single class, little more is required than the inclusion of the `qaxfactory.h` header and a suitable `QAXFACTORY_DEFAULT` macro. Once the class is compiled, its properties, slots, and signals become ActiveX properties, methods, and events to ActiveX clients. ActiveQt also provides the QAxFactory::isServer() function that can be called to determine if the application is

being run in its own right or being used as an ActiveX control, so that developers can control which functionality is available in which context.

19.2. Qt Solutions

In addition to all the classes supplied with Qt, Trolltech also produces Qt Solutions, an optional service available to Qt licensees either at the time of purchase or as an add-on product. Qt Solutions offers a regularly updated set of components and widgets, many of which are available under the same dual licensing scheme as Qt. An increasing number of Solutions made available to Qt 3 developers are also available for Qt 4, and new Solutions are planned for the future.

Online References

http://doc.trolltech.com/4.0/activeqt.html
http://www.trolltech.com/products/solutions

 © 2005 Trolltech

20. The Qt Development Community

Companies and independent developers from around the world are joining the Qt development community every day. They have recognized that Qt's architecture lends itself to rapid application development. These developers, whether they are targeting one or many platforms, benefit from Qt's consistent and straightforward API, powerful build system, and supporting tools such as Qt Designer.

Qt has an active and helpful user community who communicate using the

Index

accelerator, 15
action, 14, 15
action system, *see* actions
ActiveQt, 56
ActiveX, 56
AIX, 54
alpha-blending, 54
analog clock, 7
anti-aliasing, 8
Apple
    Xcode, 20, 24, 52
application, 13
architecture, 4, 54, 55
Athena, 54

books, 58
browser, 20

callback, 10
Carbon, 55
CDE, 54
CFPreferences, 18
collection class, 49
color, 26
    HSV, 26
    RGB, 26
COM, 56
communication, 10
    inter-process, 46
community, 58
compiler, 12
Components, 51
connect(), 10
connection, 10
container class, 49
    hash, 49
    implicit sharing, 49
    Java-style iterator, 49
    STL, 49
    STL iterator, 49
cross-platform, 54

database, 33
    DB2, 33
    Interbase, 33
    MySQL, 33
    Oracle, 33
    PostgreSQL, 33
    SQL Server, 33
    SQLite, 33
    Sybase, 33
dialog, 5, 16
    file, 16
    modal, 16
    modeless, 16
    semi-modal, 16
dock windows, 15
documentation, 20, 58
dynamic libraries, 51

encoding
    Big5, 36
    EUC-JP, 36
    GBK, 36
    ISO-8859, 36
    JIS, 36
    KOI8-R, 36
    Shift-JIS, 36
event, 8, 44
example
    application, 22
    currency convertor, 47
    signals and slots, 11

file
    reading, 18, 45
    writing, 18, 45
form
    designing, 20
FreeBSD, 54
FTP, 45

GNOME, 54
graphics, 26
    3D, 28
    image formats, 27
GUI, 13, 22

help, 17, 20
    interactive, 15
    online, 17
    What's This?, 17
HP-UX, 54
HTML, 22

i18n, *see* internationalization
image, 27

indexing, 20
input methods, 54
interactive, 17
interface
  graphical user, 13
  guidelines, 54
  multiple document, 13, 18
  single document, 13
interfaces
  dynamic, 20
internationalization, 14, 36
introspection, 12
Irix, 54
item view, 29

KDE, 54
KDevelop, 25
keyboard, 15

layout, 20, 39
  built-in, 39
  nested, 40
  right-to-left, 40
layouts, 20
Linux, 5, 54

Mac OS X, 55
mailing list
  qt-interest, 58
main, 14
main window, 20
Mandelbrot, 19
MDI, 13, 18
menu, 14
menus, 20
meta-object, 12, 19
MFC, 54
Microsoft
  .NET, 56
  Visual Studio, 20, 24, 52
  Windows, 55
moc, 12, 52
Model-View-Controller, 30
model/view, 30, 33
  SQL model, 34
Motif, 54
multithreading, *see* threading
mutexes, 19

NetBSD, 54
networking, 47

object-oriented, 10
ODBC, 33
OpenBSD, 54
OpenGL, 28

paint, 8
paintEvent(), 8
painting, 26
  Bezier curves, 26
  path, 26
  text, 26
Plastik, 42
Plastique, 5, 16, 17, 42
plugin, 25, 27, 51
popup, 15
progress bar, 16
properties, 20
property, 12

QAbstractItemModel, 30
QAbstractItemView, 30
QAbstractSocket, 48
QAction, 15
QApplication, 44
QAssistantClient, 17, 22
QBitArray, 50
QBrush, 50
QBuffer, 46
QByteArray, 50
QChar, 36
QCheckBox, 5
QColor, 26
QComboBox, 5, 15
QCommonStyle, 43
qCopy(), 49
QDataStream, 45–47
QDialog, 17
QDir, 46
QDockWidget, 15
QEvent, 44
QFile, 46
QFileDialog, 16, 17
QFileInfo, 46
qFind(), 49
QFont, 50
QFontDialog, 17
QFormBuilder, 24
QFtp, 45, 46
QGLWidget, 28
QGridLayout, 39
QGroupBox, 5

QHash<Key,T>, 49
QHBoxLayout, 39, 40
QHttp, 46
QIcon, 50
QImage, 27, 28, 46
QIODevice, 46, 47
QLayout, 41
QLineEdit, 5
QLinkedList<T>, 49
QList<T>, 49
QListView, 29
QListWidget, 29
QMainWindow, 14, 18
qmake, 12, 52
QMap<Key,T>, 49
QMenu, 14
QMenuBar, 14
QMessageBox, 16
QMultiHash<Key,T>, 49
QMultiMap<Key,T>, 49
QObject, 10, 11, 37, 44, 49
QPainter, 26
QPalette, 50
QPen, 50
QPicture, 28
QPixmap, 27, 28, 50
QPrintDialog, 31
QPrinter, 28, 31, 46
QProcess, 46
QProgressDialog, 16
QPushButton, 5
QQueue<T>, 49
QRadioButton, 5
qrc, *see* rcc
QRegExp, 50
QSA, 58
QScrollArea, 5
QSet<T>, 49
QSettings, 18
QSlider, 5
qSort(), 49
QSpinBox, 5, 15
QSplitter, 41
QSqlQuery, 33
QSqlQueryModel, 35
QSqlRelationalTableModel, 35
QSqlTableModel, 35
QStack<T>, 49
QString, 36, 50
QStyle, 42, 43, 51
Qt Assistant, 17, 20
Qt Designer, 5, 16, 17, 20, 37, 39, 43
Qt Forum, 58
Qt Linguist, 16, 36–38
Qt Quarterly, 58
Qt Script for Applications, 58
Qt Solutions, 57
Qt/Mac, 55
Qt/Windows, 55
Qt/X11, 54
QTableView, 29
QTableWidget, 29
QTabWidget, 5
qtconfig, 43
QTcpServer, 48
QTcpSocket, 46, 47
QtDesigner, 25
QTextBrowser, 31
QTextCodec, 36, 45
QTextDocument, 31
QTextEdit, 14, 22, 31
QTextLayout, 32
QTextStream, 45–47
QToolButton, 15
QToolTip, 17
QTranslator, 37
QTreeView, 29
QTreeWidget, 29
QUdpServer, 48
QUdpSocket, 46–48
QUrl, 46
QVBoxLayout, 39, 40
QVector<T>, 49
QWhatsThis, 17
QWidget, 5, 7, 17, 28, 49
QWorkspace, 14, 18

rcc, 52, 53
RENDER, 54
reusability, 10

SDI, 13
semaphores, 19
serialization, 18
settings, 18
signal, 10
signals, 19
slot, 10
slots, 19
Solaris, 54
SQL, 30, 33

STL, *see* collection class
style, 42, 54
    custom, 43
system
    build, 12, 52
    resource, 53

TCP/IP, 47
    IPv4, 47
    IPv6, 47
templates
    form, 20
text
    bidirectional, 37
    editing, 31
    entry, 5, 36
    rich, 22, 32
theme, 42, 54
thread-global storage, 19
threading, 19
threads, 19
timer, 7
toolbar, 15
toolbars, 20
tooltip, 15, 17
tr(), 14, 37
translation, 12, 36–38

uic, 22, 52
Unicode, 36
Unix, 54
URL, 46

W3C, 46
What's This?, 17
widget, 5
    built-in, 5
    central, 18
    custom, 7
widgets, 20

X11, 54
XIM, 54
Xinerama, 54
XML, 46
    DOM, 46
    SAX, 46
Xt, 54

Qt, the Qt logo, Qtopia, the Qtopia logo, Trolltech and the Trolltech logo are registered trademarks of Trolltech AS and/or its subsidiaries in the U.S. and other countries. Additional company and product names are the property of their respective owners and may be trademarks or registered trademarks of the individual companies and are respectfully acknowledged.

Trolltech AS operates a policy of continuous development. Therefore, we reserve the right to make changes and improvements to any of the products described herein without prior notice. All information contained herein is based upon the best information available at the time of publication. No warranty, express or implied, is made about the accuracy and/or quality of the information provided herein. Under no circumstances shall Trolltech AS be responsible for any loss of data or income or any direct, special, incidental, consequential or indirect damages whatsoever.

Copyright © 2005 Trolltech AS. All rights reserved.