Code less.
Create more.
Deploy everywhere.

Qt

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 and Vista, 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 and Windows CE platforms. Qt introduces a unique inter-object communication mechanism called "signals and slots". Qt has excellent cross-platform support for multimedia and 3D graphics, internationalization, SQL, XML, unit testing, 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

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®, Google®, 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.qtsoftware.com/.

1.1. Executive Summary

Qt includes a rich set of widgets ("controls" in Windows terminology) that provide standard GUI functionality (see page 5). Qt introduces an innovative alternative for inter-object communication, called "signals and slots" (see page 6), that replaces the old and unsafe callback technique used in many legacy frameworks. Qt also provides a conventional event model (page 43) for handling mouse clicks, key presses, and other user input. Qt's cross-platform GUI applications (page 8) can support all the user interface functionality required by modern applications, such as menus, context menus, drag and drop, and dockable toolbars. Desktop integration features (page 14) provided by Qt can be used to extend applications into the surrounding desktop environment, taking advantage of some of the services provided on each platform.

Qt also includes *Qt Designer* (page 15), a tool for graphically designing user interfaces. *Qt Designer* supports Qt's powerful layout features (page 6) 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 multimedia and 3D graphics (page 18). Qt is the *de facto* standard GUI framework for platform-independent OpenGL® programming. Qt's painting system offers high quality rendering across all supported platforms. A sophisticated canvas framework (page 21) enables developers to create interactive graphical applications that take advantage of Qt's advanced painting features.

Qt makes it possible to create platform-independent database applications using standard databases (page 30). 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 40). From a single source tree, recompilation is all that is required to produce applications for Windows® 98 to XP®and Windows Vista™, Mac OS X®, Linux®, Solaris™, HP-UX™, and many other versions of Unix® with X11™. Qt's `qmake` build tool produces makefiles or `.dsp` files appropriate to the target platform (page 50).

Since Qt's architecture takes advantage of the underlying platform, many customers use Qt for single-platform development on Windows, Mac OS X, and Unix because they prefer Qt's approach. Qt includes support for important platform-specific features, such as ActiveX® on Windows, and Motif™ on Unix. See the section on Qt's Architecture (page 53) for more information.

Qt uses Unicode™ throughout and has considerable support for internationalization (page 33). Qt includes *Qt Linguist* and other tools to support translators. Applications can easily use and mix text in Arabic, Chinese, English, Hebrew, Japanese, Russian, and other languages supported by Unicode.

Qt includes a variety of domain-specific classes. For example, Qt has an XML module (page 44) that includes SAX and DOM classes for reading and manipulating data stored in XML-based formats. Objects can be stored in memory using Qt's STL-compatible collection classes (page 47), and handled using styles of iterators used in Java® and the C++ Standard Template Library (STL). Local and remote file handling using standard protocols are provided by Qt's input/output and networking classes (page 44).

Qt applications can have their functionality extended by plugins and dynamic libraries (page 49). Plugins provide additional codecs, database drivers, image formats, styles, and widgets. Plugins and libraries can be sold as products in their own right.

The QtScript module (see page 37) enables applications to be scripted with Qt Script, an ECMAScript-based language related to JavaScript. This technology allows developers to give users restricted access to parts of their applications for scripting purposes.

Qt is a mature C++ framework that is widely used around the world. In addition to Qt's many commercial uses, the Open Source edition of Qt is the foundation of KDE, the Linux desktop environment. Qt makes application development a pleasure, with its cross-platform build system, visual form design, and elegant API.

Online References

http://www.qtsoftware.com/qt-in-use
http://www.qtsoftware.com/about/partners

2. Graphical User Interfaces

Qt provides a rich set of standard widgets that can be used to create graphical user interfaces for applications. Layout managers are used to arrange and resize widgets to suit the user's screen, language and fonts.

Widgets are visual elements that are combined to create user interfaces. Buttons, menus and scroll bars, message boxes and application windows are all examples of widgets.

Layout managers (see page 6) organize child widgets within their parent widget's area. They perform automatic positioning and resizing of child widgets, provide sensible minimum and default sizes for top-level widgets, and automatically reposition widgets when their contents change.

Signals and slots (see page 6) connect application components together so that they can communicate in a simple, type-safe way. This form of inter-object communication is enabled in all standard widgets and can be used by developers in their own custom widgets.

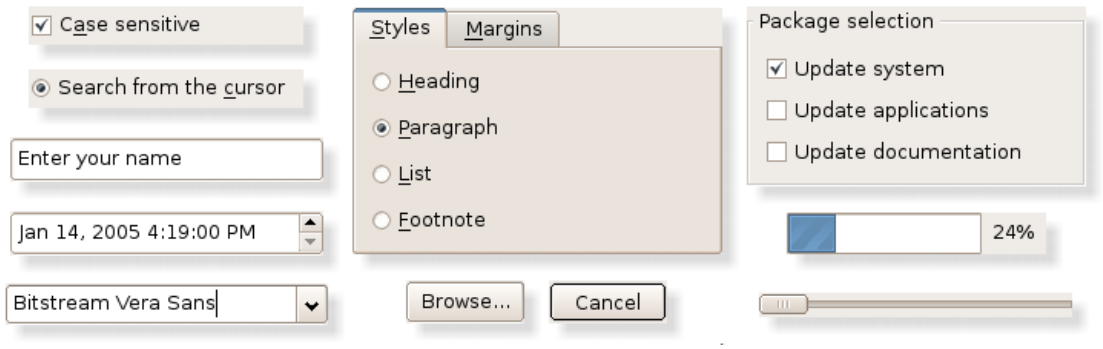

Figure 1: A selection of widgets provided by Qt.

2.1. Widgets

The images above present a selection of widgets. These include standard input widgets like **QLineEdit** for one-line text entry, **QCheckBox** for enabling and disabling simple independent settings, **QDateTimeEdit** and **QSlider** for specifying quantities, **QRadioButton** for enabling and disabling exclusive settings, and **QProgressBar**. Clickable buttons are provided by **QPushButton** and **QToolButton**.

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 page 26).

Container widgets such as **QTabWidget** and **QGroupBox** are also shown. These widgets are managed specially in *Qt Designer* to help designers create new user interfaces. More complex widgets, such as **QScrollArea**, are often used more by developers than by user interface designers because they are used to display specialized or dynamic content.

User interfaces can easily be written by hand, using widgets like **QWidget** or **QFrame** to group collections of standard widgets together, and employing layouts to arrange them appropriately

```
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();
```

Figure 2: Qt is supplied with examples that show how to make custom widgets.

in the available space.

Developers can create their own widgets and dialogs by subclassing **QWidget** or one of its subclasses. Specialized widgets like these can be completely customized to render their own content, respond to user input, and provide their own signals and slots.

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

2.2. Layouts

Layouts provide flexibility and responsiveness to user interfaces, enabling them to adapt when their styles, orientations or text fonts are updated.

Layouts help developers to support internationalization in their applications. With fixed sizes and positions, translated text is often truncated; 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.

Layouts can also run from right-to-left and from 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 layouts are fully integrated with Qt's style system (see page 40) to provide a consistent look and feel on reversed displays.

Qt Designer (see page 15) is fully able to use layouts to position widgets.

2.3. Signals and Slots

Widgets emit signals when events occur. For example, a button will emit a "clicked" signal when it is clicked. A developer 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.

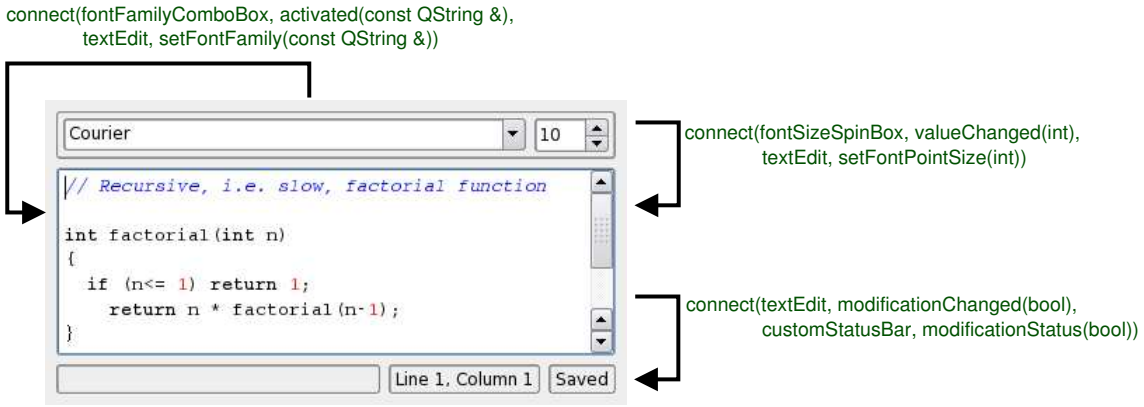

Figure 3: An example of signals and slots connections.

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 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 50). 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/4.5/gallery.html
http://doc.trolltech.com/4.5/examples.html#widgets
http://doc.trolltech.com/4.5/layout.html
http://doc.trolltech.com/4.5/object.html
http://doc.trolltech.com/4.5/signalsandslots.html

3. Application Features

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 features necessary to create modern GUI applications with menus, toolbars and dock windows. Qt supports both SDI (single document interface) and MDI (multiple document interface). Qt also supports drag and drop and the clipboard.

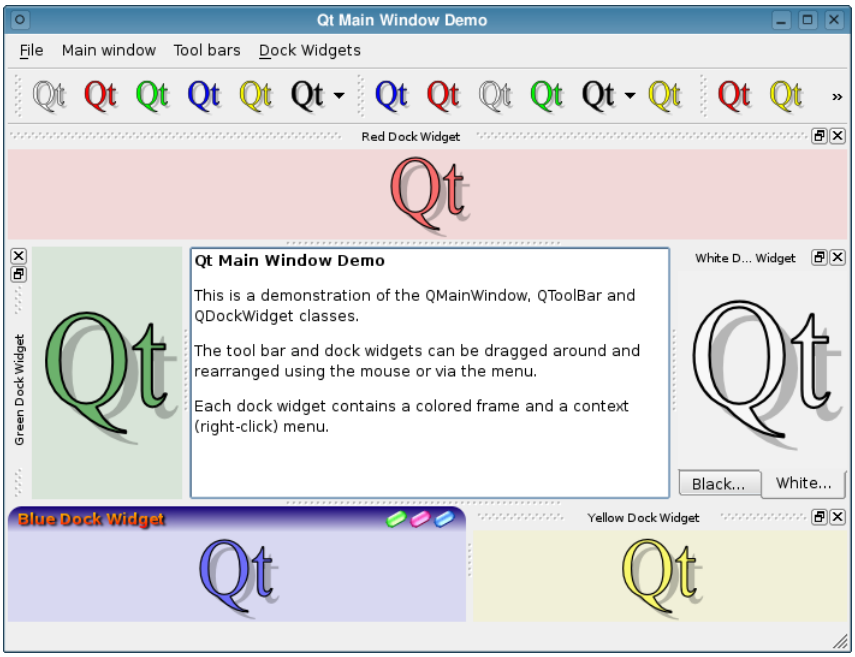

Figure 4: The Qt Main Window demonstration shows an application main window with main menus, toolbars, and dock windows arranged around a central widget.

Qt uses actions (see page 10) to simplify user interface 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.

Message boxes, wizards, and a full set of standard dialogs are provided in order 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.

Support for multithreading programming is provided by a collection of classes that represent common constructs, making it possible to write Qt applications that take advantage of threads to perform calculations, long duration tasks, or just to improve responsiveness.

Applications can also use Qt's desktop integration features to interact with services provided by the user's desktop environment.

3.1. Main Window Features

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. For MDI applications, the central area will usually be occupied by a **QMdiArea** widget which is populated with **QMdiSubWindow** widgets.

Menus

The **QMenu** widget presents menu items to the user in a vertical list. Menus can be standalone (e.g., a context pop-up menu), can appear in a menu bar, or can be a sub-menu of another pop-up 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") and cause their associated slots to be executed when selected by the user.

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.

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

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 pop-up 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 or detach them from the main window completely. Toolbars can contain widgets of almost any type; for example, **QComboBox** and **QSpinBox** widgets are often used.

Figure 5: Unified toolbar support on Mac OS X improves the look and feel of applications by blending adjacent toolbars and window title bars together.

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 toolbar button with an appropriate icon), and as an accelerator (Ctrl+S). The **QAction** class encapsulates this concept. It allows programmers to define an action in one place.

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.

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). Dock areas can also be nested to allow dock windows to be stacked in multiple rows or columns.

The main window's dock window handling includes animations that give the user feedback about where dock windows can be docked into the application. These animations are enabled by default, but they can be disabled if required.

Dock windows can be displayed with vertical title bars, and they can also share areas – when this occurs, the dock widgets are held in tabs. Dock windows can also be given individually-styled title bars and window controls (see Figure 6 above).

Some applications, including *Qt Designer* (see page 15) and *Qt Linguist* (page 35), 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.

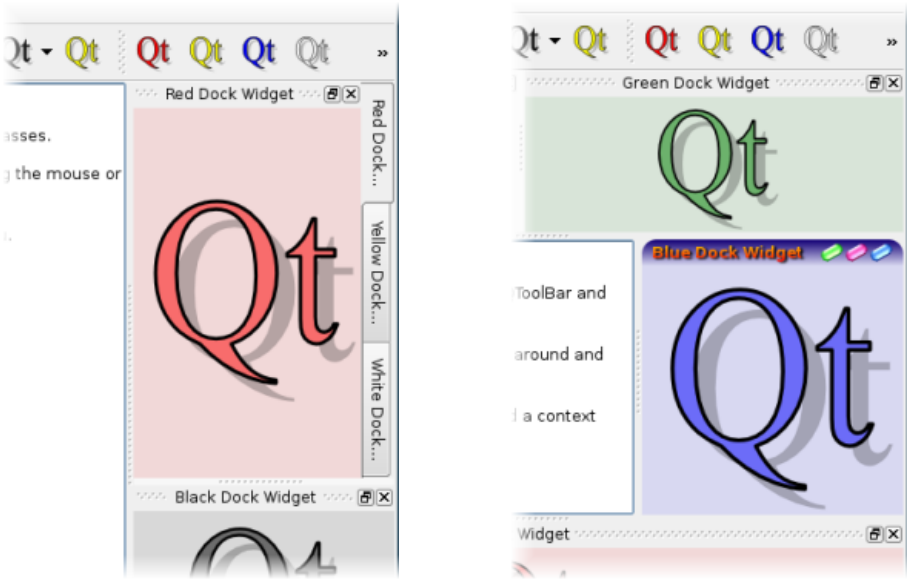

Figure 6: A dock area (left) containing three dock windows, each of which is held in a vertical tab; a dock window (right) with a custom title bar and window controls.

Multiple Document Interface

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

Child widgets of **QMdiArea** are provided by the **QMdiSubWindow** class, and can contain widgets of any type. They are rendered with a frame similar to the frame around top-level widgets, and provide buttons for functions to show, hide, maximize and set the window title; these work in the same way for **QMdiSubWindow** widgets as for ordinary windows.

QMdiArea 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.

3.2. 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 other windows 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.

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 Mac OS X.

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.

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.

3.3. 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. 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.

The **QToolTip** and **QWhatsThis** classes can also be used to implement more specialized behavior, such as providing dynamic tooltips that display different text depending on the position of the mouse within a widget.

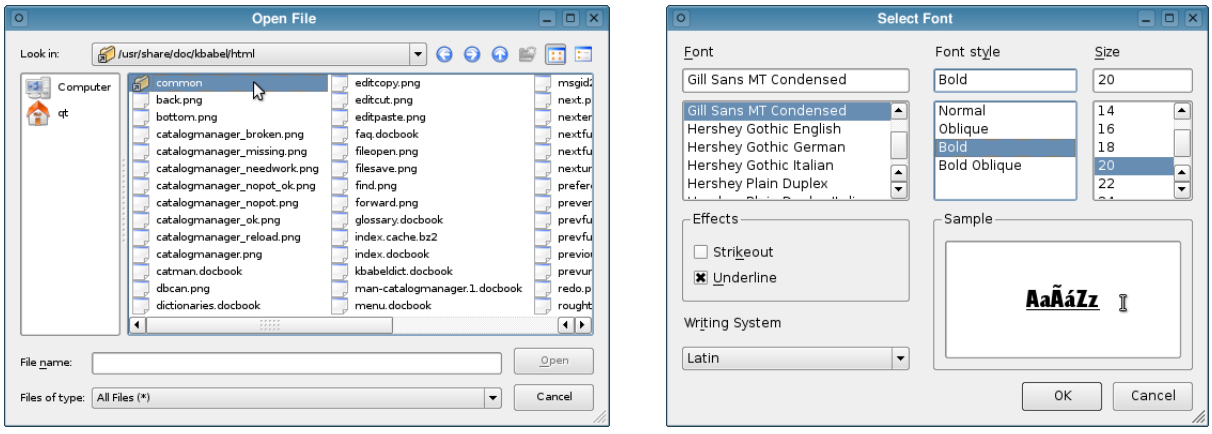

Figure 7: A **QFileDialog** and a **QFontDialog** shown in the Plastique style. On Windows and Mac OS X, native dialogs are used instead.

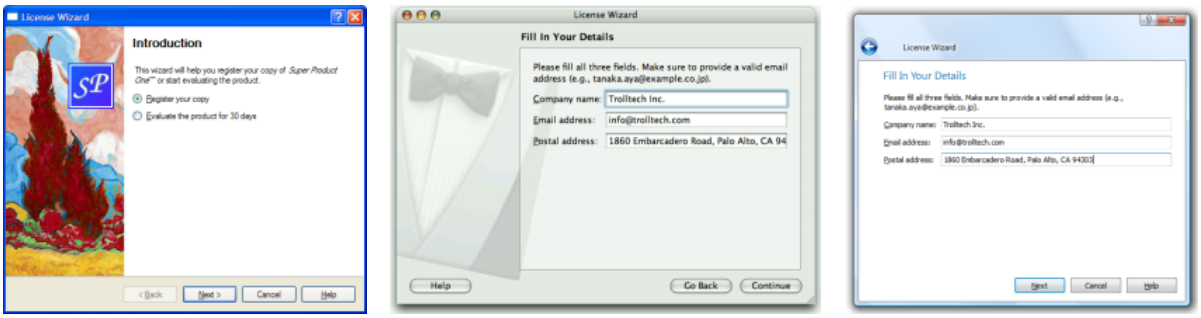

Figure 8: The **QWizard** class provides an intuitive API for creating native-looking wizards on each supported platform.

3.4. Wizards

Wizards are used to guide users through common tasks and processes, taking them step by step through the available options and providing help where necessary. Qt provides a flexible, yet intuitive API for building wizards with the appropriate native look and feel on each supported platform.

A typical wizard consists of a **QWizard** object and one or more **QWizardPage** objects that are assigned identifiers for navigation purposes.

The **QWizard** class provides features for customizing the appearance of the wizard beyond the basic platform-specific look and feel. Instances of this class also control the order in which pages are presented to the user. **QWizardPage** is a **QWidget** subclass that provides features to store and validate user input.

3.5. 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 File Handling on page 44 for more information about serialization of Qt's data types.

3.6. Multithreading and Concurrent Programming

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 supports multithreading and provides classes to represent threads, mutexes, semaphores, thread-global storage, and classes that support various locking mechanisms.

Qt'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, each components can run its own event loop, simplifying the process of communicating across thread boundaries with events. Certain types of object can also be moved between threads.

The provision of mechanisms for multithreaded communications simplifies the process of using worker threads – this can also provide benefits for GUI applications as well as for console-based computational applications.

Qt also provides facilities for concurrent programming, including implementations of the well-known map-reduce and filter-reduce algorithms. These are integrated with Qt's object model, using standard container classes to make it more convenient to use concurrent techniques in Qt applications.

A synchonous (blocking) API is supplemented with an asynchronous (non-blocking) API which uses the concept of futures to handle synchronization between a main thread and several worker threads, simplifying the delivery of results.

3.7. Desktop Integration

Applications can be extended to interact with services provided by the user's desktop environment by using Qt's desktop integration classes. These range from **QSystemTrayIcon**, which is often used by long-running applications to provide a persistent indicator in the system tray, to **QDesktopServices**, which allows resources such as `mailto:` and `http://` URLs to be processed by the most appropriate applications on each platform.

QDialogButtonBox is a specialized container widget used in Qt's standard dialogs to position buttons according to the style guidelines for each platform. It can also be used in custom dialogs.

Online References

http://doc.trolltech.com/4.5/qt4-mainwindow.html
http://doc.trolltech.com/4.5/threads.html
http://doc.trolltech.com/4.5/desktop-integration.html

4. 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 6 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. Forms can be previewed using device "skins" to simulate the display constraints and appearance of the target device.

Commercial licensees on Windows can enjoy *Qt Designer*'s user interface design facilities from within Microsoft Visual Studio®. Qt Software also produces a Qt integration plugin for the cross-platform Eclipse™ IDE that embeds *Qt Designer* alongside other Qt technologies into the IDE framework.

4.1. 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 to produce dynamically-generated user interfaces. Qt's build system (see page 50) is able to automate the compile-time construction of user interfaces to make the design process easier.

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 the Eclipse environment.

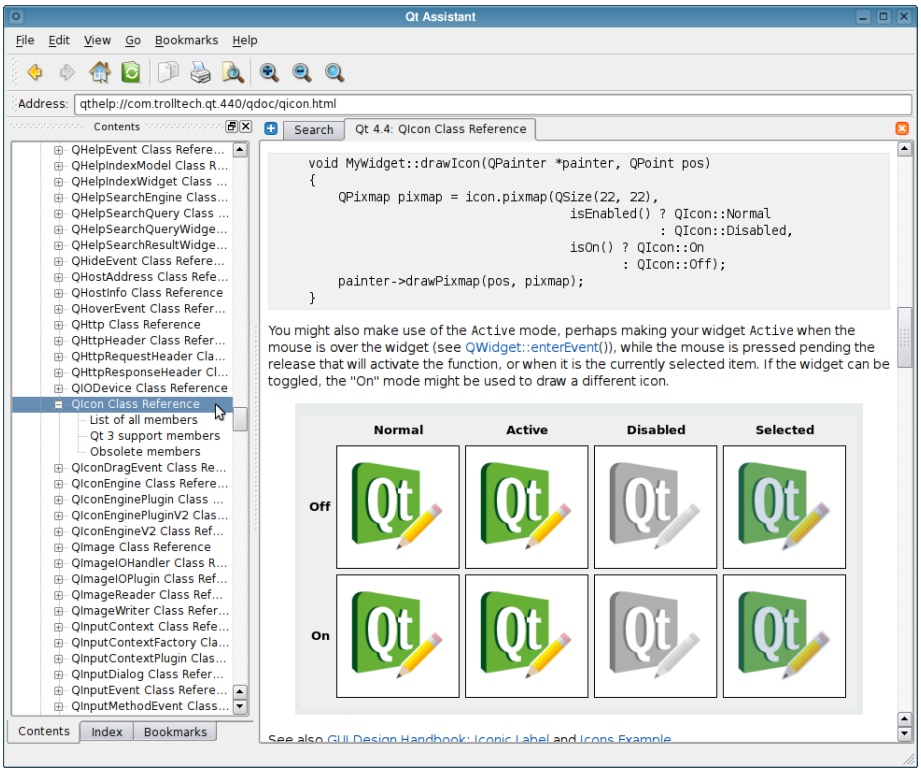

Figure 9: *Qt Assistant* displaying a page from the Qt documentation.

4.2. 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 and employs configurable filters to help users manage multiple documentation sets.

Developers can deploy *Qt Assistant* as the help browser for their own applications and documentation sets by using the classes in Qt's Help module. This module also provides an API that developers can use to access documentation for custom display purposes, perhaps using the **QToolTip** and **QWhatsThis** classes to show small pieces of relevant information to users.

Qt Assistant renders Qt's HTML reference documentation using **QWebView**, one of the classes that exposes the WebKit browser engine to Qt (see Web Integration with WebKit on page 28).

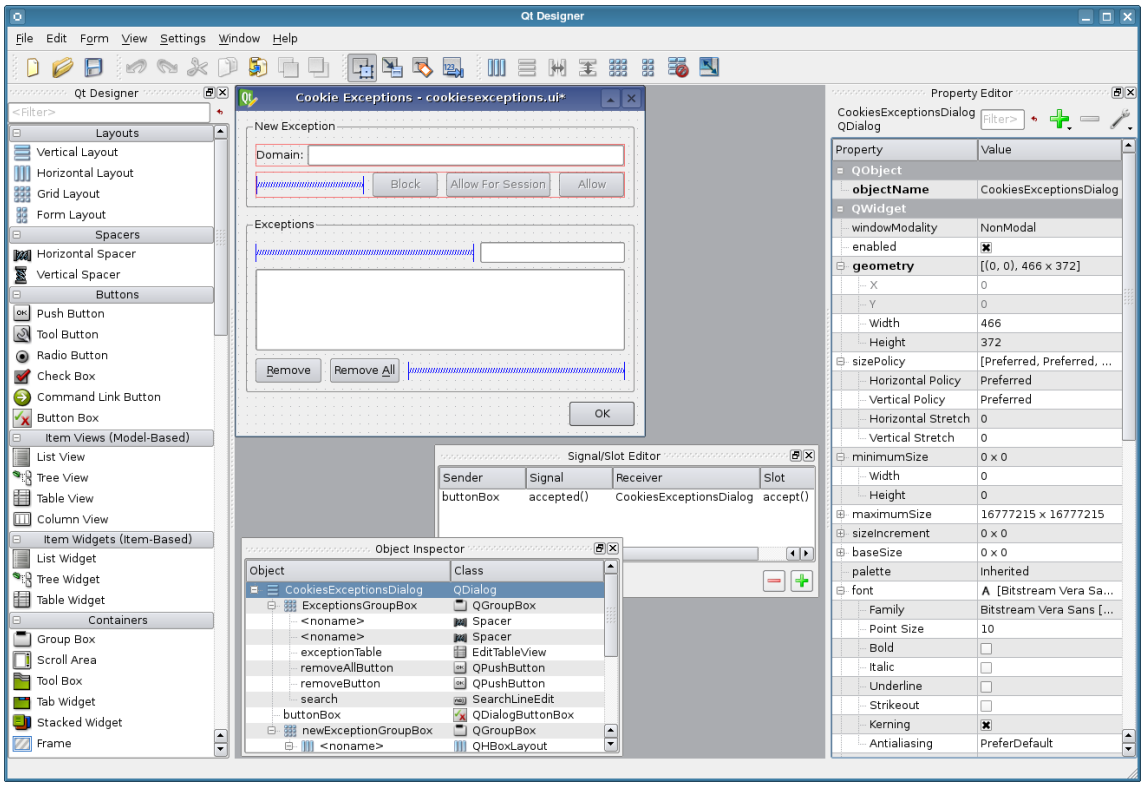

Figure 10: *Qt Designer* in "Docked Window" mode.

4.3. 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 over 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 49.

Online References

http://doc.trolltech.com/4.5/designer-manual.html
http://www.qtsoftware.com/products/appdev/developer-tools
http://doc.trolltech.com/4.5/qtdesigner.html
http://doc.trolltech.com/4.5/examples.html#qt-designer

5. Graphics and Multimedia

Qt provides excellent support for 2D and 3D graphics. Qt's 2D graphics classes support raster and vector graphics, can load and save a wide and extensible range of image formats, and can export text and graphics to Portable Document Format (PDF) files. Qt can draw transformed Unicode rich text, Scalable Vector Graphics (SVG) drawings, and provides a fully-featured canvas for demanding interactive applications. Qt also provides features for playing audio and video files and streams.

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 19). This approach ensures that a wide range of powerful painting operations are available for each of the devices supported by Qt, and also allows developers to choose the devices that are most suitable for their needs.

Figure 11: The Boxes demonstration presents a range of Qt's graphical features.

Graphical applications that require an interactive canvas can take advantage of the Graphics View framework (see page 21) to manage and render scenes with large numbers of interactive items, using multiple views if necessary.

Qt's support for OpenGL and OpenGL ES (see page 23) helps developers to integrate 3D graphics into their applications, yet it also enables them to take advantage of modern graphics hardware to improve 2D rendering performance.

Support for device-independent colors is provided by the **QColor** class. Colors are specified by ARGB, AHSV, or ACMYK values, or by common names. **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.

5.1. 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.

5.2. Images

The **QPixmap** and **QImage** classes supports input, output, and manipulation of images in several formats, including BMP, GIF, JPEG, MNG, PNG, PNM, TIFF, 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 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.

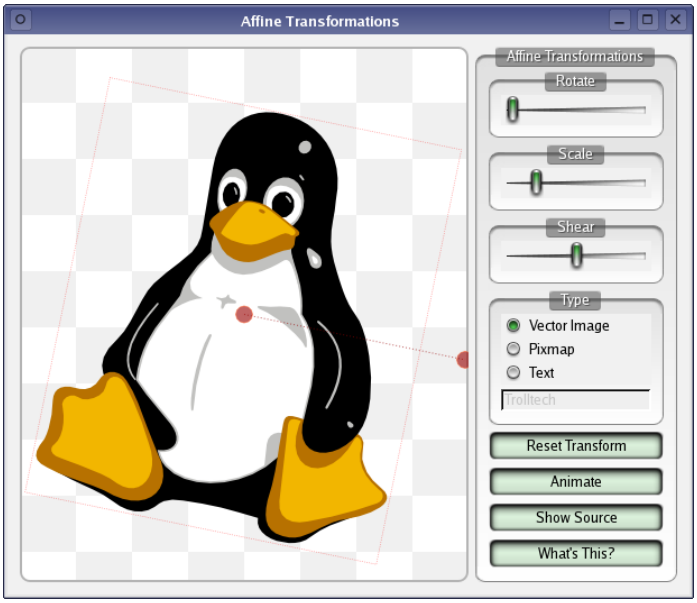

Figure 12: The Qt Affine Transformations demonstration shows how transformations can be used to achieve a desired effect.

5.3. Paint Devices and Printing

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

Qt supports the following paint devices:

- All **QWidget** subclasses are paint devices. Qt uses a *backing store* to reduce flickering during the painting process. Translucent and shaped windows can be created on suitably configured systems.
- All **QGLWidget** subclasses are paint devices that convert standard **QPainter** calls to OpenGL calls, enabling two-dimensional graphics to be accelerated on devices with appropriately supported hardware.
- 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 **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 **QSvgGenerator** represents a Scalable Vector Graphics (SVG) drawing. Paint commands are translated to the corresponding structures in the SVG file format.
- 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. Portable Document Format (PDF) and PostScript files can be generated on all platforms, enabling applications to create high quality documents that can be viewed using suitable reader applications.

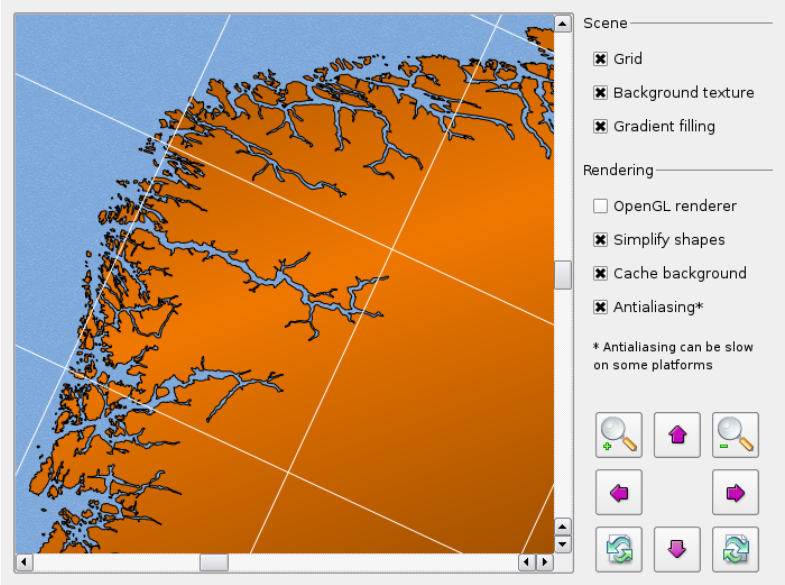

Figure 13: The Graphics View framework enables graphical applications to be created which combine high quality rendering with comprehensive features for user interaction.

5.4. Graphics View Framework

Qt introduces a powerful new framework for interactive graphical applications that is used to manage and display large numbers of items in a two-dimensional scene. Graphics View provides both an object-based API for adding new items to a scene and a traditional canvas-style API containing convenience functions for creating predefined items.

Once created, items can be placed with the required position, orientation, and scale in a scene. The display and item management functionality are implemented separately in the **QGraphicsView** and **QGraphicsScene** classes, enabling features such as multiple views onto the same scene and support for switchable renderers.

A selection of standard item types are provided, each of which is a subclass of the **QGraphicsItem** base item class, and these can be extended through subclassing to provide custom item types. Items can be grouped using **QGraphicsItemGroup** to allow higher-level control over the transformations applied to parts of a scene. Each scene, view, and the items themselves provide a comprehensive set of functions to allow coordinates to be transformed conveniently between coordinate systems. Both standard and custom items can be made selectable and movable, enabling a basic level of interactivity with a minimum of code.

Graphics View has been designed with animations in mind: **QGraphicsItemAnimation** objects can be used to create animated objects that are transformed according to a series of transformations defined at certain points on a timeline. Each animated item is run according to a **QTimeLine** object that controls the rate and duration of its animation.

A fully-featured event model, specifically designed for graphics items, enables events to be handled efficiently by dispatching them only to the items that require them. Since basic item handling is performed by the framework, items only need to respond to events if they need particular information about their environment. In such cases, **QGraphicsItem** provides a set of standard event handlers to allow developers to implement specialized interactive behavior as required.

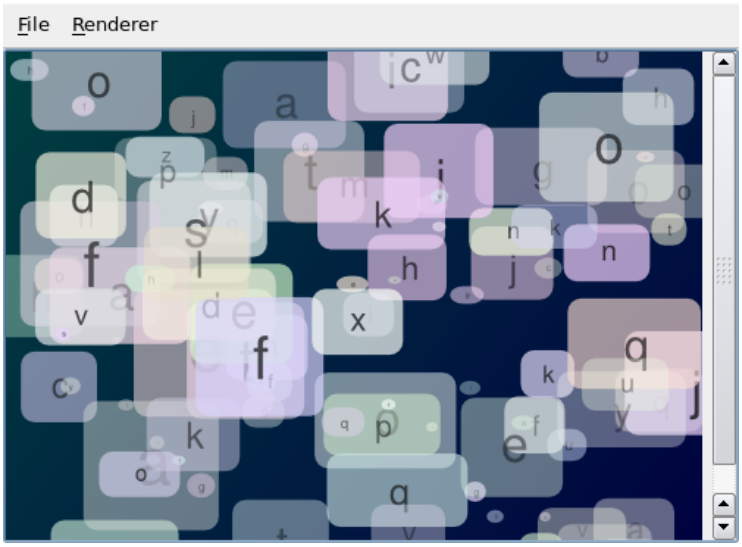

Figure 14: SVG drawings can be rendered onto any paint device supported by Qt.

QGraphicsScene objects are also able to render their contents independently of any attached views, using a **QPainter** object to perform drawing operations on any **QPaintDevice** subclass. This automatically provides support for printing, either directly to a printer or to a PDF file.

Some of the standard items bring features found in other parts of Qt to the Graphics View framework. **QGraphicsTextItem** can be used to include plain and rich text, allowing the user to perform editing tasks on documents that have been transformed with scaling, rotation and other operations. **QGraphicsSvgItem** displays Scalable Vector Graphics drawings as items in the scene. Shapes and paths are supported by **QGraphicsPathItem** and images can be added to scenes in the form of **QGraphicsPixmapItem** objects.

Applications that need to mix classical user interface elements with interactive content can embed widgets directly into a scene using **QGraphicsProxyWidget**, or create widget-like elements from scratch with **QGraphicsWidget**. As with conventional user interfaces, layout managers can be used to arrange widgets and items in a scene.

5.5. Scalable Vector Graphics (SVG)

SVG is an XML-based file format and language for describing graphical applications that is commonly associated with two-dimensional Web-based graphics. SVG support in Qt is based on the SVG 1.1 standard, a World Wide Web Consortium (W3C) Recommendation, and provides additional features to support the Tiny profiles of SVG 1.1 and 1.2.

Support for SVG rendering in Qt is provided by the **QSvgWidget** and **QSvgRenderer** classes. These use Qt's paint system to display and animate SVG drawings and, as a result, can render the contents of drawings onto any **QPaintDevice** subclass, including **QImage** and **QGLWidget**.

The **QIcon** class also supports the SVG file format, making it practical for developers to use SVG drawings instead of bitmap images for icons.

Qt developers are also able to generate SVG drawings by using **QPainter** functions to draw on **QSvgGenerator** paint devices, allowing graphics used in applications to be exported as SVG drawings with little additional effort.

5.6. 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.

Qt also enables OpenGL features and extensions to be used conveniently from within Qt applications. Convenience functions allow textures to be created from **QImage** and **QPixmap** objects, and support for pixel buffers and framebuffer objects are provided by the **QGLPixelBuffer** and **QGLFramebufferObject** classes. Support for features such as sample buffers can be enabled for use with **QGLWidget** if they are available on the underlying platform.

As well as providing an API that allows 3D graphics created with OpenGL to be integrated into a Qt application, **QGLWidget** subclasses use a specialized paint engine that translates **QPainter** operations into OpenGL calls. Applications can take advantage of this feature to improve 2D rendering performance on appropriate hardware, or it can be used to overlay controls and decorations drawn using **QPainter** onto 3D scenes drawn using OpenGL. On embedded platforms, where hardware acceleration is often limited, this paint engine is restricted to using the functionality of OpenGL ES 2.0, ensuring that it works well on as many devices as possible.

On suitable hardware, support for anti-aliased rendering can be enabled to enhance both the rendering speed and quality of graphics produced using the OpenGL paint engine. On less-capable hardware, developers can give users the choice between quality and speed by exposing these rendering options to users at run-time.

5.7. Multimedia

Qt 4.5 uses the Phonon multimedia framework, an open source project originating from the KDE project, to provide media playback features that can be accessed using a consistent, cross-platform API. Qt ensures that applications on Linux/Unix, Windows and Mac OS X transparently use the appropriate multimedia framework for each platform – this means that applications can also take advantage of platform-specific support for audio and video codecs and formats.

Phonon's features can be integrated into other technologies provided by Qt. For example, movie widgets can be added to Web pages displayed using the WebKit browser engine and to scenes rendered using the Graphics View framework.

Examples are provided that demonstrate the use of Phonon to play audio and video in Qt applications. Since Phonon can also obtain information about the multimedia capabilities on a user's device at run-time, applications can inform users about the kinds of media that they support.

Online References

http://doc.trolltech.com/4.5/qt4-arthur.html
http://doc.trolltech.com/4.5/opengl.html
http://doc.trolltech.com/4.5/qpainter.html
http://doc.trolltech.com/4.5/graphicsview.html
http://doc.trolltech.com/4.5/qtsvg.html
http://doc.trolltech.com/4.5/phonon-overview.html

6. 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 like data sharing, sorting and filtering, multiple views, and multiple data representations to be used for the same data.

When writing applications that process large quantities of data, developers typically rely on "item view" widgets to display items of data quickly and efficiently. Standard item views found in modern GUI toolkits include list views containing simple lists of items, tree views with hierarchical lists of items, and table views which provide layout features similar to those found in spreadsheet applications.

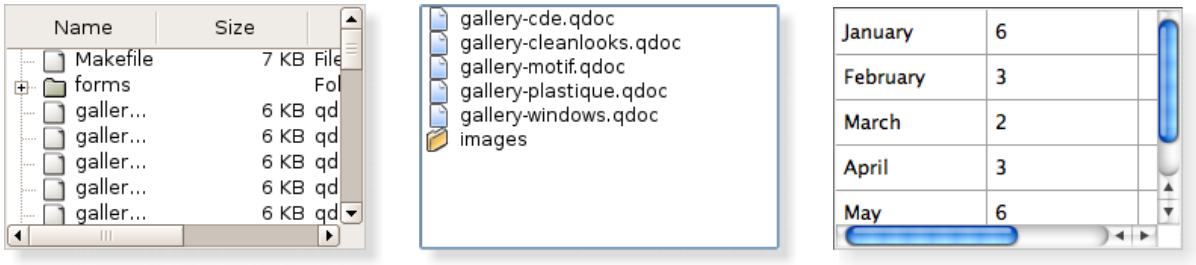

Figure 15: Qt provides standard item views for trees, lists and tables of items.

Qt's item view classes are available in two different forms: as classic item view widgets and as model/view components. Classic item view widgets such as **QListWidget**, **QTableWidget** and **QTreeWidget** are self-contained views that manage item objects explicitly created by the developer. **QListView**, **QTableView** and **QTreeView** are the equivalent model/view components to the classic item views. These model/view components provide a cleaner, component-oriented way to handle data sets.

6.1. Standard Item Views

Standard implementations of list widgets, icon views, tree widgets, and tables are supplied with Qt. These fully support drag and drop operations within the same view and between different views. As with all Qt widgets, they are also fully integrated with Qt's resource system (see page 51).

Item view classes are used to display data for various standard dialogs in Qt (Figure 7) and are extensively used in *Qt Designer*, *Qt Assistant*, and *Qt Linguist*.

Classic item views are typically used to display and manage a few hundred items of data, using an architecture that uses individual objects to encapsulate pieces of data. This approach should be familiar to existing Qt developers, and provides a convenient way to rapidly construct rich user interfaces for handling moderate amounts of data.

For consistency and reliability, the classic item views are built upon Qt's model/view framework, which provides a more scalable and customizable way to handle item view data.

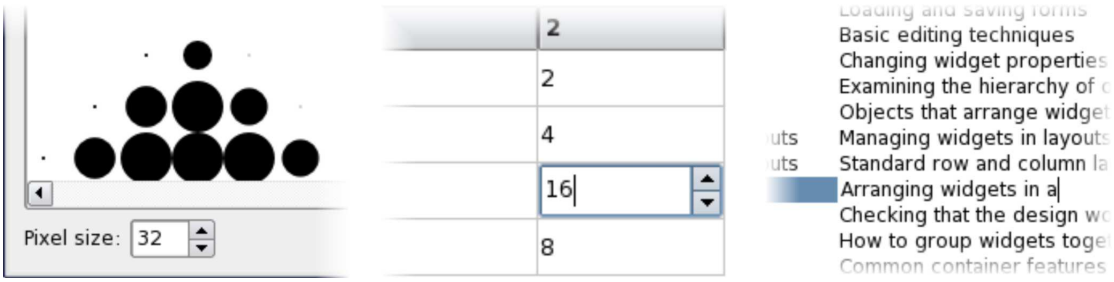

Figure 16: The component-oriented architecture of the model/view framework makes it easy to customize item views.

6.2. 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.
- By using *proxy models*, data supplied by models can be transformed before it is supplied to views. This enables applications to provide sorting and filtering facilities that can be shared between multiple views.

The model/view system is also used by Qt's SQL models (see page 30) to make database integration simpler for non-database developers.

Online References

http://doc.trolltech.com/4.5/model-view-programming.html
http://doc.trolltech.com/4.5/examples.html#item-views

7. Text Handling

Qt provides a powerful text editor widget that allows the user to create and edit rich text documents, 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.

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.

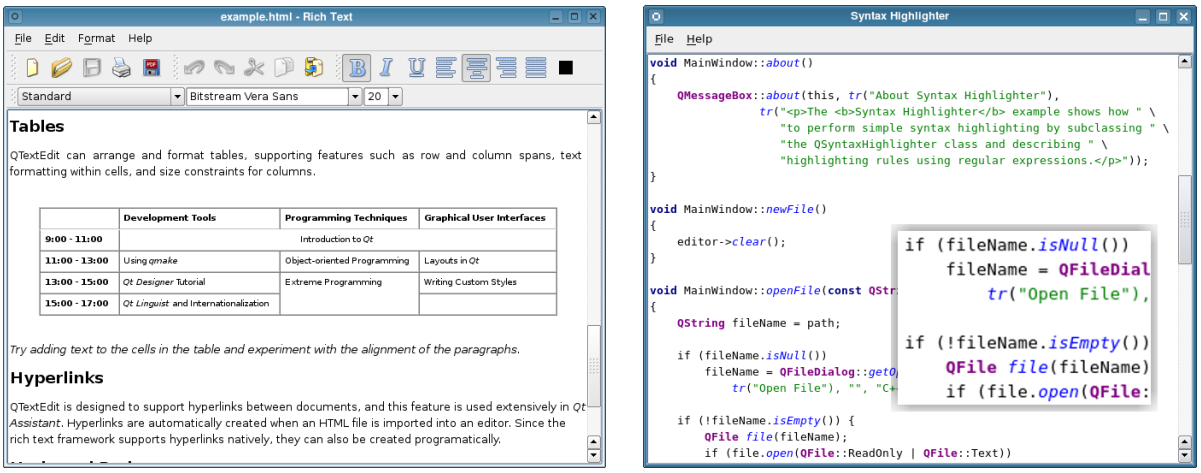

Figure 17: Qt's advanced rich text document features allow complex documents to be created and edited in **QTextEdit**. Syntax highlighting can be used to change the appearance of a document.

7.1. Rich Text Editing

Interactive rich text display and editing are 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 importing and exporting a subset of HTML 4.0, full undo/redo capabilities (including grouping of operations), and resource handling.

Qt provides an object-based API for documents that helps developers obtain a high-level overview of their structures. A cursor-based API is also provided to allow convenient exploration, processing and transformation of documents. 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 for tables, lists, frames, and ordinary paragraphs to be customized to give documents the desired appearance.

Documents created programatically 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.

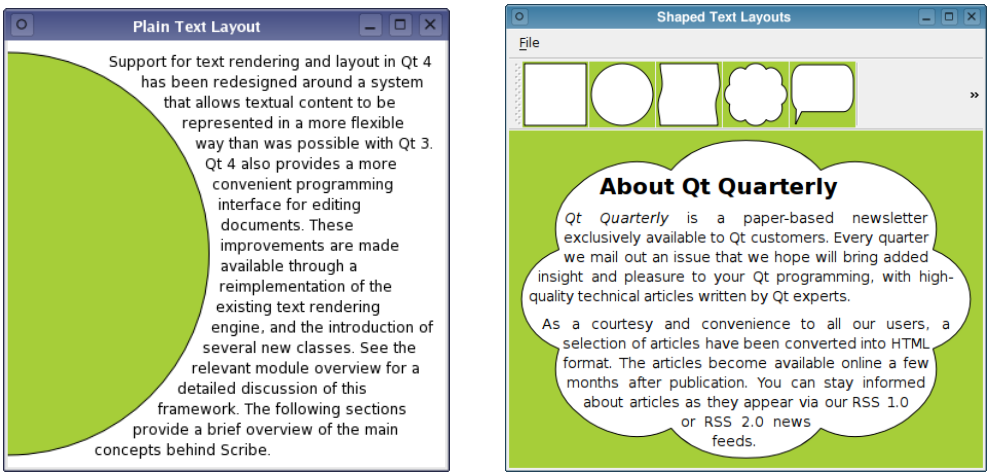

Figure 18: Low-level text handling features can be used to create specialized custom widgets with specialized text formatting requirements.

7.2. Customization, Printing and Document Export

Qt'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**.

Syntax highlighting rules can also be applied to rich text documents with the **QSyntaxHighlighter** class. This allows a standard **QTextEdit** widget to be used as the basis for a code editor, or to provide highlighting facilities for document search tools.

Documents can also be formatted according to information obtained from a **QPrintDialog** into a series of pages suitable for printing with a **QPrinter**.

The **QTextDocumentWriter** class provides support for document export to HTML, plain text and OpenDocument Format (ODF) files. This class exposes its functionality via a generic API and is designed to be extended to support additional formats in future releases.

Online References

http://doc.trolltech.com/4.5/qt4-scribe.html
http://doc.trolltech.com/4.5/richtext.html
http://doc.trolltech.com/4.5/qtextdocumentwriter.html

8. Web Integration with WebKit

Qt's integration with the WebKit browser engine enables developers to introduce Web functionality into their applications by using Qt-style APIs and paradigms to display and interact with Web content.

Qt includes integrated support for WebKit, a fully-featured open source Web rendering engine with a focus on stability and performance. The version of WebKit supplied with Qt supports a number of Web standards, including HTML 4.01, XHTML 1.1, CSS 2.1 and JavaScript 1.5. More advanced features are also available – these are presented in the *WebKit in Qt* whitepaper.

WebKit's networking is transparently handled using Qt's networking classes, providing browser components with a fully-compliant HTTP 1.1 implementation support for Secure Sockets Layer (SSL) communication and proxy support.

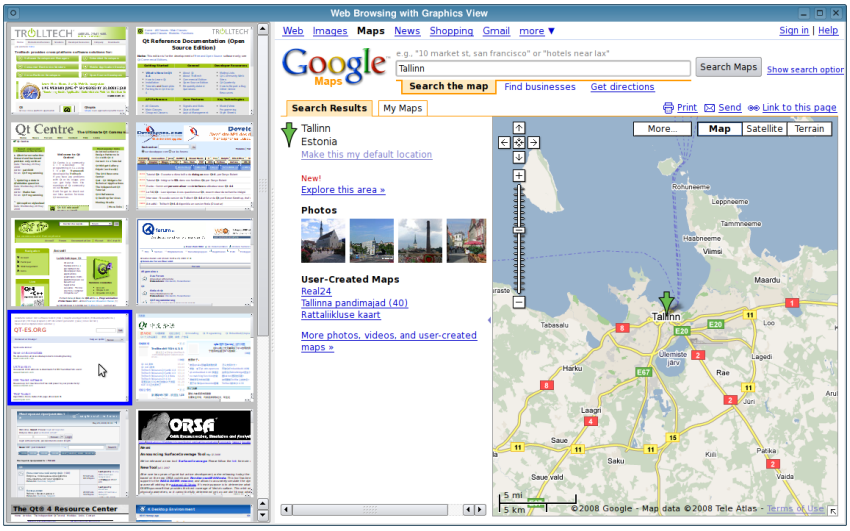

Figure 19: Qt widgets can be embedded into Web pages displayed with Qt's WebKit integration; Web Pages can be displayed as items in the Graphics View framework.

8.1. Native Application Integration

Qt's support for WebKit goes beyond just rendering HTML by exposing features of WebKit to applications using Qt's paradigms. For example, support for Qt's signals and slots communication mechanism makes it easier for developers to connect Web components to widgets and other application objects.

Conversely, the integration between Qt and the browser engine enables native Qt controls to be included within Web pages, making it possible to combine Web content with highly-dynamic native user interfaces.

WebKit also enables Web applications to use native storage for persistent data, and this features is supported by Qt. Developers can enable native storage for applications that interoperate with remote services, and take advantage of configuration options to set an appropriate location and quota for it on the user's system.

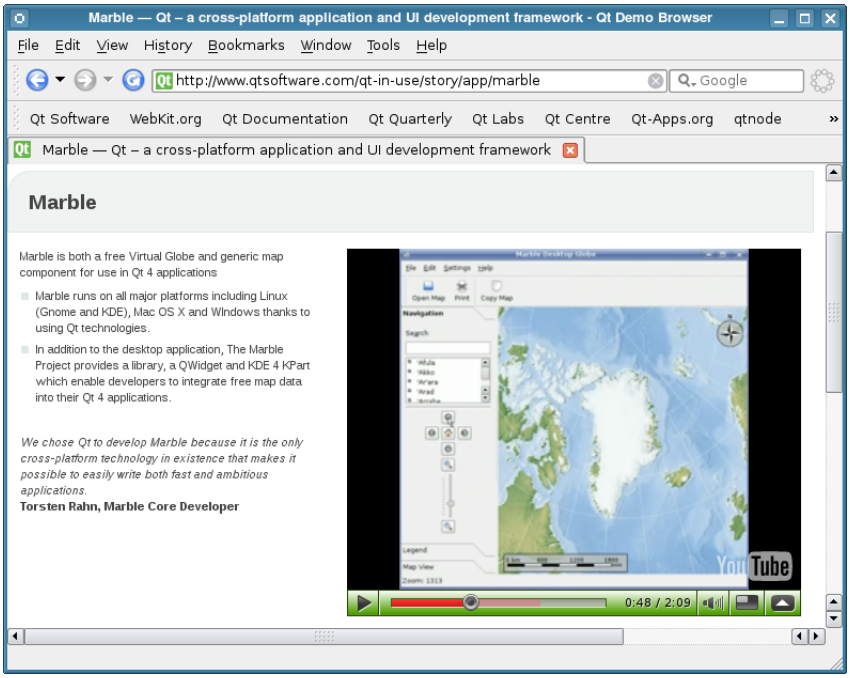

Figure 20: Third party browser plugins are supported by Qt and WebKit via the Netscape plugin API.

8.2. Netscape Plugin Support

Plugins conforming to the Netscape plugin API, a de-facto standard for third party browser components, can be embedded and displayed in Web pages rendered by Qt's WebKit integration. Configuration of this feature is performed via a Qt class that is also used to configure other kinds of plugins, such as widget plugins exposed by the application to the Web environment.

Online References

http://doc.trolltech.com/4.5/qtwebkit.html
http://doc.trolltech.com/4.5/demos-browser.html
http://www.qtsoftware.com/forms/whitepapers/reg-whitepaper-hybrid/

9. Databases

The Qt SQL module simplifies the creation of cross-platform GUI database applications. Programmers can easily execute SQL statements, use database models to supply information to item views for visualization and data entry purposes, and use widget mappers to relate database tables to specific widgets in form-based user interfaces.

The Qt SQL module provides a cross-platform interface for accessing SQL databases, and 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. Distributions of Qt include the SQLite database, and the Qt SQL module is built with support for this database by default.

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 (see page 25). 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 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; **QSqlQuery** provides the **first()**, **prev()**, **next()**, **last()** and **seek()** functions to assist with the navigation between each of the records in a result.

The `INSERT`, `UPDATE`, and `DELETE` statements are equally simple to use with **QSqlQuery**.

Qt's SQL module also supports value binding and prepared queries. Value binding can be achieved using named binding and named placeholders, or using positional binding with named or positional placeholders. 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 (see page 25). 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 work for developers. Setting up a query model is simply a matter of specifying a query and choosing which headers to examine, and setting up a table view to display the results of the query is similarly straightforward.

Models are provided for accessing SQL tables in different ways:

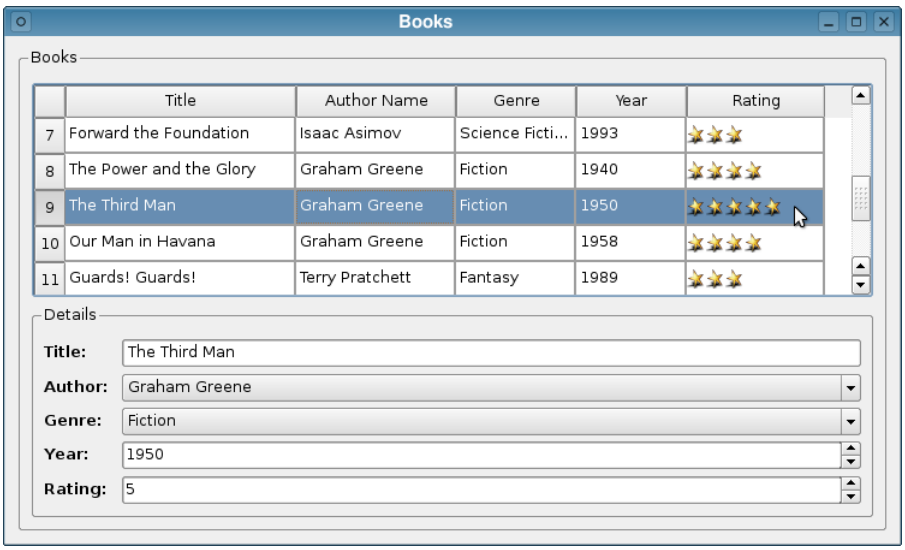

Figure 21: The Qt Books demonstration shows the integration between Qt's SQL classes and the model/view framework.

- **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 Books demonstration shown in Figure 21 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.

9.3. Data-Aware Widgets

Qt provides facilities to allow data obtained from models, such as the SQL models described above, to be related to specific widgets in a window, allowing the user to see a cross-section of the data available from different locations in the underlying data store. This approach helps to provide a form-based user interface that is more familiar to users of established data-entry and recording applications.

The **QDataWidgetMapper** class is used to set up the mapping between a model and a selection of widgets. In the Books demonstration supplied with Qt (see figure 21), a data mapper is constructed and assigned to a model, then each of the widgets used to edit fields in the database is mapped to a column:

The data mapper can be used directly to step through rows in a model, mapping the item in each column to a specific widget, and using it to display the data obtained. Functions such as **toFirst()**, **toNext()**, and **toPrevious()** make it possible to provide easy-to-use navigation controls for users.

Since the class also provides a similar API to the item view classes, different cross-sections of a hierarchical model can be obtained by setting the root index for the mapping. The model/view

API also makes it possible for a data mapper to respond to changes in item views that are also associated with a given model. The Books demonstration updates the mapping whenever the user selects a different row in a table view; this behavior is set up with a simple signal-to-slot connection.

The mapper uses the model index supplied to obtain model indexes for each of the items on the new row, and updates all the editor widgets automatically.

Online References

http://doc.trolltech.com/4.5/qtsql.html
http://doc.trolltech.com/4.5/qt4-sql.html
http://doc.trolltech.com/4.5/examples.html#sql

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 supports the Unicode version 5.0 character encoding and comes with a wide range of text codecs for converting between character sets, enabling application to use and convert textual data obtained from a variety of sources.

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* (see page 35) 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's regular expression engine, provided by the **QRegExp** class, uses Unicode strings both for the regular expression pattern and the target string.

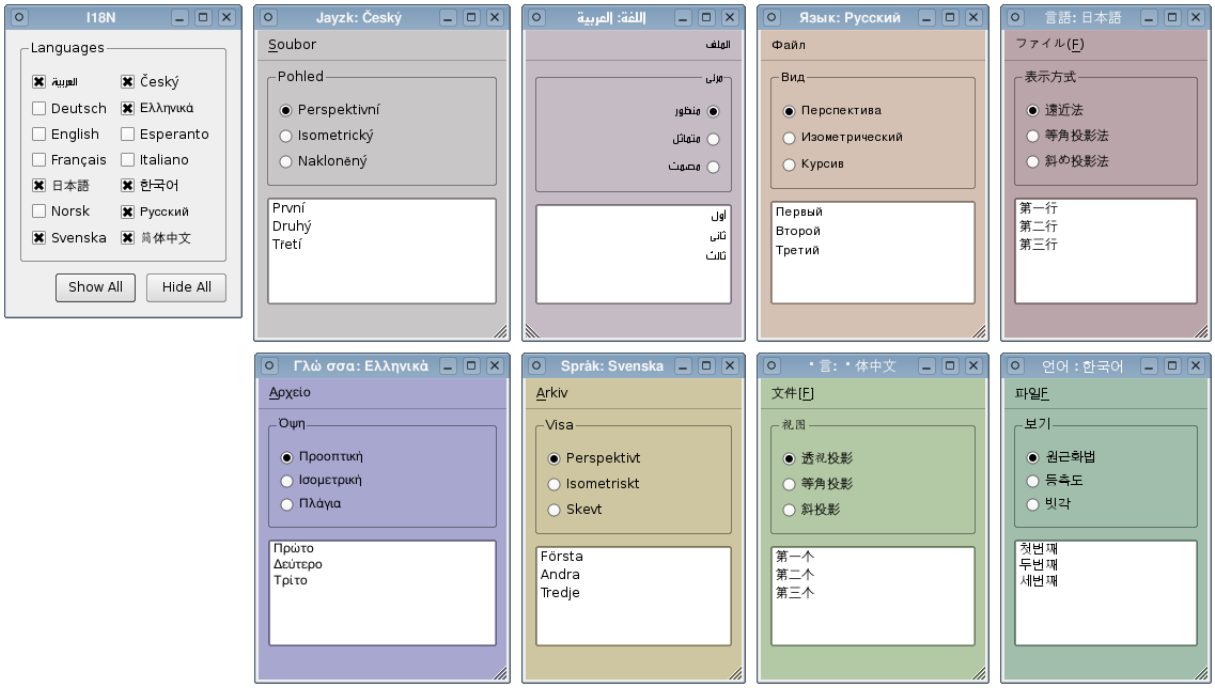

Figure 22: The same user interface shown in a variety of different languages.

Qt uses the **QString** class to store Unicode strings, both for the API and internally. Individual characters are represented by the **QChar** class, which provides functions such as **QChar::lower()** and **QChar::isPunct()** that work over the entire Unicode range. Constructors and operators are provided that automatically convert to and from 8-bit and STL strings. Programmers can copy strings by value, since they are implicitly shared (copy on write; see page 47), making them fast and memory efficient to use.

Built-in 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.

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 Qt Software provides French, German and Simplified Chinese 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")
```

*ISO is a registered trademark of the International Organization for Standardization.

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.

Another optional argument can also be used to specify an integer value to help with the translation of plural forms:

```
Context::tr("%n message(s) saved", "number of saved messages", messages.count())
```

The special "(s)" notation is used to indicate that a plural form is to be used, ensuring that suitable text for the specified integer is used for the user's language.

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 that are used to prepare `.qm` files and handle the translation process:

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. **Qt Linguist** is used by translators to provide translations for the source texts in the `.ts` files.
3. **lrelease** is used to process the `.ts` files to generate highly compressed `.qm` 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.

An additional tool, `lconvert`, is used to convert between XML Localization Interchange File Format (XLIFF), GNU Gettext PO format and Qt's `.ts` files.

10.3. Qt Linguist

Qt Linguist is a Qt application that helps translators translate Qt applications. Translations for Qt applications are traditionally stored in `.ts` format files. In addition, Qt Linguist supports GNU Gettext PO format, which is commonly used in Open Source projects, and XLIFF, which is widely supported by third party tools and services.

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.

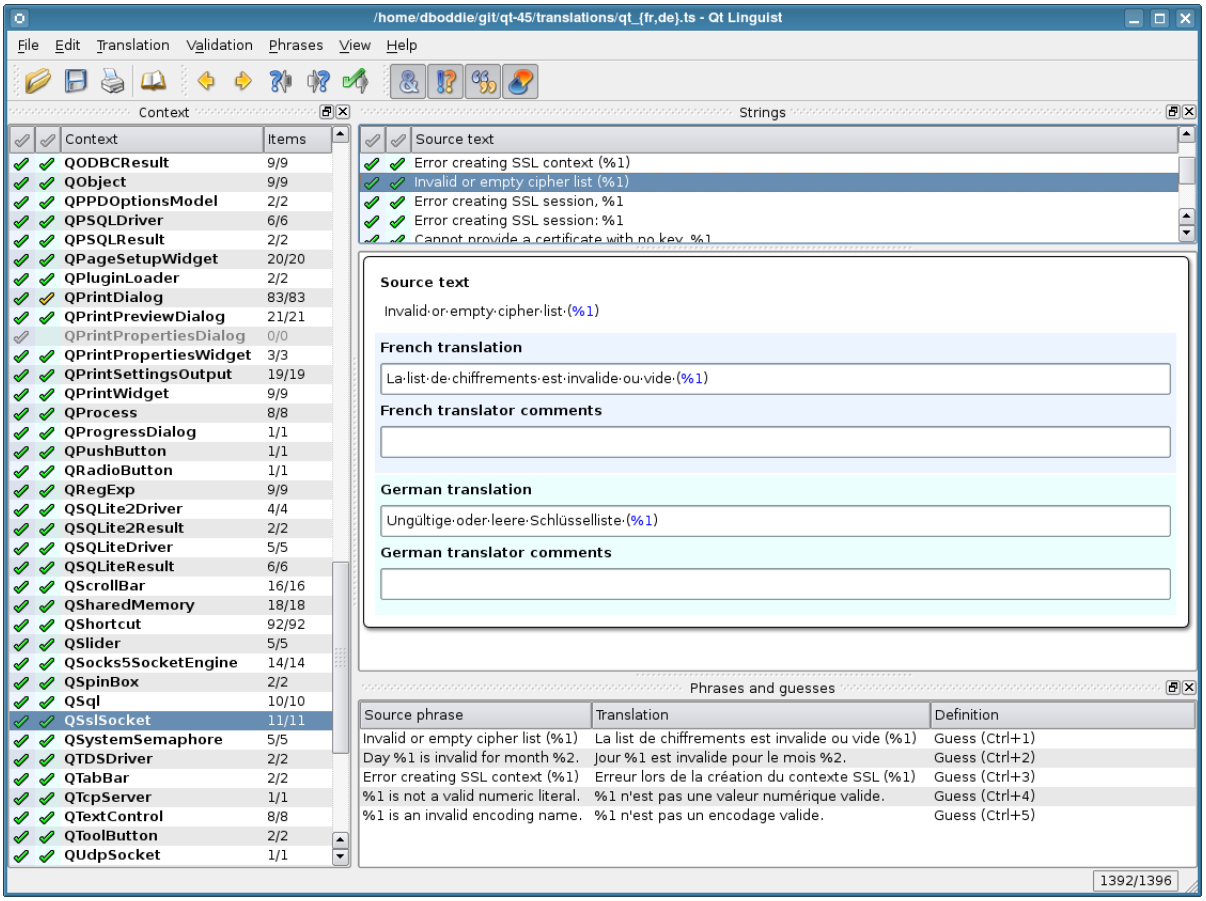

Figure 23: Working on French and German translations simultaneously with *Qt Linguist*.

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/4.5/i18n.html
http://doc.trolltech.com/4.5/unicode.html
http://doc.trolltech.com/4.5/qtextcodec.html
http://doc.trolltech.com/4.5/linguist-manual.html

11. Qt Script

Qt Script is an interpreted ECMAScript-based language which can be used to script Qt applications. The QtScript module provides an API that makes it easy to expose parts of an application to the scripting environment, including support for signals-slot communication and other standard Qt features.

Application scripting allows users to customize and extend the features of applications by accessing simple APIs for user-oriented scripting languages. Traditionally, Qt developers have used a variety of separate solutions to provide scripting support in their applications. The inclusion of Qt Script as a Qt module is intended to reduce the amount of effort required to implement scripting, while also making the process easier for developers who simply require scripting "out of the box".

11.1. Scripting Architecture

The QtScript module provides an API based around the **QScriptEngine** and **QScriptValue** classes. These provide execution and data marshalling facilities.

Instances of **QScriptEngine** are responsible for executing Qt Script code within a scripting environment and provide facilities to expose instances of **QObject** subclasses to this environment. Additionally, **QScriptEngine** allows C++ data types to be converted to their Qt Script equivalents and inserted into the scripting environment. Qt Script data types are represented in C++ as **QScriptValue** instances which provide functions to convert data to C++ types.

Additionally, signals and slots can be used to communicate between Qt Script and applications. Qt Script objects can emit the signals of the objects they wrap, and any Qt Script function can be used as a slot. This provides the additional level of flexibility that script authors require and expect from a dynamic scripting language.

The simplest use of the QtScript module is to provide an interpreter for Qt Script that executes user-defined code.

```
QScriptEngine engine;
QScriptValue result = engine.evaluate(userCode);
```

The **QScriptEngine** instance is also able to provide information about any syntax and run-time errors that occur.

In the above image, we can see the use of the QtScript module's features for integrating Qt Script into an application. A **QObject**-based wrapper for an image, providing a variety of functions and properties, is exposed to the scripting environment in the following way:

```
image = new ImageWrapper(this);
QScriptValue imageObject = engine.newQObject(image);
engine.globalObject().setProperty("image", imageObject);
```

Once defined in the environment, the user can manipulate the image using a simplified API that hides all the low-level details of managing the application's user interface.

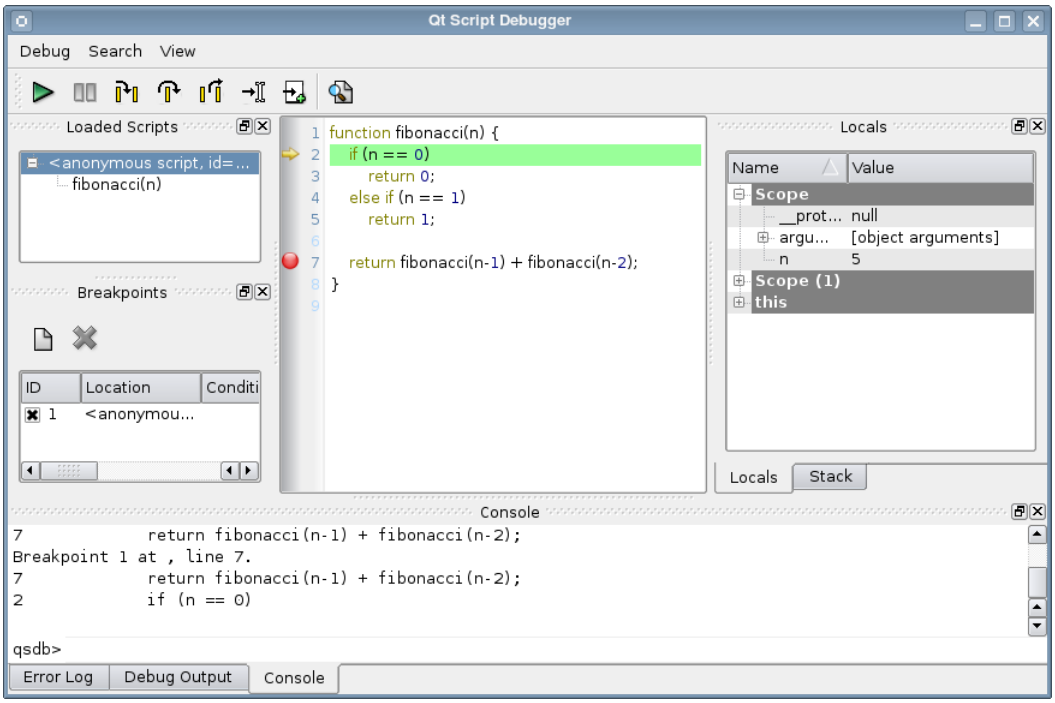

Figure 24: Debugging a function using the Qt Script debugger's user interface.

11.2. Debugging

Support for debugging of scripts is provided by the QtScriptTools module, which includes a set of integrated graphical components that developers can use when creating scriptable applications.

The **QScriptEngineDebugger** class has been designed to be simple to use. The separation between the implementation of the debugger and the script engine means that the process of attaching a debugger is simple and non-invasive; code that attaches the debugger to a script engine is self-contained and can be removed when no longer required:

```
engine = new QScriptEngine(this);
debugger = new QScriptEngineDebugger(this);
debugger->attachTo(engine);
```

The debugger provides a set of common debugging widgets that show source code, breakpoints, the contents of variables, and other useful information about the script that is being executed. These can be used together in a standard window (see Figure 24) or as separate widgets.

In addition to these widgets, the debugger also provides a standard menu and toolbar that can be used separately to create a custom debugging interface. The toolbar contains a set of buttons that are linked to actions (see Actions on page 10) which can be triggered programmatically to control the execution of scripts.

If preferred, the debugger can be used without showing any of the user interface components; the developer has the option of keeping these hidden until an error occurs or a breakpoint is encountered.

A comprehensive user manual documents how to use the debugger's user interface, and contains information on the commands that can be entered at the debugging console.

11.3. Benefits and Uses

The availability of application scripting in Qt provides opportunities and benefits both for skilled users and application developers, depending on the scripting APIs exposed by individual applications:

- Users are less dependent on the application developers for new features since scripting allows them to implement minor features themselves.
- System administrators can write scripts to customize applications before deploying them to a group of users.
- Developers can use scripting in modular applications to prototype new features that can later be incorporated into the next version of the application.
- The choice of an ECMAScript-based language makes it easier for existing Web developers to customize and extend applications.

The QtScript and QtScriptTools modules are actively maintained and developed by Qt developers, and is used internally at Qt Software for a variety of projects. While we do not anticipate that it will replace the use of other languages and solutions in some existing applications, it is expected that developers who are looking for a simple, Qt-style application scripting solution will find that the QtScript module meets their requirements.

Online References

http://doc.trolltech.com/4.5/qtscript.html
http://doc.trolltech.com/4.5/qtscripttools.html
http://doc.trolltech.com/4.5/examples.html#qtscript

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, Cleanlooks, GTK, Motif, Mac OS X, Plastique, Windows, Windows XP, and Windows Vista. 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.

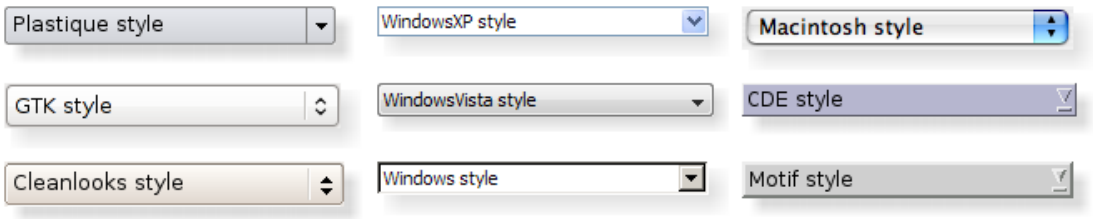

Figure 25: Combo boxes in the different native styles.

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. On the GNOME desktop environment, the GTK widget style is the default style; this uses the GTK+ theme engine to ensure that Qt applications blend in with the user's other applications. An alternative style for GTK-based environments is Cleanlooks, a style designed to look like the Clearlooks theme for GNOME.

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.

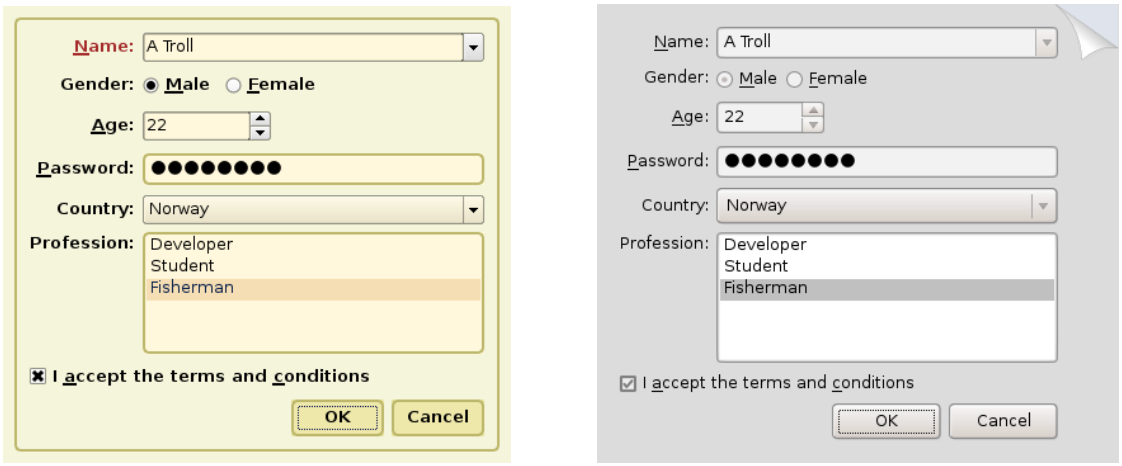

Figure 26: The Qt Style Sheet example allows the user to interactively experiment with the appearance of a set of widgets by applying changes to a style sheet as they are made.

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.

12.2. Widget Style Sheets

Qt supports the use of widget style sheets with almost all standard widgets. These textual descriptions, written in a language similar to Cascading Style Sheets (CSS), are used to customize the appearance of widgets in much the same way that CSS descriptions are used to customize HTML rendering in WWW browsers. Each widget's style sheet is accessed via its **styleSheet** property, available in **QWidget** and its subclasses, and this enables customizations to be easily applied to style-aware widgets while an application is running. Since this property is also available for editing in *Qt Designer*, graphic designers can directly influence the look and feel of applications. For many common situations where customizations to standard widgets are required, the use of style sheets can eliminate the need for a custom style to be written.

12.3. 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);
```

Figure 27: The Qt Styles example shows the built-in styles, and includes a custom style for comparison.

A style can also be compiled as a plugin (see page 49). 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 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/4.5/qt4-styles.html
http://doc.trolltech.com/4.5/widgets-styles.html
http://doc.trolltech.com/4.5/stylesheet.html
http://doc.trolltech.com/qq/qq13-styles.html
http://doc.trolltech.com/qq/qq20-qss.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, the default implementation of **QWidget::event()** forwards the most common types of event to dedicated handlers, such as the low-level **QWidget** functions, **mouseReleaseEvent()** and **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/4.5/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 fully supported, and information about the network interfaces available can be easily obtained.

14.1. File Handling

At the heart of Qt's device handling infrastructure is **QIODevice**, a general base class for files, sockets and other devices, which can be subclassed to add support for custom devices. All devices are able to communicate using signals and slots, making it straightforward to integrate file and network communications into applications.

The **QFile** class supports large files, long file names, and internationalized file names. The **QDir** and **QDirIterator** classes are used to read and traverse directories, and can be used to manipulate path names, create directories, delete files, and perform other common operations. **QFileInfo** provides more detailed information about a file, such as its size, permissions and last modification time.

Alternatively, applications can use instances of the **QDirModel** class to obtain information about files and directories in a form that is compatible with Qt's model/view classes (see page 25). This makes it possible for developers to re-use existing views to help users visualize the file system.

Qt includes classes similar to the standard `iostream` classes that operate on any **QIODevice** subclass. **QTextStream** is used to stream text to and from devices, and supports the encodings provided by **QTextCodec**. **QDataStream** is used to serialize the basic C++ types and many Qt types in a platform-independent binary format.

Transparent access to remote files is provided via a unified network access API, though specialized classes for HTTP and FTP protocols can also be used if required, building on Qt's networking classes (see page 45) . URLs can easily be parsed and reconstructed by using the **QUrl** class.

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 (see page 19). Printing text and images is handled by **QPrinter** (page 20).

Qt can also be used to monitor files and directories for changes made by other applications and services. The **QFileSystemWatcher** class acts as a registry of file paths that need to be monitored, and emits a signal whenever a file or directory on any of these paths is changed.

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.

Qt's **QXmlStreamReader** and **QXmlStreamWriter** classes present an alternative approach to reading and writing XML files in which tokens are "pulled" from an input stream and "pushed"

to an output stream. The design of these classes makes it easy to write lightweight, high-level parsers for XML-based file formats.

Higher level XML manipulation, including support for XQuery 1.0 and XPath 2.0, is provided by an additional module. This separation between basic XML handling and more powerful querying facilities allows developers to decide the level of XML support used in applications. Partial support for XSLT 2.0 adds another processing option for developers familiar with common XML technologies.

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**.

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, and can redirect input and output from external programs to files.

Additionally, higher-level communication between applications, components and the operating system can be achieved on certain platforms using the QtDBus module (see D-Bus Interoperability on page 55).

Access to low-level shared resources, such as shared memory and system semaphores, is provided by dedicated classes. These provide the basis for building alternative communication mechanisms.

14.4. Networking

Qt provides a cross-platform interface for writing TCP/IP clients and servers, supporting IPv4 and IPv6. All of the networking classes provided are reentrant and can be used from any **QThread** (see page 14).

The **QTcpSocket** class provides an asynchronous, buffered TCP connection. Since it is a **QIODevice** subclass, **QTextStream** and **QDataStream** can be used to handle socket-level communications. Similarly, **QUdpSocket** handles UDP socket operations. Both classes support blocking and non-blocking modes of operation. All of Qt's networking classes are reentrant and can be used from any **QThread**.

Custom TCP servers can be implemented by subclassing **QTcpServer**, which provides an asynchronous framework for handling incoming connections and serving clients. **QTcpServer** can operate in blocking and non-blocking modes.

Support for proxy servers is available through the **QNetworkProxy** class, enabling both application-wide and per-socket proxying facilities. HTTP, FTP and SOCKS 5 proxy types are supported, and caching facilities can be employed to improve performance. Customization features enable an application-wide policy to be employed that can set up proxies based on the socket type, the protocol in use, and other criteria.

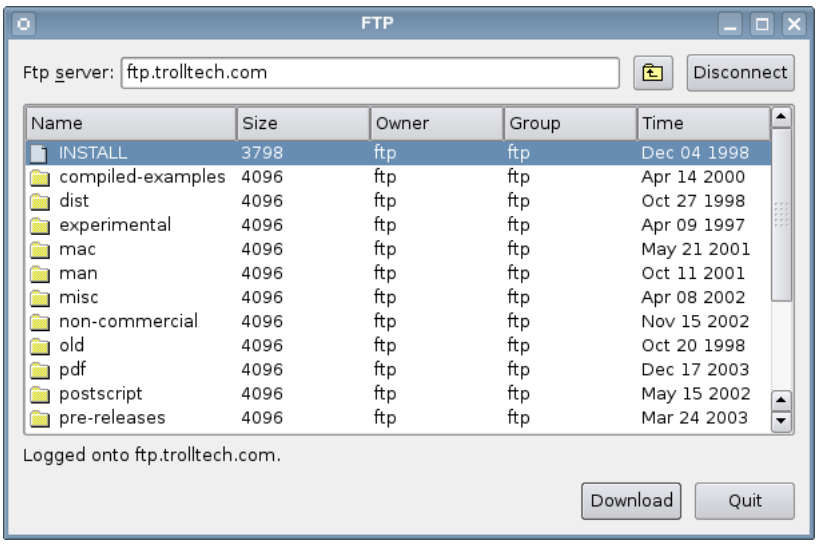

Figure 28: The Qt FTP example uses Qt's networking features to provide simple FTP browsing capabilities.

The **QAbstractSocket** class is a platform-independent wrapper for native socket APIs. It provides the underlying functionality for **QTcpSocket**, **QTcpServer**, and **QUdpSocket**. Support for proxy servers is available through the **QNetworkProxy** class, enabling both application-wide and per-socket proxying facilities.

A management infrastructure for network operations is provided in the form of **QNetworkAccessManager**, which is used to dispatch requests over common protocols, such as HTTP and FTP, and handle replies. Specific classes for requests and replies make common communication easy, while allowing the developer to customize particular requests.

Information about a machine's network interfaces is provided by the **QNetworkInterface** class. This exposes details of each interface, their capabilities, the IP addresses assigned to them, and other interface-dependent information. For example, for Ethernet interfaces, the MAC address of the underlying hardware can be obtained, and the broadcast address and netmask can be obtained in addition to the IP address.

14.4.1. Encrypted Communications

Qt includes features for secure network communications through the use of encrypted TCP connections based on Secure Socket Layer (SSL) protocols, including SSLv3 and TLSv1.

QSslSocket provides an SSL encrypted socket that can be used for both clients and servers. Abstractions for other aspects of the encryption and authentication process are addressed by the **QSslCipher**, **QSslKey** and **QSslCertificate** classes which represent cryptographic ciphers, public and private keys, and X509 certificates respectively.

Online References

http://doc.trolltech.com/4.5/qiodevice.html
http://doc.trolltech.com/4.5/qtxml.html
http://doc.trolltech.com/4.5/qtnetwork.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. Programmers can also use STL containers on the platforms that support them, at the cost of losing Qt's optimizations.

Unlike many template classes, which increase the size of executables dramatically when used, Qt's template collection classes are optimized for minimal code expansion.

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()**, and so on. 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>**, **QMultiHash<Key,T>**, **QSet<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/4.5/containers.html
http://doc.trolltech.com/4.5/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/4.5/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.

Qt is designed to work with a range of development tools and environments, from simple command line tools to integration with popular integrated development environments (IDEs).

Qt can also be used with Qt Creator (see page 52), a lightweight, cross-platform IDE that is specifically tailored to development of Qt applications.

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 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` (see page 6) 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 15).

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.

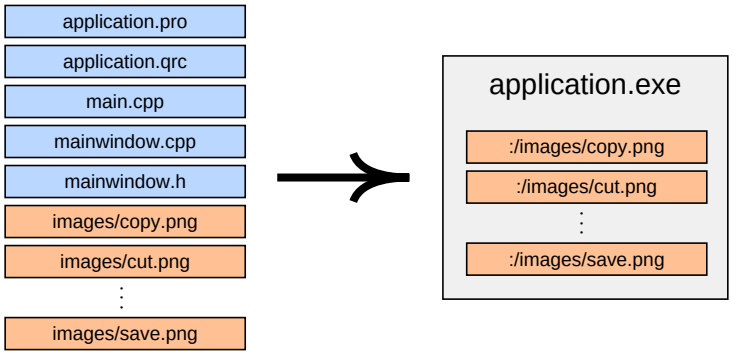

Figure 29: In this example, a set of images are packaged with the application when it is built. The application references them using the naming scheme shown.

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, and `qmake` supports the creation of universal binaries to support both Intel and PowerPC-based Macs.

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.

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.

The resource system can also be extended at run-time with the **QResource** class, allowing additional paths to be searched for resources, and enabling additional resources not specified in the `.qrc` to be loaded on demand to augment those built into the application.

17.3. Testing and Benchmarking Qt Applications

Support for unit testing is provided as a standard Qt module. Unit tests are written in C++ as **QObject**-based classes that contain test functions, and these are compiled into executables that can be run independently of any testing framework. Qt's unit testing library also provides extensions to allow graphical user interfaces to be tested.

Unit tests can also be set up to perform benchmarking operations with the use of a simple macro. Test cases can be configured to measure and report performance via the use of different backends, allowing performance data to be visualized using standard tools.

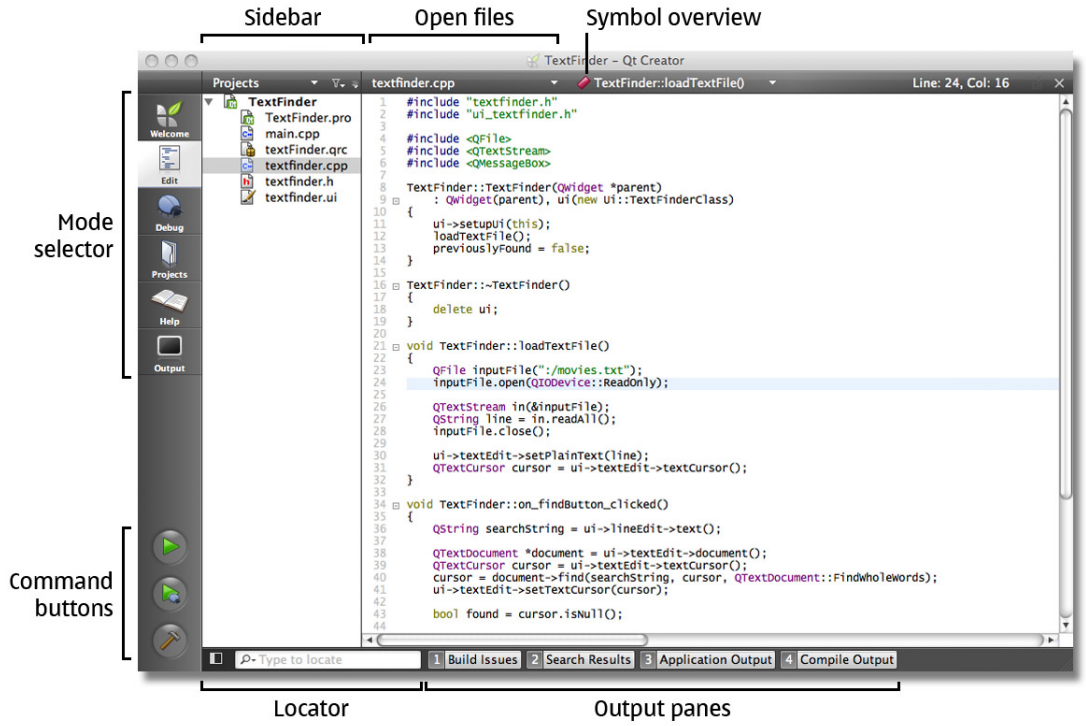

Figure 30: An overview of Qt Creator.

17.4. Qt Creator

Qt Creator is Qt Software's lightweight IDE for C++ and Qt software development. Although it is designed to be easy to use, Qt Creator provides all the features developers have come to expect from IDEs, including syntax coloring and code completion, quick location of classes, functions and other C++ structures, and integrated debugging support.

Qt-specific features include signals and slots signature completion, integrated support for Qt Designer, and built-in Qt documentation.

The *Qt Creator whitepaper* contains a more detailed introduction to this product.

Online References

http://doc.trolltech.com/4.5/qmake-manual.html
http://doc.trolltech.com/4.5/resources.html
http://doc.trolltech.com/4.5/qtestlib-manual.html
http://www.qtsoftware.com/developer/qt-creator
http://www.qtsoftware.com/products/qt/vs-integration.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 (see page 40). 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 for 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, Linux, NetBSD, OpenBSD, and Solaris. See the Qt Software 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 (see page 34). 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.

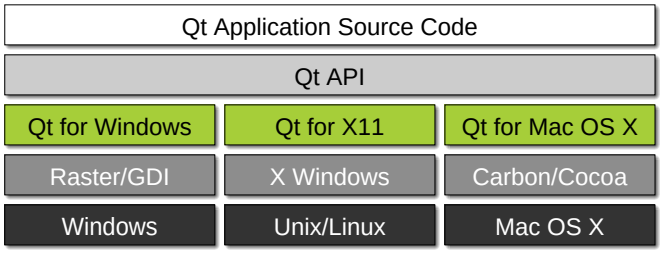

Figure 31: An overview of Qt's architecture on supported desktop platforms.

18.2. Microsoft Windows

Qt for Windows uses the Win32® API and GDI for low-level events and drawing. Qt does not use MFC or any other toolkit, but provides its own more powerful, customizable widgets that are rendered using a fast, yet accurate painting engine. (For non-specialized uses, Qt uses the native Windows file and print dialogs.)

Customers using Windows can create Qt applications using Microsoft Visual Studio® that will run on Windows 98, NT4, ME, 2000, XP and Vista. Qt performs a run-time check for the Windows version, and uses the most advanced capabilities available. Qt developers are insulated from differences in the Windows APIs.

The Microsoft accessibility interfaces are supported by Qt. 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 by using a combination of Cocoa®and Carbon® APIs. On 64-bit Macintosh hardware, Qt uses the Cocoa libraries to enable integration with Mac OS X native widgets and Cocoa views.

Qt for Mac OS X 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. Support for universal binaries on Mac OS X means that Qt applications can be created for Intel and PowerPC-based Macs. If desired, developers can take advantage of Qt's integration with native components to add platform-specific features to their applications.

Online References

http://doc.trolltech.com/4.5/supported-platforms.html
http://doc.trolltech.com/4.5/installation.html
http://doc.trolltech.com/4.5/deployment.html

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.

In addition to the ActiveQt extension outlined below, there are additional extensions available from third party suppliers.

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. Qt for Windows provides the ActiveQt module that 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.

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.

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 a single header file. Once the class is compiled, its properties, slots, and signals become ActiveX properties, methods, and events to ActiveX clients. ActiveQt also provides facilities 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. D-Bus Interoperability

On Unix platforms that support the D-Bus protocol, the QtDBus module provides facilities for inter-process communication, enabling applications to expose services for use by other processes and applications. Services are specified in interface files using a standard XML-based file format, and converted to C++ source code using the `qdbusxml2cpp` tool supplied with Qt.

19.3. Qt Solutions

In addition to all the classes supplied with Qt, Qt Software 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. Almost all of the Solutions made available to Qt 3 developers are also available for Qt 4, and many new Solutions for Qt 4 have already been released.

Online References

http://doc.trolltech.com/4.5/activeqt.html
http://doc.trolltech.com/4.5/intro-to-dbus.html
http://www.qtsoftware.com/products/solutions

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 `qt-interest` mailing list, the *Qt Centre* Web site at www.qtcentre.org, and a number of other community Web sites and Weblogs. In addition, many Qt developers are active members of the KDE community. Qt customers receive our quarterly developers' newsletter, Qt Quarterly. A growing number of commercial and open-source add-ons from third parties are also available; see the Qt Software Web site for the most up-to-date information.

Qt's extensive documentation is available on-line at doc.trolltech.com. There are also a number of books in English, French, German, Russian, Chinese and Japanese, that present and explain Qt programming. Qt's official book is **C++ GUI Programming with Qt 4** (ISBN 0-13-235416-0).

Qt Software and its partners provide a range of training options for Qt and C++, including open enrollment courses for the general public and on-site courses for customers who have more specific training needs. See the online Qt Software training pages for more information.

As well as providing a comprehensive framework for C++ developers, Qt can also be used with a variety of other programming languages. Qt itself includes the QtScript module (see page 37), a JavaScript-oriented technology that enables developers to give users access to restricted parts of their applications for scripting purposes.

Qt Software provides a binding to the Java programming language under the LGPL license – formerly called Qt Jambi.

Language bindings for Python, Ruby, JavaScript, BASIC, Ada 2005, C#, Pascal and Perl are also available from Qt Software's partners and various third parties; many of these solutions are produced and maintained by teams of open source developers.

Developers can evaluate Qt, with support, for 30 days on their preferred platform. For further information, visit the Qt Software Web site.

Online References

http://www.qtsoftware.com/about/partners
http://www.qtsoftware.com/support-services/training
http://lists.trolltech.com/qt-interest/
http://doc.trolltech.com/qq/

About Qt Software:

Qt Software (formerly Trolltech) is a global leader in cross-platform application frameworks. Nokia acquired Trolltech in June 2008, renamed it to Qt Software as a group within Nokia. Qt allows open source and commercial customers to code less, create more and deploy everywhere. Qt enables developers to build innovative services and applications once and then extend the innovation across all major desktop, mobile and other embedded platforms without rewriting the code. Qt is also used by multiple leading consumer electronics vendors to create advanced user interfaces for Linux devices. Qt underpins Nokia strategy to develop a wide range of products and experiences that people can fall in love with.

About Nokia

Nokia is the world leader in mobility, driving the transformation and growth of the converging Internet and communications industries. We make a wide range of mobile devices with services and software that enable people to experience music, navigation, video, television, imaging, games, business mobility and more. Developing and growing our offering of consumer Internet services, as well as our enterprise solutions and software, is a key area of focus. We also provide equipment, solutions and services for communications networks through Nokia Siemens Networks.

Code less.
Create more.
Deploy everywhere.

NOKIA

Oslo, Norway +47 21 60 48 00 ■ Redwood City, CA, USA +1 650-551-1676 ■ Beijing, China +86 10 6588 8836 ■ qt-info@nokia.com www.trolltech.com