Qt Creator Advances with Version 1.2

Qt Creator is the integrated development environment that now comes with Qt, and was released for the first time with Qt 4.5. Its goal is to provide a complete cross-platform IDE for development of Qt applications—making it easier and more pleasant to both get started with, and hang on to, Qt development.

Qt Creator comes with all the features one expects from a modern, fully-fledged IDE. Among other things, it has project management, integrated debugging, and an advanced code editor with syntax highlighting, code folding, and rich search features. Also, the Qt Help System and Qt Designer integrate nicely into the IDE.

Version 1.2 of Qt Creator is now available. It brings with it many new features and improvements based on feedback from Qt Creator’s users. We have especially good news for Windows developers as creator now has new support for debugging applications using the Microsoft Visual Studio toolchain. For a full overview of the changes for this version, you can see http://qtsoftware.com/developer/changes/changes-qtcreator-1.2

A new tutorial shows how to create an application from scratch. You can download Qt Creator either as part of the Qt SDK, or by itself; you can find the packages available for all supported platforms at http://www.qtsoftware.com/downloads.

New Qt for S60 Preview Released

The fifth pre-release from the Qt for S60 porting project, “Tower”, is now available. Based on the Qt 4.5 codebase, the focus in this release has been on improving graphics and network performance, better GUI integration, and porting even more Qt modules to S60.

A detailed list of changes is available at:
http://pepper.troll.no/s60prereleases/packages/changes-4.5.2-tower

The developers have set up a mailing list for technical feedback. To join, send a mail to
mailto://qts60-feedback-request@trolltech.com
with the subject set to “subscribe”. The list is read by all the Qt for S60 developers, so it’s the most direct way to give feedback on the release.

The next release will focus on finishing the experimental features of this release and will be released as part of Qt 4.6.

Designing a Benchmarking Library for Qt

One of our goals for Qt 4.5 was to improve the performance of Qt. To help us achieve this goal we decided to extend QTestLib with support for benchmarking. In this article, we will give a practical example of how you can use this new functionality to benchmark your own classes and applications. Among many candidates for this benchmarking test, we picked out QVector, and decided to test out the performance of its append() function.

There are many interesting questions to ask about QVector::append(). What is the performance in Qt 4.5 compared with Qt 4.4? How does it compare with QList::append()? How does it perform with different vector sizes? We’ll now take a look at how these questions can be answered using the benchmarking extension to QTestLib.

Benchmarking can be done in several ways, ranging from small unit-test-like benchmarks, “How fast is QVector::append()?” to full scale system tests, “Does the GUI on this new handled device stay responsive after spending one hour sending and receiving mails?” Since QTestLib is primarily a unit testing library we decided to add support for unit test benchmarks.

Writing and Running a Simple Test

The standard test function looks something like this:
void tst_QVector::append()
{
  QVector<int> vector;
  QVERIFY(vector.isEmpty());
  vector.append(2501);
  QVERIFY(vector.isEmpty() == false);
}

Extending it with a benchmark, will make it look like this:
void tst_QVector::append()
{
  QVector<int> vector;
  QVERIFY(vector.isEmpty());
  QBENCHMARK
  { vector.append(2501);
  }
  QVERIFY(vector.isEmpty() == false);
}

Run the test the same way as standard test cases and you’ll see the benchmark result appear in the output:
./tst_vector append
RESULT : tst_vector_vs_std::append():
0.000023 msec per iteration
 {total: 25, iterations: 1008576}

This output shows that 0.000023 milliseconds is the average runtime for one iteration, and that the measurements were accumulated over approximately 1M calls to append (the contents of the QBENCHMARK macro is repeated). This gives us a rough estimate of the performance of QVector::append(), which can be used for comparisons with other versions of Qt or QLinkedList::append() for example. One caveat is that calling append modifies the vector, so we’re not measuring how long inserting the first item takes, but rather an
average insertion time as the vector grows. We’ll see how to deal
with this issue later on.

The reason for repeating the contents of QBenchmark is that the
default measurement backend is QTime, which has a resolution of
one millisecond. This is too coarse for a lot of benchmarks, some of
which have a run-time in the range of a few hundred CPU cycles.
All is not lost, however. On some platforms we provide alternative
backends with higher resolution: tickcounter which uses hardware
CPU cycle counters, and callgrind which uses the Valgrind debugging
and profiling tool to count the number of instruction loads by
simulating a CPU.

<table>
<thead>
<tr>
<th>Backend</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tickcounter</td>
<td>Windows, Linux, OS X</td>
</tr>
<tr>
<td>callgrind</td>
<td>Linux, OS X (Qt 4.6 and later)</td>
</tr>
</tbody>
</table>

Example output with tickcounter:

```
./tst_vector append -tickcounter
RESULT : tst_vector::append();
    728 ticks per iteration (total: 728, iterations: 1)
```

More options are available for further tuning. Use the
-iterations option to explicitly set the number of accumulation iterations,
-and_median to run the test function several times and take the median
of the results. In the following example I also use -vb to get verbose
benchmarking output.

```
./tst_vector append -tickcounter -median -vb
QBENCHMARK : tst_vector appealing() warmup stage result:
    134904
QBENCHMARK : tst_vector appealing() accumulation stage result:
    1008
QBENCHMARK : tst_vector appealing() accumulation stage result:
    805
QBENCHMARK : tst_vector appealing() accumulation stage result:
    798
RESULT : tst_vector appealing();
    805 ticks per iteration (total: 805, iterations: 1)
```

One interesting thing here is the warm-up stage. During testing we
found that the cycle counter and Valgrind measures produced usually
large results during the first iterations. Our first guess was
that this was due to CPU cache warm-up effects, but when analyz-
ing the Valgrind call graphs we found that the time was spent by the
dynamic linker resolving the symbols for the functions under test
(for example QVector::append()). This is something we don’t want
to include in this results, so whenever the tickcounter or callgrind
backend is selected QTestLib adds a warm-up stage.

### Using Data Functions

Like standard test functions, benchmarks can also use _data() functions
to run one test function several times with different input
data. This data function creates a 2D data set that can be used to
test QLinkedList::append() and QVector::append() for different
data sizes:

```c
void tst_QVector::appendMany_data()
{
    QTest::addColumn< QByteArray >( "type" );
    QTest::addColumn<int>( "size" );
    for (int size = 10; size < 2000; size += 100 ) {
        const QByteArray sizeString =
            QByteArray::number(size);
        QTest::newRow( "QLinkedList-" +
            sizeString ).addColumn( ) << sizeString;
        QTest::newRow( "QVector-" +
            sizeString ).addColumn( ) <<
            QByteArray("QVector") << size;
    }
}
```

Here is the appendMany() function:

```c
void tst_QVector::appendMany()
{
    QFETCH(QByteArray, type);
    QFETCH(int, size);
    if (type == "QVector") {
        QVector<int> vector;
        QBenchmark {
            for (int i = 0; i < size; ++i)
                vector.append(2501);
        }
    } else {
        QLinkedList<int> list;
        QBenchmark {
            for (int i = 0; i < size; ++i)
                list.append(2501);
        }
    }
}
```

This will produce a lot of output, so to visualize it we can use the
-chart option (available with Qt 4.6, get a preview at

```
./tst_vector append_many -tickcounter -chart
```

The chart is saved to a file named results.html which can be
viewed in any modern Web browser. You can see an image of the
chart below.

![Chart Image](image)

### Summary

In this article, we have seen how the new benchmarking exten-
tion to QTestLib can be used. In this case, we benchmarked QVec-
tor’s append(), and we compared the performance between it
and QLinkedList’s append(). These results were also visualized
using QTestLib.

**Morten Sørvig** is a software engineer at Qt Soft-
ware. This electronics drop-out is obsessed with
multithreaded programming. He originates from
Playa Los Fjordos on Norway’s southern coast.

---

Qt Quarterly is published by Nokia Corporation, Oslo, Norway.
Copyright © 2009 by Nokia Corporation or original authors.
Authors: Kristian Amlie, Kent Hansen, Trenton Schulz, Morten
Sørvig.
Contributors: David Boddie, Knut Yrvin, Kavindra Devi Palaraja.
Production: Katherine Barrios.
Editor: Geir Vattekar.
A selection of Qt Quarterly’s articles and source code is available from
http://doc.trolltech.com/qq/. Potential contributors should contact the editors at
qt.comments@nokia.com.
No part of this newsletter may be reproduced, in any form or by any
means, without permission in writing from the copyright holder.
Nokia, Qt and their respective logos are trademarks of Nokia Corpo-
ration in Finland and/or other countries worldwide. All other prod-
ucts named are trademarks of their respective owners. Produced in
Norway.
New Ways of Using Dialogs

One of the new features in Qt 4.5 is a collection of new methods in QDialog that make it much easier to use application dialogs beyond the traditional modal way. These include window modal dialogs (sheets in Mac OS X) and “live feedback” dialogs. They came as a result of our re-examination of dialogs, which allowed us to create a more unified way of presenting dialogs and more flexibility in using native dialogs—first on Mac OS X and later on the other platforms.

Dialogs and Modalities

The Qt documentation provides the following description for dialogs: “A dialog window is a top-level window mostly used for short-term tasks and brief communications with the user. QDialogs may be modal or modeless.”

Traditionally when using dialogs in Qt, the code looks something like this:

```cpp
MyQDialogSubclass dialog;
// Various bits of initialization
if (dialog.exec() == QDialog::Accept) {
    // Set new values or do extra work
    // based on results.
}
```

This code creates a dialog and then calls its `exec()` method, which halts any further progress in the function until `exec()` returns. This is what is known as an application modal event loop. User events, such as mouse press and key press events, are still dispatched to the window that started the application modal event loop, but not to any other windows in the application (this is the “mode” that modal refers to).

Another way of using dialogs is to treat them like normal windows. This is accomplished by using the standard QWidget methods (i.e., `show()`, `close()`, etc.). When used this way, dialogs don’t block interaction with any of the other application windows. They are said to be non-modal, but they do offer other ways of dismissing the window used by the dialog and communicating results back from it.

There is another way of interaction called window modal or document modal interaction. This mode is between application modal and non-modal. The idea behind a window modal dialog is that the dialog blocks interaction with the window that it is a child of, but does not block the interaction with other windows in the application. An example where this could be used is an application where each “document” is represented in a separate window. Opening up a window modal dialog asking a question, e.g., where to save a file, allows the user to continue working on other windows in the application (or even create more). This example shows when window modality can be preferable compared to putting everything else in the application on hold to deal with one dialog. Naturally, this concept only makes sense for dialogs that have a parent.

Qt supports the concept of window modal dialogs via the windowModality property of QWidget that was introduced in Qt 4.1. This property allows the programmer to select any of the modalities mentioned above. The modality takes effect when the window is shown. By default, all windows are created with windowModality set to Qt::NonModal. So, by default, any window that is shown is non-modal and does not block interaction with other windows, which is what we expect.

The `windowModality` property is also used in the implementation of QDialog::exec(), where the current modality of the dialog is saved, then set to be Qt::ApplicationModal, and then restored back when `exec()` finishes. Assuming we don’t alter the `windowModality` property ourselves, we can see a mapping between functions in QDialog and the `windowModality` property:

- `QDialog::show()` → `Qt::NonModal`  
- `QDialog::exec()` → `Qt::ApplicationModal`

The mapping above does not include `Qt::WindowModal`, however. In the past, the solution was to set the windowModality property to `Qt::WindowModal` before calling `show()` or `exec()`. This works, but it is a bit different than just calling one function, and requires the developer to remember one extra step.

The mapping above is also incomplete when we look at Mac OS X’s concept of sheets. The Apple Human Interface Guidelines says, “A sheet is a modal dialog attached to a particular document or window ensuring that the user never loses track of which window the dialog applies to. Sheets also allow users to perform other tasks before dismissing the dialog, with no sense of the system being ‘hijacked’ by the application.” An example of a sheet is shown in the image above.

If we look at this closely, we see that sheets are simply a specialization of window modal windows that makes them easier to know which window the sheet is modal for. Qt has offered the ability to use sheets since Qt 4.0. Like window modality, it requires some extra thought on the part of the developer. An example of using sheets is provided in Qt Quarterly 18. Here is a reprint of the code:

```cpp
void MainWindow::maybeSave()
{
    if (!messageBox)
        return;

    if (messageBox->exec() == QMessageBox::Yes || messageBox->exec() == QMessageBox::No)
        return;

    connect(messageBox, SIGNAL(finished(int)),
            this, SLOT(finishClose(int)));
    messageBox->show();
}
```

This creates a QMessageBox with the type `Qt::Sheet`, creates a connection from the `finished()` signal to a slot in the main window for finishing the close, and then shows the message box. The code is very straightforward and it is not unreasonable to think that a Mac


OS X developer would write this code, but it’s very unlikely that a cross-platform developer would bother to write it—they would probably just use the static function instead.

The example above shows how to use sheets correctly—that is, as a window modal sheet that returns immediately and doesn’t block the function. The result of what happens in the sheet is handled later in a slot. This ensures that the user interface works correctly with the other windows that are not blocked by the sheet.

We mentioned above that the function should return after showing the sheet and not block until the sheet has finished its run. It is worth exploring why this is. The typical way to prevent the function from returning while still dispatching events is to create a local QEventLoop and quit the event loop when the window closes. A bad example to compare with the code above is shown below:

```cpp
// BAD! DON’T DO SHEETS LIKE THIS!
void MainWindow::maybeSave()
{
    if (!messageBox)
        return;
    QMessageBox::warning,
    QMessageBox::Yes | QMessageBox::Default,
    QMessageBox::No,
    QMessageBox::Cancel | QMessageBox::Escape,
    this, Qt::Sheet);
    messagebox->setButtonText(QMessageBox::Yes,
        isEntitled ? tr("Save...") : tr("Save"));
    messagebox->setButtonText(QMessageBox::No,
        tr("Don’t Save");
)
QEventLoop eventLoop;
connect(messageBox, SIGNAL(closed()),
    &eventLoop, SLOT(quit()));
messageBox->show();
eventLoop.exec();
finishClose(messageBox->result());
}
```

While this works in some cases, it does not work in all situations. Consider the situation where a user has two document windows with unsaved changes and you are handling the window close using code like that shown above. The user clicks the close button on one of the documents and gets a sheet for it. The user then clicks on the close button for the other document and a sheet is shown for this window as well.

The user then goes back and clicks on the “Don’t Save” button on the first sheet that was shown. The sheet is dismissed, but the window doesn’t close, because it is still locked in the second event loop created by the second sheet. Clearly this is not the behavior that is desired.

Another possible place where sheets can be misused is to use them as an application modal sheet. While Qt has allowed this in the past (and Qt’s static file dialogs did exactly this), it is not recommended by the Apple Human Interface Guidelines. It also creates confusion because the sheet doesn’t function this way in other applications. In other words, Qt’s static file dialog functions shouldn’t have been using sheets.

Introducing QDialog::open()

During the Cocoa port, some time was taken to revisit the sheet problem and attempt to make them easier to use.

Another thing that was noticed was that QDialog::exec() always made the dialog work in an application modal way if the modality of the window was Qt::NonModal. It seemed logical that it would make sense to add a function that would do something similar for window modal dialogs. This resulted in the QDialog::open() method.

QDialog::open() will show a dialog as a window modal dialog. On Mac OS X, the dialog will be a sheet. To cooperate correctly with all the event loop, open() returns immediately. This requires you to use some sort of signal-slot connection to finish up the work on the window. Luckily, QDialog has a built-in finished() signal that contains the value that is set when finish() is called, something that both accept() and reject() call.

Of course, nothing stops you from adding your own signals to your QDialog subclasses. The best part is that it works as a drop-in replacement for the code that was written in the earlier Qt Quarterly article, and, as a bonus, we don’t need to specify Qt::Sheet when creating the message box. Instead of calling show(), we call open() and get a sheet that works in a window modal way. You also have the benefit that the application is still able to handle work in other windows. The user can then come back to the sheet when they want to. This works well on all platforms.

With the advent of QDialog::open(), we now have a good mapping of functions to match the different types of modality:

- QDialog::show() -> Qt::NonModal
- QDialog::exec() -> Qt::ApplicationModal
- QDialog::open() -> Qt::WindowModal

This should make choosing window modality for dialogs much easier than in the past. Create your dialog and call the function that matches the modality you want. It is also important for another part of the puzzle, which we show below.

Static Subclass Functions

Another goal was to make it impossible to misuse sheets. The way to do this is to have a strong coupling with window modality. This means that specifying the sheet type is no longer necessary. Also, if you tried to open an application modal sheet, you would instead get a normal application modal dialog. This also means that, whenever you asked for a window modal dialog, you get a sheet instead. This was more a result of porting to Cocoa as Cocoa had no mechanism for showing a window modal dialog without using a sheet.

Having made the decision that application modal dialogs would never be sheets had a consequence for all the static functions that existed in QDialog subclasses; e.g., QColorDialog, QFontDialog, QFileDialog, QInputDialog and QMessageBox. We think in particular of the creator functions, e.g., the QColorDialog::getColor() family of functions, which create a QColorDialog and display it in a modal way. Each function returns the color picked by the user of the dialog—or an invalid QColor if the user canceled the operation.

The problem about these static functions is that, since they require an application modal dialog, they can never use sheets. This may be a bit of a shock for users of QFileDialog since Qt in some cases ran this dialog as a sheet (despite the effects it had on usability). Since other Mac OS X applications can run the file dialog as a sheet, it would be nice to have this ability in Qt as well.

With QDialog::open() giving us a way to access all three types of modality, we decided to see if we could include some additional functionality to these QDialog subclasses. We are pleased with the result and believe that it offers access to more idioms of interacting with documents that required more work in the past.

For many of the subclasses, we added convenience overloads of open() that allow developers to specify the slots for processing results. The dialogs take care of connecting the appropriate signals to these slots, so no extra code is required.
The overloads are given in the following list:

- `QFileDialog::open(QObject *receiver, const char *slot);`
- `QColorDialog::open(QObject *receiver, const char *slot);`
- `QFontDialog::open(QObject *receiver, const char *slot);`
- `QPageSetupDialog::open(QObject *receiver, const char *slot);`
- `QInputDialog::open(QObject *receiver, const char *slot);`
- `QPrintDialog::open(QObject *receiver, const char *slot);`
- `QProgressDialog::open(QObject *receiver, const char *slot);`
- `QPrintPreviewDialog::open(QObject *receiver, const char *slot);`

The idea is to connect the arguments passed to the typical signal that is used for the dialog. Below is a list that shows which signals the various dialogs will connect to.

- `QColorDialog` will connect the slot passed in to its `colorSelected(QColor)` signal.
- `QFontDialog` will connect to the `fontSelected(QFont)` signal.
- `QFileDialog` will connect to the `fileSelected(QString)` or `filesSelected(QStringList)` signal, depending on its mode.
- `QProgressDialog` will connect to its `canceled()` signal.

You can also check out the official Qt documentation to learn more about the signals the dialog will connect to. Doing the setup in `open()` keeps the code clean and simplifies setup of the dialog. You also get the benefit of a native dialog in cases where we can provide it, as is currently the case with `QFileDialog`, `QColorDialog`, `QFontDialog`, and `QPrintDialog`.

### New Interaction

Since we had gone to the extra effort of making native dialogs work with `open()`, it wasn’t much more of a stretch to have it work also when the dialogs are shown non-modally with `show()`. While at first glance this may not seem to be that useful, it opens up a new interaction paradigm that is a standard metaphor on Mac OS X, the “live feedback” dialogs. Something that is incredibly easy to do with Qt.

The static get functions of the `QDialog` subclasses encourage people to write applications where you stop the user to ask a question (such as “which font do you want to use?”). However, this can in some situations slow down the workflow and even become annoying for the user.

For example, consider a scenario in which the user is experimenting with a color using a `QColorDialog`. They have to click through a dialog to choose the color, dismiss the dialog, and then see how it applies. If they are not happy with the color choice, the dialog has to be re-opened before a new color can be selected. Naturally, it is more time consuming and tedious to repeat this process rather than keeping the dialog open while choosing a color and seeing its effects immediately.

One can of course provide alternatives to using dialogs to achieve a non-modal way of experimentation like that described above. In the case of font selection, for instance, Qt has the `QFontComboBox` class, which makes it possible to select fonts in a non-modal way. But it doesn’t have all the power and ease of use that `QFontDialog` has, and using dialogs is often a more intuitive way to go.

With the addition of the `QFontDialog::currentFontChanged()` signal that is emitted whenever the font is changed in the `QFontDialog`, we can connect to this signal, and, by calling `show()`, we have a dialog that doesn’t block the other windows and that provides live feedback to the application. The `QColorDialog` class also benefits from a similar setup. We can show how this is done by reworking the plug-and-paint example to use a live feedback color dialog. It’s straightforward and makes the program feel a bit faster.

```cpp
class MainWindow
{
    Q_OBJECT
    //...
private:
    // ...
    QColorDialog *globalColorDialog;
    // ...
};

class PaintArea
{
    Q_OBJECT
    //...
public slots:
    void setBrushColor(const QColor &color);
    // ...
};
```

Since we will not be using a `QColorDialog` created with one of its static creator functions (e.g., `QColorDialog::getColor()`), we need to keep a pointer to it around. We do that in the main window subclass. We also need move the `QPaintAreas::setBrushColor()` method to a slot.

```cpp
void MainWindow::brushColor()
{
    if (!globalColorDialog)
    {
        globalColorDialog = new QColorDialog(this);
        globalColorDialog->setCurrentColor(paintArea->brushColor());
        globalColorDialog->setOption(QColorDialog::NoButtons, true);
        connect(globalColorDialog,
                SIGNAL(currentColorChanged(QColor)),
                paintArea,
                SLOT(setBrushColor(QColor)));
    }
    globalColorDialog->show();
}
```

We set the color to match that of the current brush, and we set the `QColorDialog::NoButtons` option to avoid showing the OK and Cancel buttons. The main reason for doing this is that they do not make sense here, since we will be setting the color immediately (cancel will not undo the color change). However, it may be problematic on certain window managers on X11 where the window decorations do not include a close button. Finally, we set up a connection between the `currentColorChanged()` signal and the `setBrushColor()` slot that we promoted. We then call `show()` on the dialog. If the color dialog is already visible, the call to `show()` will raise it above other windows.

It comes as no surprise that it’s not much different to do the same for `QFontDialog`. We create a font dialog that we hold onto and hook it up to method in the `PaintArea` that selects a font. You can also look further at the changed example to see how it is done for a live feedback font dialog, and when using sheets for opening and saving files.

### Conclusion

A lot of effort has gone into making this new way of working with dialogs run smoothly, and we think that most people will find these new ways of interacting much more consistent and usable. While there are some issues in Qt 4.5.0 and 4.5.1, most of them have been addressed, so there is no better time to give it a go. We also hope to have more native dialogs hooked up in future versions of Qt.

---

**Trenton Schulz** is a Senior Software Engineer at Qt Software in Oslo. He is soon bidding the Qt development world farewell and beginning a career in researching.
Qt for S60 was released on June 25, 2009. This means that you now can use Qt to write applications for an S60 device. In this article, we give a little insight into the peculiarities of the S60 platform, and show how to best utilize Qt on it.

Let’s start by looking a little bit at how the Symbian operating system and the S60 platform came to be. Symbian was first released back in the early nineties by Symbian Ltd. (under a different name), and was programmed specifically for embedded devices with strict resource constraints. Later on, in the early 2000s, several companies created their own UI specializations on top of Symbian, which resulted in UIQ, S60, S80, S90 and MOAP. Nokia created Series 60, or S60, which soon became their main Symbian based platform.

The Nokia N95 is based on S60.

How does Qt fit into all this? Well, most of the Qt code is layered on top of Symbian directly, and only a relatively small portion layered on top of S60. The latter portion deals with aspects of Qt’s integration into the S60 environment, such as displaying the title and icon in the status pane, and using the editing capabilities for input methods. The remaining Symbian code deals with more generic platform APIs, such as those for painting, window management and networking. This means that Qt sits on top of both of these technologies.

### Building Applications

We really want to build our own application though! Rest assured that for most programs, the process of creating a .pro file and building the project is almost the same as on the other platforms Qt supports. The real difference lies in executing the binary.

Let’s start with the simplest Qt application there is:

```cpp
#include <QtGui>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QPushButton button;
    button.setText("Hello");
    button.show();
    return app.exec();
}
```

This is accompanied by the very simple .pro file:

```bash
SOURCES = main.cpp
```

The build process then works as usual, by executing qmake and make. Running the binary can be done using epoc and the user interface.

But for the emulator, we provide a shortcut in Qt by running make run.

That was easy, wasn’t it? Now, let’s get into a slightly more tricky area: debugging.

For debugging, we use the Carbide C++ IDE. Carbide is based on Eclipse, so if you have used Eclipse before, you should feel at home. First, we need to import our project into Carbide. This is done by choosing Import from the File menu. The latest version of Carbide can import Qt projects directly, so choose that from the available options. On the next screen, locate the .pro file we just made and leave the rest alone. You choose the correct SDK on the next screen.
Hierarchical Finite State Machines in Qt

Qt 4.6 will add a framework for defining and executing hierarchical finite state machines. Qt State Machines allow you to effectively model how some system (a widget, a component, a server process—anything) reacts to events over time; these state machines are a natural extension to Qt’s event-driven programming model. This article gives an overview of the core concepts and functionality of Qt State Machines.

The article is laid out as follows. The first section attempts to give reasons for why you would ever want to use a state machine framework in your Qt application, and why we at Qt Software felt there’s a need for it. The second section gives an overview of state machine terminology and concepts. The third section shows how to get started using the Qt State Machine framework (“Show me the code!”). The fourth section explains the basics of Qt State Machine’s execution model and how it is reflected in the APIs. The fifth section details an example state machine that brings together the features discussed in previous sections. The sixth section talks about error handling. The seventh and final section concludes the article, and offers some pointers about where to go for more information.

Thunderbolts and Lightning

Imagine that you are standing in the middle of a large, open field during a thunderstorm. You are feeling great. Then, all of a sudden, you get struck by lightning, and you fall to the ground, paralyzed. Moments later, you get struck again, thus dispelling the old myth that lightning never strikes in the same place twice. However, since you’re already paralyzed at this point, the second strike inflicts practically no damage (only trimming your eyebrows).

Apart from teaching you that you should never, ever stand in the middle of a large, open field during a thunderstorm, this anecdote illustrates how similar events (in this case, lightning strikes) can have different outcomes depending on which state a system (in this case, you, the protagonist) is in. How would you model this behavior in a computer application?

In a conventional event-driven application, the main loop takes the next event from a queue and dispatches it. The event loop itself is not context-aware: i.e., it does not take previous events into account when deciding how to dispatch the current event. Thus, it is left up to subscribers to that type of event to maintain state and handle the event accordingly. For example, we could image that the developer reimplements the lightningStrikeEvent() handler function as follows:

```cpp
void Protagonist::lightningStrikeEvent(event)
{
    if (!paralyzed) {
        paralyzed = true;
        fallToTheGround();
    }
}
```

The event handler is using a member variable to decide what needs to be done. This code might look innocent enough, but the moment we start introducing more complexity into our model we could get into trouble. For example, the protagonist could be able to pull out a magic conductor gadget from his pocket, providing a shield against lightning. Or, on the third strike of lightning, the protagonist’s moustache could catch fire. This approach can then lead to some serious spaghetti code—non-al dente.

As another (perhaps more familiar) example, consider a standard calculator application. The handling of some input symbols (for example, whether + should be treated as a unary or binary opera-
tor) depends on the calculator’s current state, which in turn was derived from previous input. In this case, also, using variables to implicitly represent and infer states is a proven way of producing buggy code that’s hard to read, verify and maintain—even for something as “trivial” as a calculator (for an enlightening analysis of the calculator case, see Miro Samek’s book “Practical Statecharts in C/C++”).

Who’s really to blame for spaghetti code—is it the application developers? Arguably, yes, but one could also rightfully argue that there should be tools and APIs available that help the developer construct state models for their application. This lets the developer focus on implementing the logic that’s unique to their application (i.e., doing their job!), rather than having to deal with mundane (but oh-so-important) plumbing work as well. Why, if such an API were readily available and easy to use, maybe developers would actually feel inclined to use it, rather than going the way of the pasta. The result would arguably be more robust applications that are easier to maintain and extend, since state would be explicitly embodied in the implementation.

This is exactly the motivation for introducing a state machine framework in Qt. The framework provides you with a means of separating context-determining logic (the ifs of spaghetti code, so to speak) from useful work. In other words, instead of your code always having to check whether some conditions are satisfied before it can do what it really wants to do, you express the logic in a state machine that guarantees a well-defined context at the time a piece of your code is executed.

### State Machine Preliminaries

Before delving into API and code, it can be helpful to briefly review some relevant terminology and concepts.

In mathematics, a finite state machine is defined in terms of an input alphabet, a set of states, an initial state, a set of final states, and a state transition function. Take care not to confuse state machine transitions with graphical (UI) transitions. In state machine terminology, a transition is an instantaneous function, a mapping from (state, input symbol) pairs to the next state. During execution, the current state is the state that the machine is currently in. The machine starts in the initial state, reads the first input symbol, consults the transition function to determine the next state, and transitions to that state. The machine then reads the next input symbol, and so on. Many state machines will typically want to perform some action (useful work) when a state change occurs. When the machine transitions to a final state, it will halt.

A state machine can be thought of as a directed graph where the vertices represent states and the edges represent transitions. By using a simple notation consisting primarily of circles (states) and labeled arrows (transitions), a state machine can be visualized in a state diagram. In software engineering, UML (Unified Modeling Language, see [http://www.uml.org](http://www.uml.org)) is a popular language that provides a notation for drawing state diagrams; there are many tools available, both commercial and non-commercial, that support this syntax.

A well-known usage of finite state machines in software is in lexical analysis and parsing. In this case, the state transition function is typically implemented as a lookup table or a switch statement. The input alphabet typically consists of values of a native type (e.g., characters/ integers), and the actions are embedded in the state machine implementation itself, which may be automatically generated, e.g., from a grammar definition file. This is a highly specialized form of state machine, and is great as such, but it can be hard to adapt to model other kinds of behavior (with different input alphabets), since that’s not what the tool/language was designed for.

In the earlier definition of a finite state machine, all states are atomic—there is no concept of nested states. When the number of states grows large, it becomes difficult for human brains to manage such a machine. Furthermore, there is no way to capture redundancy in the machine (i.e., a subset of the states should share behavior under a certain input).

In a hierarchical state machine, states can have child states (which can have child states, and so on). Such child states are also known as sub-states. In such a machine, the machine’s current state actually refers to a set of states, known as the machine’s current configuration. This set contains the innermost (atomic) active state, but also the ancestors of that state.

State composition provides abstraction and encapsulation which makes it possible for mere mortals to manage very complex state machines. It makes it possible to construct state machines that can be “plugged into” other state machines, or that can be further specialized and extended.

With nested states, transitions that originate from a sub-state can override transitions that originate from the parent state; in other words, if a sub-state doesn’t handle an event, the parent state gets a chance to handle it instead. This is a very powerful mechanism, and is similar in concept to class-based inheritance (where you create a subclass and either reimplement a method, or rely on the base implementation); hence, the mechanism is commonly referred to as behavioral inheritance.

Other important features that hierarchical state machines can provide are the ability to define states whose child states are entered in parallel and history states that enable save/restore functionality for composite states. In total, such a hierarchical state machine “toolbox” enables you to succinctly model a wide range of behavior.

The additional power that a hierarchical state machine provides over a flat state machine comes at a price: increased complexity of the state machine machinery. Rather than attempting to invent our own wheels (only to helplessly watch them all fall off at the first turn), the Qt State Machine framework adopts the algorithm from the SCXML ([http://www.w3.org/TR/scxml/](http://www.w3.org/TR/scxml/)) specification developed by the World Wide Web Consortium (W3C). Qt Software is actively contributing to this open standard.

### Getting Started

The core Qt State Machine C++ API is part of the QTCore module, so it’s instantly available to any application that uses Qt 4.6 and beyond. Here’s a small example: it shows a button that says “Hello, states!” and quits when the button is clicked:

```cpp
int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QPushButton button;
    QStateMachine machine;
    QState *s0 = new QState(machine.rootState());
    s0->setProperty(&button, "size", QSize(200, 100));
    s0->addTransition(&button, SIGNAL(clicked()), s1);
    QObject::connect(s0, SIGNAL(entered()), &button, SLOT(show()));
    QFinalState *s1 = new QFinalState(machine.rootState());
    s1->addTransition(&button, SIGNAL(clicked()), s1);
    QObject::connect(&machine, SIGNAL(finished()), &app, SLOT(qtExit()));
    machine.setInitialState(s0);
    machine.start();
    return app.exec();
}
```
The following paragraphs explain what’s going on in this example.

The QStateMachine class represents a state machine; you can create as many instances of this class as you want, and they run independently of each other. You populate the QStateMachine object with QState objects which represent the possible states of the machine; in the previous example, there are only two states, s0 and s1. (Although not shown in this example, creating nested states is only a matter of passing another state—the parent state— to the QState constructor.)

The QState::assignProperty() function is used to specify the value of an object’s property “in the future”, that is, when the state is actually entered. In the example, the size and text properties of the button will be set when state s0 is entered. When a state is entered, it emits the QState::entered() signal; in the example, we connect this signal of s0 to the button’s show() slot. (Although not shown in the example, a corresponding signal is emitted when a state is exited, QState::exited().)

The QState::addTransition() function is used to add a transition from one state (the transition’s source) to another (the transition’s target). The previous example creates a transition based on a Qt signal, in this case a button’s clicked() signal. Consequently, when the button is clicked, the machine will transition from its current state, s0, to the transition’s target state, s1, and s1 becomes the current state.

Once the machine transitions to a top-level final state, it will emit the QStateMachine::finished() signal. (Note that a state machine doesn’t have to have any final states, though; it can run forever if it wants.) Final states are represented by the QFinalState class; simply create a QFinalState object and use it as the target of one or more transitions. In the example, the machine’s finished() signal is connected to the application’s quit() slot; the effect is that the application will quit as soon as the final state, s1, is entered.

Before a state machine can be started, its initial state must be set by calling the QStateMachine::setInitialState() function. The machine may then be started by calling the QStateMachine::start() function. The machine executes asynchronously, so nothing will actually happen until the application’s event loop is entered. At that time, the machine will enter the initial state, and wait for events to process.

Events and Transitions

Using only the basic API covered in the previous section, it’s already possible to construct useful state machines. Before we can take it to the next level, however, we need to peek behind the curtain and understand state machine events and how they relate to transitions.

Each QStateMachine object maintains its own event queue, consisting of QEvent objects. Similar to how you can post events to application objects using QCoreApplication::postEvent(), you can post events to a QStateMachine object by calling QStateMachine::postEvent(). If you’ve ever used custom events for things like thread communication in Qt, and reimplemented QObject::event() to handle those events, this should sound familiar; events and signals are the principal ways in which Qt objects communicate.

When a state machine receives an event, it will basically ask all transitions that have the current state as their source state, “Should you be triggered by this event?” If a transition says, “Yes,” the state machine will transition to that transition’s target state. In this way, the state machine can be said to provide the context (by only asking those transitions that originate from the current state), and the transitions are the “direction indicators” that tell the state machine which path to follow next.

Transitions are represented by QAbstractTransition objects. The QAbstractTransition class defines an interface that the state machine uses to 1) query a transition whether it should be triggered by a particular event (the QAbstractTransition::eventTest() function); and 2) inform the transition that it has been taken, thus giving the transition a chance to do additional processing if it wants to (the QAbstractTransition::onTransition() function).

The transition’s eventTest() function receives the QEvent object to test as its argument. The first thing an implementation of this function typically does is check the event’s type; it might then go on and check attributes of the event, thus making it a conditional, or guarded, transition for that event type. For instance, a particular transition that handles key events might be triggered when the key is A, but not by any other key.

At this point you might be saying, “So hold on...If the state machine is asynchronous and event-based, how do signal transitions work?” Glad you asked! What in fact happens behind the scenes is that the state machine constructs a QSignalEvent object that represents the signal emission, and posts the event to itself. The signal event contains, in addition to the sender object and signal ID, the arguments (if any) that the signal carried. There is a matching class QSignalTransition that handles signal events. QSignalTransition’s eventTest() implementation returns true only if the event’s type is QEvent::Signal and the sender object and signal ID of the signal event match those of the transition itself.

To create a signal transition that’s conditional on the signal arguments as well (for example, a transition that should be triggered only if the first argument to the signal is a number greater than 10), you would subclass QSignalTransition and reimplement eventTest(), where you have access to the signal arguments through the QSignalEvent::arguments() function.

Example: String Comparator

This section shows how to create a state machine that uses custom events and custom transitions. It also shows how to use nested—including parallel—states.

We examine a small application that reads and matches a series of characters, i.e., a parser of sorts, implemented using the state machine API. In a real-world application you would probably not use the Qt State Machine framework for such low-level data processing (although you certainly can, as the example application shows!) since there are highly specialized tools and languages available for that (e.g., parser generators like yacc). For API illustration purposes, however, the author believes the example is suitable.

Informally, the behavior of the state machine can be stated as follows. The machine reads characters from a stream, one at a time. As each character is read, it is compared to an expected character. If the character is different from the one expected, the state machine should print an error and terminate. Likewise, if the end of the character stream is reached before all expected characters have been encountered, the state machine should print an error and terminate. When all expected characters have been matched and there are no more characters in the reader’s stream, the machine terminates silently, indicating success.

The state machine consists of two principal parts: a reader and a matcher. Each of these is defined as a composite state, and they run in parallel; the reader and matcher don’t communicate directly, but rather by the reader posting the characters as custom events to the state machine, one at a time. When the reader reaches the end of the character stream, it will post an End-Of-File (EOF) event. The matcher, meanwhile, attempts to match the characters against
those of another particular string, until it has matched everything and receives the EOF event. Error handling is achieved by having “catch-all” transitions for character events and EOF events in the top-level matcher state. If the event is not accepted by the active sub-state (e.g., because a character fails to match), the parent state gets to handle the event instead, and it has a custom transition that treats the event as an error. The diagram below illustrates this design.

We define a custom `CharEvent` type as follows:

```cpp
class CharEvent : public QEvent {
public:
    CharEvent(const QChar &value)
        : QEvent(QEvent::Type(QEvent::User+1)), m_value(value) {}  
    QChar value() const { return m_value; } 
private:
    QChar m_value;
};
```

We define a corresponding transition type that recognizes `CharEvents`:

```cpp
class CharTransition : public QAbstractTransition {
public:
    CharTransition() {}  
protected:
    bool eventTest(QEvent *e) {
        return (e->type() == QEvent::User+1); }
    void onTransition(QEvent *) {} 
};
```

We define a similar pair of classes for End-Of-File events:

```cpp
class EofEvent : public QEvent {
public:
    EofEvent() : QEvent(QEvent::Type(QEvent::User+2)) {} 
};

class EofTransition : public QAbstractTransition {
public:
    EofTransition() {}  
protected:
    bool eventTest(QEvent *e) {
        return (e->type() == QEvent::User+2); }
    void onTransition(QEvent *) {} 
};
```

We define a small helper function that generates a new event from the character stream (a `QTextStream` object):

```cpp
void generateStimulus(QTextStream &in, QStateMachine &machine)
{
    if (!in.atEnd()) {
        QChar value;
        in >> value;
        machine->postEvent(new CharEvent(value));
    } else {
        machine->postEvent(new EofEvent());
    }
}
```

We define a transition that posts a new event every time it receives a `CharEvent`:

```cpp
class ReadTransition : public CharTransition {
public:
    ReadTransition(QTextStream &in)
        : m_in(in) {}  
protected:
    void onTransition(QEvent *) {
        generateStimulus(m_in, machine()); }
private:
    QTextStream &m_in;
};
```

We define a custom state that, when entered, posts a new input event. The state has a self-transition (a `ReadTransition` object) that effectively makes the state “loop” until there are no more `CharEvents` posted:

```cpp
class ReadState : public QState {
public:
    ReadState(QTextStream &in, QState *parent = 0)
        : QState(parent), m_in(in) {
        addTransition(new ReadTransition(in));
    }
protected:
    void onEntry(QEvent *) {
        generateStimulus(m_in, machine()); }
private:
    QTextStream &m_in;
};
```

We can now start to build up the state machine and the composite reader state. Firstly, we create a parallel top-level group state (the matcher will later be added to this group as well):

```cpp
QStateMachine machine;
QState *group = new QState(machine.rootState());
```

We define the composite reader state. It consists of two sub-states: one that’s active as long as there are characters to read, and a final sub-state that’s entered when End-Of-File is encountered. For simplicity, the string of characters the reader receives always spells “Qt 4.6”. The diagram below shows the sub-states of the reader.

Here is the code that builds the reader:

```cpp
QState *reader = new QState(group);
QString str("Qt 4.6");
QTextStream in(&str);
```

```cpp
    ReadState *read = new ReadState(in, reader);
    QFinalState *eof = new QFinalState(reader);
    EofTransition *trans = new EofTransition();
    trans->setTargetState(eof);
    read->addTransition(trans);
    reader->setInitialState(read);
}
```

At this point, if we run the state machine, it will generate events for all the characters from the stream, and finish when End-Of-File is reached, as you would expect. However, it doesn’t do any useful processing of those characters. Which brings us to the matcher.
We define a transition that matches a particular character:

```cpp
class GuardedCharTransition : public CharTransition
{
public:
    GuardedCharTransition(const QChar &value) :
        m_value(value) {}

protected:
    bool eventTest(QEvent *e)
    {
        if (!(CharTransition::eventTest(e))
            return false;
        QChar ce = static_cast<QChar*>(e)->value();
        return (m_value == ce->value());
    }

    QChar m_value;
};
```

The diagram illustrates the matcher’s sub-states; notice that we create one sub-state for each character in the input string.

We define a transition that matches a particular character:

```cpp
class GuardedCharTransition : public CharTransition
{
public:
    GuardedCharTransition(const QChar &value) :
        m_value(value) {}

protected:
    bool eventTest(QEvent *e)
    {
        if (!(CharTransition::eventTest(e))
            return false;
        QChar ce = static_cast<QChar*>(e)->value();
        return (m_value == ce->value());
    }

    QChar m_value;
};
```

The diagram illustrates the matcher’s sub-states; notice that we create one sub-state for each character in the input string.

We define a helper function that creates a composite matcher state from a string.

```cpp
QState *createMatcher(const QString &str, 
QState *parent = 0)
{
    QState *matcher = new QState(parent);
    QState *s = new QState(matcher);
    matcher->setInitialState(s);
    for (int i = 0; i < str.length(); ++i) {
        QState *ss = new QState(matcher);
        GuardedCharTransition *trans =
            new GuardedCharTransition(str.at(i));
        trans->setTargetState(ss);
        ss = addTransition(trans);
    }
    QFinalState *done = new QFinalState(matcher);
    EofTransition *trans = new EofTransition();
    trans->setTargetState(done);
    s->addTransition(trans);
    s = ss;
}
```

We create the matcher as a child of the parallel state group (the string to match is taken to be the first argument passed to the application):

```cpp
QString arg(QString::fromLatin1(argv[1]));
QState *matcher = createMatcher(arg, group);
```

If everything goes well, both the reader and the matcher will enter their final sub-states, which will cause the parallel group state to finish, which will cause the state machine to transition to a top-level final state, which will cause the state machine to finish:

```cpp
QFinalState *doneGood = new QFinalState(
    machine.rootState());
group->addTransition(group, SIGNAL(finished()),
    doneGood);
```

### Error Handling

We define transitions that handle errors; these are specializations of the generic CharTransition and EofTransition classes:

```cpp
class ExpectCharTransition : public CharTransition
{
public:
    ExpectCharTransition() {}

protected:
    void onTransition(QEvent *e)
    {
        CharEvent *ce = static_cast<CharEvent*>(e);
        fprintf(stderr, "expected '%c'\n",
            ce->value().toLatin1());
    }
};
```

```cpp
class PrematureEofTransition : public EofTransition
{
public:
    PrematureEofTransition() {}

protected:
    void onTransition(QEvent *e)
    {
        fprintf(stderr, "premature end of file\n";
    }
};
```

The transitions are added to the matcher state, and will cause the machine to transition to a “bad” final state:

```cpp
QFinalState *doneBad = new QFinalState(machine.rootState());
```

```cpp
PrematureEofTransition *trans =
    new PrematureEofTransition();
trans->setTargetState(doneBad);
matcher->addTransition(trans);
```

```cpp
ExpectCharTransition *trans =
    new ExpectCharTransition();
trans->setTargetState(doneBad);
matcher->addTransition(trans);
```

When the state machine finishes, the machine’s configuration can be consulted to determine whether it did “good” or “bad”.

```cpp
ds./stringmatcher "Qt 4.6"
prints nothing (success).
```

```cpp
ds./stringmatcher "Qt 4.5"
prints expected ‘6’.
```

```cpp
ds./stringmatcher "Qt 4.4"
prints expected ‘.’.
```

```cpp
ds./stringmatcher "Qt 4.6.1"
prints premature end of file (file here refers to the reader’s character stream; i.e., the fixed string “Qt 4.6”).
```

### Wrapping Up

The Qt state machine framework scheduled for Qt 4.6 provides a powerful and robust way of defining logic within Qt applications. The framework integrates deeply with Qt’s meta-object system and event system, and is a natural extension to Qt’s foundations.

There are a lot of Qt State Machine goodies that haven’t been covered in this article; for example, how to use history states to model pause-and-resume behavior, and how to use the state machine framework together with the animation classes (also to be introduced in Qt 4.6). Naturally, the official Qt documentation is a good resource for more details and working examples. The blogs on the Qt Labs Web site is also a place to watch for tips, tricks and new developments. Finally, be sure to check out the QtSCXML project in the Labs, which aims to provide full SCXML support based on the Qt State Machine framework.

**Kent Hansen** is a Software Engineer at Qt Software. When he’s not busy helping the Qt State Machines take over the world, he enjoys playing the harmonica and the Irish whistle.
## Qt News

### Qt Development Opened to the Community

On May 11, Qt Software started the process of opening the development of Qt and related projects to the Qt user and developer community. The first step in this process is the opening of Qt's source code and revision history via a publicly accessible source code management system based on the Git version control system.

Interested parties can access the source code via the Gitorious Web interface at [http://qt.gitorious.org/](http://qt.gitorious.org/) or use their favorite Git front-end to obtain a copy of the code and its history.

Motivated developers can submit changes to Qt via an online contribution system that enables both contributors and core Qt developers to keep track of new code and cleanly integrate accepted changes into official Qt releases.

Some discussion of core Qt development now takes place on the #qt-labs IRC channel at Freenode ([http://freenode.net](http://freenode.net)), complementing the existing #qt discussion channel for users.

### Qt 4.5.2 and Qt Creator 1.2 Released

Nokia has released Qt 4.5.2. This release coincides with version 1.2 of Qt Creator. In addition, Nokia has provided additions and updates to the Qt Visual Studio Add-In and the Qt Eclipse Integration.

Qt 4.5.2 includes bugfixes and optimizations made since the release of Qt 4.5.1. Aside from these, the release also includes a new example (Fancy Browser) that shows how to use jQuery in QtWebKit. You can find a detailed list of changes here: [http://www.qtsoftware.com/developer/changes/changes-4.5.2](http://www.qtsoftware.com/developer/changes/changes-4.5.2)

Both Qt 4.5.2 and Qt Creator 1.2 are available in a new build of the Qt SDK (Build 2009.3). The Qt SDK contains everything needed to begin cross-platform development with Qt, and is freely available for download at [http://www.qtsoftware.com/downloads](http://www.qtsoftware.com/downloads).


### Qt Mobility

Qt Mobility is a project within Qt Software to create a new suite of Qt APIs for mobile device functionality with the aim of enabling cross-platform mobile application development.

The new APIs aim to make Qt an even more comprehensive framework. Developers will be able to write cross-platform Qt applications that take advantage of mobile device functionality, making it possible to create rich applications targeting Series 60, Windows CE and Maemo.

You can find it here: [http://qt.gitorious.org/qt](http://qt.gitorious.org/qt)

### KDE Plasma for Netbooks

The popularity of ultra-portable and small netbooks presents several problems for established user interface managers like KDE and GNOME. Most of these result from the reliance on large screen resolutions that desktop users have enjoyed for so long. This requires the developers of these environments to find new solutions and perhaps move away from the “desktop” concept altogether.

KDE has now started development of Plasma for netbooks. The Plasma executable for netbooks is simpler than the one used for desktops. It has a main view (we don’t use the word desktop here) in which it can show an activity; i.e., a GUI program. For now, the project is focusing on two activities. Firstly, a full screen application launcher, document browser, search interface, etc. Secondly, a sneak peek at it in our public Gitorious repository. We are also integrating it into the Qt mainline very soon, so you will be able to play around with it long before the 4.6 release, which is planned for later in the year.

You can find it here: [http://qt.gitorious.org/qt](http://qt.gitorious.org/qt)

## Qt Developer Days 2009

Nokia, Qt Software is pleased to announce the dates for our sixth annual conference, Qt Developer Days 2009! Each event offers two full days of in-depth technical workshops and seminars on Qt topics, plus a day of introductory seminars and training sessions.

As usual, attending Developer Days gives you the chance to ask core Qt developers your toughest Qt questions and gives you the opportunity to network with other Qt users and developers in the wider world.

### Munich, Germany

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>October 12</td>
<td>Pre-Conference Seminars and Training</td>
</tr>
<tr>
<td>October 13–14</td>
<td>Exhibition, Keynotes and Seminars</td>
</tr>
</tbody>
</table>


### San Francisco, California

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2</td>
<td>Pre-Conference Seminars and Training</td>
</tr>
<tr>
<td>November 3–4</td>
<td>Exhibition, Keynotes and Seminars</td>
</tr>
</tbody>
</table>


### Multi-Touch API

Devices that come with multi-touch capabilities have become more commonplace in recent times—for instance, Apple’s iPhone and iPod touch—and desktops are quickly catching up with this trend—for instance, Dell and HP have multi-touch Tablet PCs. At Qt Software, we are frequently asked when Qt will contain support for such devices.

Qt Software is developing a new multi-touch API that is planned to be released with 4.6. However, it is already possible to take a sneak peek at it in our public Gitorious repository. We are also integrating it into the Qt mainline very soon, so you will be able to play around with it long before the 4.6 release, which is planned for later in the year.

You can find it here: [http://qt.gitorious.org/qt](http://qt.gitorious.org/qt)

### Multi-Touch API

- Devices that come with multi-touch capabilities have become more commonplace in recent times—for instance, Apple’s iPhone and iPod touch—and desktops are quickly catching up with this trend—for instance, Dell and HP have multi-touch Tablet PCs. At Qt Software, we are frequently asked when Qt will contain support for such devices.
- Qt Software is developing a new multi-touch API that is planned to be released with 4.6. However, it is already possible to take a sneak peek at it in our public Gitorious repository. We are also integrating it into the Qt mainline very soon, so you will be able to play around with it long before the 4.6 release, which is planned for later in the year.

You can find it here: [http://qt.gitorious.org/qt](http://qt.gitorious.org/qt)

### KDE Plasma for Netbooks

The popularity of ultra-portable and small netbooks presents several problems for established user interface managers like KDE and GNOME. Most of these result from the reliance on large screen resolutions that desktop users have enjoyed for so long. This requires the developers of these environments to find new solutions and perhaps move away from the “desktop” concept altogether.

KDE has now started development of Plasma for netbooks. The Plasma executable for netbooks is simpler than the one used for desktops. It has a main view (we don’t use the word desktop here) in which it can show an activity; i.e., a GUI program. For now, the project is focusing on two activities. Firstly, a full screen application launcher, document browser, search interface, etc. Secondly, a newspaper article, which can scroll a view of selectable widgets on the screen; this can, for instance, be used to select between running applications.

Development is still in an early phase, but it looks promising. You can find information about it at the following site: [http://www.notmart.org/index.php](http://www.notmart.org/index.php)