

Qt auf der Managed Platform

Nach reinen GUI-Toolkits wie Gtk oder wxWindows hält nun ein wahrer Allrounder Einzug in die Welt von Mono und .NET: Mit Qyoto ist Qt4 nun auch von Managed Code aus verfügbar.

Qyoto basiert, wie viele Qt- beziehungsweise KDE-Bindings, auf dem Qt-Wrapper SMOKE. SMOKE wird bereits erfolgreich bei den Projekten PerlQt und QtRuby eingesetzt. Der Wrapper arbeitet wie eine große Bibliothek von Qt-Methoden, die alle mit einer ID gekennzeichnet sind. Um eine bestimmte Methode aufzurufen, benötigt man nur ihre ID und eventuell einen Zeiger zu einer gültigen Instanz eines Qt-Objekts. Dies alles wird von der Qyoto-Runtime verwaltet.

Mit „Qyoto“ wird nur die Qt4-Version der C#-Bindings bezeichnet. Die KDE4-Bindings werden später „Kimono“ genannt werden.

Installation

Bis jetzt ist Qyoto nur auf Linux getestet worden, wobei es mit einigen Modifikationen auch problemlos auf Windows oder

Mac OS X laufen sollte. Im Folgenden werde ich von einem Linux-System mit einer aktuellen Mono-Version ausgehen.

Benötigt wird vorab Qt4 [1], CMake [2] sowie eine aktuelle Version von QtRuby-4 [3]. Grundkenntnisse in Sachen Konsole werden vorausgesetzt.

QtRuby-4 wird nur benötigt, da in diesem Paket eine aktuelle Version von SMOKE enthalten ist. Von dieser Bibliothek wurde bis jetzt noch kein separates Release erstellt.

Für alle drei Programme beziehungsweise Bibliotheken sollten Pakete für Ihre Distribution verfügbar sein. Falls dies nicht der Fall ist, müssen Sie sie selbst kompilieren. Ich werde darauf hier allerdings nicht näher eingehen. Für Qt4 und, falls verfügbar, auch für QtRuby-4 müssen weiterhin die entsprechenden Entwicklerpakete installiert werden. Diese haben meist die Endung *-devel*, beispielsweise *qt-4.2.3-devel*.

Bis zum Redaktionsschluss lag leider noch kein Release von Qyoto vor. Allerdings läuft es schon sehr stabil, und das Release ist in den nächsten Wochen zu erwarten. Bis zum Erscheinen dieser Ausgabe könnte ein Release vorhanden sein. Dies können Sie auf der Homepage von Qyoto [4] überprüfen. Falls dem nicht so sein sollte, müssen Sie sich eine Version aus dem SVN herunterladen.

Dazu öffnen Sie eine Konsole, navigieren zu einem passenden Verzeichnis (zum Beispiel *~/src*) und geben dort folgendes Kommando ein:

```
svn co
svn://anonsvn.kde.org/home/kde/trunk/KDE/
kdebindings/csharp/ qyoto
```

Damit laden Sie die aktuelle Entwicklungsversion von Qyoto aus dem SVN-Repository von KDE in den Ordner *qyoto* herunter.

Mit *cd qyoto* wechseln Sie in selbiges Verzeichnis und geben dort zum Konfigurieren *cmake .* ein (den Punkt beachten, dieser ist wichtig!) und danach *make* zum Kompilieren.

Falls alles ohne Fehlermeldung durchläuft, ist Qyoto kompiliert und bereit für den ersten Einsatz. Falls nicht, überprüfen Sie bitte, ob alle benötigten Pakete installiert sind.

Da die Binärdateien erst mit dem Release von Qyoto automatisch installiert werden, sind Sie im Moment darauf angewiesen, mit sogenannten Symlinks zu arbeiten. Ein Symlink ist eine symbolische Verknüpfung, also keine echte Kopie einer Datei oder eines Verzeichnisses, sondern nur ein Verweis.

Als letzten Schritt erstellen Sie ein Sandkastenverzeichnis, in dem Sie nach Belieben mit Qyoto herumexperimentieren können. Dafür gehen Sie auf der Konsole wieder eine Ebene nach oben (*cd ..*), erstellen dort einen Ordner namens *qyototest* und navigieren auch gleich in diesen hinein (*mkdir qyototest; cd qyototest*). Hier erstellen Sie Symlinks zum C#- als auch zum C++-Teil von Qyoto:

```
ln -s ../qyoto/qyoto/libqyoto.so .
ln -s ../qyoto/qyoto/qt-dotnet.dll .
```

Achten Sie auch hier auf die Punkte am Ende!

Damit ist das Testverzeichnis mit allen benötigten Dateien eingerichtet.

Erste Schritte

Listing 1 zeigt ein erstes Programm. Der Code ist so gut wie selbsterklärend, dennoch will ich ihn etwas näher erläutern:

Alle Qt-Klassen befinden sich im Namensraum *Qyoto*, also holen wir diesen mit *using Qyoto* herein. *QApplication* ist die Klasse, die zuständig für die Main

Auf einen Blick

Autor



Arno Rehn programmiert seit seinem 9. Lebensjahr. Seit 2005 entwickelt er aktiv am Qyoto-Projekt mit. Sie erreichen ihn über arno@arnorehn.de.

dotnetpro.code
A0708Mono

Sprachen C#

Technik GUI-Programmierung mit Qyoto

Voraussetzungen Linux, Mono > 1.2, Qt4, SMOKE, CMake

Listing 1

Hallo, Welt!

```
using System;
using Qyoto;

class MainClass : QObject {
    public static void Main(string[] args) {
        new QApplication(args);
        QPushButton button =
            new QPushButton("Hello World!");
        button.Resize(100, 30);
        button.Show();
        QApplication.Exec();
    }
}
```

Event Loop ist. Durch sie läuft das Programm erst. Wichtig: Von den meisten Qt-Klassen kann keine Instanz erstellt werden, solange noch keine *QApplication*-Instanz existiert. Die Main Event Loop muss dazu nicht gestartet werden, allerdings muss eine Instanz vorhanden sein!

Danach erstellen wir einen einfachen Button mit der Aufschrift *Hello World!* mittels der Klasse *QPushButton*. Dieser wird auf die Größe 100x30 Pixel gebracht und angezeigt. Nun wird die Main Event Loop gestartet.

Was sicherlich ins Auge fällt, ist, dass wir kein Hauptfenster benötigen, in dem wir Buttons oder Ähnliches erstellen. In Qt kann jede Subklasse von *QWidget* (das ist die Klasse, von der alle Steuerelemente erben) als eigenständiges Fenster angezeigt werden. Weiterhin sind die meisten Methoden von *QApplication* statisch, sodass wir keine direkte Objektinstanz benötigen.

Auch wird die Main Event Loop automatisch beendet, sobald das letzte Fenster geschlossen ist. Um das Ganze auszuprobieren, schreiben wir obigen Code in eine Datei *test1.cs* in unserem Sandkastenordner. Danach wird das ganze wie folgt kompiliert:

```
gmc -r:qt-dotnet.dll -out:test1.exe test1.cs
```

Achtung: Die *qt-dotnet.dll* (also die C#-Assembly unseres Wrappers) wird noch nicht in den GAC installiert, genauso wenig wie die *libqyoto.so* in den Linker-Suchpfad kopiert wird. Daher müssen beide Dateien immer in dem Verzeichnis liegen, in dem die Applikation ausgeführt wird! Beim Kompilieren muss zur *qt-dotnet.dll* dann gegebenenfalls ein absoluter Pfad angegeben werden. Um unser Programm nun auszuführen, reicht

ein einfaches *mono test1.exe*. Erweitern wir diesen Code nun noch etwas (siehe Listing 2).

Am Programmcode hat sich wenig geändert, außer dass nun ein merkwürdiges *Connect* mit *SIGNAL* und *SLOT* hinzugekommen ist.

Hier sind wir bei einem der Kernfeatures von Qt angelangt: Dem Signal/Slot-System, Qt's Eventsystem. Jede Klasse, die von *QObject* ableitet, kann mehrere Signals und Slots haben. Ein Signal ist vergleichbar mit einem Event in C#. Es wird ausgelöst, wenn etwas Bestimmtes im Programm passiert ist, auf das der Programmierer reagieren kann. Ein Slot ist eine Methode, die mit einem Signal verbunden („connected“) werden kann, um so auf das Signal reagieren zu können. Weiterhin ist es auch möglich, Signale mit Signalen zu verbinden, sodass beim Auslösen eines bestimmten Signals ein anderes emittiert wird.

Zu beachten ist noch, dass sowohl Signals als auch Slots eigentlich Methoden sind, sodass bei der Angabe die Klammern am Ende des Namens notwendig sind. Natürlich können den Slots auch Signale mit Parametern übergeben werden, dazu aber später mehr.

Schauen wir uns nun noch einmal obigen Code an. Wir verbinden das Signal *clicked()* mit der Methode *quit()* des Objektes *qApp*. Damit Qt weiß, ob wir nun ein Signal zu einem Slot oder einem anderen Signal verbinden, sind die Methoden *SIGNAL* beziehungsweise *SLOT* notwendig. Diese bearbeiten den übergebenen String entsprechend, sodass Qt den Unterschied erkennen kann.

Diese beiden Methoden sind in der Klasse *Qt* implementiert, von der jede andere Klasse in Qyoto ableitet, sodass wir meist nicht explizit diese Klasse angeben müssen. *Connect* jedoch befindet sich in der Klasse *QObject*, sodass wir, falls wir nicht von *QObject* ableiten, *QObject.Connect* schreiben müssen.

Weiterhin ist *qApp* eine globale Variable, die nach einem Aufruf von *new QApplication(...)* eine Instanz von eben diesem *QApplication*-Objekt enthält.

Kompilieren wir das Programm nun wieder und führen es aus:

```
gmc -r:qt-dotnet.dll -out:test2.exe test2.cs
mono test2.exe
```

Wenn wir nun auf den Button klicken, wird sich das Programm beenden.

Erstellen wir eine weitere Datei, *test3.cs*, mit dem Inhalt von Listing 3.

Listing 2

Ein SIGNAL mit SLOT.

```
using System;
using Qyoto;
class MainClass : QObject {
    public static void Main(string[] args) {
        new QApplication(args);
        QPushButton button =
            new QPushButton("Hello World!");
        button.Resize(100, 30);

        Connect(button, SIGNAL("clicked()"),
            qApp, SLOT("quit()"));

        button.Show();
        QApplication.Exec();
    }
}
```

In einem selbst erstellten Widget wird hier komplett gezeigt, wie man Signals und Slots erstellt und einsetzt. Darüber hinaus wird die Verwendung von Layouts dargestellt. Bleiben wir aber zunächst bei den Signals und Slots:

Wir verbinden beim ersten *Connect* das Signal *clicked()* des Buttons mit einem Slot namens *ButtonClicked()* der aktuellen Klasse. Wo kommt dieser nun aber her? Schauen wir eine Methode weiter unten, dort haben wir ja unsere Methode *ButtonClicked()*. Damit diese auch als Slot erkannt wird, markieren wir sie mit dem Attribut *Q_SLOT*.

In der Methode selbst wird nun zuerst Text auf der Konsole ausgegeben. Danach wird ein benutzerdefiniertes Signal ausgelöst namens *TestSignal()*. Dafür wird die Eigenschaft *Emit* verwendet, die Zugriff auf die Signals gibt. Wo kommt dieses Signal nun aber her? Schauen wir ans Ende des Codes – dort ist ein Interface definiert, und dort finden wir auch unser Signal wieder. Es ist, ähnlich wie Slots mit einem *Q_SLOT*-Attribut, mit einem *Q_SIGNAL*-Attribut markiert. Das Interface selbst erbt von dem Signal-Interface der Elternklasse, *IQWidgetSignals*. Um aber überhaupt unser Signal „aufrufen“ zu können, müssen wir die Eigenschaft *Emit* überschreiben, damit diese auch unser neues Interface zurückgibt. Ansonsten könnten wir nur Signale der Elternklasse, in diesem Fall *QWidget*, auslösen. Der Rückgabewert von *Emit* ist immer die geschützte Variable *Q_EMIT*, allerdings immer zum entsprechenden Interface gecastet. *Q_EMIT* ist eigentlich ein transparenter Proxy, der die Methodenaufrufe

Listing 3

Layouts nutzen.

```
using System;
using Qyoto;
class MainWidget : QWidget {
    public MainWidget() : base() {
        QVBoxLayout vbox = new QVBoxLayout(this);
        QPushButton button = new QPushButton("Quit");
        QLCDNumber number = new QLCDNumber();
        QSlider slider = new QSlider(Orientation.Horizontal);
        vbox.AddWidget(button);
        vbox.AddWidget(number);
        vbox.AddWidget(slider);
        slider.Minimum = 0;
        slider.Maximum = 100;
        Connect(button, SIGNAL("clicked()"), this, SLOT("ButtonClicked()"));
        Connect(this, SIGNAL("TestSignal()"), qApp, SLOT("quit()"));
        Connect(slider, SIGNAL("valueChanged(int)"), number, SLOT("display(int)"));
    }
    [Q_SLOT]
    public void ButtonClicked() {
        Console.WriteLine("Button clicked!");
        Emit.TestSignal();
    }
    protected new IMainWidgetSignals Emit {
        get {
            return (IMainWidgetSignals) Q_EMIT;
        }
    }
    public static void Main(string[] args) {
        new QApplication(args);
        MainWidget w = new MainWidget();
        w.Show();
        QApplication.Exec();
    }
}
interface IMainWidgetSignals : IQWidgetSignals {
    [Q_SIGNAL]
    void TestSignal();
}
```

abfängt und auswertet. Dies aber nur am Rande, eine komplette Erklärung der Interna würde hier den Rahmen sprengen.

Wieder zurück im Konstruktor können wir feststellen, dass nun unser benutzerdefiniertes Signal mit dem Slot `quit()` der `QApplication`-Instanz verbunden wird. Also wird unser Programm beendet, wenn das Signal ausgelöst wird.

Zu guter Letzt wird das Signal `valueChanged(int)` eines Sliders (`QSlider`) mit dem Slot `display(int)` einer LCD-Nummernanzeige (`QLCDNumber`) verbunden. Bei dieser Verbindung werden dem Slot auch direkt Parameter übergeben. Dies ist einfach zu realisieren, indem man einfach die Typen der Parameter in die Klammern schreibt. Das ist so, als ob man die Signatur einer Methode angibt, nur Rückgabety und Parameternamen weglässt. Ein Signal mit mehreren Parametern könnte also wie folgt aussehen:

```
meinSignal(int, int, int)
```

Der Effekt der obigen Verbindung ist einfach zu verstehen: Wird der Wert des Sliders verändert, wird der neue Wert in der LCD-Nummernanzeige angezeigt.

Kommen wir nun zu den Layouts, die schon angesprochen wurden. Am Anfang des Konstruktors finden wir diese Zeile:

```
QVBoxLayout vbox = new QVBoxLayout(this);
```

Hiermit erstellen wir ein sogenanntes Vertical Box Layout, also ein Layout, das Steuerelemente vertikal anordnet. Als Elternfenster geben wir die aktuelle Klasse

(`this`) an. Alle Steuerelemente, die später mittels `AddWidget()` hinzugefügt werden, werden vom Layout vertikal angeordnet und bekommen automatisch die Elternklasse des Layouts gesetzt.

Mittels `AddLayout()` kann man darüber hinaus auch andere Layouts von einem Layout verwalten lassen, sodass eine gut strukturierte Oberfläche erstellt werden kann.

Neben `QVBoxLayout` für eine vertikale Anordnung existieren auch noch `QHBoxLayout` für horizontale Anordnung und `QGridLayout` für eine Anordnung in einem Gitter.

Ferner werden von Layouts verwaltete Steuerelemente automatisch an Größenänderungen des Elternfensters angepasst, sodass immer der gesamte Platz ausgefüllt wird.

Einige Steuerelemente sollten nun aber nur in eine Richtung vergrößert werden, zum Beispiel `QPushButtons` oder `QLineEdit`s (einzeilige Eingabefelder), die nur in der Breite verändert werden sollten, nicht aber in der Höhe. Für solche Fälle gibt es die Eigenschaft `QWidget.SizePolicy`, die regelt, welche Größe das Steuerelement bevorzugt. `QPushButtons` geben hier beispielsweise an, dass sie nur horizontal in ihrer Größe verändert werden dürfen. Die Standardeinstellung der meisten Steuerelemente ist jedoch „Bevorzugt“. Das bedeutet: Das Widget darf in seiner Größe frei verändert werden, es bevorzugt jedoch die Größe, die in der Eigenschaft `QWidget.SizeHint` angegeben ist.

Dies soll nur eine Information am Rande sein. Auf diese Eigenschaft kommen wir später noch einmal zu sprechen.

Mehr Informationen zu Layouts und generell allen Themen rund um Qt finden Sie in der Hilfe von Qt, die mit dem Programm „Qt Assistant“ aufgerufen werden kann. Geben Sie dafür einfach `assistant` auf der Kommandozeile ein. Die Hilfe beschreibt das originale C++-API, das sich allerdings nicht viel vom C#-API unterscheidet. Unterschiede sind nur, dass in Qyoto alle Funktionen am Anfang groß geschrieben werden und einige Zugriffsfunktionen für Eigenschaften direkt durch `.NET`-Eigenschaften ersetzt wurden. Es gibt beispielsweise statt `setWindowTitle()` und `windowTitle()` die `.NET` Eigenschaft `WindowTitle`.

Systemereignisse abfangen

Oft ist es nützlich, Systemereignisse wie beispielsweise das Neuzeichnen eines Fensters oder den Klick eines Mausbuttons irgendwo im Fenster abfangen zu können, um diese auszuwerten.

Um ein solches Ereignis abzufangen, muss nur der sogenannte Eventhandler neu implementiert werden. Dieser Eventhandler ist einfach eine geschützte virtuelle Methode, die beim Auftreten des Systemereignisses von Qt aufgerufen wird. Listing 4 zeigt ein kurzes Beispiel, wie sich das Paint-Event abfangen lässt. Es zeichnet einen Kreis in die Mitte eines Widgets.

Listing 4

Das Paint-Ereignis abfangen.

```
using System;
using Qyoto;
class MainWindow : QWidget {
    public MainWindow() : base() {
        WindowTitle = "Paint Event Beispiel";
    }
    protected override void PaintEvent(QPaintEvent e) {
        QPainter painter = new QPainter(this);
        painter.SetRenderHint(QPainter.RenderHint.Antialiasing);
        int width = 100;
        int height = 100;
        painter.DrawEllipse(Width() / 2 - width / 2, Height() / 2 - height / 2, width, height);
        painter.End();
    }
    public static void Main(string[] args) {
        new QApplication(args);
        MainWindow w = new MainWindow();
        w.Show();
        QApplication.Exec();
    }
}
```

Wir überschreiben hier einfach die Methode *PaintEvent()*, um das Ereignis abzufangen. In der Methode selbst wird ein Instanz eines *QPainter* erstellt. Das ist die Klasse, mit der alle Zeichenoperationen in Qt ausgeführt werden. Als erstes Argument nimmt der Konstruktor von *QPainter* ein *QPaintDevice*, also ein „Gerät“, auf dem gezeichnet werden kann. Jede Subklasse von *QWidget* als auch *QWidget* selbst ist ein *QPaintDevice*, also kann darauf gezeichnet werden.

Im nächsten Schritt schalten wir *Antialiasing* für den *QPainter* ein, damit unser Kreis auch wirklich schön rund wird und wir keine Ecken sehen. Danach werden Höhe und Breite festgelegt, und eine Ellipse wird in der Mitte des Widgets gezeichnet.

Im letzten Schritt wird mit *End()* der Zeichenvorgang beendet. Alle Ressourcen, die während des Zeichnens genutzt wurden, werden freigegeben.

Arbeiten mit dem Qt Designer

Das Gestalten von grafischen Benutzeroberflächen mittels selbst geschriebenem Code ist bei kleinen Programmen oft noch einfach. Doch je komplexer das Programm wird, desto aufwändiger werden auch meist die Oberflächen. Damit sich die GUI-Gestaltung trotzdem noch einfach und schnell vollzieht und man sich auf den wichtigeren Teil des Programms, die Funktionalität, konzentrieren kann,

liefert Qt auch gleich ein entsprechendes Designertool mit. Gestalten wir nun auch gleich ein Fenster mit dem Qt Designer. Er sollte im Menü ihrer Desktopumgebung zu finden sein, alternativ kann man ihn auch mit *designer* direkt von der Konsole aus aufrufen.

Nach dem Start wählen wir vom Menü *File* den Unterpunkt *New Form* aus und erstellen ein normales *Widget*. Wir bekommen ein Fenster ohne Inhalt angezeigt.

Zur Orientierung: Auf der linken Seite befindet sich die *Widget Box*, in der alle verfügbaren Widgets aufgelistet sind. Sie können per *Drag-and-drop* einfach auf die Form gezogen werden. Auf der rechten Seite haben wir ganz oben den sogenannten *Object Inspector*, der Zugriff auf alle Widgets über eine Liste gibt. Darunter ist der *Property Editor* positioniert, mit dem man die verschiedenen Eigenschaften der Widgets bearbeiten kann, wie Name, Beschriftung et cetera. Noch eine Etage tiefer finden wir den *Signal/Slot Editor*, in dem alle *Connections* angezeigt werden und mit dem man auch ohne den *Connection-Modus* *Signals* und *Slots* verbinden kann (dazu später mehr). Darunter befindet sich wiederum der *Ressourcen Editor*, der aber für diesen Artikel ohne Belang ist. Am Schluss steht der *Action Editor*, mit dem man sogenannte *Actions* bearbeiten kann. Dies sind Objekte, die sowohl in *Toolbars* als auch in *Menüs* eingefügt werden können, sodass man nicht jeden Eintrag doppelt erstellen muss. Auch sie können ein-

fach durch *Signals* und *Slots* verbunden werden.

Ziehen wir also zu Beginn erst einmal einen *Push-Button* auf das *Widget* und positionieren ihn unten rechts in der Ecke. Damit das Ganze etwas mehr nach einem sinnvollen Dialog aussieht, ziehen wir auch noch zwei *Frames* auf das *Widget* und drei *Checkboxes* beziehungsweise *Radio Buttons* in jeden *Frame*.

Nun sieht das Ganze noch etwas ungeordnet aus, deshalb erstellen wir im nächsten Schritt *Layouts*, die unsere Steuerelemente verwalten. Zuerst geben wir den zwei *Frames* ein *Layout*. Dazu klicken wir mit der rechten Maustaste an eine freie Stelle in jedem *Frame*, wählen aus dem *Popupmenü* den Punkt *Layout* und im *Untermenü* *Layout Vertically*.

Jetzt sieht es in den *Frames* schon etwas geordneter aus, allerdings ist das *Widget* an sich immer noch nicht das, was wir wollen. Geben wir ihm doch auch ein *Layout*. Dazu auf eine freie Stelle im *Widget* klicken und wieder den Punkt *Layout Vertically* auswählen. Doch was ist das? Der *Button* unten ist komplett in die Länge gezogen, die *Frames* sind vertikal angeordnet ... Ein schöner Dialog sieht anders aus.

Machen wir das noch einmal rückgängig, mittels *[Strg]+[Z]* oder über die *Menüleiste*, *Edit/Undo*. Damit wir unser gewünschtes Ergebnis bekommen, müssen wir die zwei *Frames* zuerst einmal selbst in ein *Layout* zwängen, und zwar in ein horizontales. Markieren wir also beide *Frames* (*Shift* gedrückt halten und freie Flächen anklicken) und wählen im *Popupmenü* von einem der beiden *Frames* *Layout Horizontally*. Das sieht schon besser aus: zwei *Frames* nebeneinander in einem roten Kasten. Dieser rote Kasten ist später nicht mehr zu sehen, er zeigt nur, dass dort ein *Layout* am Werke ist.

Damit nun der *Button* auch seine Größe behält, brauchen wir einen sogenannten *Spacer*. Dazu ziehen wir aus der *Widget Box* einfach einen solchen *Horizontal Spacer* neben unseren *Button*. Die beiden werden nun markiert und auch in ein horizontales *Layout* gepackt. Klicken wir nun wieder auf eine freie Stelle auf dem *Widget* und wählen *Layout Vertically*. Das sieht jetzt schon fast gut aus, nur scheint der *Button* im unteren *Layout* zentriert zu sein und sitzt nicht am unteren Rand, wie er eigentlich sollte. Hier kommt nun die Eigenschaft *SizePolicy*, die oben schon kurz angeschnitten wurde, zum Einsatz. Wählen wir die beiden

Frames also aus, schauen rechts im Property Editor nach der Eigenschaft *SizePolicy* und klappen diese auf. Den Unterpunkt *vSizeType* ändern wir nun von *Preferred* auf *Expanding* und schwups füllen die Frames einen Großteil des Widgets aus. Der Button ist nun wie gewünscht am unteren Rand des Widgets.

Hiermit haben wir den Frames praktisch nur gesagt: Anstatt immer die bevorzugte Größe zu haben, nutzt ihr ab sofort den gesamten euch zur Verfügung stehenden Platz aus. Da der Freiraum im unteren Layout nur ungenutzt war, wird er also nun von den Frames genutzt.

Nun wäre es natürlich auch noch schön, wenn das Widget geschlossen werden würde, sobald wir auf den Button klicken. Natürlich wäre es am bequemsten, würden wir das mit Signals und Slots lösen. Damit wir die Connections nicht manuell in unseren Code schreiben müssen, hat der Designer auch für dieses Problem eine Lösung: ein integrierter Signal/Slot Editor. Gehen wir nun also in den Connection-Modus über den Menüpunkt *Edit/Edit Signals/Slots*. Was nun sofort auffällt, ist, dass die Widget Box deaktiviert ist und dass alle Steuerelemente rot hinterlegt werden, wenn wir mit dem Mauszeiger über sie fahren. Dies zeigt an, dass wir uns im Signal/Slot Editor Modus befinden.

Halten wir nun die linke Maustaste über unserem Button gedrückt und ziehen eine Linie bis an den Rand des Widgets, damit wir sicher sind, dass wir das Widget und nicht eventuell ein Layout treffen. Es erscheint eine rote Linie von dem Button ausgehend zum Mauszeiger. Lassen wir jetzt die Maustaste los, erscheint ein Fenster mit den Signalen des Buttons auf der linken Seite und den Slots des Widgets auf der rechten Seite. Gegebenenfalls muss die Checkbox *Show all signals and slots* aktiviert werden, um die Slots des Widgets anzuzeigen. Wählen wir nun als Signal *clicked()* und als Slot *close()* aus und bestätigen den Dialog. Das Ergebnis ist eine konstante rote Linie vom Button zur Form, die die bestehende Verbindung anzeigt. Außerdem wird die neue Verbindung rechts im Signal/Slot-Toolfenster angezeigt.

Damit haben wir unser erstes Widget mittels des Qt Designers erstellt. Wechseln wir mit *Edit/Edit Widgets* zurück in den GUI-Editor und testen es mit *Form/Preview* einmal. Wir können den Dialog beliebig vergrößern/verkleinern, und wenn wir auf den Button klicken, schließt er sich sogar. Damit haben wir unser Ziel erst einmal

erreicht. Speichern wir das Widget als *dialog.ui* in unserem Sandkastenordner. Doch wie bekommen wir den Dialog nun in unser Programm hinein?

Generieren von Code mittels UI-Dateien

Auch das ist relativ einfach. Wechseln wir in der Konsole wieder in unser Sandkastenverzeichnis und erstellen mit dem Programm *uics* (ein Fork von *uic*, dem User Interface Compiler, für C#) die Quellcodedatei für unseren Dialog:

```
../qyoto/qyoto/tools/uics/uics dialog.ui -o ui_dialog.cs
```

Die Ausgabe des Programms wird durch den Parameter *-o* in die Datei *ui_dialog.cs* geschrieben.

Dies ist aber nur der Code für die Oberfläche an sich, wir benötigen nun noch den eigentlichen Programmcode, wie in Listing 5.

Wie wir sehen, ist der eigentliche Code, der benötigt wird, um das GUI auf ein Widget zu übertragen, genau drei Zeilen lang. Zuerst wird mit

```
Ui.Form ui;
```

eine Variable deklariert, mit der wir auf das GUI zugreifen können. Im Konstruktor wird dann eine Instanz der UI-Klasse erzeugt und mit der Methode *SetupUi()* das GUI auf das aktuelle Widget übertragen.

uics erzeugt einen eigenen Namespace, *Ui*, in dem alle *Ui*-Klassen verweilen. Der Name der Klasse ist der gleiche wie der Objektname, der der Form im Designer zugewiesen wurde. Über die UI-Klasse sind auch alle Steuerelemente der Form zu erreichen, wie das etwa durch die Zeile

```
Console.WriteLine(ui.pushButton.Text);
```

verdeutlicht wird. Der Name der Steuerelemente ist wieder der gleiche wie der Objektname im Designer. Aus diesem Grund ist es oft eine gute Idee, den Objekten im Designer gleich aussagekräftige Namen zu geben. Ansonsten kann es leicht zu großer Verwirrung beim Programmieren kommen, da die Namen nicht eindeutig sind. Um unser Beispiel nun zu kompilieren, müssen wir nur die generierte Datei als zusätzlichen Parameter anhängen:

```
gmsc -r:qt-dotnet.dll dialog.cs ui_dialog.cs
```

Beim Ausführen kann es unter Umständen sein, dass die Widgets durch die

Listing 5

UI-Dateien nutzen.

```
/*
 * Dieser Code wird nur funktionieren,
 * wenn im Designer die Objektnamen der
 * Form und des PushButtons nicht geändert wurden.
 */
using System;
using Qyoto;

public class MainWindow : QWidget {
    Ui.Form ui;
    public MainWindow() : base() {
        ui = new Ui.Form();
        ui.SetupUi(this);
        Console.WriteLine(ui.pushButton.Text);
    }
    public static void Main(string[] args) {
        new QApplication (args);
        MainWindow w = new MainWindow();
        w.Show();
        QApplication.Exec();
    }
}
```

Layouts nicht vollkommen korrekt ausgelegt werden. Dies ist ein bekannter Bug und es wird bereits daran gearbeitet.

Fazit

Qyoto ist ein vielversprechendes Projekt, mit dem sowohl das Qt- als auch das KDE-API unter Mono/.NET verfügbar ist beziehungsweise sein wird. Damit wird die Funktionalität von Qt nicht nur für eine einzige weitere Programmiersprache bereitgestellt. Ein ganzer Pool an Programmiersprachen – neben C# und VB.NET auch IronPython, Boo, Ruby.NET oder Nemerle – kann von dem Funktionsumfang profitieren.

Weiterhin ist mit Qt4 eine Bibliothek vorhanden, deren Umfang wesentlich größer ist als der des .NET Frameworks. In diesem Artikel konnte leider nur auf die GUI-Programmierung eingegangen werden. Allerdings umfasst Qt noch weitere Features wie D-BUS, SQL, SVG, OpenGL, Netzwerk und vieles mehr. Der Endbenutzer muss nicht mehr tausende von Zusatzbibliotheken installieren, alles ist unter einem Dach vereint. Es dürfte sich lohnen, dieses Projekt weiter im Auge zu behalten. |||||

[1] www.trolltech.com

[2] www.cmake.org

[3] rubyforge.org/projects/korundum/

[4] www.qyoto.org