

# QT 4.x unter Windows und Visual Studio 2005

Volker Wiendl

8. Mai 2006

# Inhaltsverzeichnis

<b>1</b>	<b>Installation der OpenSource Version von QT 4.x unter Windows</b>	<b>3</b>
1.1	Installation des inoffiziellen Visual Studio Patches . . . . .	3
1.2	Übersetzen des QT Source Codes . . . . .	4
1.3	Aufräumen . . . . .	5
1.4	Umgebungsvariablen . . . . .	6
1.5	Wichtige Verknüpfungen . . . . .	7
<b>2</b>	<b>Erstellen eines neuen Anwendungsprojekts unter Visual Studio 2005</b>	<b>7</b>
<b>3</b>	<b>Erzeugen einer Beispielanwendung</b>	<b>8</b>
3.1	Hinzufügen einer CPP Datei . . . . .	8
3.2	Einsprungspunkt der Anwendung . . . . .	9
3.3	Benutzung des QT Designers . . . . .	10
3.4	Integration der UI Datei in Visual Studio 2005 . . . . .	12
3.5	Erzeugen der Hauptklasse . . . . .	14
3.6	Der Meta-Object-Compiler von QT . . . . .	19
3.7	Erzeugen eigener Slot Funktionen . . . . .	20
<b>A</b>	<b>Copyright Notiz</b>	<b>25</b>
A.1	Haftungsausschluss: . . . . .	26

# 1 Installation der OpenSource Version von QT 4.x unter Windows

Um die aktuelle Version von Trolltech's QT unter Windows zu benutzen, lädt man sich als erstes die aktuellste Version unter der OpenSource Seite von Trolltech<sup>1</sup> herunter. Möchte man QT mit Visual Studio verwenden, so wählt man das ZIP Archiv ohne den MinGW Installer. Andernfalls gibt es einen Installer der einem die meiste Arbeit abnimmt, anschließend aber den MinGW GCC Compiler voraussetzt und QT nicht zusammen mit Visual Studio laufen lässt. Im weiteren wird davon ausgegangen, dass Visual Studio verwendet wird und min. 2GB freier Festplattenspeicher vorhanden sind.

Nach dem Herunterladen des ZIP Archivs entpackt man die darin enthaltenen Dateien in das Verzeichnis in dem man später QT installieren möchte. Es ist dabei von Vorteil ein Verzeichnisnamen zu wählen, der keine Leerzeichen enthält. Prinzipiell funktioniert die Installation zwar vermutlich auch mit einem Verzeichnisnamen mit Leerzeichen, jedoch könnte es dabei zu Problemen bei der Integration unter Visual Studio kommen. Im folgenden wird davon ausgegangen, dass alle Dateien in das Verzeichnis `C:\Programme\Qt4` entpackt wurden (Achtung: Standardmäßig ist im ZIP Archiv von Trolltech bereits ein Ordnername enthalten, der dem Namen des Archivs entspricht. Dieser wird in diesem Fall nicht verwendet, sondern es werden nur die Dateien und Unterverzeichnisse innerhalb dieses Ordners in das Verzeichnis `C:\Programme\Qt4` entpackt).

## 1.1 Installation des inoffiziellen Visual Studio Patches

Um die OpenSource Edition zu Visual Studio kompatibel zu machen ist es nötig sich von der Qtwin Projektseite<sup>2</sup> den „Unofficial patches for Qt4“ in der Sektion *Files* herunter zu laden. Beim Erstellen dieses Tutorials war der aktuellste Patch der für die QT Version 4.1.2 (`acs4qt412p1.zip`). Nach dem Herunterladen des ZIP Archivs, entpackt man sämtliche Dateien ebenfalls in das Verzeichnis in dem bereits die Dateien von QT liegen. Anschließend installiert ein Doppelklick auf die Datei `installpatch.bat` die nötigen Änderungen am Sourcecode von QT.

### Optional:

Eine weitere Änderung am Source Code die nicht im inoffiziellen Patch enthalten ist, würde einen in späteren QT Projekten von einigen Duzend Warnmeldungen befreien.

Man fügt dazu selbst mit Hilfe eines Editors in die Datei

`C:\Programme\QT4\src\corelib\thread\qatomic.h` folgende Zeilen ein:

---

<sup>1</sup><http://www.trolltech.com/download/qt/windows.html>

<sup>2</sup><http://sourceforge.net/projects/qtwin>

```
#pragma warning(push)
#pragma warning(disable:4311)
#pragma warning(disable:4312)
```

Diese Anweisungen schalten vorübergehend die Warnungen 4311 und 4312 ab. Damit diese in anderen Dateien wieder gemeldet werden, muss am Ende der Datei *qatomic.h* noch ein weiterer Eintrag

```
#pragma warning(pop)
```

hinzugefügt werden. Der beste Platz für die ersten drei Zeilen ist in Zeile 30 (Stand: Qt 4.1.2) nach der Zeile:

```
#if !defined(Q_CC_GNU) && !defined(Q_CC_BOR)
```

während die pop Anweisung am Besten vor der Zeile *#endif // \_MSC\_VER ...* in Zeile 150 (Stand: Qt 4.1.2) platziert wird.

## 1.2 Übersetzen des QT Source Codes

Zum Übersetzen des Source Codes ist es nötig eine Visual Studio Konsole zu öffnen. Diese findet sich bei einer normalen Visual Studio Installation unter

Microsoft Visual Studio 2005

-> Visual Studio Tools

-> Visual Studio 2005 Command Prompt

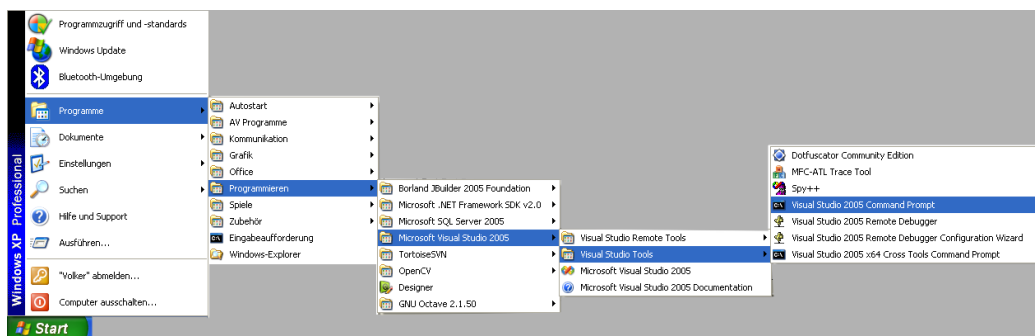


Abbildung 1: Starten der Visual Studio Konsole

Auf der sich öffnenden Kommandozeile wechselt man nun in das Verzeichnis der QT Installation:

```
C:
cd C:\Programme\Qt4
```

Anschließend ruft man *qconfigure.bat* mit dem entsprechenden Parameter für die eigene Visual Studio Version auf.

```
qconfigure msvc2005
```

Möchte man standardmäßig die PlugIns für GIF Bilder, ODBC Sql Treiber und SQLite mit übersetzen lassen, können zusätzlich die entsprechenden Parameter *-qt-gif*, *-plugin-sql-odbc* und *-plugin-sql-sqlite* angegeben werden.

```
qconfigure msvc2005 -qt-gif -plugin-sql-odbc -plugin-sql-sqlite
```

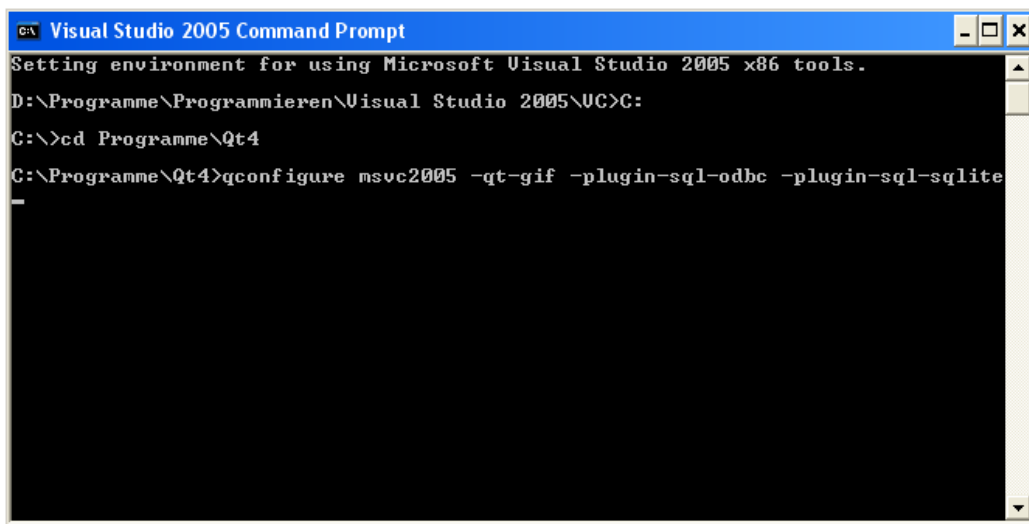


Abbildung 2: Wechseln des Verzeichnisses und starten von *qconfigure.bat*

Die folgende Frage nach Akzeptanz der Lizenz bestätigt man mit der Eingabe von *y*. Anschließend wird der QT Source Code übersetzt. Dies kann einige Zeit dauern und nimmt außerdem einiges an Festplattenspeicher in Anspruch. Es sollten daher min. 2GB freier Speicher zur Verfügung stehen bevor man den Code übersetzt.

### 1.3 Aufräumen

Nachdem der komplette Source Code ohne Fehler übersetzt wurde und wieder die Konsole zur Eingabe bereit ist, können die temporär angelegten Übersetzungsdateien durch löschen der tmp Verzeichnisse entfernt werden, was einiges an Plattenplatz wieder frei gibt. Ein Aufruf von *nmake clean* löscht

zusätzlich die lib Dateien, was dazu führt, das die Software-Entwicklung in Visual C++ nicht mehr möglich ist. Eine Möglichkeit dies zu umgehen wäre, den Inhalt des lib Verzeichnisses vor dem Aufruf von `nmake clean` zu sichern. Sollten beim Übersetzen irgendwo Fehler auftreten, kann der Übersetzungsvorgang mit dem Aufruf von `nmake` erneut gestartet werden.

## 1.4 Umgebungsvariablen

Für eine problemlose Ausführung von QT Anwendungen sollte das `bin` Verzeichnis der QT Installation in die `Path` Umgebungsvariable von Windows aufgenommen werden.

Durch das Drücken der Tastenkombination „Windows-Taste“+Pause öffnet sich das Systemeigenschaften Fenster. Dort findet sich in der Karteikarte *Erweitert* die Schaltfläche *Umgebungsvariablen*. Im Bereich Systemvariablen sollte sich nun bereits ein Eintrag mit dem Namen `Path` befinden, der durch den Eintrag `C:\Programme\Qt4\bin` erweitert wird. Für die spätere Integra-

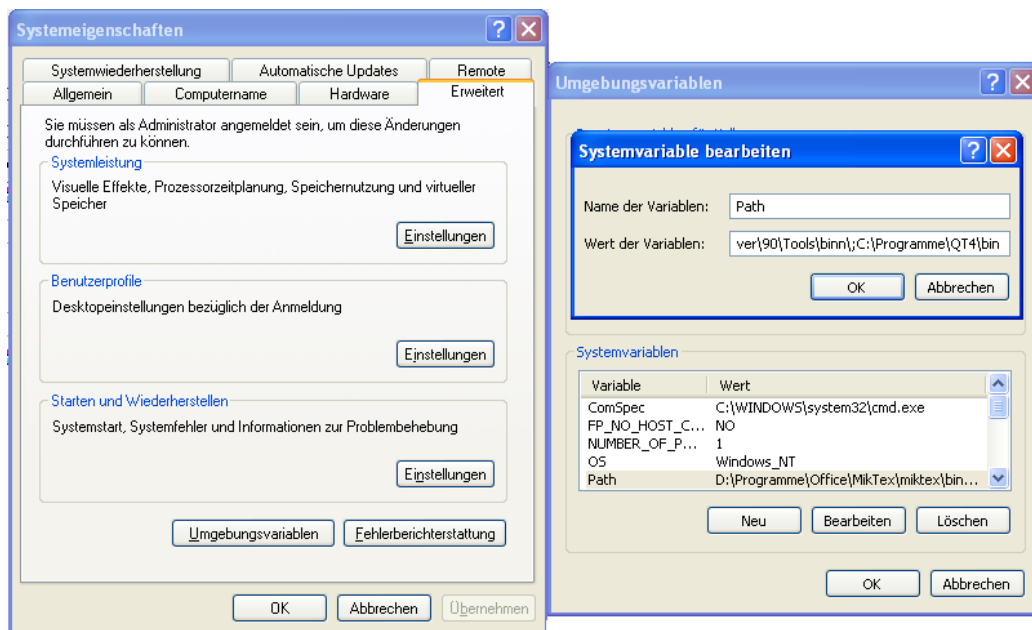


Abbildung 3: Anpassen der `Path` Umgebungsvariable

tion in Visual Studio ist es jetzt noch sinnvoll eine weitere System Umgebungsvariable mit dem Namen `QTDIR neu` zu definieren. Diese bekommt als Wert den Pfad zum QT Installationsverzeichnis. In dem hier gezeigten Beispiel also `C:\Programme\Qt4`

### Optional:

Möchte man auch mit *qmake* bzw. *nmake* arbeiten (siehe späteres Tutorial zum Umgang mit Makefiles), so sollte man noch die entsprechenden Variablen *INCLUDE* und *LIB* die Visual Studio standardmäßig bereits angelegt hat um die entsprechenden Pfade von QT erweitern. Dazu trägt man bei der Umgebungsvariable *INCLUDE* folgendes ein:

```
C:\Programme\Qt4\include
```

Die Umgebungsvariable *LIB* wird ebenfalls um folgendes ergänzt:

```
C:\Programme\Qt4\lib
```

## 1.5 Wichtige Verknüpfungen

Für die Arbeit von QT sind zwei Elemente für die tägliche Arbeit wichtig. Zum einen der Designer (zu finden nach dem Übersetzen unter *bin\designer.exe* des QT Installationsverzeichnisses) zum anderen die Einstiegsseite zur Dokumentation von QT (zu finden unter *doc\html\index.html*). Es bietet sich an sich für diese beiden Dinge entsprechende Verknüpfungen im Startmenü bzw. Browser anzulegen.

## 2 Erstellen eines neuen Anwendungsprojekts unter Visual Studio 2005

Der erste Schritt für ein gut strukturiertes QT Projekt in Visual Studio ist die Erzeugung einer neuen leeren *Solution*.

Über den Menüpunkt *File -> New -> Project...* wählt man *Other Project Types* dort *Visual Studio Solutions* und schließlich *Blank Solution*. Im folgenden Dialog gibt man den Namen und Ort der neuen Projektmappe an.

Es öffnet sich nun der Solution Explorer. Dort klickt man mit der rechten Maustaste auf die Solution und wählt dort *Add -> New Project...* Im sich öffnenden Dialog wählt man ein neues *Visual C++ - Win32 Projekt*. Man hat nun die Wahl eine Konsolen- oder eine Windows Applikation zu erstellen. In diesem Beispiel wird eine Applikation mit Konsole erstellt. Als Verzeichnispfad wird zusätzlich zum voreingestellten Pfad der Solution noch *\src* angehängt (siehe Abb. 6). Dies bewirkt, dass das neue Projekt im Unterverzeichnis *src* erstellt wird.

Anschließend öffnet sich ein Dialog zur Einstellung der Projekteigenschaften (siehe Abb. 7). Unter *Application Settings* wählt man nun aus, dass man ein leeres Projekt erstellen möchte (*Empty Project*) und drückt auf den *Finish* Button. Nun ist ein neues Projekt der Solution Mappe hinzugefügt worden

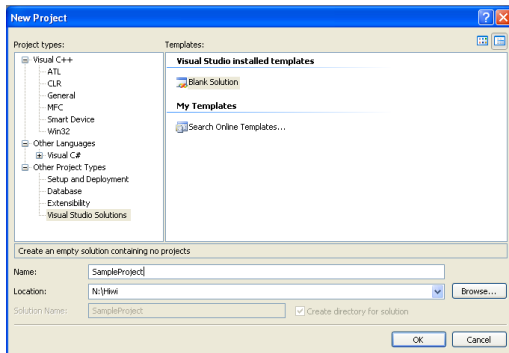


Abbildung 4: Erstellen einer leeren Solution

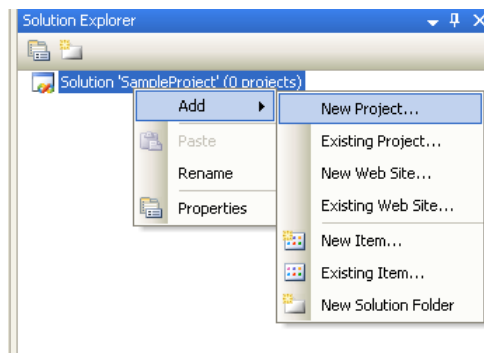


Abbildung 5: Hinzufügen eines neuen Projekts innerhalb der Solution

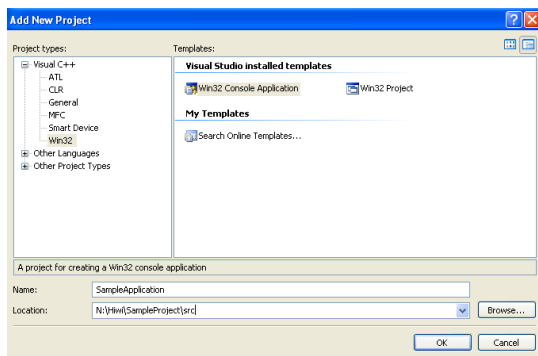


Abbildung 6: Erstellen einer neuen Applikation mit Konsole

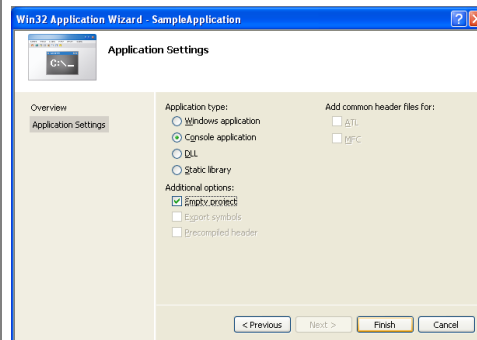


Abbildung 7: Einstellungen der Projekteigenschaften

und es existieren die drei Unterordner *Header Files*, *Resource Files* und *Source Files*.

### 3 Erzeugen einer Beispielanwendung

In den folgenden Abschnitten wird anhand einer kleinen Beispielapplikation der Umgang und die Integration des QT Designers in Visual Studio gezeigt.

#### 3.1 Hinzufügen einer CPP Datei

Für eine neue Anwendung fügt man als erstes eine *.cpp* Datei ein, indem man mit der rechten Maustaste auf den Unterordner *Source Files* klickt und aus dem aufklappenden Menü das Untermenü *Add* und anschließend *New Item...*

auswählt (siehe Abb. 8).

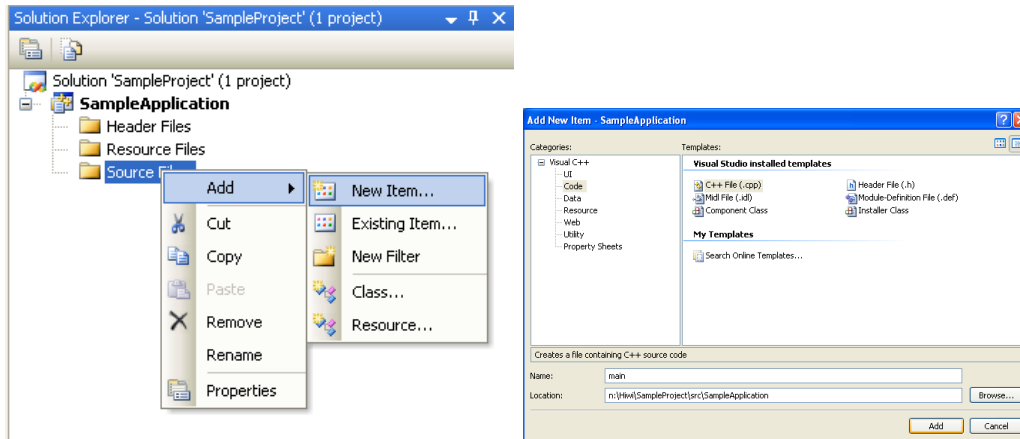


Abbildung 8: Hinzufügen eines neuen Projektelements

Abbildung 9: Hinzufügen einer neuen CPP Datei

Im sich daraufhin öffnenden Dialog wählt man *C++ File (.cpp)* und gibt unten den entsprechenden Namen (in diesem Beispiel `main.cpp`) ein. Wie man sieht ist durch das Anlegen des Projekts im Unterverzeichnis `src` bereits gewährleistet, dass standardmäßig die Quellcode Dateien im Verzeichnis `src\Projektname` abgelegt werden.

### 3.2 Einsprungspunkt der Anwendung

In der neuen Datei wird nun der Einsprungspunkt für die zu entwickelnde Anwendung implementiert. Für Konsolenanwendungen ist dieser definiert durch:

```
int main(int argc, char* argv[])
{
    ...
}
```

bei Windows Anwendungen wäre statt dessen folgende Funktion anzugeben

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR szCmdLine, int iCmdShow)
{
    ...
}
```

### 3.3 Benutzung des QT Designers

Um nun eine QT Applikation zu integrieren, öffnet man zunächst den QT Designer. Im Menü *Edit* -> *User Interface Mode* lässt sich auswählen, ob man lieber im *Docked Window* Modus arbeitet oder mit mehreren Fenstern. Als erstes erstellt man nun eine neue *Form* durch Auswählen des Menüs *File* -> *New Form*.... Der darauf folgende Dialog erscheint nach der Neuinstallation von QT bereits standardmäßig nach dem Starten des Designers. Nun wählt man *Main Window* (siehe Abb. 10) und bestätigt mit *Create*.

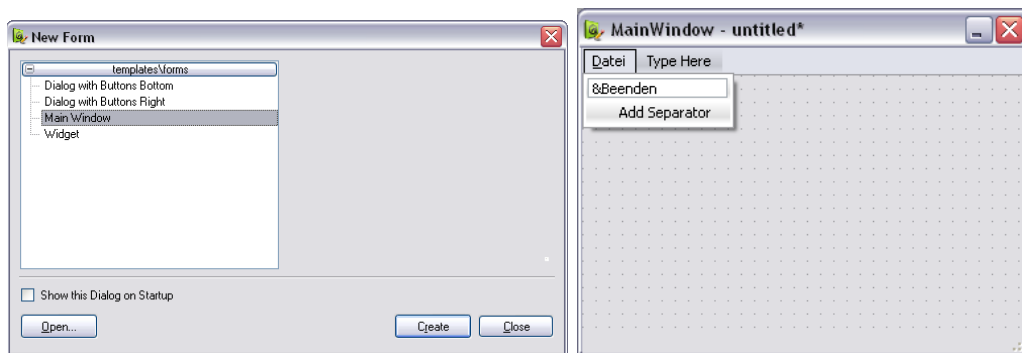


Abbildung 10: Erstellen eines neuen Main Widgets

Abbildung 11: Hinzufügen eines Menüs

Es erscheint ein neues Anwendungsfenster, in welches neue Elemente hinzugefügt werden können. Als erstes wird in diesem Beispiel ein Datei Menü mit einem Eintrag zum Beenden der Anwendung eingefügt. Dazu klickt man doppelt auf den bestehenden Eintrag *Type Here* und gibt anschließend *&Datei* ein. Das „&“ Zeichen bewirkt, dass der darauf folgende Buchstabe zusammen mit der *ALT*-Taste als Shortcut verwendet werden kann, in diesem Fall also *ALT+D*. Anschließend fügt man auf die selbe Art innerhalb des Menüs *Datei*, den Eintrag *&Beenden* hinzu (siehe Abb. 11).

Auf der rechten Seite des Designers gibt es Fenster für weitere Einstellungen.

- Der *Object Inspector* listet alle in der *Form* definierten Objekte auf.
- Der *Property Editor* lässt den Benutzer die Eigenschaften des aktuell gewählten Objekts verändern.
- Der *Signal/Slot Editor* dient zur Verknüpfung von Objekt Signalen mit entsprechenden Slots die in einem der *Form* Objekte bereits definiert sind.

- Der *Resource Editor* ist für das Hinzufügen von externen Ressourcen wie Bildern oder Icons zuständig
- Der *Action Editor* listet alle definierten *Actions* innerhalb der **Form**

Wie man sieht, werden die Namen der Objekte automatisch aus den Namen der Menüeinträge erstellt. Arbeitet man mit mehreren Leuten an einem Projekt sollte man stets auf eine gleich bleibende Namenskonvention der Variablen achten. Es gibt dafür viele Möglichkeiten.

Nach der ungarischen Notation sollten z.B. Member Variablen immer mit einem *m\_* anfangen (siehe auch Ungarische Notation<sup>3</sup>). Da allgemein Programmiersprachen meist mit englischen Ausdrücken arbeiten, sollte man auch seinen Variablen - soweit man dem Englischen einigermaßen mächtig ist - englische Namen geben. In diesem Beispiel werden deshalb nun die Menü Objekte durch Verwendung des Property Editors in *m\_menu\_file* und *m\_action\_quit* umbenannt, sowie die bereits eingetragene QMenuBar in *m\_menubar*.

Bevor man nun die Funktion des *Beenden* Menüeintrags mit dem Schließen der Anwendung verknüpft, sei kurz die Funktionsweise von QT Signalen und Slots erklärt.

QT Anwendungen besitzen intern eine Event Loop. Wird nun eine Aktion innerhalb der Anwendung ausgelöst z.B. durch das Klicken eines Menüeintrags, so wird das entsprechende Signal ausgelöst. Im Fall eines Menüeintrags handelt es sich um ein **QAction** Object, welches ein Signal *triggered()* auslöst, wenn es aktiviert wird. Nach dem Auslösen wird nun nachgesehen ob eventuell bestimmte Funktionen (so genannte Slots) mit diesem Signal verbunden sind. Ist dies der Fall wird die entsprechende Funktion aufgerufen. Dabei können mit einem Signal beliebig viele Slots verbunden sein. Näheres zu dem Signal/Slot Konzept von QT findet sich auch in der Dokumentation von QT (siehe Abschnitt 1.5).

Um nun die *Beenden* „Action“ mit dem Beenden der Anwendung zu verknüpfen, klickt man im Signal/Slot Editor auf das *Plus*-Icon. Anschließend wählt man als Sender das QAction Objekt *m\_action\_quit*, als Signal *triggered()*, als Receiver *MainWindow* und als Slot *close()* (siehe Abb. 12).

Abschließend kann man - wenn man möchte - noch den WindowTitle des MainWindows ändern bevor man schließlich die **Form**-Datei abspeichert. Als Pfad wählt man das Verzeichnis in dem man auch die übrigen Quellcode Dateien des Projekts abgespeichert hat, oder man erstellt ein eigenes Unterverzeichnis *Forms* (siehe Abb. 13). Für den Namen der Datei sollte man den

---

<sup>3</sup><http://www.uni-koblenz.de/daniel/Namenskonventionen.html>

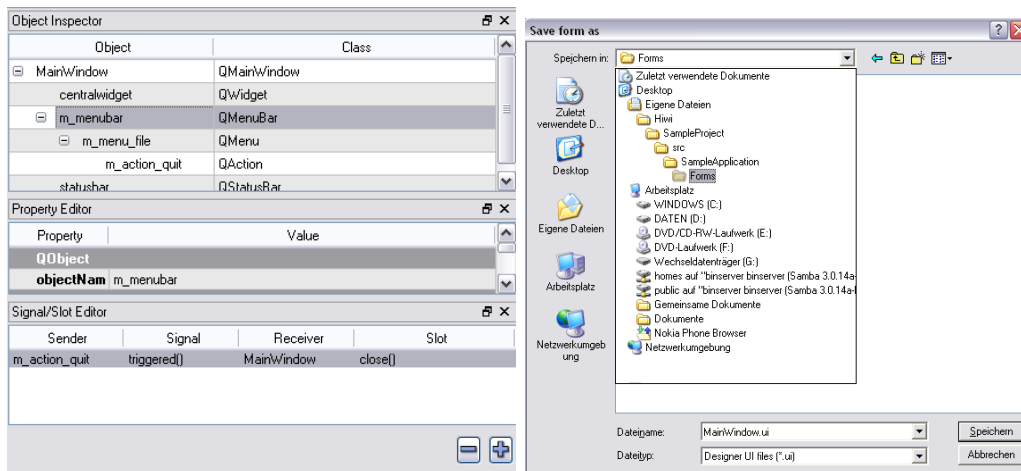


Abbildung 12: Verknüpfen der Signale mit Slots      Abbildung 13: Abspeichern der Form-Datei

selben wählen, den auch später die C++ Klasse erhalten soll und die Form im Designer hatte (im Beispiel MainWindow).

### 3.4 Integration der UI Datei in Visual Studio 2005

Um die gespeicherte Form Datei in Visual Studio 2005 zu integrieren, klickt man nun wieder mit der rechten Maustaste auf das Projekt und wählt diesmal *Add -> Existing Item....* Im sich öffnenden Dialog muss zunächst als Dateityp *All Files (\*.\*)* ausgewählt werden. Anschließend wählt man die zuvor gespeicherte UI Datei.

Beim ersten Mal wenn eine solche UI-Datei hinzugefügt wird, bekommt man eine Meldung angezeigt, dass keine passende *Custom Build Step* Regel gefunden werden konnte und man wird gefragt ob eine solche jetzt angelegt werden soll (siehe Abb. 14). Man bestätigt das nun mit einem Klick auf den *Yes*-Button.

Daraufhin öffnet sich ein weiterer Dialog. Innerhalb diesem wird zunächst angegeben, wo die benutzerdefinierten Build-Regeln abgespeichert werden sollen. In diesem Beispiel wird diese im Installationsverzeichnis von Visual Studio abgelegt und bekommt den Dateinamen *custom\_rules*. Als Display Name wird *Custom Build Rules* angegeben (siehe Abb. 15). Bevor man nun das ganze mit *OK* bestätigen darf, fügt man noch eine neue Build Regel mit einem Klick auf den *Add Build Rule...* Button hinzu.

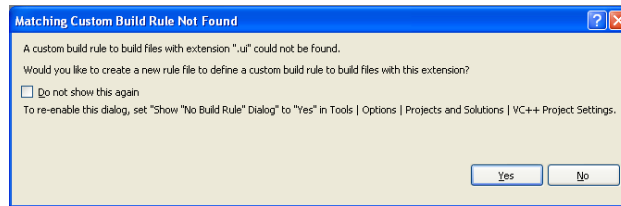


Abbildung 14: Neue Build Regel erstellen

Erneut öffnet sich ein Dialog, in dem nun die eigentliche Regel für UI Dateien eingetragen wird (siehe Abb. 16). Folgende Felder sind dabei auszufüllen:

- *Command Line:*

```
$(QTDIR)/bin/uic.exe "$(InputPath)" -o ui_$(InputName).h
```

- *Display Name*

```
Compile QT UserInterface File (*.ui)
```

- *Execution Description:*

```
Creating UI File ui\_\_$(InputName).h
```

- *File Extensions:*

```
*.ui
```

- *Name:*

```
UI
```

- *Outputs:*

```
ui_$(InputName).h
```

Nun kann man alle Dialog über das jeweilige Drücken der *OK* Buttons schließen. Damit sollte es jetzt möglich sein mit der rechten Maustaste auf die UI Datei zu klicken und den Eintrag *Compile* ohne Fehler auszuführen. Durch das anlegen dieser neuen Build Regel werden ab sofort alle dem Projekt hinzugefügten UI Dateien automatisch mit dem *UIC* Tool von QT übersetzt. Der UI-Compiler erzeugt dabei im Source Code Verzeichnis eine neue Header Datei welche die Definition einer Klasse mit sämtlichen Implementierungen der im Designer angelegten Objekte enthält. In FallIm Fall der Datei *MainWindow.ui* wäre dies die Header Datei *ui\_MainWindow.h* mit der Klasse *Ui\_MainWindow*.

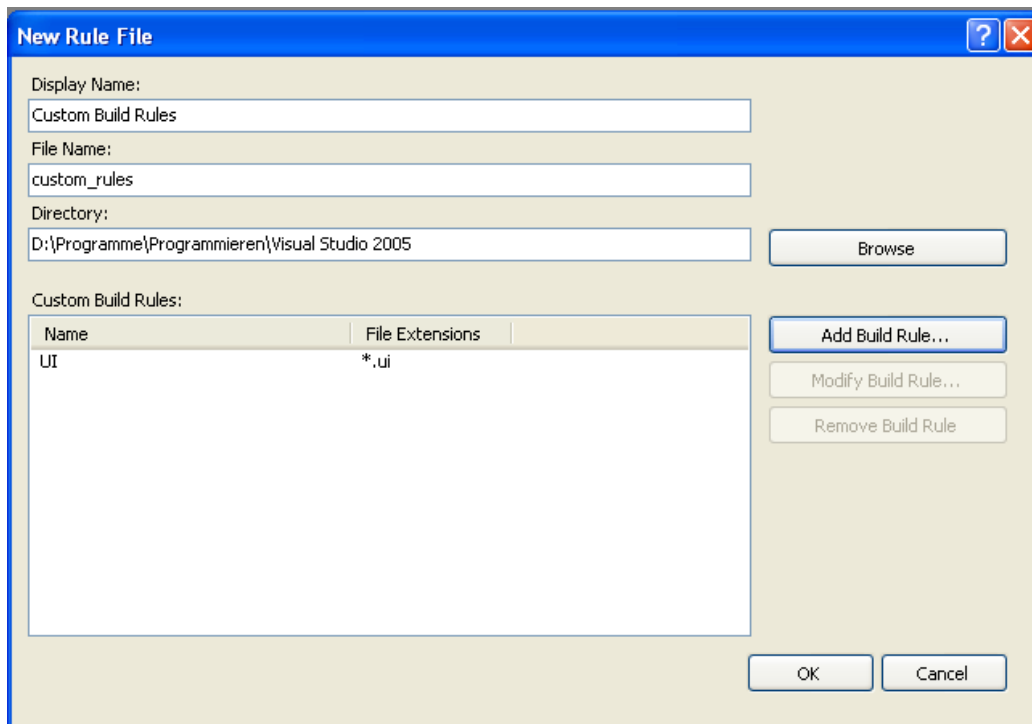


Abbildung 15: Erzeugen einer neuen Regel Datei

### 3.5 Erzeugen der Hauptklasse

Ähnlich wie schon in den vorherigen Schritten erwähnt (siehe Abschnitt 3.1), werden nun zwei neue Elemente dem Projekt hinzugefügt (rechte Maustaste auf Source, bzw. Header Ordner des Projekts und neues Element hinzufügen). Zum einen eine neue Header Datei *MainWindow.h* und eine neue CPP Datei *MainWindow.cpp*. Damit sind nun folgende Projekte in dem Projekt enthalten:

1. main.cpp
2. MainWindow.ui
3. MainWindow.h
4. MainWindow.cpp

Was nun noch fehlt ist der entsprechende Source Code. Die Dateien *MainWindow.h* und *MainWindow.cpp* werden später die Definition und Implementierung der eigentlichen GUI Klasse enthalten, während in der Datei *main.cpp* die Instanzierung dieser Klasse erfolgt.

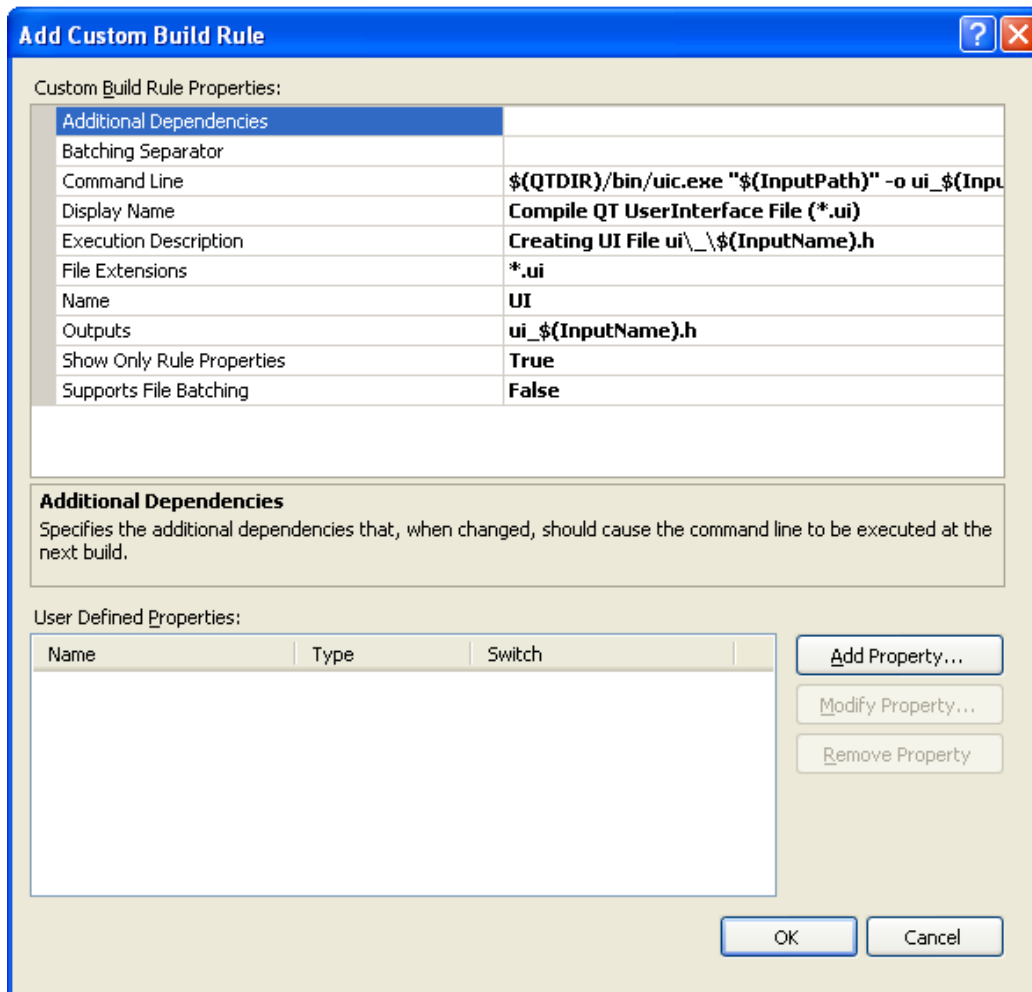


Abbildung 16: Definieren der *UI* Build Regel

Die Definition der GUI Klasse `MainWindow` erfolgt durch folgenden Source Code in der Datei `MainWindow.h`:

```
// to avoid multiple class definitions by including this file more than once
// we have to surround the class definition by this compiler flag
#ifndef MAINWINDOW_H_
#define MAINWINDOW_H_

#include "Ui_MainWindow.h"

/**
 * Sample MainWindow Class
 * The class is a simple implementation of a QDesigner created form file
 * defining a simple QMainWindow application
 */
```

```

    * The class inherits from QMainWindow and Ui_MainWindow. The Ui_MainWindow
    * provides the QDesigner part of the implementation, while the QMainWindow
    * provides the main functions of a QT Application
    */
class MainWindow : public QMainWindow, protected Ui_MainWindow
{
    Q_OBJECT

public:
    /**
     * Constructor of the MainWindow class
     * @param parent this optional parameter defines a parent widget the
     *         created instance will be child of
     * @param flags optional parameter defining extra widget options
     *         (see also the QT documentation)
     */
    MainWindow(QWidget* parent = 0, Qt::WindowFlags flags = 0);
    /**
     * Destructor of the MainWindow class, defined virtual to guarantee that
     * the destructor will be called even if the instance of this class is
     * saved in a variable of a parent class type
     */
    virtual ~MainWindow();

};
#endif // end of #ifndef MAINWINDOW_H_

```

Mit Ausnahme des *Q\_OBJECT* Makros (siehe Abschnitt 3.6) wird - neben den hier im Source Code gegebenen Erklärungen - im folgenden nicht weiter auf die einzelnen Anweisungen eingegangen. Hierfür sei auf ein entsprechendes C++ Buch verwiesen. Es ist jedoch anzumerken, dass eine ausgiebige Dokumentation des eigenen Source Codes immer hilfreich ist und deshalb dringend durchgeführt werden sollte. Die Art der Beispiel Dokumentation entspricht dabei einem Doxygen kompatiblen Syntax. Damit lassen sich später automatisch HTML Dokumentationen ähnlich des von QT oder Javadoc bekannten Stils erstellen.

Die Implementierung der Klassendefinition erfolgt in der Datei *MainWindow.cpp*. Bei dem aktuellen Funktionsumfang beschränkt sich diese auf folgende Zeilen:

```

#include "MainWindow.h"

MainWindow::MainWindow(QWidget* parent /* = 0 */, Qt::WindowFlags flags /* = 0 */)
    : QMainWindow(parent, flags)
{
    // create gui elements defined in the Ui_MainWindow class
    setupUi(this);
}

```

```

}

MainWindow::~MainWindow()
{
    // no need to delete child widgets, QT does it all for us
}

```

Damit ist die Implementierung der GUI Klasse abgeschlossen. Es fehlt nun noch die Instanzierung in der Datei *main.cpp*.

```

// Header file to get a new instance of QApplication
#include <Qt/qapplication.h>
// Header file for MainWindow class
#include "MainWindow.h"

int main(int argc, char* argv[])
{
    // create a new instance of QApplication with params argc and argv
    QApplication app( argc, argv );
    // create a new instance of MainWindow with no parent widget
    MainWindow* mainWindow = new MainWindow(0, Qt::Window);
    // The main widget is like any other, in most respects except that if it is deleted,
    // the application exits.
    app.setActiveWindow(mainWindow);
    // resize window to 640 x 480
    mainWindow->resize(640,480);
    // Shows the widget and its child widgets.
    mainWindow->show();
    // Enters the main event loop and waits until exit() is called
    // or the main widget is destroyed, and returns the value that
    // was set via to exit() (which is 0 if exit() is called via quit()).
    return app.exec();
}

```

Versucht man nun das Projekt zu übersetzen, so erhält man noch diverse Fehlermeldungen. Dies liegt daran, dass in den Projekteinstellungen keinerlei Angaben über den Ort der Header Dateien von Qt definiert sind. Hier ist es wieder hilfreich, dass während der Installation das QT Installationsverzeichnis in der Umgebungsvariablen *QTDIR* abgespeichert wurde. Um nun die Header Dateien dem Projekt bekannt zu machen, klickt man mit der rechten Maustaste auf das Projekt und wählt im Popup Menü den Punkt *Properties*. Im sich öffnenden Dialogfenster wählt man in der Combobox *Configuration* am oberen Rand den Eintrag *All Configurations*. Anschließend wählt man auf der linken Seite die Kategorie *C/C++* und dort die Unterkategorie *General* (siehe Abb. 17). In der Zeile *Additional Include Directories* trägt man nun folgendes ein:

"\$(QTDIR)/include";

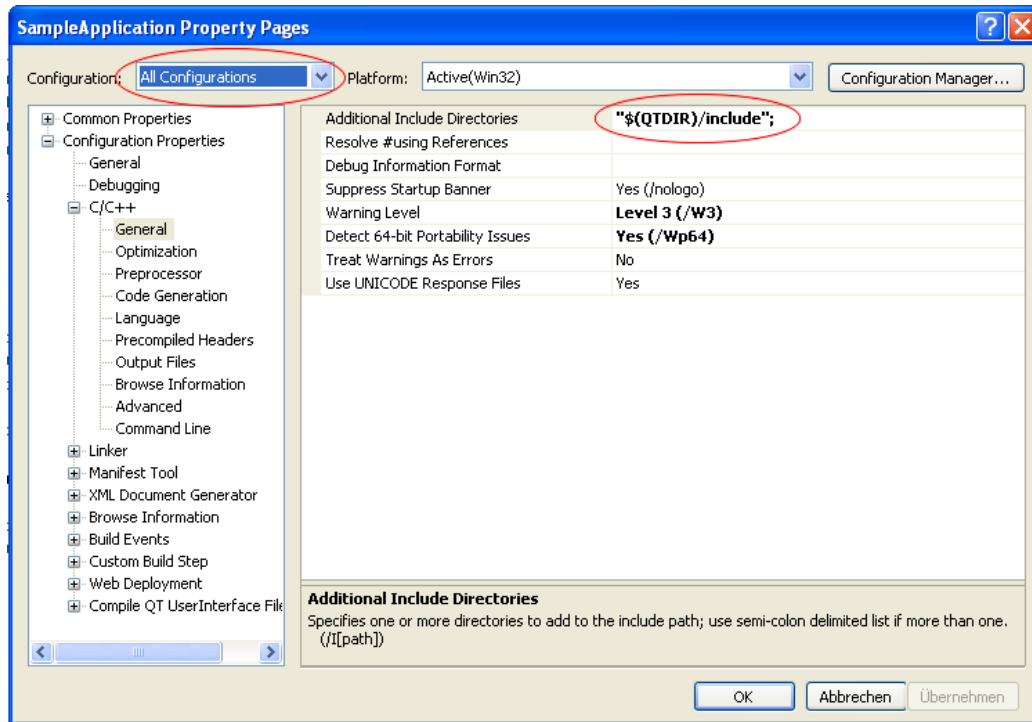


Abbildung 17: Einstellen der Include Pfade in den Projekteinstellungen

Neben den Header Dateien werden auch die Library Dateien von QT benötigt. Für eine einfache Anwendung mit GUI Elementen werden die Lib Dateien *QtGuid4.lib* und *QtCored4.lib* in der Debug Konfiguration bzw. *QtGui4.lib* und *QtCore4.lib* in der Release Konfiguration benötigt. Dazu wählt man zunächst die entsprechende Konfiguration über die erwähnte Combobox *Configuration*. Anschließend klickt man in der Kategorie *Linker* auf die Unterkategorie *Input*. Dort trägt man in der Zeile *Zusätzliche Abhängigkeiten* im Fall der Debug Konfiguration folgendes ein:

QtGuid4.lib QtCored4.lib

im Fall der Release Konfiguration:

QtGui4.lib QtCore4.lib

Damit der Linker die angegebenen Bibliotheken findet, muss auch hier die entsprechende Pfadangabe eingetragen werden. Dazu wechselt man wieder auf *All Configurations* und anschließend in die Kategorie *Linker*, Unterkategorie *General*. Dort trägt man in die Zeile *Additional Library Directories* folgendes ein:

```
"$(QTDIR)/lib";
```

Damit hat man vorerst sämtliche wichtigen Projekteinstellungen abgeschlossen. Jedoch treten beim Übersetzen des Projekts immer noch Fehler auf. Im Detail sind dies drei Linker Fehler, die angeben, dass bestimmte Symbole (die Funktionen `qt_metacall`, `qt_metacast` und `metaObject`) nicht aufgelöst werden können.

### 3.6 Der Meta-Object-Compiler von QT

Der Meta Object Compiler (*moc*) von QT ist für die SIGNALs und SLOTs von QT notwendig. QT erlaubt es *Signale* zu emitieren und damit verbundene *Slots* aufzurufen. Verbunden wird ein Signal und ein Slot über die Funktion *connect*.

Jede Klasse die von `QObject` oder einer davon abgeleiteten Klasse erbt, kann selbst Signale und Slots definieren, bzw. in Oberklassen definierte Signale und Slots nutzen. Wenn eine Klasse eigene Slots oder Signale definiert, muß das Makro `Q_OBJECT` in die Klassendefinition aufgenommen werden (siehe Abschnitt 3.5). Ist dieses Makro in der Header Datei vorhanden, muß diese vor dem Übersetzen des Source Codes mit dem C++ Compiler erst noch vom Meta Object Compiler von QT kompiliert werden.

Am einfachsten integriert man diesen zusätzlichen Verarbeitungsvorgang in Visual Studio, indem man einen benutzerdefinierten Build Schritt den entsprechenden Header-Dateien (im Beispiel *MainWindow.h*) hinzufügt. (Rechte Maustaste auf Header Datei -> *Properties*) Es erscheint ein Dialog in dem auf der linken Seite unter *Configuration Properties* der Eintrag *Custom Build Step* zu finden ist. Bevor man dort die nötigen Einstellungen vornimmt, wechselt man noch über die Combobox *Configuration* auf den Eintrag *All Configurations*. Anschließend trägt man jeweils die folgenden Sachen in die entsprechenden Felder ein:

- *Command Line:*

```
$(QTDIR)\bin\moc.exe "$(InputPath)" -o "$(InputDir)moc_$(InputName).cpp"
```

- *Description:*

```
Performing moc on $(InputName).h
```

- *Outputs:*

```
$(InputDir)moc_$(InputName).cpp
```

Alternativ könnte auch eine allgemeine Build Regel wie in Abschnitt 3.4 erstellt werden. Dies ist in diesem Fall aber nicht zu empfehlen, da die Regel ja nicht auf alle \*.h Dateien angewendet werden soll, sondern nur auf jene, die die Q\_OBJECT Anweisung enthalten. Der Nachteil ist allerdings, dass die entsprechenden Einträge bei jeder neuen Header Datei manuell neu gesetzt werden müssen.

Wer lieber eine allgemeine Regel definiert möchte, kann die *Rule* Datei mit den Custom Build Rules des Projekts bearbeiten, in dem man mit der rechten Maustaste auf das Projekt klickt und anschließend den Eintrag *Custom Build Rules...* auswählt. Die weiteren Einstellungen erfolgen dann analog zu Abschnitt 3.4.

Mit einem Rechtsklick auf die Header Datei *MainWindow.h* kann man nun über den Eintrag *Compile* den MO'Compiler seine Arbeit leisten lassen. Daraufhin sollte eine entsprechende Datei *moc\_MainWindow.cpp* im Verzeichnis abgelegt werden, welches auch die übrigen Quelldateien enthält. Diese Datei muss dem Projekt hinzugefügt werden (auf dem Projekt rechte Maustaste drücken -> Add -> Existing Item...).

Nun sind folgende Dateien im Projekt vorhanden:

1. main.cpp
2. MainWindow.ui
3. MainWindow.h
4. MainWindow.cpp
5. moc\_MainWindow.cpp

Wird das Projekt nun ohne Fehler übersetzt, kann es über den Play Button von Visual Studio oder das entsprechende Kommando in den Menüs gestartet werden. Damit ist die erste QT Anwendung fertig gestellt!

### 3.7 Erzeugen eigener Slot Funktionen

Um der Beispielanwendung ein wenig mehr Funktion zu verleihen, wird im Folgenden gezeigt wie eigene Slot Funktionen definiert werden können und diese mit Signalen verbunden werden.

Der erste Schritt ist das Erzeugen eines neuen `QAction` Objects, welches in das Menü integriert wird. Dazu öffnet man zunächst im Designer von QT

wieder die UI Datei *MainWindow.ui*. Dies kann auch direkt aus Visual Studio erfolgen.

Durch einen Klick mit der rechten Maustaste auf die Datei und anschließender Wahl des Menüpunkts *Öffnen mit...* bzw. *Open with...* erscheint ein Dialog, in dem man wählen kann, mit welchem Programm man die Datei öffnen möchte. Wählt man anschließend dabei den QT Designer und definiert diesen als Standard, so lässt sich zukünftig jede UI Datei mit einem Doppelklick im Designer öffnen.

Im QT Designer selbst wird nun analog zum *Beenden* Eintrag ein neuer Menüpunkt erzeugt. In diesem Beispiel bekommt der neue Eintrag den Namen *Hello World...* und wird vor den Eintrag *Beenden* gezogen (siehe Abb. 18). Auch der Name des `QAction` Objekts wird wieder in Anlehnung an die im Abschnitt 3.3 angesprochenen ungarischen Notation gewählt: *m\_action\_hello*. Damit sind die Arbeiten im Designer vorerst beendet.

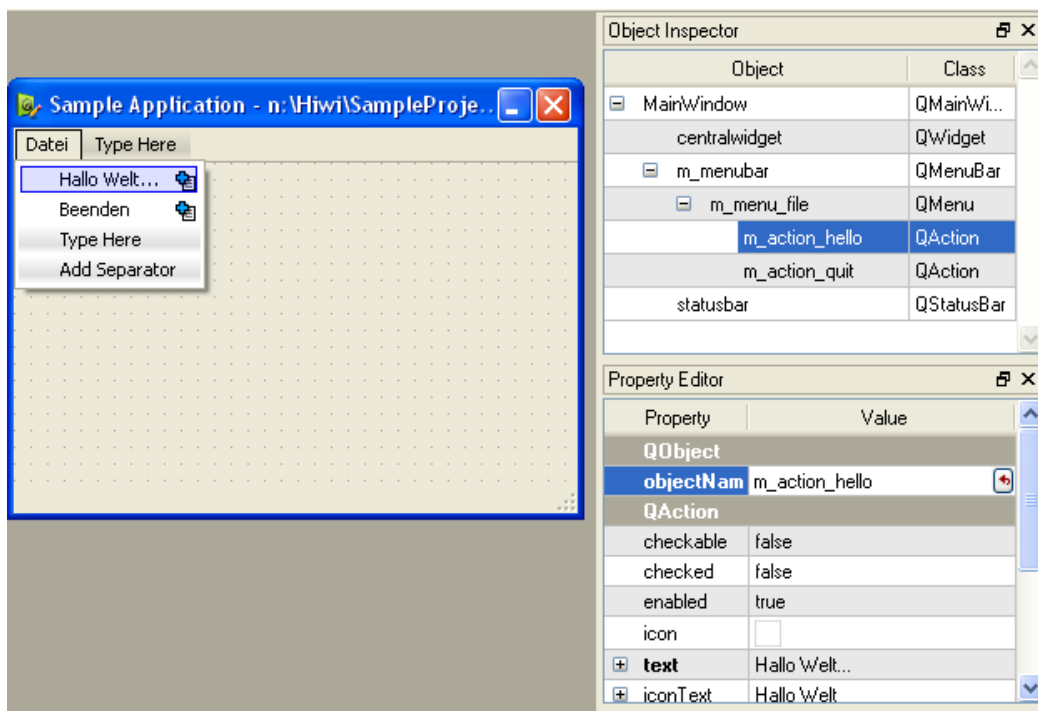


Abbildung 18: Erzeugen eines neuen `QAction` Objekts

Probiert man nun die Anwendung zu starten, existiert bereits der neue Menüeintrag *Hello World...* jedoch passiert noch nichts, wenn man darauf klickt. Um nun eine Funktionalität zu programmieren definiert man zunächst in der

Header Datei *MainWindow.h* eine entsprechende Slot Methode. Damit der MO'Compiler später weiß, welche Methoden Slots darstellen, werden diese in einen eigenen Bereich gesetzt, welcher durch das Schlüsselwort *slots* definiert wird. In diesem Beispiel wird eine *protected* slot Methode erzeugt indem ein entsprechender Abschnitt *protected slots*: innerhalb der Header Datei *MainWindow.h* erzeugt wird. Danach erfolgt die Definition ganz normal wie bei anderen C++ Methoden auch durch die Angabe von Rückgabetyt, Methodenname und Parameterliste. In diesem Fall:

```
void helloWorldSlot();
```

Im folgenden nochmal der gesamte Code der Header Datei *MainWindow.h* nach den erfolgten Ergänzungen:

```
// to avoid multiple class definitions by including this file more than once
// we have to surround the class definition by this compiler flag
#ifndef MAINWINDOW_H_
#define MAINWINDOW_H_

#include "Ui_MainWindow.h"

/**
 * Sample MainWindow Class
 * The class is a simple implementation of a QDesigner created form file
 * defining a simple QMainWindow application
 * The class inherits from QMainWindow and Ui_MainWindow. The Ui_MainWindow
 * provides the QDesigner part of the implementation, while the QMainWindow
 * provides the main functions of a QT Application
 */
class MainWindow : public QMainWindow, protected Ui_MainWindow
{
    Q_OBJECT

public:
    /**
     * Constructor of the MainWindow class
     * @param parent this optional parameter defines a parent widget the
     *         created instance will be child of
     * @param flags optional parameter defining extra widget options
     *         (see also the QT documentation)
     */
    MainWindow(QWidget* parent = 0, Qt::WindowFlags flags = 0);
    /**
     * Destructor of the MainWindow class, defined virtual to guarantee that
     * the destructor will be called even if the instance of this class is
     * saved in a variable of a parent class type
     */
    virtual ~MainWindow();

protected slots:
    void helloWorldSlot();

};
#endif // end of #ifndef MAINWINDOW_H_
```

Zu jeder Deklaration muss auch irgendwo eine Implementierung vorhanden sein, weshalb als nächster Schritt die gerade erzeugte Methode *helloWorldSlot()* in der Datei *MainWindow.cpp* einen Rumpf bekommt.

```
void MainWindow::helloWorldSlot()
{
    QMessageBox::information(this, tr("MainWindow Message"), tr("Hello World!"));
}
```

Wie man sehen kann, beschränkt sich die Funktionalität auf die Anzeige einer *MessageBox*, die den Text *Hello World!* beinhaltet. Für nähere Erläuterungen zum Funktionsumfang von *QMessageBox* sei hier einmal mehr auf die *QT* Dokumentation verwiesen.

Damit auf die Funktion von *QMessageBox* zugegriffen werden kann muss noch die entsprechende Header Datei am Anfang der *MainWindow.cpp* eingebunden werden. Dies erfolgt durch folgende Zeile:

```
#include <qt/qmessagebox.h>
```

Würde man das Programm nun versuchen zu starten, so würde zwar keine Probleme beim Übersetzen auftreten, jedoch würde nach wie vor nichts passieren, wenn man auf den Menüeintrag *Hello World...* klickt. Dazu fehlt noch die Verbindung zwischen dem Signal des *QAction* Objekts und dem Slot *helloWorldSlot()*. Diese kann z.B. im Constructor der *MainWindow* Klasse erfolgen in dem man folgenden Code einfügt:

```
connect(m_action_hello, SIGNAL(triggered()), this, SLOT(helloWorldSlot()));
```

Prinzipiell kann die Erstellung einer solchen Verbindung auch an anderer Stelle im Programm erfolgen, also theoretisch auch in einer anderen Slot Methode. Um eine solche Verbindung zur Laufzeit wieder zu lösen, kann ein analoger Aufruf mittels *disconnect* erfolgen.

Im folgenden noch einmal die komplette *MainWindow.cpp*

```
#include "MainWindow.h"
#include <qt/qmessagebox.h>

MainWindow::MainWindow(QWidget* parent /* = 0 */, Qt::WindowFlags flags /* = 0 */)
    : QMainWindow(parent, flags)
{
    // create gui elements defined in the Ui_MainWindow class
    setupUi(this);
    connect(m_action_hello, SIGNAL(triggered()), this, SLOT(helloWorldSlot()));
}

MainWindow::~MainWindow()
```

```

{
    // no need to delete child widgets, QT does it all for us
}

void MainWindow::helloWorldSlot()
{
    QMessageBox::information(this, tr("MainWindow Message"), tr("Hello World!"));
}

```

Prinzipiell ist es auch Möglich Slot Funktionen Parameter zu übergeben. Diese können dann mit Signalen die Parameter des selben Typs senden verbunden werden. Ein Beispiel wäre die Verbindung des *currentIndexChanged(int)* Signals von `QComboBox` mit einem eigenen Slot, der als Parameter ebenfalls einen Integer Wert erwartet. Um die Fülle an Möglichkeiten kennen zu lernen, empfiehlt sich ein Blick auch auf die Beispiele die bei QT bereits dabei sind oder ein weiterer Blick in die Dokumentation von QT.

## A Copyright Notiz



### Namensnennung-NichtKommerziell-Weitergabe unter gleichen Bedingungen 2.0 Deutschland

Sie dürfen:

- den Inhalt vervielfältigen, verbreiten und öffentlich aufführen
- Bearbeitungen anfertigen

Zu den folgenden Bedingungen:

- *(by)* **Namensnennung.** Sie müssen den Namen des Autors/Rechtsinhabers nennen.
- *(nc)* **Keine kommerzielle Nutzung.** Dieser Inhalt darf nicht für kommerzielle Zwecke verwendet werden.
- *(sa)* **Weitergabe unter gleichen Bedingungen.** Wenn Sie diesen Inhalt bearbeiten oder in anderer Weise umgestalten, verändern oder als Grundlage für einen anderen Inhalt verwenden, dann dürfen Sie den neu entstandenen Inhalt nur unter Verwendung identischer Lizenzbedingungen weitergeben.
- Im Falle einer Verbreitung müssen Sie anderen die Lizenzbedingungen, unter die dieser Inhalt fällt, mitteilen.
- Jede dieser Bedingungen kann nach schriftlicher Einwilligung des Rechtsinhabers aufgehoben werden.

**Die gesetzlichen Schranken des Urheberrechts bleiben hiervon unberührt.**

Das Commons Deed ist eine Zusammenfassung des Lizenzvertrags<sup>4</sup> in allgemeinverständlicher Sprache.

---

<sup>4</sup><http://creativecommons.org/licenses/by-nc-sa/2.0/de/legalcode>

## **A.1 Haftungsausschluss:**

The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) - it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license.

Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

Für die Anwendung der in diesem Tutorial erwähnten Vorgänge wird keine Haftung übernommen, ebenso wenig wie für daraus entstehende Schäden.