The Amadeus Leisure Group

Einführung in die GUI-Programmierung mit Qt 4

Erstbetreuer: Prof. Dr. rer. nat. Gerhard Dikta
Zweitbetreuer: Uli Schmidt

Company	TravelTainment GmbH		
Author	Yannick Müller, Matr.-Nr.: 833948		
Version	Date	Change	By
1.0	14.12.2011	Erstellt	Y. Müller

Inhalt

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Einführung in die GUI-Programmierung mit Qt 4

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name: Yannick Müller

Aachen, den 14. Dezember 2011

1 Einleitung

In dieser Seminararbeit wird zu Beginn kurz umrissen, was Qt ist und welche Vor- und Nachteile sich durch den Einsatz dieses Frameworks ergeben. Es folgt eine kurze Erläuterung der Grundlagen der Programmierung mit Qt. Danach wird ein wichtiges Grundkonzept, Signale und Slots, erläutert und anhand eines Beispiels veranschaulicht.
Auf diese Grundlagen aufbauend wird anschließend das eigentliche Thema der GUI-Programmierung behandelt. Dort werden zunächst verschiedene Layouts diskutiert und dann einige Widgets vorgestellt.
Der theoretische Hintergrund wird mit der Ereignisverarbeitung von Qt, so wie mit einem Ausblick auf die eingebauten Systeme zur Internationalisierung und zum Ressourcenmanagement beendet.
Das Thema dieser Seminararbeit wurde in Hinblick auf eine zukünftige Bachelorarbeit gewählt, für die ein Programm mit Qt entwickelt werden soll.

2 Das Framework Qt

2.1 Was ist Qt?

Derzeit wird Qt von Nokia entwickelt. Es war ursprünglich ein Framework zur plattformunabhängigen Programmierung grafischer Benutzeroberflächen (GUI) in C++. Es beinhaltet jedoch noch sehr viele andere Bibliotheken, z.B. für die Netzwerk-Programmierung, SQL-, XML-, SVG-, JavaScript- und OpenGL-Unterstützung. Zudem stellt Nokia zur Entwicklung mit Qt eine vollständige Entwicklungsumgebung, einen GUI-Builder und einige andere Tools zur Verfügung, sodass die gesamte Programmierung allein mit diesen bewältigt werden kann.
Qt erweitert die Sprache C++ um einige Komponenten, weshalb ein zusätzlicher Präprozessor, *qmake*, verwendet wird, der aus Qt-Quellcode standardkonformen C++-Quellcode erstellt. *qmake* kann aber auch z.B. aus den XML-Dateien, die der GUI-Builder generiert, C++-Code erzeugen.
Es gibt zudem noch Anbindungen an andere Programmiersprachen, z.B. an C#, Java und Python.
Beispiele für mit Qt erstellte Software sind der Linux-Desktop KDE, Google Earth, Skype und VLC.

2.2 Vorteile von Qt

Es gibt viele Gründe für den Einsatz von Qt. Es ist ein sehr mächtiges und trotzdem sehr einfach zu benutzendes Framework. Es lässt sich auf vielen Plattformen, z.B. auf Desktopsystemen wie Windows, Linux, Mac OS und auf mobilen Plattformen wie Symbian OS oder Maemo einsetzen. Qt steht unter einem dualen Lizenzsystem, was bedeutet, dass es sowohl unter einer proprietären (kommerziellen) Lizenz als auch unter der freien LGPL verfügbar ist. Qt bietet eine strukturierte Dokumentation und bringt eine der umfangreichsten Entwicklungsumgebungen mit, den Qt Creator.

Das Qt-Framework codiert Zeichen durchgängig in UTF-16, wodurch Sonderzeichen oder beispielsweise chinesische Schriftzeichen dargestellt werden können. Dies ermöglicht internationale Anwendungen. Klassische C-Strings hingegen können pro Zeichen nur ein Byte verwenden und daher nur einen begrenzten Zeichensatz von maximal 256 verschiedenen Zeichen darstellen. Zudem kann bei Qt jederzeit ein angezeigter String mit HTML-Tags formatiert werden.

Für einige Datentypen hat Qt eigene Bezeichnungen eingeführt, die kürzer und damit schneller zu tippen sind, als die Standard-C++-Datentypen (z.B. `uint` statt `unsinged int`). Des Weiteren gibt es Datentypen, die eine bestimmte Bit-Anzahl haben, was ein Vorteil gegenüber Standard-C++ ist, da die hier Länge der Datentypen nicht festgelegt ist (z.B. `qint32` für 32 Bits).

Außerdem bietet Qt ein Internationalisierungs-System an, mit dem man Anwendungen in beliebige Sprachen übersetzen kann.

2.3 Nachteile von Qt

Für die Benutzung von Qt-Programmen werden Qt-Bibliotheken benötigt. Sind diese im Ziel-System nicht vorhanden, müssen sie mitgeliefert werden. Verglichen mit anderen Frameworks erzeugt dies relativ große Programme. Das Hello-World-Programm aus Kapitel 2.4 benötigt beispielsweise unter Windows 18KB, während die zwei benötigten DLLs zusammen 10MB groß sind.

Zudem ist die Dokumentation von Qt recht knapp gehalten und manchmal wird bemängelt, dass die Programmierung nicht in reinem C++ stattfindet, was eine direkte Kompilierung verhindert. Somit wird ein zusätzliches Programm zur Generierung von komprimierbarem Code benötigt.

2.4 Grundlagen

```
01 #include <QApplication>
02 #include <QWidget>
03 #include <QLabel>
04 #include <QPushButton>
05 #include <QVBoxLayout>
06 //alternativ:
07 //#include <QtGui>
08
09 int main(int argc, char *argv[])
10 {
11     //wird von jeder GUI-Anwendung benötigt
12     QApplication app(argc, argv);
13
14     //Ein Widget erzeugen, dass als Fenster fungiert.
15     QWidget window;
16     //Fenstertitel setzen
17     window.setWindowTitle("Qt4 Example");
18
19     //Ein Label mit einem Text erzeugen
20     QLabel *label = new QLabel("Hello World!");
21
22     //Einen Button mit dem Tastenkürzel Alt-E erzeugen
23     QPushButton *button = new QPushButton("&Exit");
24
25     //Label und Button untereinander anordnen
26     QVBoxLayout *layout = new QVBoxLayout(&window);
27     layout->addWidget(label);
28     layout->addWidget(button);
29
30     //Fenseter sichtbar machen
31     window.show();
32
33     //Die Eventschleife starten
34     return app.exec();
35 }
```

Dies ist ein Hello-World-Programm in Qt. Es wird ein Fenster erzeugt, in dem ein Text und ein Button zu sehen sind.

Jede Qt-GUI-Anwendung muss eine `QApplication` erstellen (Zeile 12), die für die Eventschleife zuständig ist und diese zum Schluss (in Zeile 34) starten. Mehr dazu in Kapitel 5.

In Zeile 15 wird ein `QWidget` angelegt, um es als Fenster zu benutzen. `QWidget` ist die Basisklasse aller Benutzersteuerelemente. Sie erbt von der Basisklasse der meisten Qt-Klassen: `QObject`. So wird es möglich, die Speicherverwaltung, die in C++ normalerweise manuell organisiert werden muss, ein Stück weit zu automatisieren. `QObjects` werden hierarchisch strukturiert, indem ein `QObject` ein Elternelement und Kindelemente besitzt. Wird ein `QObject` zerstört, werden alle Kindelemente ebenfalls zerstört. Besitzt es selber

einen Vater, wird dieser ebenfalls benachrichtigt. So wird vermieden, dass der bei der Zerstörung des Vaters versucht wird, das bereits zerstörte Kind zu löschen. Aus diesem Grund müssen alle `QObjects`, die nicht vaterlos sind, mit dem `new`-Operator auf dem Heap angelegt werden, sodass sie nicht zu früh zerstört werden. Qt verwendet daher an vielen Stellen Zeiger als Funktionsparameter, da dadurch in der Regel alle `QObjects` als Zeiger angelegt werden. Normalerweise bietet eine Subklasse von `QObject` immer einen Konstruktor mit einem Zeiger auf das Vaterelement an. Somit kann die Eltern-Kind-Beziehung sofort hergestellt werden. Aber auch das nachträgliche Eingliedern in die Hierarchie ist möglich. Ein elternloses `QWidget` wird immer als Fenster angezeigt, andere `QWidgets` werden innerhalb des Vaterelements dargestellt.

In Zeile 20 und 23 werden ein Label zur Darstellung eines Textes und ein Button erzeugt. Diese sind zu Beginn elternlos. Zur sinnvollen Anordnung der Komponenten innerhalb des Fensters wird ein Layout benötigt. Dieses wird in Zeile 26 als Kindelement von `window` angelegt. Anschließend werden das Label und der Button mithilfe des Layouts untereinander angeordnet. Dies geschieht durch den Aufruf von `addWidget` in Zeile 27f. Dadurch übernimmt `window` die Elternschaft von dem Label und dem Button. Auf das Thema Layouts wird in Kapitel 4.1 näher eingegangen.

Damit das Fenster angezeigt wird, wird es in Zeile 31 die Methode `show()` aufgerufen. Als letztes wird nun die Eventschleife gestartet. Das Programm läuft so lange, bis das Fenster geschlossen wird.

Um Qt-Klassen verwenden zu können, müssen entsprechende Header-Dateien eingebunden werden (Zeile 2.4ff.). Diese tragen in Qt stets denselben Namen wie die zu verwendende Klasse. Alternativ steht eine Header-Datei für jedes Qt-Modul bereit, das alle Header des Moduls einbindet. Der Name dieser Datei setzt sich zusammen aus „Qt“ und dem Namen des Moduls. Um alle GUI-Komponenten verwenden zu können, kann alternativ `QtGui` eingebunden werden.

Die Übergabe von Objekten an Methoden kann in C++ auf drei Weisen erfolgen: Es kann ein Zeiger auf das Objekt, eine Referenz oder eine Kopie des Objektes übergeben werden. Keine dieser Möglichkeiten ist unproblematisch. Bei Zeigern und Referenzen kann es passieren, dass das übergebene Objekt durch Verlassen seines Gültigkeitsbereiches zerstört wird, das aufgerufene Objekt aber später noch einmal versucht, darauf zuzugreifen. Die Alternative wäre, das Objekt auf dem Heap anzulegen. Allerdings muss dann darauf geachtet werden, dass das Objekt auch wieder korrekt gelöscht wird. Die einfachste Möglichkeit ist, eine Kopie

zu übergeben. Dies ist aber wiederum aufwändig und nicht alle Objekte können kopiert werden. Qt versucht, dieses Problem zu umgehen. Zum einen wird die Speicherverwaltung von `QObject`s wie oben beschrieben durchgeführt. Zum anderen wird das Prinzip des Copy-On-Write verwendet. Dies bedeutet, dass beim Anfertigen einer Kopie zunächst eine flache Kopie angefertigt wird, deren Komponenten auf dieselben Speicherstellen zeigen wie die des Originals. Das geht deutlich schneller als eine tiefe Kopie anzufertigen und reicht in vielen Fällen aus, wenn z.B. nur gelesen werden muss oder sowieso Kopie oder Original sofort wieder zerstört wird. Erst wenn versucht wird, die Kopie zu verändern, wird eine tiefe Kopie angefertigt.

3 Signale und Slots

Bei jeder Aktion auf einer Benutzeroberfläche wird eine Funktion aufgerufen. Wird beispielsweise bei einem Klick auf einen Button eine Nachricht angezeigt, wird eine Funktion aufgerufen, die die Erzeugung und Anzeige des Nachrichtenfensters übernimmt. Häufig werden von GUI-Frameworks sogenannte Callback-Funktionen verwendet. Dies sind Funktionszeiger, die zwar schnell, aber nicht typsicher sind. Bei Qt wird dies anders umgesetzt: Es richtet Signal-Slot-Verbindungen ein. Diese haben den Vorteil, dass sie typsicher sind und dass die Verbindung automatisch getrennt wird, sobald eines der kommunizierenden Objekte gelöscht wird.

3.1 Grundlagen

Um in dem Hello-World-Beispiel (Kapitel 2.4) bei einem Button-Klick das Programm zu beenden, kann folgende Zeile nach der Erzeugung des Buttons eingefügt werden:

```
24 QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));
```

Hier wird der Slot `quit()` des Objekts `app` mit dem Signal `clicked()` des Objekts `button` verbunden. Der Slot `quit()` bewirkt, dass das Programm beendet wird. Slots sind herkömmliche C++-Funktionen mit Rückgabetyp `void`. Das Signal `clicked()` wird aufgerufen, wenn der Benutzer den Button drückt. Zum Einrichten einer Signal-Slot-Verbindung wird die statische Methode `QObject::connect()` verwendet:

```
static bool QObject::connect (
  const QObject * sender,   const char * signal,
  const QObject * receiver, const char * method,
  Qt::ConnectionType type = Qt::AutoConnection )
```

Sie benötigt als Parameter den Sender, dessen Signal, den Empfänger und dessen Slot. Die Argumente `signal` und `slot` sind keine Funktionszeiger, sondern C-Strings. Daher muss man die Makros `SIGNAL()` bzw. `SLOT()` verwenden, die die korrekten Argumente einsetzen. Die Verbindungen können mittels `QObject::disconnect()` auch wieder gelöst werden.

In obigem Beispiel haben die beiden Methoden keine Übergabeparameter. Falls ein Signal oder ein Slot einen oder mehrere Parameter erfordert, müssen die Typen der Parameter in den Klammern angegeben werden. Werte sind nicht erlaubt. Der Slot des Empfängers muss immer dieselbe oder eine kürzere Argumentliste haben, wie das Signal des Senders. So definieren beispielsweise die Klassen `QSpinBox` und `QSlider` jeweils diese beiden Signale und Slots:

```
void valueChanged ( int i ) [signal]
void setValue ( int val ) [slot]
```

`QSpinBox` repräsentiert, wie der Name schon sagt, einen Spinner in Qt, `QSlider` analog einen Slider. Wird der Wert der Komponente geändert, wird entsprechend das Signal `valueChanged(int)` aufgerufen. Angenommen es existieren eine `QSpinBox * spinner` und ein `QSlider * slider`, dann lassen sich diese also z.B. wie folgt gegenseitig verknüpfen:

Abbildung 3.1 Eine QSpinBox

Abbildung 3.2 Ein QSlider

```
01 QObject::connect(spinner, SIGNAL(valueChanged(int)),
   slider,  SLOT(setValue(int)));
02 QObject::connect(slider,  SIGNAL(valueChanged(int)),
   spinner, SLOT(setValue(int)));
```

3.2 Eigene Signale und Slots

Um Signale und Slots zu erstellen, muss man eine Klasse schreiben, die sich von `QObject` ableitet. Die Ableitung muss `public` sein, damit die Klasse zu `QObject` gecastet werden kann. Außerdem muss das Makro `Q_OBJECT` definiert sein. Dann stehen die beiden Qt-Schlüsselwörter `signals` und `slots` zur Verfügung, mit denen Methoden als Signal bzw. Slot markiert werden können. Für eine Beispielklasse könnte die Header-Datei so aussehen:

```
01 #ifndef MYOBJECT_H
02 #define MYOBJECT_H
03
04 #include <QObject>
05
06 class MyObject : public QObject
07 {
08     Q_OBJECT
09 private:
10     int number;
11 public:
12     //Konstruktor
13     MyObject(QObject * parent = 0);
14     //Zugriffsmethoden
15     int getNumber() const;
16 public slots:
17     //funktionert weiterhin als Setter, ist aber zusätzlich ein Slot
18     void setNumber(int num);
19 signals:    //signals erlaubt keinen Modifier.
20     void numberChanged(int num);
21 };
22 #endif // MYOBJECT_H
```

Diese Klasse ist ein Wrapper für ein `int`. Sie bietet einen Getter und einen Setter für `number`. Der Setter fungiert hier zusätzlich als Slot. Das bedeutet, dass ein Objekt dieser Klasse als Empfänger einer Signal-Slot-Verbindung eingetragen werden kann. Der Setter kann so auf jedes Signal registriert werden, das ein `int` als Parameter hat. Er ist auch weiterhin eine klassische C++-Methode, hat die gleichen Funktionen wie eine solche und kann auch weiterhin so aufgerufen werden. Ein Slot darf jedoch nicht statisch sein. Zusätzlich existiert ein Signal, das ausgelöst wird, wenn `number` geändert wird. Eine cpp-Datei dazu könnte so aussehen:

```
01 #include "myobject.h"
02 #include <QDebug>
03
04 MyObject::MyObject(QObject * parent)
05     : QObject(parent)
06 {
07     number = 0;
08 }
09
10 void MyObject::setNumber(int num)
11 {
12     if (number != num)
13     {
14         number = num;
15         qDebug() << "setNumber" << num;
16         emit numberChanged(num);
17     }
18 }
```

Auffällig ist, dass die Implementierung der Methode `numberChanged(int)` fehlt. Die Implementierung darf nicht vorhanden sein, sie wird stattdessen vom Präprozessor übernommen. Diese Implementierung sorgt dafür, dass die Slots aller registrierten Verbindungen zu diesem Signal benachrichtigt werden, wenn die Methode aufgerufen wird.
Der Setter erzeugt eine Debug-Ausgabe mittels `qDebug()`. Diese Variante hat, im Vergleich zu `std::cout` und `std::err`, den Vorteil, dass sie komfortabler zu handhaben ist und sich leicht durch das Makro `QT_NO_DEBUG_OUTPUT` deaktivieren lässt.
Außerdem fällt das Wort `emit` auf. Es ist ein weiteres Qt-Schlüsselwort und signalisiert, dass ein Signal angestoßen wird. Es stellt keine Funktionalität zur Verfügung und kann auch weggelassen werden, sollte aber wegen besserer Lesbarkeit verwendet werden.
`MyObject` kann also für Signal-Slot-Verbindungen verwendet werden. Angenommen, es existiert eine `QSpinBox * spinner`, dann kann ein gegebenes `MyObject * obj` wie oben beschrieben mit dem Spinner verbunden werden:

```
01 QObject::connect(spinner, SIGNAL(valueChanged(int)),
   obj, SLOT(setNumber(int)));
```

Meistens wird man jedoch eine Klasse nicht direkt von `QObject` ableiten, sondern von einer Subklasse, die bereits zum Teil den nötigen Anforderungen entspricht.

4 GUI

Die Basis einer jeden grafischen Oberfläche bildet in Qt die Klasse `QWidget`. Diese erbt von `QObject` und `QPaintDevice`. `QPaintDevice` ist die Basis für alles, was gezeichnet wird. Widgets haben auch die Eltern-Struktur, da sie von `QObject` ableiten. Wird ein

elternloses Widget erzeugt und angezeigt, wird es als neues Fenster dargestellt. `QWidget` selbst fungiert zunächst nur als Container für andere Widgets.
Das Grundkonzept jeder Qt-Anwendung besteht im Wesentlichen aus Ableitung. Wie bereits erläutert, müssen eigene Klassen geschrieben werden, um Signale und Slots zu definieren. Ebenso verfährt man bei der Erstellung von Benutzeroberflächen. Normalerweise leitet man ein Widget von der gewünschten Klasse ab, fügt innerhalb der Klasse die Kindelemente hinzu und schreibt benötigte Signale und Slots. Am Ende bleibt oft für die Main-Funktion nicht mehr zu tun, als eine `QApplication` und das geschriebene Widget zu erzeugen, das Widget anzuzeigen und die Eventschleife zu starten.

4.1 Layout

Sollen mehrere Elemente zu einem Widget hinzugefügt werden, müssen diese positioniert werden. Auch in Qt gibt es die Möglichkeit, die Elemente manuell anzuordnen. Dazu bietet `QWidget` folgende Methoden an, mit denen sich Position und Größe des Elementes bestimmen lassen:

```
void QWidget::setGeometry ( int x, int y, int w, int h )
void QWidget::setGeometry ( const QRect & )
```

Der Nachteil dieser Variante ist, dass die Möglichkeit fehlt, auf Größenänderungen des Fensters bzw. des Elternwidgets zu reagieren. Außerdem muss ggf. alles neu positioniert werden, wenn ein weiteres Element hinzukommt. Daher gibt es Layout-Manager. Diese steuern die Position und die Größe der Kindelemente und das Verhalten bei Größenänderung. Die Basisklasse eines jeden Layout-Managers in Qt ist die abstrakte Klasse `QLayout`. Sie ist von `QObject` und von `QLayoutItem` abgeleitet.

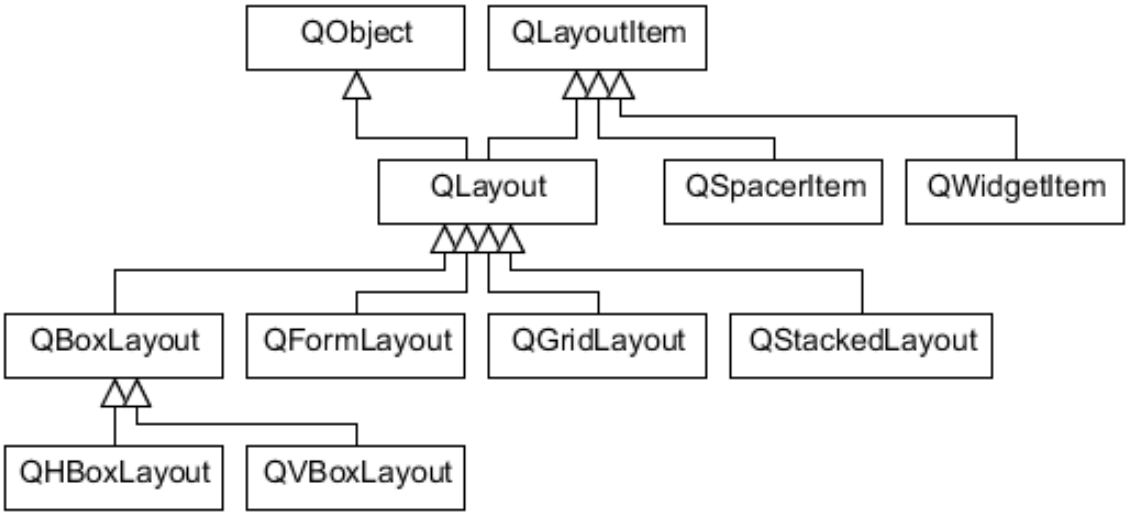

Abbildung 4.1 Klassenstruktur der Qt-Layouts

Soll einem Widget ein Layout zugeordnet werden, existieren zwei Möglichkeiten:
entweder

```
01 QVBoxLayout *layout = new QVBoxLayout(parent);
```

oder

```
01 parent->setLayout(layout);
```

Hier wurde exemplarisch `QVBoxLayout` verwendet. Analoges gilt für alle Layouts. Dabei ist `parent` der Zeiger auf das Widget, welches das Layout erhalten soll. In den meisten Fällen wird es `this` sein. `parent` übernimmt die Elternschaft von `layout`. Komponenten werden dann wie folgt hinzugefügt:

```
02 layout->addWidget(child);
```

Die Elternschaft von `child` wird jetzt jedoch nicht von `layout` übernommen, sondern von `parent`.
`QVBoxLayout` ordnet Komponenten untereinander an, `QHBoxLayout` nebeneinander und `QGridLayout` in einem Gitter. Bei `QStackedLayout` ist immer nur eine Komponente sichtbar, die den gesamten Platz einnimmt und `QFormLayout` ist speziell für Eingabeformulare gedacht. Im Folgenden werden vor allem die drei grundlegenden Layouts `QVBoxLayout`, `QHBoxLayout` und `QGridLayout` behandelt. Zunächst wird jedoch auf

einige allgemeine Funktionen eingegangen, die durch `QLayout` für alle Layouts definiert sind.

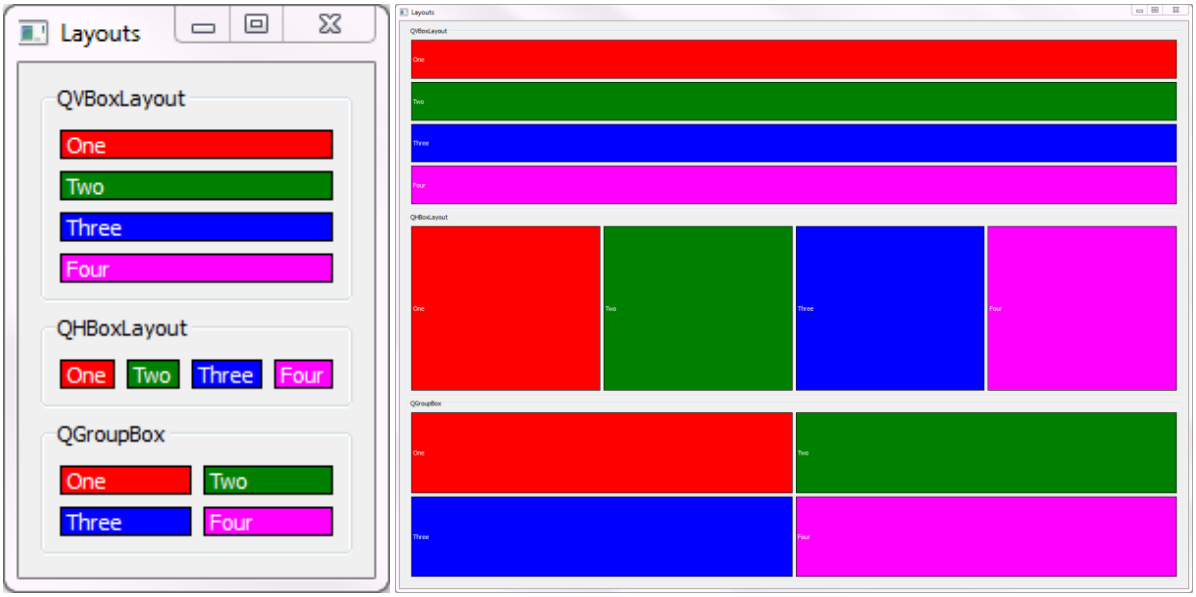

Abbildung 4.2 Verschiedene Layouts bei kleinem und bei großem Fenster

4.1.1 QLayout

Properties von Qt-Klassen besitzen immer einen Getter, der denselben Namen wie die Eigenschaft trägt und einen Setter. Dem Namen des Setters wird zusätzlich ein „`set`“ vorangestellt. Im Folgenden werden einige Properties vorgestellt, die von `QLayout` definiert sind und daher allen Layouts zur Verfügung stehen.

ContentsMargins

```
QMargins QLayout::contentsMargins () const
void QLayout::setContentsMargins ( int left, int top,
                                   int right, int bottom )
void QLayout::setContentsMargins ( const QMargins & margins )
```

Diese Eigenschaft legt den Abstand zwischen den Komponenten des Layouts und der übergeordneten Komponente fest. In Abbildung 4.3 wird der Zustand nach Aufruf von `setContentsMargins(20, 30, 40, 50)` illustriert. Links von den Komponenten sind 20 Pixel Platz, oben 30, rechts 40 und unten 50.

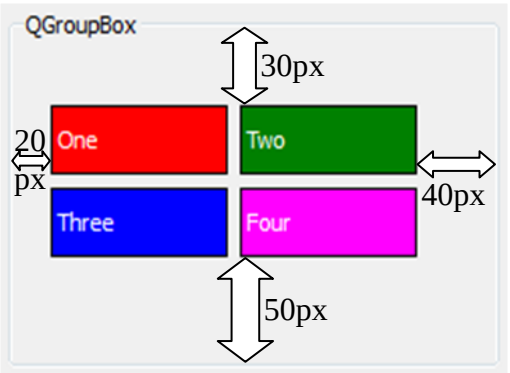

Abbildung 4.3 setContentMargins(20, 30, 40, 50)

Spacing

```
int QLayout::spacing () const
void QLayout::setSpacing ( int )
```

Mit der Property `spacing` lässt sich der Abstand zwischen den Komponenten des Layouts festlegen. Abbildung 4.4 demonstriert die Änderung des Komponentenabstandes mithilfe von `setSpacing(30)`.

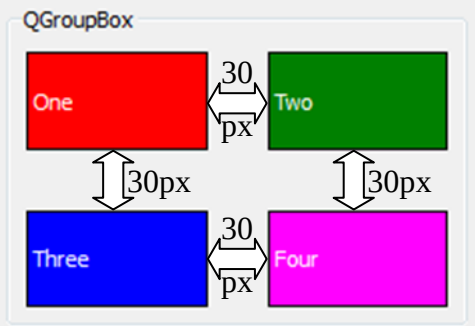

Abbildung 4.4 spacing(30)

4.1.2 QBoxLayout

`QVBoxLayout` und `QHBoxLayout` leiten beide von `QBoxLayout` ab und unterscheiden sich nur dadurch, dass `QVBoxLayout` die Komponenten vertikal und `QHBoxLayout` horizontal auf den verfügbaren Platz aufteilt. Die gesamte Funktionalität ist daher in der Basisklasse `QBoxLayout` definiert. Hier wird auf einige ihrer Funktionen eingegangen.

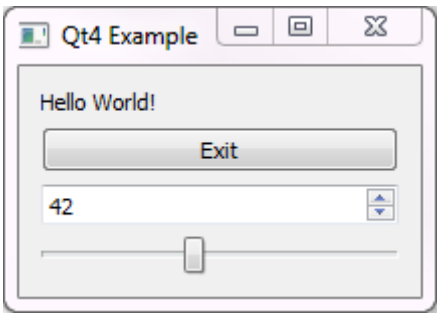

Abbildung 4.5 Anordnung mittels QVBoxLayout

Spacing

```
void QBoxLayout::addSpacing ( int size )
void QBoxLayout::insertSpacing ( int index, int size )
```

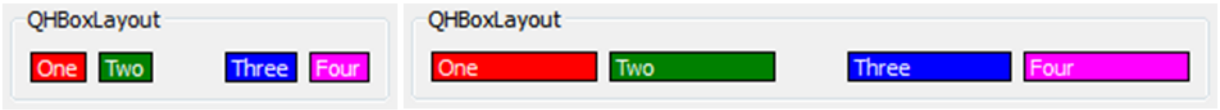

Abbildung 4.6 insertSpacing(2, 30)

Ähnlich wie Spacing bei `QLayout` fügen diese Methoden einen festen Abstand zwischen Komponenten ein. Allerdings wird hier kein Abstand für alle Komponenten eingefügt, sondern nur ein Abstand an einer bestimmten Stelle. Dabei fügt `addSpacing` den Leerraum hinter dem letzten Element ein und `insertSpacing` fügt ihn an einer beliebigen Stelle zwischen den Komponenten ein. Beim Einfügen muss beachtet werden, dass für jeden Abstand ein Kindelement erzeugt wird und dementsprechend ein Kindindex beansprucht wird. Möchte man also einen Abstand zwischen One und Two und zwischen Two und Three einfügen, funktioniert es nicht, `insertSpacing(1, 30); insertSpacing(2,`

`30);` aufzurufen, denn nach dem ersten Aufruf hat der erste Abstand den Index 1. Folglich fügt der zweite Aufruf den zweiten Abstand nach dem ersten ein, statt nach dem zweiten Label.

Stretch

```
void QBoxLayout::addStretch ( int stretch = 0 )
void QBoxLayout::insertStretch ( int index, int stretch = 0 )
```

Mit `addStretch` und `insertStretch` lassen sich Abstände einfügen, die den gesamten verfügbaren Platz einnehmen. Beim Einfügen muss, genau wie bei Spacing, beachtet werden, dass ein Kindelement für den Stretch angelegt wird und damit ein Kindindex benötigt wird. Werden mehrere Stretches eingefügt, wird der Platz entsprechend dem `stretch`-Argument gewichtet. Zum Beispiel bekommen zwei Abstände mit dem selben `stretch`-Argument gleich viel Platz und ein Abstand mit einem doppelt so hohen Wert doppelt so viel Platz. Dabei spielt der absolute Wert keine Rolle. Nur der relative Wert im Vergleich mit den anderen ist von Bedeutung.

Abbildung 4.7 insertStretch(2)

Es gibt jedoch nicht nur die Möglichkeit, Abstände einzufügen und zu gewichten, sondern auch die, den Platz der Komponenten zu gewichten.

```
void QBoxLayout::setStretch ( int index, int stretch )
bool QBoxLayout::setStretchFactor ( QWidget * widget, int stretch )
int QBoxLayout::stretch ( int index ) const
void QBoxLayout::addWidget ( QWidget * widget, int stretch = 0,
  Qt::Alignment alignment = 0 )
```

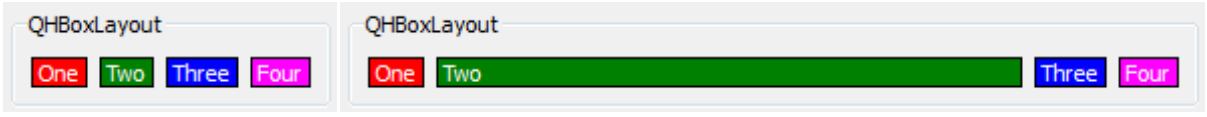

Abbildung 4.8 setStretch(1, 1)

Die Funktionsweise ist analog zu der der Abstände. Auch hier muss die Gewichtung nicht nachträglich eingefügt werden, sondern kann direkt bei `addWidget` mit übergeben werden.

Direction

```
void QBoxLayout::setDirection ( Direction direction )
Direction QBoxLayout::direction () const
```

Interessant ist die Möglichkeit, die Ausrichtung des Layouts zu ändern. Da diese Funktion von `QBoxLayout` zur Verfügung gestellt wird, ist sie auch in `QVBoxLayout` und `QHBoxLayout` verfügbar. Somit ist es z.B. möglich, mit einem `QHBoxLayout` die Komponenten untereinander anzuordnen. Die Elemente können aber auch von rechts nach links oder von unten nach oben angeordnet werden.

4.1.3 QGridLayout

`QGridLayout` ordnet die Komponenten in einem Gitter an. Aus diesem Grund ist die `addWidget`-Methode überladen, damit sie zusätzlich die Gitterkoordinaten des Elements aufnehmen kann:

```
void QGridLayout::addWidget ( QWidget * widget, int row, int column,
  Qt::Alignment alignment = 0 )
void QGridLayout::addWidget ( QWidget * widget,
  int fromRow, int fromColumn, int rowSpan, int columnSpan,
  Qt::Alignment alignment = 0 )
```

Dabei geben `row` und `column` jeweils die Zeile und Spalte an, in der das Element stehen soll. Die zweite Variante erlaubt zusätzlich eine Angabe, über wie viele Zeilen und Spalten sich das Element erstrecken soll. Es ist auch möglich, Plätze im Gitter unbenutzt zu lassen. Abbildung 4.9 demonstriert die Verwendung von spans und freien Gitterplätzen. Für das Element One wurde ein `rowSpan` von 1 und ein `columnSpan` von 2 benutzt und für das Element Four ein `rowSpan` von 3 und ein `columnSpan` von 1.

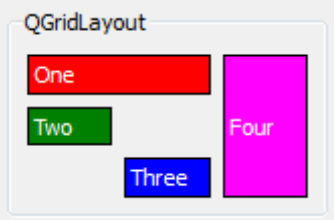

Abbildung 4.9 QGridLayout mit span

Count

```
int QGridLayout::rowCount () const
int QGridLayout::columnCount () const
```

Mit diesen beiden Methoden kann die aktuell benötigte Anzahl der Zeilen und Spalten abgefragt werden.

Minimum Size

```
int QGridLayout::rowMinimumHeight ( int row ) const
void QGridLayout::setRowMinimumHeight ( int row, int minSize )
int QGridLayout::columnMinimumWidth ( int column ) const
void QGridLayout::setColumnMinimumWidth ( int column, int minSize )
```

Diese Eigenschaften repräsentieren die Mindesthöhe bzw. –breite für eine bestimmte Zeile bzw. Spalte.

Stretch

```
int QGridLayout::rowStretch ( int row ) const
void QGridLayout::setRowStretch ( int row, int stretch )
int QGridLayout::columnStretch ( int column ) const
void QGridLayout::setColumnStretch ( int column, int stretch )
```

Mit diesen Methoden kann die `stretch`-Eigenschaft für eine bestimmte Zeile bzw. Spalte ausgelesen oder festgelegt werden. Die Eigenschaft funktioniert analog zu der in `QBoxLayout` (siehe Kapitel 4.1.2).

Spacing

```
int QGridLayout::verticalSpacing () const
void QGridLayout::setVerticalSpacing ( int spacing )
int QGridLayout::horizontalSpacing () const
void QGridLayout::setHorizontalSpacing ( int spacing )
```

Spacing legt fixe Abstände zwischen allen Zeilen bzw. Spalten fest. Die Funktionsweise ist analog zu spacing in QLayout (siehe Kapitel 4.1.1).

4.2 Widgets

Der Begriff „Widget“ ist eine Mischung der englischen Wörter Window und Gadget[1,2]. In Deutsch kann der Begriff „Steuerelement“ verwendet werden. Aber auch das Wort Widget ist im Deutschen gebräuchlich. Die Klasse `QWidget` ist die Basisklasse aller Steuerelemente in Qt. Auf alle oder sogar nur auf die wichtigen detailliert einzugehen, würde den Rahnem dieser Arbeit bei Weitem sprengen. Daher kann hier nur grob auf die gebräuchlichsten Widgets eingegangen werden. Eine detailliertere, aber immer noch gekürzte Übersicht ist mit

[1] http://dict.leo.org/?lp=ende&search=widget
[2] http://de.wikipedia.org/wiki/Steuerelement

Abbildung 4.10 gegeben. Die meisten Klassennamen sind selbsterklärend. Sollte ein Widget benötigt werden, kann leicht in der Qt-Dokumentation die Benutzung nachgelesen werden.

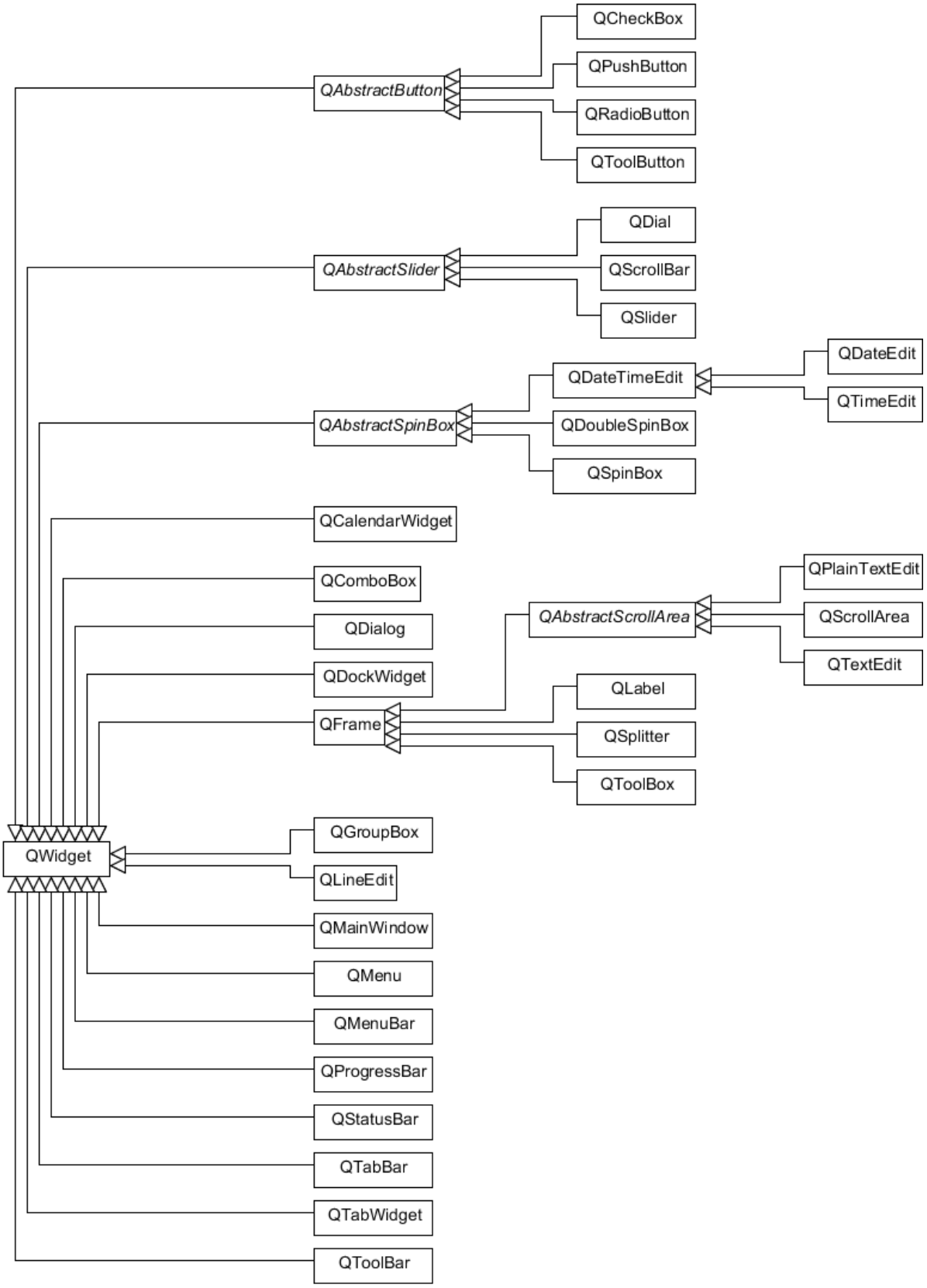

Abbildung 4.10 Übersicht der wichtigsten Widgetklassen in Qt

4.2.1 QLabel

Abbildung 4.11 QLabel

Zur Darstellung von Texten wird `QLabel` benutzt. Die Klasse hat aber mehr Funktionen als nur Texte auf den Bildschirm zu zeichnen. Es können mit HTML-Tags oder Rich Text formatierte Texte, Zahlen, Bilder und Animationen (z.B. GIF) dargestellt werden. Ein Label kann immer nur ausschließlich einen Inhalt haben. Ein Bild und einen Text beispielsweise gleichzeitig in einem Label darzustellen, ist nicht möglich. Der Text kann über die Property `text`, eine Zahl über `num`, ein Bild über `pixmap` und eine Animation über `movie` gesetzt werden. Der Inhalt kann durch `clear()` gelöscht werden.

4.2.2 QAbstractButton

`QAbstractButton` ist die abstrakte Basisklasse aller Widgets, die vergleichbare Funktionen wie Buttons zur Verfügung stellen. Sie beinhaltet z.B. den Beschriftungstext und ein Icon. Diese werden über die Properties `text` und `icon` gesetzt. Manche Buttons können durch Anklicken „aktiviert“ und „deaktiviert“ werden. Ob dies möglich ist, wird mit der Property `checkable` angezeigt. In Ermangelung eines allgemeinen deutschen Terminus für die englischen Wörter checked und checkable werden hier die Begriffe „aktiv“ und „aktivierbar“ verwendet. Ob ein Button aktiv ist, zeigt die Property `checked` an. Durch Veränderung von `checkable` wird es möglich, Togglebuttons zu erzeugen. Das sind Buttons, die nach dem Anklicken gedrückt bzw. aktiv bleiben und durch erneutes Klicken wieder deaktiviert werden. Es stehen u.a. folgende Slots zur Verfügung:

```
void QAbstractButton::click () [slot]
void QAbstractButton::setChecked ( bool ) [slot]
void QAbstractButton::toggle () [slot]
```

`click` simuliert den Klick eines Benutzers. Mit `setChecked` lässt sich bei aktivierbaren Buttons der Status festlegen und mit `toggle` der Status wechseln.

Zudem sind folgende Signals definiert:

```
void QAbstractButton::clicked ( bool checked = false ) [signal]
void QAbstractButton::pressed () [signal]
void QAbstractButton::released () [signal]
void QAbstractButton::toggled ( bool checked ) [signal]
```

`clicked` wird bei jeder Betätigung eines Buttons ausgelöst. Dabei ist es unerheblich, ob der Button vom Benutzer oder vom Programm gesteuert wurde. Wenn der Button aktivierbar ist, zeigt das angegebene Argument an, ob der Button aktiv ist. Entsprechend werden `pressed` und `released` aufgerufen, wenn der Button nach unten gedrückt bzw. losgelassen wurde.

`toggled` wird bei aktivierbaren Buttons immer aufgerufen, wenn der Status durch den Benutzer oder das Programm verändert wurde. Das Argument zeigt auch hier den Status an.

4.2.3 QPushButton

`QPushButton` stellt einen klassischen Button bereit. Zusätzlich zur Funktionalität von `QAbstractButton` bietet `QPushButton` die Eigenschaft `flat`, mit der der Button flach dargestellt werden kann. Dies empfiehlt sich allerdings nicht, da er optisch meist nicht von einem Text-Label zu unterscheiden ist und so vom Benutzer nicht als Button wahrgenommen wird.

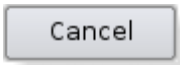

Abbildung 4.12 QPushButton

4.2.4 QCheckBox

`QCheckBox` implementiert eine Checkbox auf Basis von `QAbstractButton`. Die Klasse unterstützt zusätzlich einen Modus, in dem drei Zustände zugelassen sind. In der Benutzung bedeuten diese typischerweise: Ja, Nein, Unverändert lassen. Dieser Modus kann über die Property `tristate` aktiviert werden. Hier reicht die boolesche Eigenschaft `checked` nicht mehr aus. Daher ist zusätzlich die Eigenschaft `checkState` definiert. Diese repräsentiert keinen Wahrheitswert, sondern ein `enum`. Mögliche Werte sind `Qt::Unchecked`, `Qt::PartiallyChecked` und `Qt::Checked`.

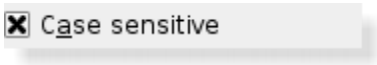

Abbildung 4.13 QCheckBox

4.2.5 QRadioButton

Radiobuttons bieten dem User normalerweise eine von mehreren Optionen zur Auswahl, wobei sich die Alternativen ausschließen. Ähnlich wie eine Checkbox, kann ein Radiobutton ausgewählt sein oder nicht. `QRadioButtons` sind standardmäßig `autoExclusive`. Das bedeutet, dass sich alle Radiobuttons, die dasselbe Eltern-Widget haben, gegenseitig ausschließen. Sind andere Gruppierungen gewünscht, müssen die Radiobuttons in `QButtonGroups` gruppiert werden. Wird ein `QRadioButton` ausgewählt, wird das Signal toggled ausgelöst.

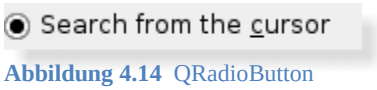

Abbildung 4.14 QRadioButton

4.2.6 QSlider

Da hier nur auf `QSlider` als einzige Subklasse von `QAbstractSlider` eingegangen wird, wird nicht differenziert, ob die beschriebene Funktionalität in der Sub- oder in der Basisklasse definiert ist.

Abbildung 4.15 QSlider

Mit einem Slider können `int`-Werte vom Benutzer eingestellt werden. Zur Festlegung des Wertebereichs werden die Properties `minimum` und `maximum` verwendet. Mit `setRange` können beide Werte gleichzeitig gesetzt werden:

```
void QAbstractSlider::setRange ( int min, int max )
```

Der aktuelle Wert des Sliders wird durch die Property `value` ausgedrückt. Der Setter `setValue` fungiert zusätzlich als Slot.

Wird der Wert des Sliders verändert, wird das Signal `valueChanged` aufgerufen:

```
void QAbstractSlider::valueChanged ( int value ) [signal]
```

Dieses Signal wird aufgerufen, sobald der Slider bewegt wird. Es kann durch Deaktivierung der Eigenschaft `tracking` in der Form geändert werden, dass `valueChanged` erst aufgerufen wird, wenn der Slider losgelassen wird. Die aktuelle Position des Sliders kann dann weiterhin über die Eigenschaft `sliderPosition` erfragt werden.

4.2.7 QAbstractSpinBox

Die Basisklasse für Spinner ist `QAbstractSpinBox`. Spinner sind Steuerelemente, die aus einem Textfeld und zwei kleinen Buttons bestehen. Dabei können die Werte in dem Textfeld sowohl mithilfe der Tastatur als auch mit den Buttons verändert werden können. Durch die Property `text` wird der Text repräsentiert, der im Textfeld des Spinners steht. Es gibt zwei Möglichkeiten, um auf die Verringerung des minimal zulässigen Wertes zu reagieren: Es kann der maximal zugelassene Wert angenommen oder die Verringerung ignoriert werden. Diese beiden Möglichkeiten können durch die Property wrapping eingestellt werden. Analog gilt dies für das Erreichen des Maximums. Mit der Property `accelerated` kann eingestellt werden, ob bei längerem Halten eines Buttons die Änderungsrate zunimmt.

Bei einem Klick auf einen der Buttons wird die Funktion `stepBy()` aufgerufen. Zum Inkrementieren wird `1` übergeben, zum Dekrementieren `-1`.

```
virtual void QAbstractSpinBox::stepBy ( int steps )
void QAbstractSpinBox::stepDown () [slot]
void QAbstractSpinBox::stepUp () [slot]
```

Das Argument `steps` in der Funktion `stepBy()` zeigt an, um wie viele Schritte der Wert erhöht werden soll. Zusätzlich stehen die beiden Slots `stepUp()` und `stepDown()` zur Verfügung, deren Aufruf analog zum Aufruf von `stepBy(1)` bzw. `stepBy(-1)` ist.

```
void QAbstractSpinBox::editingFinished () [signal]
```

Dieses Signal wird aufgerufen, wenn der Benutzer die Enter-Taste drückt oder der Spinner den Fokus verliert.

4.2.8 QSpinBox

Abbildung 4.16 QSpinBox

Die Klasse `QSpinBox` implementiert einen Spinner für `int`-Werte. Der Benutzer kann demnach Ganzzahlen einstellen. Genau wie `QSlider` bietet `QSpinBox` die Properties `minimum` und `maximum`. Des Weiteren stehen äquivalent die Methode `setRange()` zur Festlegung des Wertebereiches und die Property `value` für den aktuellen Wert zur Verfügung. Auch das Signal `valueChanged()` und der Slot `setValue()` sind äquivalent zu denen in `QSlider`. Mit der Property `singleStep` kann eingestellt werden, um welchen Betrag sich der Wert ändert, wenn der Benutzer einen der Pfeile betätigt. Standardmäßig ist dieser Wert auf `1` eingestellt. Mit den Properties `prefix` und `suffix` ist es möglich, in dem Textfeld vor bzw. hinter der Zahl noch einen Text darstellen zu lassen. So könnte man beispielsweise Einheiten angeben.

4.2.9 QDoubleSpinBox

Mit einer `QDoubleSpinBox` können Fließkommawerte eingestellt werden. Die Funktionsweise ist analog zur `QSpinBox`. Der einzige Unterschied besteht darin, die Typen für z.B. `value`, `minimum` und `maximum` keine `int`s sondern `double`s sind.

4.2.10 QLineEdit

Abbildung 4.17 QLineEdit

`QLineEdit` stellt ein einzeiliges Texteingabefeld zur Verfügung. Auf den eingegebenen Text kann über die Property `text` zugegriffen werden. Mit der Eigenschaft `placeholderText` kann ein String gesetzt werden, der angezeigt wird, solange kein Text im Eingabefeld steht und das Feld keinen Fokus hat. Mit der Property `echoMode` kann angegeben werden, wie der eingegebene Text dargestellt werden soll. So lässt sich ein Passworteingabefeld erzeugen. Möglich sind folgende Werte:

`QLineEdit::Normal` Der eingegebene Text wird angezeigt.

`QLineEdit::Password` Statt dem eingegebenen Text werden Sternchen (*) für jeden Buchstaben angezeigt.

`QLineEdit::NoEcho` Es wird gar nichts angezeigt. So kann auch die Länge eines Passworts verschleiert werden.

`QLineEdit::PasswordEchoOnEdit` Der Text wird während der Eingabe angezeigt, aber verborgen, sobald der Fokus wechselt.

5 Ereignisverarbeitung

Mit Ereignisverarbeitung ist nicht das Signal-Slot-Prinzip gemeint, obwohl man dieses ebenfalls als Ereignisverarbeitung auffassen kann. In diesem Kapitel geht es um die Verarbeitung von Ereignissen, die das Betriebssystem an das Qt-Programm sendet, wie z.B. Maus- oder Tastatureingaben. Es gibt aber auch Ereignisse, die von Qt generiert werden. Dazu benutzt Qt, wie jede GUI-Schnittstelle, eine Ereignisschleife. Sobald ein Ereignis auftritt, wird es in eine Warteschlange eingereiht. Die Ereignisschleife, auch Event-Loop genannt, prüft zyklisch in nicht wahrnehmbaren Abständen, ob Ereignisse vorliegen und leitet sie an das Ziel-Objekt weiter, wo diese dann verarbeitet werden. Die Ereignisschleife wird in Qt mit `QApplication::exec()` gestartet. Ereignisse werden durch die Subklassen von `QEvent` gekapselt.

Die Verarbeitung erfolgt, indem das `QEvent` an die Methode `event()` von QObject übergeben wird:

```
virtual bool QObject::event ( QEvent * e )
```

Der Rückgabewert dieser Methode zeigt an, ob das Event verarbeitet wurde. Um das Verhalten in Subklassen anpassen zu können, wird die Methode `virtual` bereitgestellt. `QWidget` beispielsweise überschreibt die Methode in der Weise, dass Events nicht direkt von `event()` verarbeitet werden, sondern, je nach Eventtyp, an spezialisierte Methoden weitergereicht werden. Hier sind exemplarisch zwei Methoden aufgeführt:

```
virtual void QWidget::keyPressEvent ( QKeyEvent * event )
virtual void QWidget::mousePressEvent ( QMouseEvent * event )
```

5.1 QEvent

Ein QEvent besitzt einen bestimmten Typen. Dieser Typ wird mit dem Konstruktor gesetzt und kann über `type()` ausgelesen werden. Der Wert von `type()` ist ein enum vom Typ `QEvent::Type`. Mögliche Werte sind z.B. `QEvent::MouseButtonPress` oder `QEvent::KeyRelease`. Es gibt über hundert definierte Werte und zusätzlich sind eigene Werte möglich, sodass hier nicht alle beschrieben werden können.

Accepted

```
bool QEvent::isAccepted () const
void QEvent::setAccepted ( bool accepted )
void QEvent::accept ()
void QEvent::ignore ()
```

Dieses Flag zeigt an, ob das Empfängerobjekt das Event ignoriert oder es bearbeiten kann. Mit `accept()` kann das Flag zusätzlich auf `true` und mit `ignore()` auf `false` gesetzt werden. Wird das Event ignoriert, versucht Qt, es einem Elternelement zuzustellen.

5.2 Ereignishandler überschreiben

Wie bereits erläutert, werden Events an `QObject::event` zugestellt und bei Widgets an spezialisierte Handlermethoden weitergereicht. Für die Implementierung eigener Eventhandler muss von `QObject` geerbt und `event()` überschreiben werden. Möchte man ein selbst definiertes oder angepasstes Steuerelement schreiben, muss man `QWidget` ableiten. Es empfiehlt sich dann nicht `event()` zu überschreiben, sondern je nach Verwendungszweck eine der spezialisierten Methoden.
Zur Demonstration dieses Verfahrens wird hier als Beispiel ein Label implementiert, das seinen Inhalt löscht, wenn die Taste Entfernen gedrückt wird.

`mylabel.h`:

```
01 #ifndef MYLABEL_H
02 #define MYLABEL_H
03
04 #include <QLabel>
05
06 class MyLabel : public QLabel
07 {
08 public:
09     explicit MyLabel(QWidget * parent = 0, Qt::WindowFlags f = 0);
10     explicit MyLabel(const QString & text, QWidget * parent = 0,
   Qt::WindowFlags f = 0);
11
12 protected:
13     virtual void keyPressEvent(QKeyEvent *ev);
14 };
15
16 #endif // MYLABEL_H
```

mylabel.cpp:

```
01 #include "mylabel.h"
02 #include <QKeyEvent>
03
04 MyLabel::MyLabel(QWidget *parent, Qt::WindowFlags f)
05     : QLabel(parent, f)
06 {
07 }
08
09 MyLabel::MyLabel(const QString &text, QWidget *parent,
   Qt::WindowFlags f)
10     : QLabel(text, parent, f)
11 {
12 }
13
14 void MyLabel::keyPressEvent(QKeyEvent * event)
15 {
16     if (event->key() == Qt::Key_Delete)
17         clear();
18     else
19         QLabel::keyPressEvent(event);
20 }
```

Um die Verwendung der Klasse zu testen, wird noch eine `main.cpp` implementiert:

```
01 #include <QApplication>
02 #include "mylabel.h"
03
04 int main(int argc, char *argv[])
05 {
06     QApplication app(argc, argv);
07
08     MyLabel myLabel("Some Text");
09     myLabel.show();
10
11     return app.exec();
12 }
```

Die eigentliche Reimplementierung des Eventhandlers findet in mylabel.cpp in Zeile 14ff. statt. Ist die gedrückte Taste die Entfernen-Taste, wird mit `clear()` der Inhalt des Labels gelöscht. Ansonsten wird die Implementierung der Superklasse aufgerufen. Das `accepted`-Flag des Events ist dabei standardmäßig auf `true` gesetzt, sodass dann dieses nicht verändert werden muss, wenn Entfernen gedrückt wird. Falls eine andere Taste gedrückt wird, behandelt die Super-Implementierung das korrekte Setzen des Flags.

6 Das Ressourcensystem

Möchte man Dateien wie Bilder in ein Programm einfügen, kann Qt diese von einem Pfad einlesen. Das Problem ist, dass relative Pfadangaben nicht benutzt werden können, da je nach

Ausführungsumgebung und Aufruf das Arbeitsverzeichnis unterschiedlich ist. Qt bietet zwar die Möglichkeit, das Anwendungsverzeichnis auszulesen, jedoch müssen auf diese Weise Strings für den Dateipfad umständlich zusammengesetzt werden. Außerdem müssen bei dieser Variante zusätzlich zu der binären Datei und ggf. den Qt-Bibliotheken, auch die Ordner, in denen die Bilder liegen, zur Verfügung gestellt werden.

Qt 4 bietet hier mit dem Ressourcen-System eine elegante Alternative an. Zunächst muss eine Ressourcen-Datei zum Projekt hinzugefügt werden. Dies ist eine XML-Datei, in der unter einem sogenannten `prefix` mehrere `file`s mit je einem `alias` angelegt werden können. Die Dateinamen werden hier relativ zum Projektverzeichnis angegeben. Da der Qt Creator einen grafischen Editor für Ressourcen-Dateien mitbringt, wird hier nicht genauer auf das Format eingegangen. Eingetragene Dateien werden beim Erstellen mit in die ausführbare Datei kompiliert. So müssen diese Dateien nicht mehr zur Verfügung gestellt werden.

Der Zugriff auf Ressourcen-Dateien kann nun an jeder Stelle des Programms erfolgen, in der sonst auf eine Datei zugegriffen wird. Dazu wird statt dem Pfad der Ausdruck „`:/`“ verwendet. Diesem wird der Präfix und Alias oder der Dateiname, getrennt durch einen Slash, angehängt. Angenommen, im Projektverzeichnis befindet sich ein Ordner `images` mit einem Bild `cancel.png` und in der Ressourcen-Datei wurde diese Datei unter dem Präfix `img` mit dem Alias `cancel` eingetragen. Dann kann diese zur Erzeugung eines Icons so verwendet werden:

```
01 QIcon(":/img/cancel")
```

Es ist auch möglich, in der Ressourcen-Datei eine Sprache für eine Datei mit anzugeben. Bei einer Übersetzung wird dann automatisch diejenige Datei ausgewählt, die zu der eingestellten Sprache passt.

7 Internationalisierung

Als Basis für eine übersetzbare Anwendung mit Qt sollten alle zu übersetzenden Strings mit der statischen Methode `QObjetct::tr()` gekapselt werden.

```
static QString QObject::tr ( const char * sourceText, const char *
  disambiguation = 0, int n = -1 )
```

Im Beispiel aus Kapitel 2.4 müsste also z.B.

```
20 QLabel *label = new QLabel("Hello World!");
```

durch

```
20 QLabel *label = new QLabel(QObject::tr("Hello World!"));
```

ersetzt werden. Innerhalb von Subklassen von `QObject` sollte der Klassenname vor `tr()` weggelassen werden, da die Klasse, auf der `tr()` aufgerufen wird, später als Kontext dient. In dem Beispiel gibt es jedoch keine andere sinnvolle Klasse, auf der die Methode aufgerufen werden könnte. Ist eine Übersetzung eingestellt, übernimmt `tr()` die Übersetzung des übergebenen Strings. Ist keine Übersetzung eingestellt oder wurde kein übersetzter String gefunden, wird der übergebene String zurückgegeben. Mit dem optionalen Parameter `disambiguation` kann eine Erklärung oder ein Kontext für den Übersetzer angegeben werden, da dasselbe Wort in unterschiedlichen Kontexten unterschiedlich übersetzt werden kann. Mit dem dritten optionalen Parameter `n` kann, falls ein Satz mit einer Zahl dargestellt werden soll, diese Zahl übergeben werden. Qt versucht dann, die Zahl grammatikalisch korrekt in den Satz einzubauen.
Um eine Übersetzung erstellen zu können, muss in der Projektdatei (.pro) noch ein Eintrag hinzugefügt werden. Das könnte z.B. so aussehen:

```
TRANSLATIONS = MyApp_de.ts \
    MyApp_it.ts \
    MyApp_es.ts
```

Als nächstes muss das Tool `lupdate` ausgeführt werden. Dies kann im Qt Creator über Extras −> Extern −> Linguist −> lupdate geschehen. Das Tool extrahiert alle Strings, die mit `tr()` verwendet werden und erzeugt daraus die zuvor in der Projektdatei eingetragenen Übersetzungsdateien. Dies sind XML-Dateien, die nun mit dem Qt Linguist bearbeitet werden können. So kann eine Übersetzung auch ohne Programmierkenntnisse, z.B. von professionellen Übersetzern erstellt werden.
Möchte man eine Übersetzung verwenden, muss das Tool `lrelease` verwendet werden, das aus den Übersetzungsdateien (.ts) binäre Dateien (.qm) erzeugt. Das Programm kann ebenfalls mithilfe des Qt Creators über Extras −> Extern −> Linguist −> lrelease gestartet werden.
Um die Übersetzung zu verwenden, wird folgender Code in der Regel als erstes in der Main-Funktion ausgeführt:

```
01      QApplication app(argc, argv);
02      QTranslator trans;
03      trans.load(":/MyApp_de")
04      app.installTranslator(&trans);
```

Dabei kann `load()` ein Argument mit dem Namen der Datei und ein zweites mit dem Pfad annehmen. Hier wird die Datei allerdings über das Ressourcen-System geladen (siehe Kapitel 6), sodass keine Pfadangabe nötig ist. In der Praxis bietet es sich jedoch an, den Namen der Übersetzungsdatei nicht fest in den Quelltext zu codieren, sondern z.B. mithilfe von Benutzereinstellungen oder der Systemsprache zu erzeugen.

8 Fazit

Ziel dieser Arbeit war es, einen Einblick in die GUI-Programmierung mit Qt 4 zu geben. Im Zuge der Einarbeitung in dieses Thema stellte sich heraus, dass Qt, verglichen mit reinem C++ oder anderen Bibliotheken wie boost, sehr angenehm zu benutzen ist, sodass die Einarbeitung sehr leicht war und wenig Zeit beanspruchte. Auch der Qt Creator bietet verglichen mit z.B. dem Visual Studio 2008 von Microsoft viele und intuitive Funktionen. Nachteile von Qt sind, dass eine relativ große Bibliothek mitgeliefert werden muss und dass ein zusätzliches Tool zur Kompilierung benötigt wird.

Durch den begrenzten Umfang dieser Arbeit war es nur möglich, einige Themen grob anzureißen. Qt ist ein sehr großes und mächtiges Framework, über das ganze Bücher geschrieben werden. Dennoch wurde versucht, auf die Grundlagen der GUI-Programmierung mit Qt ausreichend einzugehen und einen Ausblick auf einige fortgeschrittene Funktionalitäten wie das Ressourcen- und Übersetzungssystem zu geben.

9 Literaturverzeichnis

Blanchette, J., & Summerfield, M. (2008). *C++ GUI Programmierung mit Qt 4.* München: Addison-Wesley.

Category:Software that uses Qt. (8. September 2011). Von Wikipedia: http://en.wikipedia.org/wiki/Category:Software_that_uses_Qt abgerufen

Nokia Corporation. (2011). *Qt Reference Documentation*. Von http://doc.qt.nokia.com/4.7/ abgerufen

Qt (Bibliothek). (2. Dezember 2011). Von Wikipedia: http://de.wikipedia.org/wiki/Qt_%28Bibliothek%29 abgerufen

Qt (framework). (27. November 2011). Von Wikipedia: http://en.wikipedia.org/wiki/Qt_%28framework%29 abgerufen

Wolf, J. (2007). *Qt 4 GUI-Entwicklung mit C++*. Bonn: Galileo Press.

Abbildungsnachweis

Abbildung 3.1: http://doc.qt.nokia.com/4.7/images/plastique-spinbox.png (2011)

Abbildung 3.2: http://doc.qt.nokia.com/4.7/images/plastique-slider.png (2011)

Abbildung 4.11: http://doc.qt.nokia.com/4.7/images/plastique-label.png (2011)

Abbildung 4.12: http://doc.qt.nokia.com/4.7/images/plastique-pushbutton.png (2011)

Abbildung 4.13: http://doc.qt.nokia.com/4.7/images/plastique-checkbox.png (2011)

Abbildung 4.14: http://doc.qt.nokia.com/4.7/images/plastique-radiobutton.png (2011)

Abbildung 4.15: http://doc.qt.nokia.com/4.7/images/plastique-slider.png (2011)

Abbildung 4.16: http://doc.qt.nokia.com/4.7/images/plastique-spinbox.png (2011)

Abbildung 4.17: http://doc.qt.nokia.com/4.7/images/plastique-lineedit.png (2001)