II.

Teil II:
Bausteine komplexer Anwendungen

14

Grafische Benutzungsschnittstellen

Dieses Kapitel behandelt die folgenden Themen:

- Ereignisgesteuerte Programmierung
- Meta-Objektsystem
- Signale, Slots und Widgets
- Dialoge

Standard C++ kennt keine Elemente für grafische Benutzungsoberflächen (englisch *graphical user interfaces*) (GUI). Nichtsdestoweniger sind GUIs nicht mehr wegzudenken, weswegen hier eine Einführung in die GUI-Programmierung gegeben wird. Die entsprechenden Komponenten finden sich in verschiedenen Bibliotheken. Sehr bekannt sind die Microsoft Foundation Classes (MFC) bzw. ihre Nachfolger in .NET für Windows-Betriebssysteme. Die in Abschnitt 1.5 empfohlene Entwicklungsumgebung Code::Blocks kann unter anderem auf Basis der Windows-API Programme mit einer grafischen Benutzungsschnittstelle erzeugen. Der Nachteil der genannten Möglichkeiten ist die mangelnde Portabilität.

GTK+ (= the GIMP Toolkit) ist eine weitere Bibliothek zur Erstellung grafischer Benutzungsoberflächen. GTK+ wurde ursprünglich für GIMP, ein mächtiges Bildbearbeitungsprogramm, entwickelt. GTK+ wird zur Entwicklung des GNOME-Desktops benutzt. GNOME ist eine Benutzungsoberfläche, die auf vielen Linux-Systemen zu finden ist. GTK+ gibt es für verschiedene Betriebssysteme und ist nicht lizenzpflichtig, auch dann

nicht, wenn kommerzielle Software damit entwickelt wird. Die Homepage von GTK+ ist *http://www.gtk.org*.

Ebenfalls sehr bekannt ist die portable Qt-Bibliothek für Windows-, Mac- und Unix-Betriebssysteme. Die in manchen Linux-Systemen vorhandene Benutzungsoberfläche KDE wird mit Qt entwickelt. Im Gegensatz zu GTK+ gibt es eine sehr ausführliche Dokumentation. Qt zeichnet sich durch weitere hervorstechende Merkmale aus:

- Das Programm Qt Assistant (Aufruf `assistant`) ist ein umfangreiches Hilfesystem für alle Qt-Programme.
- Das Programm Qt Designer (Aufruf `designer`) ist ein Werkzeug, das es ermöglicht, mithilfe von Mausoperationen eine grafische Benutzungsoberfläche am Bildschirm zu erzeugen und zu konfigurieren.
- Das Programm Qt Linguist unterstützt die Anpassung von Anwendungen, die international nutzbar sein sollen, an verschiedene Sprachen.
- Das Programm Qt Creator ist eine IDE speziell für Qt-Anwendungen.
- Ab Version 4.5 ist Qt nicht nur mit einer kommerziellen Lizenz, sondern auch als Open Source Software erhältlich – der Grund, warum Sie sie mitsamt den Lizenzen auf der DVD finden. Eine vergleichende Übersicht der Lizenzen finden Sie unter *http://www.qt.nokia.com/products/licensing*.
- Es gibt Unterstützung für die Arbeit mit Verzeichnissen, ein Hilfesystem, die Verarbeitung von XML und die Netzwerkprogrammierung. Eine integrierte Datenbank ist ebenfalls vorhanden.

Zusammengefasst: Qt ist die ausgereifteste und umfangreichste Open Source Software für die portable Entwicklung grafischer Benutzungsoberflächen mit C++. Java-Kenner finden in Qt all das, was sie an Standard C++ vermissen. Aus diesem Grund wird in diesem Kapitel ein kleiner Einblick in Qt gegeben.

14.1 Ereignisgesteuerte Programmierung

Die bislang in diesem Buch dargestellte Programmierung beruht darauf, dass die Abarbeitung der Programmschritte wie vorgesehen der Reihe nach abläuft. Das gilt auch innerhalb der Threads des Kapitels 13 – nicht aber für die Programmierung grafischer Benutzungsoberflächen. Der Grund ist einfach: Die Reihenfolge der Interaktionsschritte eines Benutzers ist nicht vorhersehbar und damit auch nicht die Reihenfolge der auszuführenden Programmschritte. Die Aktion eines Benutzers, wie zum Beispiel Anklicken eines Menüpunktes oder Bewegung mit der Maus, wird als *Ereignis* aufgefasst, auf das das Programm reagieren soll. Das Ereignis wird vom Betriebssystem erkannt und an das Programm, wenn es sich dafür angemeldet (registriert) hat, weitergeleitet.

Es geht also darum, Funktionen zu schreiben, die nicht von anderer Stelle desselben Programms, sondern bei Eintreffen eines bestimmten Ereignisses aufgerufen werden und

die gewünschte Reaktion liefern. Insofern besteht eine Ähnlichkeit mit den auf Seite 225 erwähnten Callback-Funktionen. Diese Art der Programmierung ist typisch für alle *Frameworks*. Ein Framework (dt. etwa Rahmenwerk) ist eine Software mit bestimmten Funktionen, die die Struktur einer Anwendung vorgibt. Die Anwendung kann diese Funktionen nutzen, indem sie konkrete Implementierungen bereitstellt, die beim Framework registriert und von ihm aufgerufen werden. Der Kontrollfluss wird also wesentlich durch das Framework bestimmt.

Etwas konkreter: Wenn bekannt ist, was bei Anklicken eines Menüpunktes zu tun ist, kann eine entsprechende Funktion geschrieben werden. Dies reicht jedoch nicht aus: Dem Framework muss mitgeteilt werden, dass diese Funktion mit dem Ereignis »Anklicken des Menüpunktes« verbunden werden soll. Wenn das geschehen ist, wird das Framework die Funktion aufrufen, sobald der Menüpunkt angeklickt wird. Wie das im Einzelnen aussieht, sehen Sie weiter unten am Beispiel. Die skizzierte Verfahrensweise gilt für alle Systeme zur GUI-Programmierung, nicht nur für Qt.

14.2 GUI-Programmierung mit Qt

Hinweis: Dieses Kapitel gibt eine erste kurze Einführung in die GUI-Programmierung mit Qt und ist eher für Fortgeschrittene gedacht. Der Schwerpunkt liegt nicht auf den vielfältigen Möglichkeiten der umfangreichen Bibliothek, sondern auf den grundlegenden Konzepten: dem Programmiermodell (Meta-Objekt-Compiler, Signale und Slots) und der Speicherverwaltung. Wer sich näher mit Qt beschäftigen möchte, dem sei das Buch [BlSu] empfohlen. Dieses Kapitel bietet eine gute Grundlage für das Verständnis.

Installationshinweise finden Sie im Anhang ab Seite 938 (Windows) bzw. 946 (Linux).

Hilfe zu Qt

Es gibt viele ergiebige Informationsquellen. An erster Stelle ist das laut Nokia »einzige offizielle« Buch [BlSu] zu nennen. Als Hilfe während der Programmierung ist das Programm *assistant* unerlässlich, das unter anderem die ausführliche Dokumentation aller Klassen bietet. Bei weitergehenden Fragen können unter anderem die folgenden Webseiten herangezogen werden:

http://www.qt.nokia.com/developer,
http://www.qtforum.de/ und
http://www.qtcentre.org/

14.2.1 Meta-Objektsystem

Qt erweitert C++ nicht nur um GUI-Elemente, sondern auch um ein sprachliches Konzept, das Meta-Objektsystem genannt wird. Seine wichtigsten Bestandteile:

- **Introspektion**: Dieser Mechanismus erlaubt es, *zur Laufzeit* Informationen über andere Objekte zu bekommen, zum Beispiel den Namen der Klasse zu erfragen und die zur

Verfügung gestellten Methoden zu ermitteln. Standard-C++ bietet diese Möglichkeit nicht. Deswegen wurde das Qt-Werkzeug *moc* (Meta-Objekt-Compiler) entwickelt. Es analysiert Qt-Quellprogramme und erzeugt C++-Quellcode, der die benötigten Funktionen bereitstellt und bei der Erzeugung eines ausführbaren Qt-Programms übersetzt und eingebunden wird. Mit Hilfe der Introspektion können die Signale und Slots (siehe unten) ermittelt werden.

- **Signale**: Ein Signal entspricht einem Ereignis im obigen Sinn und ist nicht mit dem Begriff »signal« der Unix-Programmierung zu verwechseln. Ein Signal ist einem Sender zugeordnet, zum Beispiel einem Button oder einem anderen GUI-Element.
- **Slots**: Ein Slot ist eine Funktion, die dem Empfänger eines Signals zugeordnet ist und auf das Signal reagieren soll.
- **Verbindung von Signal und Slot**: Die `connect()`-Anweisung verbindet Signale mit Slots:

```
connect(sender,    // Zeiger auf Sender-Objekt
        SIGNAL(    // Makro
          signal), // einem Signal zugeordneter Funktionsname
        empfaenger, // Zeiger auf Empfänger-Objekt
        SLOT(      // Makro
          slot));  // Name der Funktion, die reagieren soll
```

Die Funktionsnamen sind mit runden Klammern `()`, aber ohne Parameter anzugeben. Mehrere `connect()`-Anweisungen können ein Signal mit mehreren Funktionen verbinden, es ist aber auch möglich, mehrere Signale mit derselben Funktion zu verbinden oder auch Signale an andere Signale zu koppeln.

Konkrete Anwendungen finden Sie weiter unten. `SIGNAl` und `SLOT` sind nur zwei der verwerndeten Makros. Makros sind das Mittel, auf den ersten Blick merkwürdige anmutende syntaktische Qt-Konstruktionen in C++ zu verwandeln. Die Makros werden durch den Präprozessor abgearbeitet, sodass der Compiler sie nicht sieht.

14.2.2 Der Programmablauf

Für jede Qt-Anwendung mit einem GUI existiert ein `QApplication`-Objekt. Es dient zur Initialisierung und sorgt für die Ereignisverarbeitung. Letztere wird durch Aufruf der Methode `exec()` angestoßen, die den sogenannten Event Loop enthält. Darunter ist zu verstehen, dass in einer Endlosschleife auf ein Ereignis gewartet wird. Wenn eins eintrifft, wird es verarbeitet und auf das nächste Ereignis gewartet. Die Schleife bricht erst ab, wenn das letzte grafische Element geschlossen ist. Sehen Sie sich das folgende kleine Beispiel an:

Listing 14.1: Das erste Qt-Programm

```
// cppbuch/k14/label/main.cpp
#include <QApplication>
#include <QLabel>
#include<iostream>

class MeinLabel : public QLabel{
public:
  MeinLabel(const char* text)
```

```
      : QLabel(text) {
   }
   ~MeinLabel() {
      std::cout << "Destruktor ~MeinLabel gerufen!" << std::endl;
   }
};

int main(int argc, char *argv[]) {
   QApplication app(argc, argv);
   MeinLabel* mlabel = new MeinLabel("Erstes Qt-Programm");
   mlabel->show();
   int ergebnis = app.exec();
   delete mlabel;
   return ergebnis;
}
```

Sinn der Klasse `MeinLabel` ist, über die Funktionalität der Klasse `QLabel` hinaus den Destruktoraufruf zu protokollieren. Der Start des Programms erzeugt ein Label als Mini-Fenster auf dem Bildschirm (Abbildung 14.1).

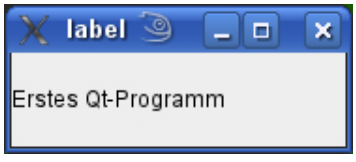

Abbildung 14.1: Label

Zuerst wird das `QApplication`-Objekt angelegt. Sie sehen, dass es einige Parameter von `main()` übernimmt. Dies ist für Qt-interne Zwecke notwendig, sodass `main()` diese Parameter stets aufführen muss. Nach der Konstruktion des `MeinLabel`-Objekts auf dem Heap und seiner Anzeige wird `exec()` ausgeführt. `exec()` kehrt erst dann zurück, wenn das Label durch Anklicken des Kreuzes in der rechten oberen Ecke geschlossen wird. Wenn alles gutgegangen ist, wird 0 zurückgegeben.

Hinweis für Windows-Nutzer

Um Ausgaben für Testzwecke mit `cout` oder `cerr` auf der Konsole zu sehen, müssen Sie in die *.pro*-Datei, die mit `qmake -project` erzeugt wird, die Zeile `CONFIG += console` einfügen. Danach wieder `qmake` und `make` aufrufen.

14.2.3 Speicher sparen und lokal Daten sichern

Qt führt mithilfe einer Baumstruktur Buch über die auf dem Heap angelegten Objekte. Wenn einem Konstruktor ein Zeiger auf ein Eltern-Objekt mitgegeben wird, wird das Objekt in die Liste der Kind-Objekte, die das Eltern-Objekt führt, eingetragen. Der Destruktor des Eltern-Objekts sorgt dafür, dass alle Kind-Objekte gelöscht werden. Damit muss `delete` nur noch für Heap-Objekte aufgerufen werden, die kein Elternobjekt haben.

Tipp
Qt-Objekte nicht auf dem Stack, sondern mit `new` anlegen!

Außer der beschriebenen automatischen Löschung gibt es weitere Gründe dafür, die in der Qt-Dokumentation unter der Überschrift »Qt Object Model«, Abschnitt »Qt Objects: Identity vs Value« beschrieben werden.

Ein `QMainWindow` ist das Hauptfenster einer Anwendung. Um es zu verwenden, wird eine eigene Klasse davon abgeleitet. Das folgende Beispiel zeigt, dass `delete` nicht aufgerufen werden muss, wenn das Attribut `WA_DeleteOnClose` gesetzt ist. Es können mehrere Fenster geöffnet sein. Um Speicher zu sparen und möglicherweise noch Daten zu sichern, bewirkt das Attribut den Aufruf des Destruktors und die Löschung des Objekts, wenn es geschlossen wird. Eine zweite Möglichkeit ist das Überschreiben der geerbten `protected`-Methode `closeEvent()`, die automatisch aufgerufen wird, wenn das Fenster geschlossen wird. Beide Möglichkeiten sind dargestellt, obwohl eine davon in der Praxis ausreicht. `#include<QtGui>` schließt *alle* GUI-Elemente ein. Bei großen Projekten sollten nur die tatsächlich benötigten Dateien inkludiert werden, um Compilationszeit zu sparen.

Listing 14.2: Einfaches Window

```
// cppbuch/k14/window/Window.h
#ifndef WINDOW_H
#define WINDOW_H
#include<QtGui>
#include<fstream>

class Window : public QMainWindow {
public:
   Window() {
      setAttribute(Qt::WA_DeleteOnClose);
   }
   ~Window() {
      // save(); Alternative zu closeEvent(), ohne Benutzerinteraktion
   }
protected:
   void closeEvent(QCloseEvent* ce) {
      int antwort = QMessageBox::warning(this, "Titel",
                                "closeEvent gerufen. Sichern?",
                                QMessageBox::Yes|QMessageBox::No);
      if(antwort == QMessageBox::Yes) {
         save();
      }
   }
private:
   void save() {
      std::ofstream log("daten.txt");
      log << "gesicherte Daten" << std::endl;
      log.close();
   }
};
```

Listing 14.3: `delete` ist nicht erforderlich.

```
// cppbuch/k14/window/main.cpp
include <QApplication>
#include"Window.h"

int main(int argc, char *argv[]) {
   QApplication app(argc, argv);
   Window* win1 = new Window();
   win1->show();
   Window* win2 = new Window();
   win2->show();
   return app.exec();
}
```

Im Gegensatz zu Aufräumarbeiten auf der Ebene des `main`-Programms können so pro Fenster abschließende Aktionen wie zum Beispiel eine Datensicherung durchgeführt werden. Eine Benutzerinteraktion wie in `closeEvent()` ist im Destruktor nicht möglich.

14.3 Signale, Slots und Widgets

Die oben kurz beschriebenen Signale und Slots werden in diesem Abschnitt in einem konkreten Beispiel eingesetzt, um das Zusammenwirken mit verschiedenen GUI-Elementen zu demonstrieren. Die GUI-Elemente werden *Widgets* genannt. Die Abbildung 14.2 zeigt ein Fenster des Typs `QMainWindow` mit verschiedenen Widgets. Im Wesentlichen geht es um das Zeichnen eines Kreises, dessen Radius mit der Maus (Widget `QDial`) oder durch Wahl einer Zahl (Widget `QSpinBox`) verändert werden kann.

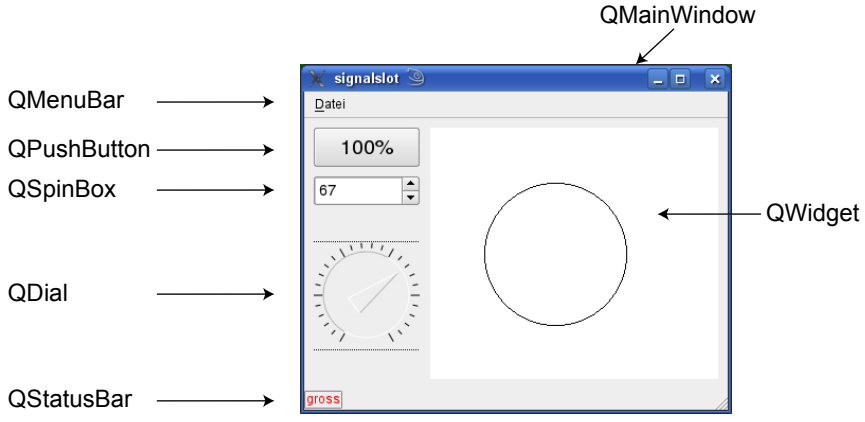

Abbildung 14.2: Qt-Beispiel

Die Elemente sind:

- Ein Menübalken des Typs `QMenuBar`, dem ein Datei-Menü »Datei« zugeordnet ist. Mit der Tastenkombination Alt+D (Hotkey) oder durch Mausklick kommt man zu den Einträgen des Menüs. In diesem Fall ist es nur der im Bild nicht sichtbare Eintrag »Ende«, der zum Beenden des Programms führt.
- Eine Statuszeile, die anzeigt, ob der Kreis groß oder klein ist. Falls der genannte Menüeintrag »Ende« aktiviert wird, zeigt die Statuszeile den erläuternden Text »Programm beenden«.
- Mehrere Widgets mit verschiedenen Aufgaben. `QWidget` ist die Basisklasse aller GUI-Objekte. Außer dem Hauptfenster gehören alle Widgets zu einem Eltern-Widget. Sie sind Elemente, die selbst angezeigt werden, oder aber Container für weitere Widgets. Das in der Abbildung 14.2 eingezeichnete `QWidget` bezieht sich nur auf die Zeichenfläche, das umhüllende Widget `QtBeispiel` ist nicht direkt sichtbar. Die Struktur ist am besten im UML-Diagramm unten (Abbildung 14.3) zu erkennen. Die Klasse `QtBeispiel` dient als Container für die nachfolgend aufgeführten Widgets. Das UML-Diagramm zeigt vereinfachend nicht, dass alle Widgets von `QWidget` erben.

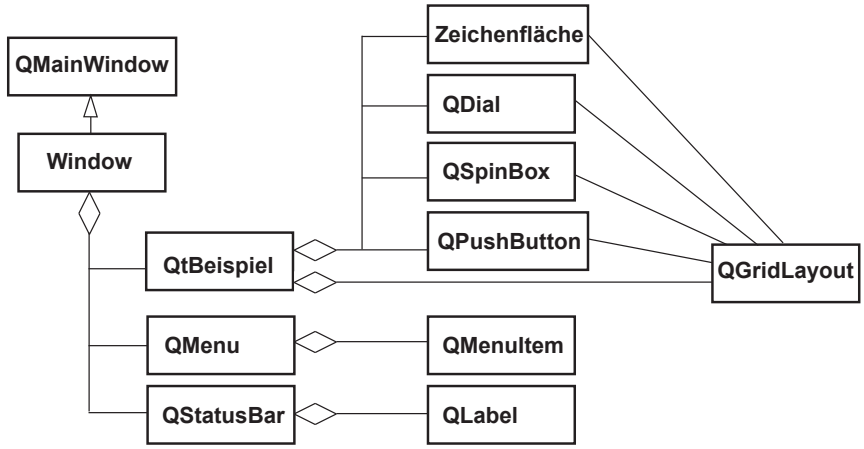

Abbildung 14.3: Vereinfachtes UML-Diagramm zum Beispiel

- Ein »100%«-Button (`QPushButton`), dem das Signal `clicked()` zugeordnet ist. Anklicken bewirkt, dass der Kreis maximal groß gezeichnet wird.
- Ein Eingabefeld des Typs `QSpinBox`, in das eine Zahl eingetippt werden kann. Alternativ kann sie durch Anklicken der Elemente rechts im Eingabefeld hoch- bzw. heruntergezählt werden. So ein Eingabefeld wird »SpinBox« genannt, vom englischen *to spin* (sich drehen), auch wenn die Bewegung innerhalb einer Reihe aufeinanderfolgender Elemente nicht kreisförmig ist. Eine Änderung der Zahl bewirkt eine Radiusänderung des Kreises.
- Eine Einstellscheibe (englisch *dial*) des Typs `QDial`, die mit der Maus bedient werden kann. Eine Änderung der Einstellung bewirkt eine Radiusänderung des Kreises.
- Eine Zeichenfläche, auf der der Kreis dargestellt wird.

Das `main`-Programm ist gewohnt kurz. Das Löschen des Fensters wird mit dem Schließen erledigt (Attribut `WA_DeleteOnClose` im `Window`-Konstruktor).

Listing 14.4: Beispielprogramm mit Qt: main()

```
// cppbuch/k14/signalslot/main.cpp
#include <QApplication>
#include "window.h"

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  Window* win = new Window;
  win->show();
  return app.exec();
}
```

Der Konstruktor der Klasse `Window` setzt zunächst das Attribut zum Löschen und dann die äußeren Koordinaten und Abmessungen mit `setGeometry(int x, int y, int breite, int hoehe)`. Die `int`-Werte werden in Pixeln angegeben. Ein `QMainWindow` hat verschiedene Bereiche:

- Am oberen Rand ist der Menübalken (englisch *menu bar*).
- Am unteren Rand befindet sich die Statuszeile.
- In der Mitte ist der Bereich für das zentrale Widget. Dieser Bereich kann von sogenannten »Dock Widgets« oder »Toolbars« umgeben sein. Erstere sind Widgets, die frei innerhalb des erlaubten Bereichs verschoben werden können, z.B. Widgets in Zeichenprogrammen zur Pinsel- oder Farbwahl. Letztere sind verschiebbare Leisten am Rand, die Widgets für verschiedene Aktionen enthalten. Dock Widgets und Toolbars werden im Beispiel nicht verwendet, aber das zentrale Widget sowie Menübalken und Statuszeile. Das zentrale Widget `widget` vom Typ `QtBeispiel` ist ein Container für die sichtbaren Widgets. Es wird in Zeile 14 unten festgelegt.

Listing 14.5: Klasse Window

```
1  // cppbuch/k14/signalslot/window.h
2  #ifndef WINDOW_H
3  #define WINDOW_H
4  #include "QtBeispiel.h"
5
6  class Window : public QMainWindow {
7    // Das Makro Q_OBJECT sorgt u.a. für die richtige Interpretation der Signale und Slots.
8    Q_OBJECT
9  public:
10   Window() {
11     setAttribute(Qt::WA_DeleteOnClose);
12     setGeometry(100, 100, 400, 300);
13     widget = new QtBeispiel(this);
14     setCentralWidget(widget);
15     menuAnlegen();                          // siehe unten
16     statuszeileAnlegen();                   // siehe unten
17     connect(widget->getZeichenflaeche(), SIGNAL(radiusChanged(int)),
18           this, SLOT(statuszeileAktualisieren(int)));
19   }
```

Die `connect`-Anweisung (Zeilen 17 und 18) ist wie folgt zu verstehen: Die Zeichenfläche des Widgets ist der Sender des Signals `radiusChanged(int)`. Die Zeichenfläche ruft bei Änderung die Funktion auf und übergibt ihr den aktuellen Radius. Der Empfänger des Signals ist `*this`, also das `Window`-Objekt, und für dieses Objekt soll die Funktion `statuszeileAktualisieren(int)` mit dem übergebenen Wert aufgerufen werden. `connect` verbindet das Signal `radiusChanged(int)` mit dem Slot (= der aufzurufenden Funktion des Empfängers) `statuszeileAktualisieren(int)`. Damit der Slot vom Meta-Objekt-Compiler erkannt wird, ist `private slots:` vorzuschalten bzw. `public slots:`, wenn der Slot von einem anderen Objekt aus erreichbar sein soll. In der Funktion ist zu sehen, dass auch HTML-formatierter Text möglich ist.

```
20 private slots:
21    void statuszeileAktualisieren(int radius) {
22       if(radius < 51) {
23          statusLabel->setText("<span style=\"color:green\">klein</span>");
24       }
25       else{
26          statusLabel->setText("<span style=\"color:red\">gross</span>");
27       }
28    }
29 private:
30    void menuAnlegen() {
31       fileMenu = menuBar()->addMenu(tr("&Datei"));
32       QAction* quitAction = new QAction(tr("&Ende"), this);
33       quitAction->setShortcut(tr("Ctrl+Q"));
34       quitAction->setStatusTip(tr("Programm beenden"));
35       fileMenu->addAction(quitAction);
36       // qApp ist globale Variable der Applikation
37       connect(quitAction, SIGNAL(triggered()), qApp, SLOT(quit()));
38    }
39
40    void statuszeileAnlegen() {
41       statusLabel = new QLabel("nicht bewegt");
42       statusBar()->addWidget(statusLabel);
43    }
44    QtBeispiel* widget;
45    QMenu* fileMenu;
46    QLabel* statusLabel;
47 };
48 #endif
```

In der Funktion `menuAnlegen()` wird ein Menüeintrag mit einer Aktion verbunden. Die Schritte im Einzelnen (Zeilennummern am Anfang):

31 Das Menü `tr("&Datei")` wird dem Menübalken hinzugefügt. Ein einem Buchstaben vorangestelltes `&`-Zeichen definiert diesen Buchstaben als Hotkey. `tr()` steht für *translate* und ist eine Funktion, die bei der tatsächlichen Ausgabe für die richtige Sprache sorgt, sofern dieses Feature eingestellt ist.

32 Es wird eine Aktion des Namens `tr("&Ende")` erzeugt.

33 Die Tastenkombination Ctrl+Q (= Strg-Q) wird als Shortcut definiert. Mit dieser Tastenkombination wird das Programm auch ohne Anwahl des Menüeintrags beendet.

34 Bei Aktivieren des Eintrags wird der erläuternde Text in der Statuszeile angezeigt.

35 Die Aktion wird dem Menü hinzugefügt.

37 Damit die Anwahl dieses Menü-Items tatsächlich zum Beenden des Programms führt, muss das von ihm ausgehende Signal mit der `quit()`-Methode der Applikation verbunden werden. `qApp` ist eine in jeder Qt-Anwendung definierte globale Variable, die auf das `QApplication`-Objekt zeigt. Sender des Signals ist das Objekt `*quitAction`. Das Signal ist für Aktionen vordefiniert und heißt `triggered()`. Empfänger des Signals ist das `QApplication`-Objekt `qApp`, dessen Slot-Methode `quit()` bei Auftreten des Signals gerufen wird.

Das allgemeine Aussehen wird vom aktuellen System übernommen, das heißt, unter Windows würde die Abbildung 14.2 wie ein typisches Windows-Fenster aussehen. Im Folgenden werden die noch fehlenden Dateien *QtBeispiel.h* und *Zeichenflaeche.h* dargestellt und kommentiert.

Listing 14.6: Klasse QtBeispiel

```
1  // cppbuch/k14/signalslot/QtBeispiel.h
2  #ifndef QTBEISPIEL_H
3  #define QTBEISPIEL_H
4  #include"Zeichenflaeche.h"
5
6  class QtBeispiel : public QWidget {
7    Q_OBJECT
8  public:
9    QtBeispiel(QMainWindow* parent = 0)
10     : QWidget(parent),
11       zeichenflaeche(new Zeichenflaeche),
12       qdial(new QDial),
13       qspinbox(new QSpinBox) {
14     qdial->setNotchesVisible(true); // Marken sichtbar machen
15     qspinbox->setFont(QFont("Helvetica", 12, QFont::StyleNormal));
16     QPushButton *maxWertButton = new QPushButton("100%");
17     maxWertButton->setFont(QFont("Helvetica", 16, QFont::Normal));
18
19     // Connect-Aufrufe
20     connect(qdial, SIGNAL(valueChanged(int)), zeichenflaeche,
21           SLOT(setRadius(int)));
22     connect(qdial, SIGNAL(valueChanged(int)), qspinbox, SLOT(setValue(int)));
23     connect(qspinbox, SIGNAL(valueChanged(int)), qdial, SLOT(setValue(int)));
24     connect(maxWertButton, SIGNAL(clicked()), this, SLOT(setMaxwert()));
25
26     QGridLayout *gridLayout = new QGridLayout(this);
27     gridLayout->addWidget(maxWertButton, 0, 0);
28     gridLayout->addWidget(qspinbox, 1, 0);
29     gridLayout->addWidget(qdial, 2, 0);
30     gridLayout->addWidget(zeichenflaeche, 0, 1, 3, 1);
```

```
31        // Zeile, Spalte, erstreckt sich über 3 Zeilen und eine Spalte
32        gridLayout->setColumnStretch(1, 10); // Spalte 1 breiteren Raum einräumen
33        setLayout(gridLayout);
34     }
35     qdial->setValue(1);                          // Startwert
36     const Zeichenflaeche* getZeichenflaeche() const {
37        return zeichenflaeche;
38     }
39  private slots:
40     void setMaxwert() {
41        qdial->setValue(100);
42     }
43  private:
44     Zeichenflaeche* zeichenflaeche;
45     QDial* qdial;
46     QSpinBox* qspinbox;
47  };
48  #endif
```

11–13 In diesen Zeilen werden drei der Widgets initialisiert, für die `QtBeispiel` als Container dient.

16 Konstruktion des 100%-Buttons, des vierten angezeigten Widgets.

20–21 Entsprechend der oben dargestellten Logik werden Signale und Slots verbunden. Es wird dafür gesorgt, dass Änderungen der Drehscheibe `*qdial` zum Aufruf der Funktion `setRadius(int)` des Objekts `*zeichenflaeche` führen. Der Aufruf von `setRadius(int)` bewirkt das Zeichnen des Kreises mit einem neuen Radius (siehe Listing der Klasse `Zeichenflaeche` unten).

22 Hier können Sie sehen, dass dasselbe Signal an mehrere Empfänger geleitet werden kann, in diesem Fall an die SpinBox, damit gleichzeitig die Zahlen aktualisiert werden.

23 Umgekehrt sollen sich Änderungen der SpinBox in der Einstellung der Drehscheibe bemerkbar machen. Dabei wird ein Problem deutlich: Wenn die Spinbox die Drehscheibe benachrichtigt und diese daraufhin nicht nur den Zustand ändert, sondern wiederum die Spinbox informiert, könnte es eine unendliche Folge gegenseitiger Aufrufe geben! In diesen von Qt vorgegebenen Klassen wird dafür gesorgt, dass so etwas nicht geschehen kann (siehe Tipp unten).

24 Das Signal `clicked()` ist für `QPushButton` vordefiniert. Anklicken des Buttons führt zum Aufruf der Methode `setMaxwert()`, die ihrerseits dafür sorgt, dass die Drehscheibe `qdial` entsprechend gesetzt wird (Zeile 41). Dieses wiederum führt durch die `connect`-Anweisungen der Zeilen 20-23 zum Neuzeichnen des Kreises mit dem Radius 100 und zur Aktualisierung der SpinBox. Der voreingestellte mögliche Bereich der SpinBox liegt zwischen 0 und 99, wenn er nicht zum Beispiel mit `setRange(int min, int max)` anders definiert wird. Die Klasse sorgt dafür, dass der angezeigte Wert nicht überschritten wird. Wenn der Wert auf 100 gesetzt wird, erscheint also tatsächlich nur 99.

26–30 Das `QGridLayout`-Objekt ist ein Layout-Manager, der die Widgets in einer Tabellenform anordnet. Der Funktion `addWidget(QWidget w, int zeile, int spalte)` wird dabei die gewünschte Position mitgegeben. Die Zählung beginnt ab 0. Nun kann es sein, dass sich ein Objekt über mehrere Zeilen und Spalten erstreckt, wie bei der Zeichenfläche in unserem Fall. Dafür gibt es eine überladene Variante: `addWidget(QWidget w, int zeile, int spalte, int anzahlDerZeilen, int anzahlDerSpalten)`.

33 Das Layout wird dem Container zugeordnet.

35 Der Startwert wird festgelegt. Dank des Signal/Slot-Mechanismus pflanzt sich der eingestellte Wert auf die SpinBox und die Zeichenfläche fort.

Tipp

Bei gegenseitigen Informationen eigener Klassen muss darauf geachtet werden, Rekursion zu verhindern.

Die einfachste Möglichkeit, Rekursion zu verhindern, ist, nur dann etwas zu tun, wenn sich der Zustand tatsächlich ändert, etwa

```
void setZustand(int neuerZustand) {
   if(zustand != neuerZustand) { // nur dann etwas tun!
      zustand = neuerZustand;
      // weitere damit verbundene Aktionen ausführen
      // beteiligte Objekte informieren
   }
}
```

Die Zeichenfläche dient ausschließlich der Anzeige und der Aktualisierung des Kreises. Der Konstruktor legt den Anfangswert des Radius und die Hintergrundfarbe fest. Erläuterungen zu den anderen Funktionen folgen nach dem Listing.

Listing 14.7: Klasse Zeichenflaeche

```
1  // cppbuch/k14/signalslot/Zeichenflaeche.h
2  #ifndef ZEICHENFLAECHE_H
3  #define ZEICHENFLAECHE_H
4  #include <QtGui>
5
6  class Zeichenflaeche : public QWidget {
7     Q_OBJECT
8  public:
9     Zeichenflaeche() {
10       aktuellerRadius = 1;
11       setPalette(QPalette(QColor(255, 255, 255))); // weiß
12       setAutoFillBackground(true);
13    }
14
15 public slots:
16    void setRadius(int radius) {
17       if(radius != aktuellerRadius) {      // nur dann etwas tun!
18          if (radius < 1) {
```

```
19            radius = 1;
20         }
21         aktuellerRadius = radius;
22         update();
23         emit radiusChanged(aktuellerRadius); // Änderung signalisieren
24      }
25   }
26
27 signals:
28   void radiusChanged(int neuerRadius);
29
30 protected:
31   void paintEvent(QPaintEvent*) { // Parameter wird nicht benutzt
32      QPainter painter(this);
33      // Transformation der Koordinaten auf die Zeichenfläche
34      painter.translate(0, rect().height());
35      int offset = rect().height()/2;
36      int r = aktuellerRadius;
37      // einen durch ein Rechteck definierten Kreis zeichnen:
38      painter.drawEllipse(QRectF(offset-r, -offset-r, 2*r, 2*r));
39   }
40 private:
41   int aktuellerRadius;
42 };
43 #endif
```

15–25 Der Slot `setRadius(int radius)` wird über den Signal-Mechanismus von den anderen Widgets aufgerufen. Es wird nur dann etwas getan, wenn sich der Radius geändert hat. Damit ist eine Zusammenarbeit mit möglichen weiteren, noch zu schreibenden Klassen ohne Rekursion gewährleistet. Allein für dieses Beispiel könnte die Abfrage auf eine Änderung des Radius entfallen, weil schon die Qt-Widgets Rekursion verhindern; es wäre jedoch schlechter Stil und riskant.

22 Die Funktion `update()` plant einen `paintEvent()`-Aufruf (Zeile 31) ein. Der eigentliche Zeichenvorgang kostet Zeit. Deshalb sammelt Qt aus Performance-Gründen intern alle Aufrufe zum Zeichnen und führt sie erst bei passender Gelegenheit zusammen aus. Außer der erhöhten Geschwindigkeit ist ein weiterer Vorteil das reduzierte Flackern eines Bildes.

23 Hier sehen Sie ein neues Schlüsselwort. `emit` ist tatsächlich ein Makro. Die Bedeutung ist klar: die Erzeugung eines Signals. Das Signal wird von der Klasse `Window` ausgewertet, die gegebenenfalls die Statuszeile aktualisiert (Zeilen 17–18 im Listing der Klasse `Window` oben).

27–28 Das Signal wird hier definiert. Sie sehen, dass es die Form einer Funktionsdeklaration hat. Wo bleibt die Funktionsdefinition? Sie wird vom Meta-Objekt-Compiler (moc) erzeugt. Wie das Ergebnis, auf das ich hier *nicht* eingehen möchte, aussieht, können Sie herausfinden, wenn Sie sich die vom moc erzeugten *.cpp*-Dateien ansehen. Der Dateiname beginnt mit *moc_*.

Zum Ablauf: Im Konstruktor wird der Startwert des Radius mit 1 festgelegt. Einmalig, ohne dass seitens des Programms etwas dazu notwendig ist, wird das Zeichnen ausgeführt. Der Aufruf der Methode `setRadius()` im Konstruktor kann deswegen entfallen. Auch würde beim ersten Mal der Signal-Mechanismus nicht funktionieren, weil die `connect`-Anweisungen im Konstruktor von `QtBeispiel` noch nicht abgelaufen sind. Das erste Zeichnen hätte keine Auswirkungen auf die anderen beteiligten Widgets. Damit alle Anzeigen mit dem Kreisradius garantiert übereinstimmen, wird am Ende des Konstruktors von `QtBeispiel`, also nach Ablauf der `connect`-Anweisungen, der Startwert für alle Widgets mit `qdial->setValue(1);` festgelegt (Zeile 35 im Listing von `QtBeispiel`).

14.4 Dialog

Ein Dialog ist ein GUI-Element zur Abfrage von Informationen, die eine Anwendung benötigt, zum Beispiel gewünschte Farben, Namen, Einstellungen für die Schriftgröße und mehr. Im folgenden Beispiel wird ein einfacher Dialog zur Eingabe eines Namens sowie seine Wechselwirkung mit dem aufrufenden Programm gezeigt. Der Dialog des Typs `NamenDialog` (siehe Listing unten) besitzt außer dem Eingabefeld einen Button zur Bestätigung der Eingabe und einen Button zum Abbruch, wie Abbildung 14.4 zeigt. `NamenDialog` erbt von der Qt-Klasse `QDialog`.

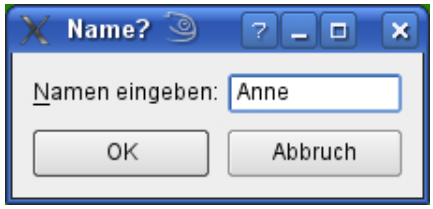

Abbildung 14.4: Einfacher Dialog

Listing 14.8: Dialog zur Eingabe eines Namens

```
1  // cppbuch/k14/dialog/NamenDialog.h
2  #ifndef NAMENDIALOG_H
3  #define NAMENDIALOG_H
4  #include<QtGui>
5
6  class NamenDialog : public QDialog {
7  Q_OBJECT
8  public:
9     NamenDialog(QWidget* parent = 0)
10       : QDialog(parent) {
11       setWindowTitle("Name?");
12       eingabezeile = new QLineEdit;
13       QLabel* text = new QLabel("&Namen eingeben:");
14       text->setBuddy(eingabezeile);
```

```
15      QPushButton* okButton = new QPushButton("OK");
16      QPushButton* abbruchButton = new QPushButton("Abbruch");
17
18      QBoxLayout* zeile1 = new QBoxLayout(QBoxLayout::LeftToRight);
19      zeile1->addWidget(text);
20      zeile1->addWidget(eingabezeile);
21      QBoxLayout* zeile2 = new QBoxLayout(QBoxLayout::LeftToRight);
22      zeile2->addWidget(okButton);
23      zeile2->addWidget(abbruchButton);
24      QBoxLayout* alles = new QBoxLayout(QBoxLayout::TopToBottom);
25      alles->addLayout(zeile1);
26      alles->addLayout(zeile2);
27      setLayout(alles);
28
29      connect(okButton, SIGNAL(clicked()),this, SLOT(accept()));
30      connect(abbruchButton, SIGNAL(clicked()),this, SLOT(reject()));
31    }
32    QString getText() {
33      return eingabezeile->text();
34    }
35  private:
36    QLineEdit* eingabezeile;
37  };
38  #endif
```

Die Erläuterung des Listings, soweit es Qt betrifft:

11–16 Diese Zeilen setzen den Titel des Dialogs und definieren alle Widgets. Der Aufruf `text->setBuddy(eingabezeile)` bewirkt, dass die Tastenkombination Alt+N den Focus auf die Eingabezeile bringt. Der Buchstabe N ist in der Variablen `text` als Shortcut festgelegt.

18 Ein `QBoxLayout` sorgt dafür, dass Widgets in einer Reihe angeordnet werden. Die Reihe kann horizontal oder vertikal sein. Der dem Konstruktor übergebene Parameter definiert die Anordnung.

19–20 Dem Layout-Objekt werden das Label und die Eingabezeile von links nach rechts hinzugefügt.

21–23 Das nächste `QBoxLayout`-Objekt `zeile2` nimmt die beiden Buttons auf.

24 Das `QBoxLayout`-Objekt `alles` definiert eine vertikale Anordnung von oben nach unten.

25–26 Diesem Objekt werden die beiden Zeilen hinzugefügt, sodass sie übereinander liegen, entsprechend der Abbildung 14.4.

27 Das Layout für diesen `QDialog` wird festgelegt.

29–30 Die Signale der beiden Buttons werden mit Slots verbunden. Die Slots `accept()` und `reject()` sind nicht im Listing zu finden, weil sie von der Oberklasse geerbt werden.

Beide Slots bewirken, dass der Dialog geschlossen (aber nicht gelöscht!) wird. `accept()` setzt den Zustand des Dialogs auf `QDialog::Accepted`, `reject()` auf `QDialog::Rejected`. Diese beiden Konstanten (1 und 0) können mit der geerbten Methode `result()` abgefragt werden.

Tipp

Wenn Sie nach Schließen des Dialogs noch Fragen an das Objekt haben, also die Methode `result()` oder `getText()` aufrufen, muss das Dialog-Objekt, obgleich geschlossen und deswegen nicht mehr sichtbar, noch existieren. Aus diesem Grund darf das Attribut `WA_DeleteOnClose` nicht gesetzt werden!

Natürlich muss das Dialog-Objekt irgendwann »entsorgt« werden. Das kann man dem Eltern-Objekt überlassen, oder, wenn man den Speicher früh freigeben möchte, nach Abfrage der gewünschten Informationen selbst erledigen. Dieser Weg wird in dem folgenden Mini-Programm, das nur die Benutzung des Dialogs zeigen soll, eingeschlagen.

Listing 14.9: Window als Dialog-Benutzer

```
// cppbuch/k14/dialog/DialogUser.h
#ifndef DialogUser_h
#define DialogUser_h
#include"NamenDialog.h"

class DialogUser : public QMainWindow {
   Q_OBJECT
public:
   DialogUser() {
      setAttribute(Qt::WA_DeleteOnClose);
      namendialog = new NamenDialog(this);
      namendialog->setModal(true);
      namendialog->show();
      connect(namendialog, SIGNAL(accepted()), this, SLOT(dialogBeendet()));
      connect(namendialog, SIGNAL(rejected()), this, SLOT(dialogBeendet()));
   }
private slots:
   void dialogBeendet() {
      // Zuerst Ergebnis abfragen (hier mit Anzeige) ...
      QString str("Dialog abgebrochen");
      if(namendialog->result() == QDialog::Accepted) { // doch erfolgreich
         str = namendialog->getText();
      }
      QMessageBox msgBox;
      QString msg("Dialogergebnis: ");
      msg += str;
      msgBox.setText(msg);
      msgBox.exec();
      // und dann Dialog schließen
      delete namendialog;
   }
private:
   NamenDialog* namendialog;
```

```
};
#endif
```

Für das Haupfenster kann `WA_DeleteOnClose` natürlich gesetzt werden. Der Konstruktor erzeugt den Dialog und setzt die Eigenschaft »modal«. Ein modaler Dialog zeichnet sich dadurch aus, dass er erst bearbeitet werden muss, ehe ein anderes Fenster der Anwendung den Focus bekommen kann. Im Beispiel ist das leicht zu sehen: Das Hauptfenster kann nicht geschlossen werden, solange der Dialog noch geöffnet ist. Wenn der Dialog nicht modal ist, kann jedes Fenster der Anwendung den Focus bekommen (aktiviert werden).

Auf Seite 462 ist zu sehen, dass dasselbe Signal an verschiedene Empfänger gesendet werden kann. Hier hingegen sehen Sie an den `connect`-Anweisungen, dass verschiedene Signale an denselben Empfänger, den Slot `dialogBeendet()`, gehen. Die Methode `dialogBeendet()` ermittelt den Zustand des Dialogs und, wenn er nicht abgebrochen wurde, den eingegebenen Text. Qt bietet eine Reihe vorgefertigter Dialoge. `QMessageBox` ist einer davon; er wird zur Anzeige des Ergebnisses verwendet. Abschließend wird der Dialog mit `delete` gelöscht. Der Destruktor der Oberklasse sorgt dafür, dass der Dialog aus der Kind-Liste des Eltern-Objekts (hier `DialogUser`, Oberklasse `QMainWindow`) ausgetragen wird, so dass der Destruktor der Klasse `QMainWindow` dieses »Kind« nicht mehr löscht. Das `main`-Programm ist kurz und schlicht:

```
#include"DialogUser.h"

int main(int argc, char *argv[]) {
   QApplication app(argc, argv);
   DialogUser* dialogUser = new DialogUser;
   dialogUser->show();
   return app.exec();
}
```

14.5 Qt oder Boost?

Qt ist, wie am Anfang des Kapitels schon erwähnt, nicht nur für GUIs geeignet, sondern bietet viele weitere Funktionen wie ein eigenes Thread-Modell, die Arbeit mit Verzeichnissen und mehr. Es zeigt sich, dass die Boost-Library und Qt eine beachtliche Schnittmenge der angebotenen Funktionalitäten aufweisen. Wenn Sie ohnehin wegen einer zu schreibenden Benutzungsoberfläche Qt verwenden, bietet es sich an, auch die anderen Qt-Funktionen zu nutzen – andernfalls genügt die Boost-Library (oder Sie nehmen auch dann Qt). Im Folgenden wird exemplarisch zum Vergleich auf die anders als in [ISOC++] realisierten Threads und das rekursive Durchwandern eines Verzeichnisbaums eingegangen. Dabei werden die Beispiele den entsprechenden Boost-Beispielen in diesem Buch nachempfunden, um einen direkten Vergleich zu bieten. Weitere Möglichkeiten wie Netzwerkprogrammierung und mehr bitte ich dem Buch [BlSu] zu entnehmen.

14.5.1 Threads

`QThread`-Objekte sind anders als Standard-C++-Threads. Um einen Thread zu erzeugen, erzeugt man eine Unterklasse von `QThread`, in der die Methode `run()` überschrieben wird.

Listing 14.10: Thread-Beispiel

```
// cppbuch/k14/thread/thread.cpp
#include<QThread>
#include <iostream>

class MyThread : public QThread {
public:
   MyThread(int t)
      : dauer(t) {
   }
   void run() {
      sleep(dauer);
      std::cout << "Thread beendet! Laufzeit = " << dauer << " s" << std::endl;
   }
private:
   int dauer;
};

using namespace std;

int main() {
   MyThread t1(4);
   MyThread t2(6);
   MyThread t3(2);
   t1.start();        // ruft implizit run() auf
   t2.start();
   t3.start();
   cout << "t1.isRunning(): " << t1.isRunning() << endl;
   cout << "t2.isRunning(): " << t2.isRunning() << endl;
   cout << "t3.isRunning(): " << t3.isRunning() << endl;
   t1.wait(); // warten auf Beendigung
   t2.wait(); // warten auf Beendigung
   t3.wait(); // warten auf Beendigung
   cout << "t1.isRunning(): " << t1.isRunning() << endl;
   cout << "main() ist beendet" << endl;
}
```

Das Beispiel entspricht demjenigen auf Seite 421, das auf der Boost-Bibliothek beruht. Die Unterschiede in den Beispielen sind:

- Ein `QThread` wird nicht schon mit dem Aufruf des Konstruktors gestartet, sondern erst mit der Methode `start()`. Diese geerbte Methode ruft dann die Methode `run()` auf. Die Methode `run()` direkt aufzurufen, würde zwar zur Abarbeitung der Methode führen, aber nicht zu einem eigenen Thread.
- Die auf Seite 421 verwendete Methode `get_id()` dient dort zur Anzeige, ob der Thread aktiv ist. Qt hat dafür die Methode `isRunning()`.
- Anstatt `join()` heißt es nun `wait()`.

Das Qt-Thread-Modell bietet sehr viele Möglichkeiten zur Parallelisierung von Prozessen wie Thread-Pools, thread-lokale Daten oder ein API zum Schreiben paralleler Anwendungen ohne low-level-Steuerungsmechanismen wie Mutex-Objekte (Namespace `QtConcurrent`).

14.5.2 Verzeichnisbaum durchwandern

Qt bietet ebenfalls viele Möglichkeiten der Dateiein- und -ausgabe. Hier sei exemplarisch das von Standard-C++ nicht unterstützte Lesen von Verzeichnissen und der Abstieg im Verzeichnisbaum gezeigt. Das Beispiel entspricht demjenigen des Abschnitts 25.1.5 auf Seite 732, das auf der Boost-Bibliothek beruht. Von den Dateien des Abschnitts ändert sich nur die Datei `tree.cpp`:

Listing 14.11: Verzeichnisbaum anzeigen

```
// cppbuch/k14/dirtree/tree.cpp
#include<iostream>
#include<stdexcept>
#include<QFileInfo>
#include<QFileInfoList>
#include<QDir>
#include"tree.h"

namespace {
    std::ostream& operator<<(std::ostream& os, const QString& qstr) {
      const char* str = qstr.toAscii().constData();
      os << str;
      return os;
   }

   void baumAnzeigen(const QDir& d, int level) {
      QStringList eintraege = d.entryList(
                    QDir::Dirs|QDir::Files|QDir::NoDotAndDotDot);
      QStringList::const_iterator dIterator = eintraege.begin();
      while(dIterator != eintraege.end()) {
         QString fn(*dIterator);
         QString absPfad(d.absolutePath() + "/" + fn);
            for(int i = 0; i < level; ++i) {
               std::cout << "| ";
            }
            std::cout << "|-- " << fn << std::endl;
            QFileInfo qi(absPfad);
            if(qi.isDir()) {
               QDir dir(absPfad);
               baumAnzeigen(dir, level+1);
            }
         ++dIterator;
      }
   }
}  // anonymer namespace
```

```
void baumAnzeigen(const std::string& verz) {
   QFileInfo pfad(verz.c_str());
   if(pfad.isDir()) {
      std::cout << verz << std::endl;
      QDir dir(verz.c_str());
      baumAnzeigen(dir, 0);
   }
   else {
      throw std::runtime_error(" ist kein Verzeichnis!");
   }
}
```

Die Ablauflogik ist dieselbe wie die des Programms auf Seite 733. Die Qt-spezifischen Unterschiede sind:

- Qt verwendet eine eigene String-Klasse `QString`, für die kein Ausgabeoperator `<<` definiert ist. Deswegen wird dieser oben im Listing definiert.
- Die Einträge eines Verzeichnisses sind unter anderem auch als `QString`-Liste erhältlich. Der Funktion `entryList()` können Filter übergeben werden. Im Beispiel sind Datei- und Verzeichnisnamen erlaubt, aber nicht, wenn sie nur aus einem oder zwei Punkten bestehen (aktuelles bzw. Oberverzeichnis). Symbolische Links oder Geräte werden oben damit ausgeschlossen.

Fazit

Dieses Kapitel bietet einen Einstieg in die GUI-Programmierung mit Qt und seine anderen Möglichkeiten. Besonders die Umwandlung in ein Modell mit gestaffelten Lizenzen, darunter auch die GNU Public License und die LGPL, lassen vermuten, dass die Verbreitung von Qt zunehmen wird. Da auch die technische Qualität und die Qualität der Dokumentation recht gut sind, ist Qt meiner Meinung nach die beste Möglichkeit zur Realisierung portabler Benutzungsoberflächen mit C++.