Einführung in Qt

Im Rahmen der Vorlesung Objektorientierte Softwareentwicklung mit C++ (WS 2002/2003)

Inhaltsangabe

1. Was ist Qt ?

- C++ Klassenbibliotheken zur Gestaltung von grafischen Benutzeroberflächen (GUI)
- Paket mit diversen Tools, die die Entwicklung von GUIs unterstützen
- Plattformübergreifende Implementation: unterstützt aktuell (Version 3.0) Windows-, UNIX- und Mac-Systeme
- Qt wird seit 1994 von der norwegischen Firma Trolltech entwickelt

2. Der Ursprung von Qt

Vor der Entwicklung von Qt:

zunächst starr auf das jeweilige System festgelegte Entwicklungsumgebung (Beispiele: Windows : MFC; UNIX : X-Windows)

→ Nachteile: **plattformgebunden**, meist **unpraktikabel** (da direkte Programmierung von Nöten)

Neuer Ansatzpunkt bzgl. des Application Programmer Interface (API - eine für Programmierer wichtige Schnittstelle zwischen dem zu programmierenden Gerät (z.B. Betriebssystem) und dem Programm. So ist es möglich, durch einfache Befehle komplexe Funktionen auszulösen):

- **API Layering**: Überlagerung von neuen Schnittstellen über vom System zur Verfügung gestellten Schnittstellen (Beispiel Java: AWT)
- **API Emulation**: (basierend auf einer vorgegebenen Plattform) Nachahmung einer anderen (Beispiel: WINE-Project)
- **GUI Emulation**: Bereitstellung aller Schnittstellen durch das Toolkit (außer den Basisbefehlen des Systems)

→ **Qt : GUI Emulation**

Vorteil: schneller als die anderen Ansätze

Nachteil: das Aussehen der entworfenen GUIs kann auf verschiedenen Plattformen leicht variieren

3. Eigenschaften von Qt

- **objektorientierter Ansatzpunkt**, insbesondere Instanzen & Klassen, private & öffentliche Klassen / Variablen, Vererbung, Polymorphismus und virtuelle Klassen / Methoden, etc.

- **Kombinationen mit anderen Sprachen** als C++ **möglich**, zum Beispiel Perl, Phyton, OpenGL, SQL

- **Kostenlos** für nicht-kommerzielle Applikationen (Beispiel Diplomarbeit ;-))

4. Konzept einer Qt-Applikation

Merkregel: **„Almost everything is a widget !“**

- widget steht für die Bezeichnung eines Fensters unter UNIX
- jede (!) Qt-Applikation enthält Instanzen von Widgets oder speziellen Versionen von Widgets

Mein erstes einfachstes Qt-Programm:

```
#include <qapplication.h>//Headerdatei für QApplication
int main( int argc, char** argv )
{
    QApplication app( argc, argv );
                // eine neue Instanz von QApplication
    QWidget *window = new QWidget();
                // eine neue Instanz von QWidget
    app.setMainWidget(window);
                // window wird zum Haupfenster der
                // Application “app”
    window->show();
                // Anzeigen des Fensters
    return app.exec();
                // Ausführen der Applikation
}
```

(sehr spektakuläres Programm: zeichnet ein 200x200 Pixel großes Bild auf den Bildschirm)

Einfache Qt-Programme allgemein compilieren:

- auf **theseus** wechseln (!), nur dort ist zur Zeit Qt 3.0 installiert
- **Umgebungsvariablen**:
  export QTDIR=/usr/local/qt3
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$QTDIR/lib
- **Compilieren**:
  g++ -c -Wall –I$QTDIR/include –o first.o first.cpp
- **Linken**:
  g++ -L$QTDIR/lib –lqt –o first first.o

(“komplizierte” Qt-Programme und deren Übersetzung werden später behandelt)

Basis-Klassen aus dem Beispielprogramm:

- **QApplication**: jede Applikation enthält genau eine (!) Instanz dieser Klasse. Sie hält das Ganze zusammen und bewerkstelligt die Kommunikation zwischen Benutzer und den Objekten. Wichtig: am Ende der Main-Methode der exec()-Aufruf.

- **QWidget**: abgeleitet von der „Urklasse“ QObject, stellt eine Instanz der Klasse QWidget ein grafisches Element dar. Die Klasse Widget bringt viele Methoden mit, um das Aussehen, Größe, Position auf dem Bildschirm, etc. zu verändern (siehe Online Referenz Dokumentation). Mit show() wird das widget angezeigt, mit hide() versteckt. Jedes Widget wird innerhalb dessen Vater-Widgets dargestellt, Position wird relativ zur li.-ob.-Ecke des Vater-Widgets angegeben, Widgets ohne Vater sind Toplevel-Widgets

5. Ereignisverarbeitung

Die schönste Benutzeroberfläche nutzt nichts, wenn man mit dem Programm nicht interagieren kann, deswegen sollte das Beispielprogramm auf Benutzerinteraktionen reagieren können. Dies wird in Qt durch den so genannten Signal / Slot – Mechanismus bewerkstelligt.

Das Prinzip des Signal / Slot Mechanismus:

- **Ein Objekt emittiert ein Signal**. Dies kann ein vordefiniertes Signal aus der jeweiligen Klassendefinition sein oder manuell durch den Befehl emit(). Es spielt für das emittierende Objekt keine Rolle, wer der Empfänger und ob eine Reaktion stattfindet.

- Zur Interaktion muss ein **Signal** mit einem **Slot verbunden** werden, dies geschieht über den Befehl connect() :

  bool QObject::connect(QObject *sender , char * signal_n, QObject *receiver, char * slot_n) const;

  sender und **signal** spezifizieren das sendende Objekt und seine Signal-Methode, **receiver** und **slot** spezifizieren das empfangende Objekt und seine Slot-Methode. Beispiel hierfür ist das folgende Programm.

- Die Aktion beim Eintreffen eines Signals nennt sich Slot. Dieser Slot muss im jeweiligen Empfängerobjekt definiert sein. Außerdem müssen etwaige Parameter (sozusagen Nachrichten) in Signal und Slot übereinstimmen. Ansonsten können beliebig viele Signale mit beliebig vielen Slots verbunden werden.

Das Beispielprogramm wird also dahingehend abgeändert:

```
#include <qapplication.h>
#include <qpushbutton.h>
int main( int argc, char** argv )
{
    QApplication app( argc, argv );
    QWidget *window = new QWidget();
    app.setMainWidget(window);
    QPushButton *Button = new QPushButton("Abbruch", window);
                    // eine neue Instanz von QPushButton
                    // mit Beschriftung “Abbruch”
    QObject::connect( Button, SIGNAL(clicked()), &app,
            SLOT(quit()) );
                    // verbinden des Signals clicked von Button
                    // (vordefiniertes Signal der Klasse QPushButton)
                    // mit dem Aufruf quit im Empfänger app
                    // (bewirkt Schließen der Applikation)
    window->show();
    return app.exec();
}
```

Achtung:

- es ist nicht möglich, eine eigene **Eventklasse** zu definieren, die nur aus Signalen und Slots besteht. Vielmehr sind Signale / Slots als Teil einer Klasse zu verstehen.

- Signale und Slots können **nur in Klassen** eingesetzt werden, die von der Klasse QObject abgeleitet sind – zum Beispiel QWidget, siehe oben.

- Es muss in neu definierten Klassen das Makro **Q_Object** gesetzt werden – siehe unteres Beispiel.

- **default**-Parameter sind für Signals und Slots nicht erlaubt.

Beispiel einer neu definierten Klasse mit Signals/Slots-Mechanismus:

```
class MyClass : public QObject{
Q_OBJECT    // Makro für Signals/Slots

  public:        // public Methoden, Konstruktoren

  signals:       // selbst definiertes Signal
    void signal1(int,const char*);

  public slots:
    void slot1(QString &q_str);

  private slots:
    void slot2();

  private:     // private Methoden und Attribute

};
```

Eine so definierte Klasse lässt sich nicht so einfach übersetzen. Aus der Datei mit der Klassendefinition (meist eine Header-Datei) muß mit Hilfe des **Meta-Object Compilers** (**moc**) eine zusätzliche Datei erzeugt werden (moc_*)

Grundsätzlich bieten sich zwei Wege bei der Benutzung des **moc** an:

Methode 1:

Das mit moc erzeugte Programm der Klassendefinition wird in einem eigenen Schritt zu einer Objektdatei kompiliert (mit **g++**) und erst beim Linken (mit **g++**) zur ausführbaren Datei hinzugefügt.

Methode 2:

Das mit moc erzeugte Programm der Klassendefinition wird mittels einer **#include** -Anweisung in den Quell-Code eingefügt.

Je nach Wahl ergeben sich folgende Kompilierungsaufrufe:

- **Kompilationsvorgang nach <u>Methode 1</u>**:

  theseus$ **moc myclass.h -o moc_myclass.cpp**
  theseus$ g++ moc_myclass.cpp -o moc_myclass.o
  (zunächst wird die moc Datei gesondert übersetzt)
  theseus$ g++ myclass.cpp -o myclass.o
  theseus$ g++ myclass.o **moc_myclass.o** main.o -o main -L... -l...

- **Kompilationsvorgang nach <u>Methode 2</u>**:

  theseus$ moc myclass.h -o moc_myclass.cpp
  Include von moc_myclass.cpp in myclass.cpp
  (Beispiel: #include "moc_myclass.cpp")
  theseus$ g++ myclass.cpp -o myclass.o
  theseus$ g++ myclass.o main.o -o main -L... -l...

(eine Alternative für Fortgeschrittene stellt das Tool qmake dar – mehr dazu siehe Online Referenz Dokumentation)

6. Feintuning – die große weite Welt der Qt-Klassen

Es erfolgt eine kurze Vorstellung einiger Klassen, die häufig in Benutzeroberflächen auftreten. Darüber hinaus bietet Qt eine exzellente Dokumentation auf der Herstellerseite von Trolltech (http://doc.trolltech.com/3.0/)

QPushButton:

erbt von der abstrakten Klasse **QButton**, die von **QWidget** erbt. In **QButton** sind Methoden definiert, die allen Button-Typen gemeinsam sind.

Beispiele:

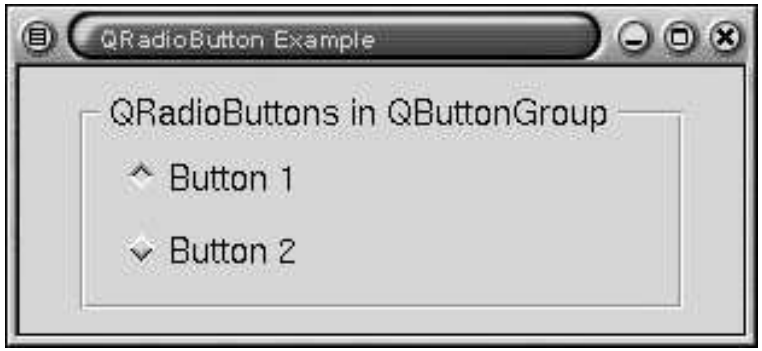
```
QString text() const;                    // liefert die Beschriftung des Buttons
QButtonGroup* group() const;             // legt eine Buttongruppe an
virtual void setText(const QString&);    //setzt die Beschriftung
```

Verbindung eines **QPushButton**‘s mit einem Slot (Beispiel siehe Programm)

- das Signal (z.B. **QPushButton::clicked())** wird verbunden mit dem gewünschten Slot (Methode eines Objekts/Funktion)

- Das Signal **clicked**() wird dann gesendet, wenn **mousePressEvent** & **mouseReleaseEvent** über dem **QPushButton** registriert wurden (= Maustaste drücken und wieder loslassen)

QRadioButtons:



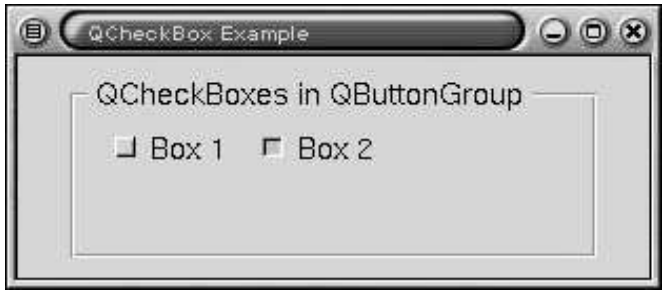
lassen den Benutzer genau eine Alternative
aus einer Reihe von Möglichkeiten auswählen

- **Konstruktoren**
  QRadioButton(const QString &text,QWidget *parent);
  … … …

- **Methoden**
  bool isChecked() const; …
  virtual void setChecked(bool ch);
  … … …
  … … …

- **Signale** *(geerbt von QButton)*
  void clicked();
  void stateChanged(int);
  … … …

QCheckBoxes



lassen den Benutzer ein Item „ankreuzen“*(1 bit Information „an/aus“)*

- **Konstruktoren**
  QCheckBox(QWidget* parent);
  QCheckBox(const QString& text, Qwidget* parent);

- **Methoden**
  bool isChecked() const; void setChecked(bool check); …

- **geerbte Methoden**
  virtual void setText(const QString& text);
  QString text() const; …

- **geerbte Signale**
  void pressed();
  void toggled(bool on); …

QLabel

lässt bequem einen einfachen Text darstellen

- **Konstruktoren**
  QLabel(QWidget *parent,const char*
  name=0,WFlags f=0);
  QLabel(const QString & text,QWidget *parent); …

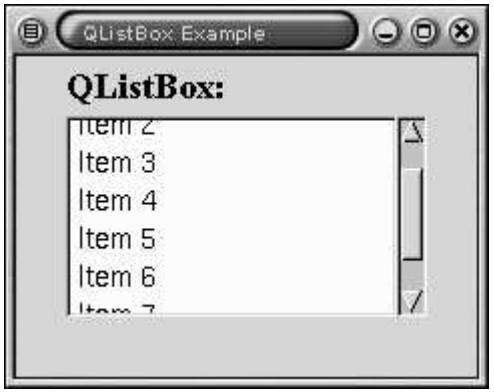
- **Methoden**
  QString text() const;
  virtual void setAlignment(int alignm);
  void setIndent(int indentation);
  int indent(void) const; …

- **Slots**
  virtual void setText(const QString& text);
  virtual void setMovie(const QMovie& movie);
  void clear(void); …

Beispiel: QLabel* text = new QLabel(parentWidget, "Dies ist ein QLabel");

QListBox

läßt den Benutzer ein oder mehrere Items aus einer Liste auswählen



- **Konstruktoren**

  QListBox(QWidget* parent=0,const char* name=0,WFlags f=0);

- **Methoden**

  void insertItem(const QString& text, int index=-1);
  void insertItem(const QPixmap& pixmap, int index=-1);
  void changeItem(const QString& text, int index);
  virtual void setCurrentItem(int index);
  int currentItem() const;
  QString currentText(void) const;
  QString text(int index) const;
  void clear(void);
  ... ...

- **Signals & Slots**

```
void selected(int index);
virtual void clearSelection(void);
void selectAll(bool select);
...          ...          ...
```

Beispiel:

```
QListBox* meineListe = new QListBox(parentWidget,"QlistBox Example");
Int i = 0;
    while( ++i <= 20 )
    l->insertItem( QString::fromLatin1( "Item " ) + QString::number( i ), i );
```

QMessageBox

stellt eine kurze Botschaft dar

- ein String, ein Icon, bis zu drei Buttons

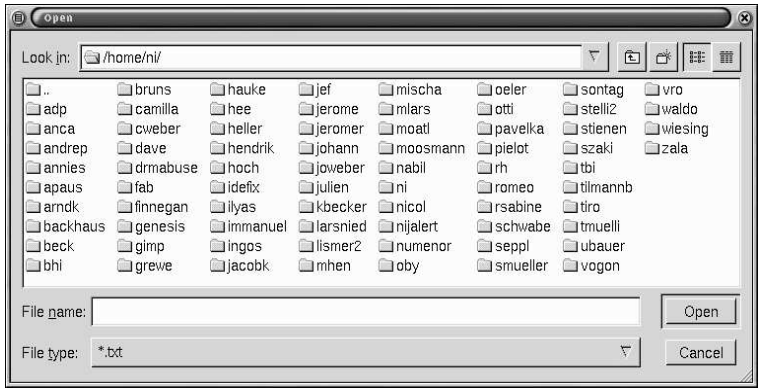
- drei verschiedene Typen:
  Information: innerhalb normalen Programmablaufs
  Warning: zur Warnung des Benutzers für ungewöhnliche Fehler
  Critical: für kritische Fehler

Es wird die Nummer des gedrückten Buttons zurückgeliefert, so kann im **switch**-Konstrukt auf die Benutzerentscheidung reagiert werden

Beispiel:
```
switch(QMessageBox::warning(...)){
    case 0: /* try again */;  break;
    case 1: /* exit      */;  break;
}
```

QFileDialog

stellt eine Dialog-Box bereit, mit der Dateien oder Verzeichnisse ausgewählt werden können



- **Statische Methoden**

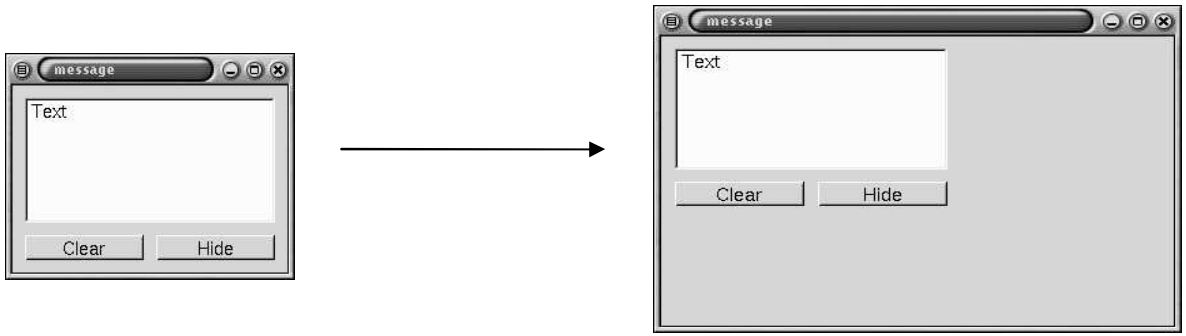
  QString getOpenFileName(...);
  QStringList getOpenFileNames(...);
  QString getSaveFileName(...); ...

Beispiel:

```
QString s(QFileDialog::getOpenFileName(QString::null,\“Images (*.jpg)“,this));
if(s.isEmpty()) return;
// your method open()
open(s);
```

7. Designprinzipien – Layout-Manager

Ein mögliches Problem bei Benutzeroberflächen ergibt sich, wenn sich die Größe des / der Fenster ändert. Meist führt dies zu unschönen oder unbrauchbaren Nebeneffekten.



Effekt beim Verändern der Größe ohne Layout-Manager

Einen Ausweg aus diesem Problem stellen die **Layout-Manager** dar. Sie werden primär eingesetzt, um:

- Unterwidgets **innerhalb** eines Eltern-Widgets anzuordnen ohne explizite Angabe/Ausrechnen deren Koordinaten
- Unterwidgets bei **resizeEvents** automatisch der neuen Größe des Eltern-Widgets anzupassen ohne explizite Neuberechnung deren Größe und Position

Es gibt zwei verschiedene Layout-Manager:

- **QBoxLayout**:
  ordnet die zugeordneten Widgets in Reihe an
  - in einer Zeile – Klasse: **QHBoxLayout** *(Horizontal)*
  - in einer Spalte – Klasse: **QVBoxLayout** *(Vertikal)*

- **QGridLayout**: ordnet die Widgets auf einem Gitter an

(Beide Layout-Manager werden abgeleitet von der abstrakten Basisklasse **QLayout;** Layouts lassen sich beliebig schachteln!)

QBoxLayout:

hat folgende Funktionalität:

Es bekommt vom Eltern-Widget den ihm zur Verfügung stehenden Platz, teilt diesen Platz in mehrere Kästchen („Boxes“) ein.

Jedes von **QBoxLayout** verwaltete Widget wird in eine „Box“ gesteckt

- Jedes Widget in so einer Box bekommt:
  - **mindestens seine: QSize QWidget::minimumSize(void);**
  - **höchstens seine: QSize QWidget::maximumSize(void);**

Der übrig bleibende Platz wird aufgeteilt gemäß von stretch-Faktoren (siehe Button-Beispiele)

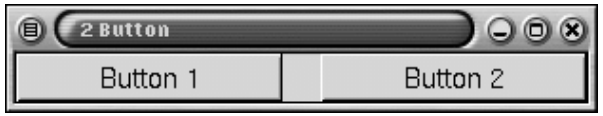
- Ist **QBoxLayout** – vertikal, dann sind die „Boxes“ untereinander
- Ist **QBoxLayout** – horizontal, dann sind die „Boxes“ nebeneinander

Beispiele des QBoxLayout anhand von Buttons:

// Layout-Programmfragment mit 2 Buttons: b1, b2

```
QHBoxLayout *l=new QHBoxLayout(topW);
l->addWidget(b1); l->addWidget(b2);
```


```
QHBoxLayout *l=new QHBoxLayout(topW);
l->addWidget(b1); l->addSpacing(20);
l->addWidget(b2);
```

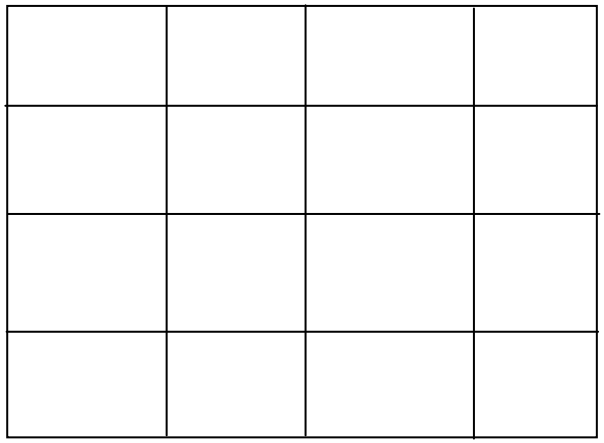



```
QHBoxLayout *l =new QHBoxLayout(topW);
l->addStretch(1); l->addWidget(b1);
l->addWidget(b2);
```


```
QHBoxLayout *l=new QHBoxLayout(topW);
l->addStretch(1); l->addWidget(b1,1);
l->addStretch(1); l->addWidget(b2,1);
l->addStretch(1);
```

QGridLayout:

- teilt den Platz in Zeilen und Spalten in Form einer Tabelle auf
- plaziert die verwalteten Widgets in den einzelnen „boxes“
- Achtung: ist wesentlich fehleranfälliger das QBoxLayout



Beispiel:
QGridLayout *grid = new QGridLayout(messageW, 4, 4);

- **Konstruktoren**

  QGridLayout(QLayout* par,int row=1,int col=1,int spc=-1);
  QGridLayout(QWidget* par,int row=1,int col=1,int bord=0,int spc=-1);

- Erste Zeile ist row=0, erste Spalte ist col=0
- Ein Unter-Layout wird mit der Methode:

  void addLayout(QLayout *layout,int row, int col);

  an der Position (row, col) auf dem Gitter eingefügt

Achtung :
Anders als bei QBoxLayout ist die Reihenfolge des Einfügens der Widgets in ein Gitter wegen der expliziten Angabe der Position (row, col) nicht wichtig !

- Ein Widget kann in ein Grid eingefügt werden:
  - in die Zeile **row**, in der Spalte **col** mit der Methode:
    void addWidget(QWidget* w, int row, int col, int alignment=0);
  - auch komplett leere Zellen/Zeilen sind erlaubt
  - auch hier ist die Reihenfolge des Einfügens beliebig

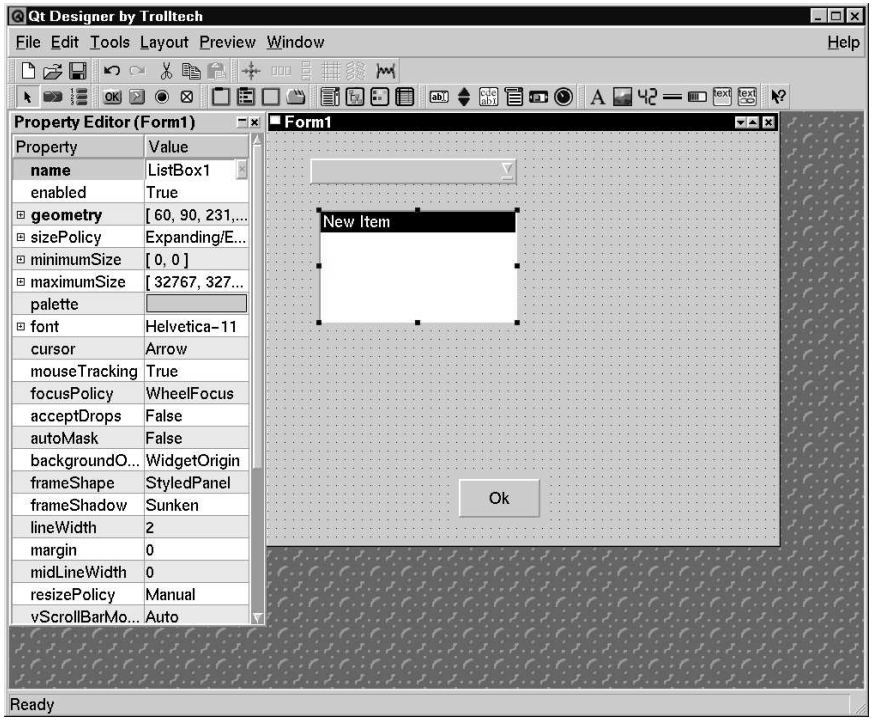
- Die Mindestmaße der Gitterzellen können definiert werden
  - Die Mindestbreite einer Spalte läßt sich einstellen mit:
    addColSpacing(int col, int space);
  - Die Mindesthöhe einer Zeile läßt sich einstellen mit:
    addRowSpacing(int row, int space);

Beispiel eines QGridLayouts:

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qlayout.h>
#include <qmultilineedit.h>

int main(int argc, char **argv){
        QApplication app(argc,argv);
        QWidget *messageW = new QWidget();
        messageW->resize(220,150);
        QMultiLineEdit *messages = new QMultiLineEdit(messageW);
        messages->append("Text");
        QPushButton *clear=new QPushButton("Clear",messageW);
        QPushButton *hide=new QPushButton("Hide",messageW);

        QGridLayout *grid = new QGridLayout(messageW, 2, 2);
        grid->addMultiCellWidget( messages, 0,0, 0,1);
        grid->addWidget(clear, 1 , 0);
        grid->addWidget(hide, 1 , 1);

        messageW->show();
        app.setMainWidget(messageW);
        return app.exec();
}
```

8. Exkurs : Der Qt-Designer

Viele zeitaufwändige Dinge (beispielsweise das Anlegen eines passenden Designs) lassen sich einfach und bequem mit dem Qt Designer bewerkstelligen (zu finden auf der theseus unter **$QTDIR/bin/designer**)



Eine vom Designer erstellte Datei enthält das User Interface *.ui (in XML-Format).

Sie kann via **uic** in C++-Code übersetzt werden, mit: theseus$ $QTDIR/bin/uic (user interface compiler) - ein ausführliches Tutorial ist in der Online Referenz Dokumentation zu finden

9. Beginn einer Diskussion

Vorteile von Qt:

- umfangreiches Toolkit mit vielen nützlichen Hilfsmitteln (moc, qmake, qtdesigner, etc.)
- aufgrund der Objektorientierung ähnliche Implementation wie zum Beispiel bei C++
- Kombination mit vielen Sprachen möglich
- Implementation ist auf verschiedenen Plattformen einsetzbar
- Viele vordefinierte Klassen verfügbar, sehr gute Online Referenz Dokumentation
- ...

Nachteile:

- häufig Kollisionen zwischen Qt- und C++-Klassenbibliotheken bei gleichzeitigem Einsatz
- Portierung von einem System auf das andere zeitweise problematisch (Beispiel: auf Windows Rechner geschriebene Applikation musste für UNIX umgeschrieben werden)
- Bei umfangreichen Applikationen verliert man bei Layout-Managern schnell den Überblick -> sehr genaue Programmierung erforderlich, speziell bei Kombination von mehreren Layout-Managern
- Qt-Designer liefert meist nur durchschnittliche Ergebnisse, „von Hand“ vielfach praktikabler
- …