Praxisorientierte Einführung in C++
Lektion: "Einführung in das GUI-Toolkit Qt"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

- Überblick: Qt
- Event-Loops
- Hello World
- Exkurs pkg-config
- Qt Überblick (2)
- Signals und Slots
- moc
- moc: Beispiel
- Anmerkungen
- qmake

Motivation

Qt Features

- Für viele Plattformen und Architekturen verfügbar
- Open Source (aber auch kommerzielle Lizenz erhältlich für Closed Source Applikationen)
- Sehr umfangreich
  - GUI
  - DB
  - Network
  - Threading
  - XML
  - ...

Qt Überblick

Qt: Download unter

http://qt-project.org/downloads

- Als Quellcode oder als fertiges Binary
- Aber in den meisten GNU/Linux/BSD-Distributionen enthalten (z.B. Package libqt4-dev in Ubuntu Linux)

Qt Überblick

Qt: Download unter

http://qt-project.org/downloads

- Als Quellcode oder als fertiges Binary
- Aber in den meisten GNU/Linux/BSD-Distributionen enthalten (z.B. Package libqt4-dev in Ubuntu Linux)
- C++-Bibliothek (aber nicht nur)
- Benutzt MOC (den Meta Object Compiler), um sog. "Signals" und "Slots" zu implementieren

Graphical User Interfaces (GUI)

- Bislang haben wir nur Kommandozeilenprogramme implementiert
- Bisher nur sehr geringer Grad an Interaktivität (z.B. mittels `std::in`)

Kommandozeilenprogramme

I.d.R. wird Input in Output umgewandelt und evtl. ab und zu auf Benutzereingabe gewartet

Graphical User Interfaces (GUI)

- Bislang haben wir nur Kommandozeilenprogramme implementiert
- Bisher nur sehr geringer Grad an Interaktivität (z.B. mittels `std::in`)

Kommandozeilenprogramme

I.d.R. wird Input in Output umgewandelt und evtl. ab und zu auf Benutzereingabe gewartet

Programm mit grafischer Benutzeroberfläche (GUI)

- Programm wartet ständig darauf, daß etwas "passiert"
- Es ereignen sich *Events*
  - Mausklicks
  - Tastatureingaben
  - Maus wurde bewegt

Event-Loops

- In GUI-Programmen ist Ablauf meistens sehr ähnlich:

Event Handling Schema

```
while(Application.isRunning()){
  Event e = wait_for_next_event();
  handle_event(e);
}
```

Event-Loops

- In GUI-Programmen ist Ablauf meistens sehr ähnlich:

Event Handling Schema

```
while(Application.isRunning()){
  Event e = wait_for_next_event();
  handle_event(e);
}
```

- Motiviert zu einer sog. "Event-Loop"

Event-Loops

- In vielen alten GUI-Bibliotheken mußte Event-Loop explizit programmiert werden (X11, win32, ...)

Explizites Abarbeiten einer Event-Loop

```
int main() {
    GUIWindow window;
    GUIEvent *event;
    while ((event = GUIGetNextEvent()) {
       switch(event->type) {
          case GUI_WINDOW_CLOSE: ... break;
          case MOUSE_CLICK: ... break;
          default: ... break;
       }
    }
}
```

Event-Loops in Qt

- Explizites Abarbeiten ist umständlich und fehlerträchtig
- Daher Event-Loop wird von Qt abgearbeitet
- "Main"-Event-Loop einer Applikation wird durch Aufruf der Methode `exec()` von einer Singleton-Instanz der Klasse `QApplication` gestartet

Schematischer Aufbau einer Qt-Application

Genereller Programmaufbau

```
#include <QtGui/QApplication>
...
int main() {
  // Erstelle QApplication-Objekt (immer als erstes!)
  // Erstelle GUI aus Widgets (Qt und/oder eigene)
  // Zeige Widget an (show())
  // Fuehre QApplication::exec() aus
}
```

- Qt sorgt dann dafür, dass Events vom Benutzer an die richtigen Teile der Applikation verteilt werden

Qt-Hello-World

hello-world.cpp

```
#include <QtGui/QApplication>
#include <QtGui/QLabel>
int main(int argc, char **argv) {
  QApplication app(argc,argv);
  QLabel label("Hello World");
  label.show();
  return app.exec();
}
```

- Übersetzen z.B mithilfe von `pkg-config`

```
g++ -o hello-world hello-world.cpp $(pkg-config --libs --cflags QtCore QtGui)
```

Exkurs Package-Config (pkg-config)

- pkg-config ist ein nettes Hilfstool um Abhängigkeiten zwischen Bibliotheken aufzulösen
- Basiert auf sog. Package-Config-Files (Endung .pc)
  - Liegen i.d.R immer in `${prefix}/lib/pkgconfig`
  - Sehr einfache Syntax
  - Werden verwendet um Compiler- und Linker-Flags zu bestimmen
  - Kann man auch leicht selbst schreiben

Beispiele

```
> pkg-config --cflags QtCore QtGui
-DQT_SHARED -I/usr/include/qt4 -I/usr/include/qt4/QtCore
> pkg-config --libs QtCore
-lQtCore
> pkg-config -variable moc_location QtCore
/usr/bin/moc-qt4
```

- Für mehr Informationen: manpage `man pkg-config`

Beispiel QtCore.pc

QtCore.pc (leicht vereinfacht)

```
prefix=/usr
exec_prefix=${prefix}
libdir=${prefix}/lib/i386-linux-gnu
includedir=${prefix}/include/qt4/QtCore
moc_location=/usr/bin/moc-qt4
Name:  Qtcore
Description:  Qtcore Library
Version:  4.8.1
Libs:  -L${libdir} -lQtCore
Cflags:  -DQT_SHARED -I/usr/include/qt4 -I${includedir}
```

- In Ubuntu 12.04: in `/usr/lib/i386-linux-gnu/pkgconfig`
- Doppelte Tokens und Standardpfade (wie `-I/usr/include`) werden von pkg-config automatisch entfernt

Hello-Qt Demo

Demo!

Mal kurz ausprobieren

GUI-Elemente

- Qt bietet eine Vielzahl an vorgefertigten GUI-Elementen ("Widgets"), die man miteinander kombinieren kann
  - Buttons
  - Radioboxes
  - SpinButtons
  - Slider

GUI-Elemente

- Qt bietet eine Vielzahl an vorgefertigten GUI-Elementen ("Widgets"), die man miteinander kombinieren kann
  - Buttons
  - Radioboxes
  - SpinButtons
  - Slider
  - ...
  - Datei-, Drucker-, Farbauswahldialoge, ...
  - Bilder ...
  - OpenGL-Widgets
  - Textein- und ausgabe, ...

Widgets

- Widgets haben Default-Verhalten
- Wenn man Default-Verhalten ändern will:
- Von Widget-Klasse ableiten und Methoden überschreiben (diese sind virtual)
- Oder: Event-Filter installieren ("nicht-invasiv")

Widgets

- Widgets haben Default-Verhalten
- Wenn man Default-Verhalten ändern will:
- Von Widget-Klasse ableiten und Methoden überschreiben (diese sind virtual)
- Oder: Event-Filter installieren ("nicht-invasiv")

Standart-Vorgehensweise

- Komponenten werden über sog. Signals und Slots "verbunden"

Signals und Slots

- Die meisten Qt-Widgets (und auch andere Komponenten) bieten eine Vielzahl von Signals und Slot
- Diese erlauben, eine Applikation dynamisch zu verschalten
- Basis-Klasse `QObject` hat Methode `connect`

Signals und Slots

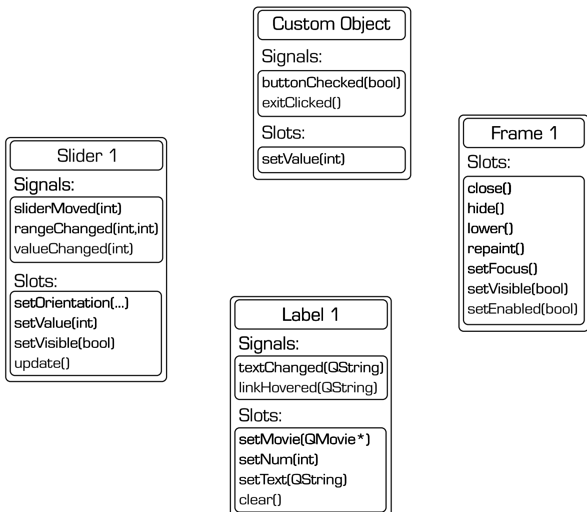
- Die meisten Qt-Widgets (und auch andere Komponenten) bieten eine Vielzahl von Signals und Slot
- Diese erlauben, eine Applikation dynamisch zu verschalten
- Basis-Klasse `QObject` hat Methode `connect`

connect-Syntax

```
connect(Src-Pointer,SIGNAL(params),Dst-Pointer,SLOT(params));
```

- Es gibt auch noch eine statische Variante
- `SIGNAL` und `SLOT` sind *Magic-Macros* welche von Qt definiert werden

Signals und Slots Grafik

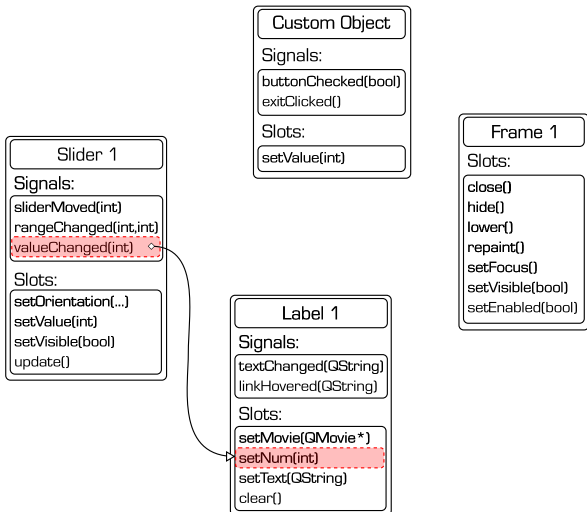


Frame 1

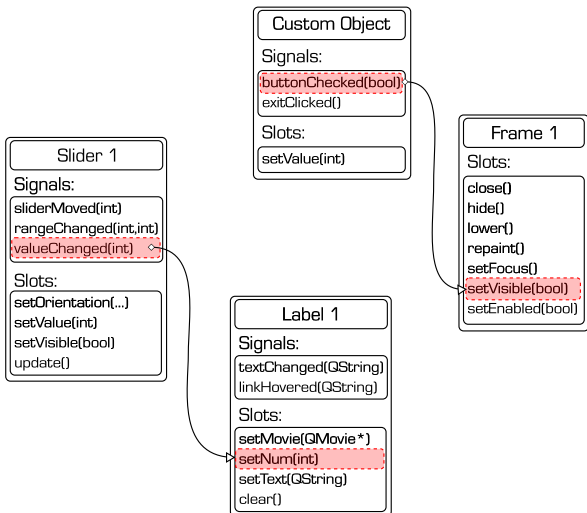
Slots:

- close()
- hide()
- lower()
- repaint()
- setFocus()
- setVisible(bool)
- setEnabled(bool)

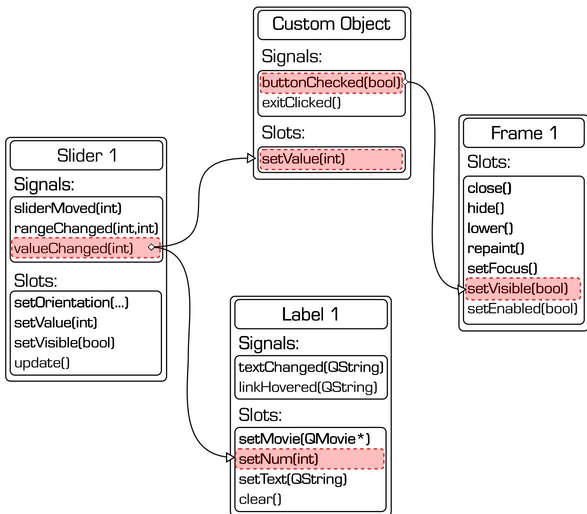
Signals und Slots Grafik



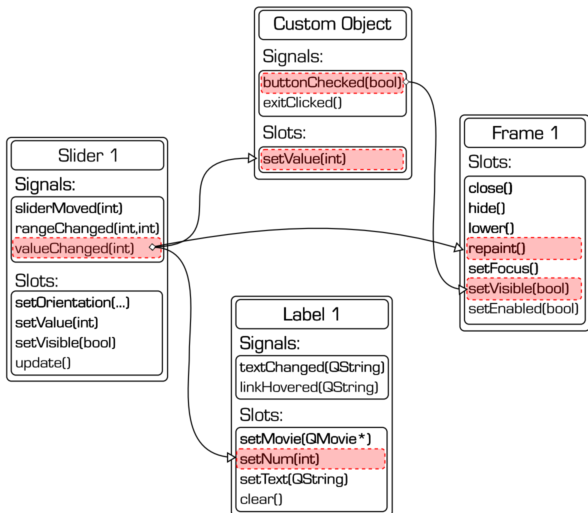
Signals und Slots Grafik



Signals und Slots Grafik



Signals und Slots Grafik



Implementationssache ...

- Slots werden im Klassenscope durch "`public slots:`" als solche deklariert (analog zu `public:` oder `private:`)
- Signale werden mittels "`signals:`" deklariert und sind immer `private`

Implementationssache ...

- Slots werden im Klassenscope durch "`public slots:`" als solche deklariert (analog zu `public:` oder `private:`)
- Signale werden mittels "`signals:`" deklariert und sind immer `private`

Wichtig!

- Signale werden von moc implementiert (dazu gleich mehr)
- Slots müssen von Klasse (also genau genommen von Programmierer selbst) implementiert werden

Slots (im Detail)

- Qt Slots sind einfache Funktionen
- Können also auch einfach 'per Hand' aufgerufen werden
- Deklariert mittels

```
[public|protected|private] slots:
```

Slots (im Detail)

- Qt Slots sind einfache Funktionen
- Können also auch einfach 'per Hand' aufgerufen werden
- Deklariert mittels

```
[public|protected|private] slots:
```

Achtung

Sichtbarkeitsqualifizierer gilt nur für direkten Aufruf → Signal-Slot-Verbindungsmechanismus berücksichtigt diese nicht!

Slots (im Detail)

- Auch virtual möglich
- (relativ) langsam (ca. 10 mal langsamer als Aufruf einer Callback-Funktion)
- Dafür natürlich deutlich erhöhte Flexibilität
  - Mehrere Verbindungen (auch über Kreuz)
  - Zur Laufzeit dynamisch

Große Frage

Wie wird das umgesetzt? $\rightarrow$ Aufschluss darüber liefert moc-Output Datei

Qt-Hello-World mit Signals und Slots

Quit-Button example

```
#include <QtGui/QApplication>
#include <QtGui/QPushButton>

int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QPushButton button("Quit");
    button.show();
    QObject::connect(&button, SIGNAL(clicked()), &app, SLOT(quit()));

    return app.exec();
}
```

- Beachte: `connect` arbeitet immer mit Pointern
- I.d.R. haben Klassen eine Vielzahl von Slots und Signals

Aber Achtung!

Die Macros `SIGNAL` und `SLOT` interpretieren ihr Argument als **string**
$\Rightarrow$ Schreibfehler werden erst zur Laufzeit erkannt

Noch ein Beispiel: Slider und Anzeige

slider.cpp

```
#include <QtGui/QApplication>
#include <QtGui/QLabel>
#include <QtGui/QSlider>
#include <QtGui/QBoxLayout>

int main(int argc, char **argv) {
    QApplication app(argc, argv);

    // Widgets
    QWidget container;
    QSlider slider(Qt::Horizontal);
    QLabel label(QString::number(slider.value()));

    // Layout
    container.setLayout(new QBoxLayout(QBoxLayout::TopToBottom));
    container.layout()->addWidget(&slider);
    container.layout()->addWidget(&label);

    // verbinden ..
    QObject::connect(&slider, SIGNAL(valueChanged(int)), &label, SLOT(setNum(int)));

    // anzeigen und Event-Loop starten
    container.show();
    return app.exec();
}
```

Slots selbst deklarieren

- In nächsten Beispiel soll ein Slot deklariert und definiert werden
- Da Slots intern herkömmliche Funktionen sind, sind hier keine Besonderheiten zu beachten
- Es soll eine Klasse `SlotToStream` implementiert werden
  - Die `SlotToStream` Instanz soll 3 (überladene) Slots aufweisen
  - Jeder dieser Slots soll empfangende Daten in einen Stream schreiben

Slot-Beispiel: SlotToStream

SlotToStream.h

```
#ifndef SLOT_TO_STREAM_H
#define SLOT_TO_STREAM_H

#include <QObject>
#include <iostream>
// Merke: um Signals und Slots zu verwenden, muessen wir von QObject
// erben. Hier direkt -- ist aber auch indirekt moeglich!
class SlotToStream : public QObject {
  Q_OBJECT // Magisches Qt-Macro: Definiert sog. Meta-Object-Daten
           // und -Funktionen welche fuer 'connect' benoetigt werden
           // wenn eine Qt-Klasses eigene Signals oder Slots definier
           // muss dieses Macro enthalten sein
  std::ostream &m_stream;
  public:
  SlotToStream(std::ostream &stream) : m_stream(stream){}
  public slots:
  void toStream(const int &i){ m_stream << i << std::endl; }
  void toStream(const float &f){ m_stream << f << std::endl; }
  void toStream(const double &d){ m_stream << d << std::endl; }
  void toStream(const std::string &s){ m_stream << s << std::endl; }
};
#endif
```

Verwendung der SlotToStream-Klasse

- Einfache Anwendung: Den aktuellen wert eines Sliders ausgeben
- Vollständige Signal-/Slot-Signatur notwendig
- Signatur muss *kompatibel* sein
- Es können auch Verbindungen hergestellt werden, in denen nur die ersten N Argumente verwendet werden z.B.:
  `SIGNAL(mySignal(char,int))` → `SLOT(mySlot(char))`

SlotToStreamMain.cpp

```
#include "SlotToStream.h"
#include <QSlider>
#include <QApplication>
int main(int n, char**ppc){
  QApplication app(n,ppc);
  QSlider slider(Qt::Horizontal);
  SlotToStream printer(std::cout);
  QObject::connect(&slider,SIGNAL(valueChanged(int)),
                   &printer,SLOT(toStream(int)));
  slider.show();
  return app.exec();
}                                              // kompilieren: kommt gleich!
```

Signals

- Neben Slots können natürlich auch Signals selbst definiert werden
- Signale sind ebenfalls Methoden
- Signale werden mittels folgendem Modifikator innerhalb einer Klasse kenntlich gemacht

```
signals:
```

- Eine Klasse die Signale anbietet, muss ebenfalls `QObject` beerben und mit dem `Q_OBJECT` Macro versehen werden
- Signals sind immer automatisch private $\rightarrow$ sie werden von dem Objekt *emmitiert*

Keine Implementation von Signals

- Im Gegensatz zu Slots sind Signals allerdings keine *normalen* Funktionen
- Signals müssen (und dürfen) nicht implementiert werden
- Signals werden immer automatisch durch Qt's sog. *Meta Object Compiler* kurz **MOC** implementiert
- Implementation der Signale und der von dem `Q_OBJECT`-Macro deklarierten Symbole, befindet sich dann immer im zugehörigen moc-File
- .moc-File muss mit übersetzt und gelinkt werden
- Symbole werden dann gefunden

Der Meta-Object-Compiler (moc)

- Um eigene Signals/Slots zu implementieren, wird `moc` verwendet
- `moc` ist ein Zusätzlicher Präprozessor

Der Meta-Object-Compiler (moc)

- Um eigene Signals/Slots zu implementieren, wird `moc` verwendet
- `moc` ist ein Zusätzlicher Präprozessor
- Erzeugt Quellcodedatei aus Klassendefinition (Header)
- Erzeugte Datei muss mit anderen Quellcodedateien übersetzt und gelinkt werden

Der Meta-Object-Compiler (moc)

- Um eigene Signals/Slots zu implementieren, wird `moc` verwendet
- `moc` ist ein Zusätzlicher Präprozessor
- Erzeugt Quellcodedatei aus Klassendefinition (Header)
- Erzeugte Datei muss mit anderen Quellcodedateien übersetzt und gelinkt werden
- Klasse, die Signals/Slots bereitstellt, muss von `QObject` `public` erben
- Klasse muss das Makros `Q_OBJECT` enthalten (ohne anschließendes Semikolon)

MOC: Anwendungsbeispiel

Das Beispielprogramm SlotToStream besteht aus 2 Source-Dateien

- SlotToStream.h: enthält die Klassendefinition (alles inline)
- main.cpp: verwendet die SlotToStream-Klasse

MOC: Anwendungsbeispiel

Das Beispielprogramm SlotToStream besteht aus 2 Source-Dateien

- SlotToStream.h: enthält die Klassendefinition (alles inline)
- main.cpp: verwendet die SlotToStream-Klasse

- moc wird auf `SlotToStream.h` angewendet

```
> moc-qt4 -o moc_SlotToStream.cpp myWidget.h
```

MOC: Anwendungsbeispiel

Das Beispielprogramm SlotToStream besteht aus 2 Source-Dateien

- SlotToStream.h: enthält die Klassendefinition (alles inline)
- main.cpp: verwendet die SlotToStream-Klasse

- moc wird auf `SlotToStream.h` angewendet

```
> moc-qt4 -o moc_SlotToStream.cpp myWidget.h
```

- Übersetzen und Linken

```
> g++ main.cpp moc_SlotToStream.cpp $(pkg-config --libs --cflags QtCore
QtGui)
```

(Alltägliches) Beispiel

- Counter für eine *Doomsday-machine* (Weltuntergangsmaschine)
- Soll von konfigurierbarer Zahl bis 0 herunterzählen (mittels Klasse `QTimer`)
- Bei Erreichen von 0 wird Methode `destroyWorld()` aufgerufen.

Rezept : Doomsday-Machine-in-C++

Man nehme ...

- 1 `QSpinButton`, um die Zahl der Sekunden einzustellen
- 1 `QPushButton`, um den Countdown zu starten
- 1 `QPushButton`, um den Countdown zu stoppen
- 1 `QTimer`, für das *Timing*
- 1 `QLabel`, um die verbliebene Zeit anzuzeigen
- 1 `QVBoxLayout`, um die Widgets anzuordnen
- 1 Instanz von Klasse `Doomsday` um das ganze zu koordinieren

Rezept : Doomsday-Machine-in-C++

Man nehme ...

- 1 `QSpinButton`, um die Zahl der Sekunden einzustellen
- 1 `QPushButton`, um den Countdown zu starten
- 1 `QPushButton`, um den Countdown zu stoppen
- 1 `QTimer`, für das *Timing*
- 1 `QLabel`, um die verbliebene Zeit anzuzeigen
- 1 `QVBoxLayout`, um die Widgets anzuordnen
- 1 Instanz von Klasse `Doomsday` um das ganze zu koordinieren

Demo

Gleich live! (Da die ganzen Dateien schlecht auf die Folien passen) Dateien sind über die Webseite verfügbar

Sonstiges

- QObject-Instanzen können immer mit einem *Parent*-Argument erstellt werden
- Daraus ergibt sich ein Zugehörigkeits-Baum
- Wenn Parent gelöscht wird, werden automatisch vorher alle Children gelöscht
- Erspart eine Menge von Destruktoraufrufen

qmake: Ein ganz nettes (meta-) build-tool

- QMake vereinfacht die Arbeit mit MOC
- QMake ist ein "Meta-Build-Tool"
- Erstellt Makefile aus Projektdefinition
- Projektdefinitionsfile kann in einfachen Fällen auch automatisch erstellt werden
- Beispiel für Doomsdaymachine

Pro/Contra qmake

Vorteile

- Plattformübergreifend
- Kann auch Bibilotheken erstellen
- Sehr komplex und mächtig
- Hierarchische Builds
- Automatische Detektion von Abhängigkeiten

Pro/Contra qmake

Vorteile

- Plattformübergreifend
- Kann auch Bibilotheken erstellen
- Sehr komplex und mächtig
- Hierarchische Builds
- Automatische Detektion von Abhängigkeiten

Nachteile

- High Level – Zugang erschwert Finetuning von Makefiles
- Sehr Qt-Spezifisch
  - Ergibt immer eine Abhängigkeit von Qt
  - Fällt nur ins Gewicht, wenn man **nicht**-Qt-Applikationen oder Bibliotheken bauen möchte

QWidgets (reloaded)

- Qt hat Vielzahl von vorgefertigten Widgets

QWidgets (reloaded)

- Qt hat Vielzahl von vorgefertigten Widgets
- Häufig benötigt man allerdings spezielles Widget
  - Anzeige von Industrieprozessen
  - Spiele

QWidgets (reloaded)

- Qt hat Vielzahl von vorgefertigten Widgets
- Häufig benötigt man allerdings spezielles Widget
  - Anzeige von Industrieprozessen
  - Spiele
- Eigene Widget-Klassen
  - Erben von `QWidget` (oder einer geeigneten Unterklasse evtl. z.B `QGLWidget`)
  - Methoden, die angepasst werden sollen, können überschrieben werden (dazu gleich noch ein Beispiel)
  - Falls gewünscht, können auch eigenen Slots/Signals hinzugefügt werden

QWidgets (reloaded)

- Qt hat Vielzahl von vorgefertigten Widgets
- Häufig benötigt man allerdings spezielles Widget
  - Anzeige von Industrieprozessen
  - Spiele
- Eigene Widget-Klassen
  - Erben von `QWidget` (oder einer geeigneten Unterklasse evtl. z.B `QGLWidget`)
  - Methoden, die angepasst werden sollen, können überschrieben werden (dazu gleich noch ein Beispiel)
  - Falls gewünscht, können auch eigenen Slots/Signals hinzugefügt werden
- Oder: Zusammenstückeln aus bereits existierenden Komponenten