

Laboraufgabe Qt

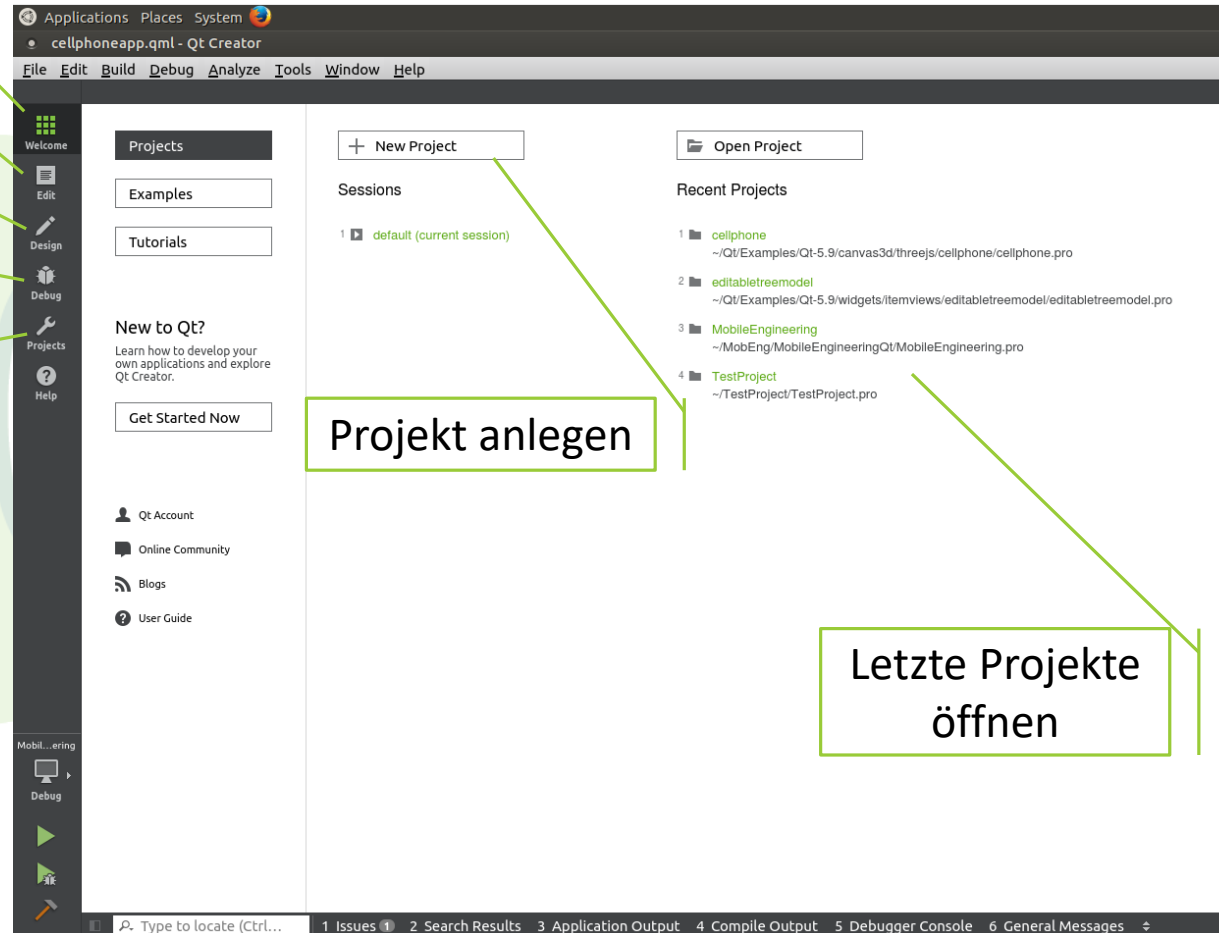
HENNER BENDIG

SVEN OLE LUX

04.04.2017

QtCreator

- Startseite
- Quellcodes
- GUI Designer
- Debugger
- Projekt-einstellungen

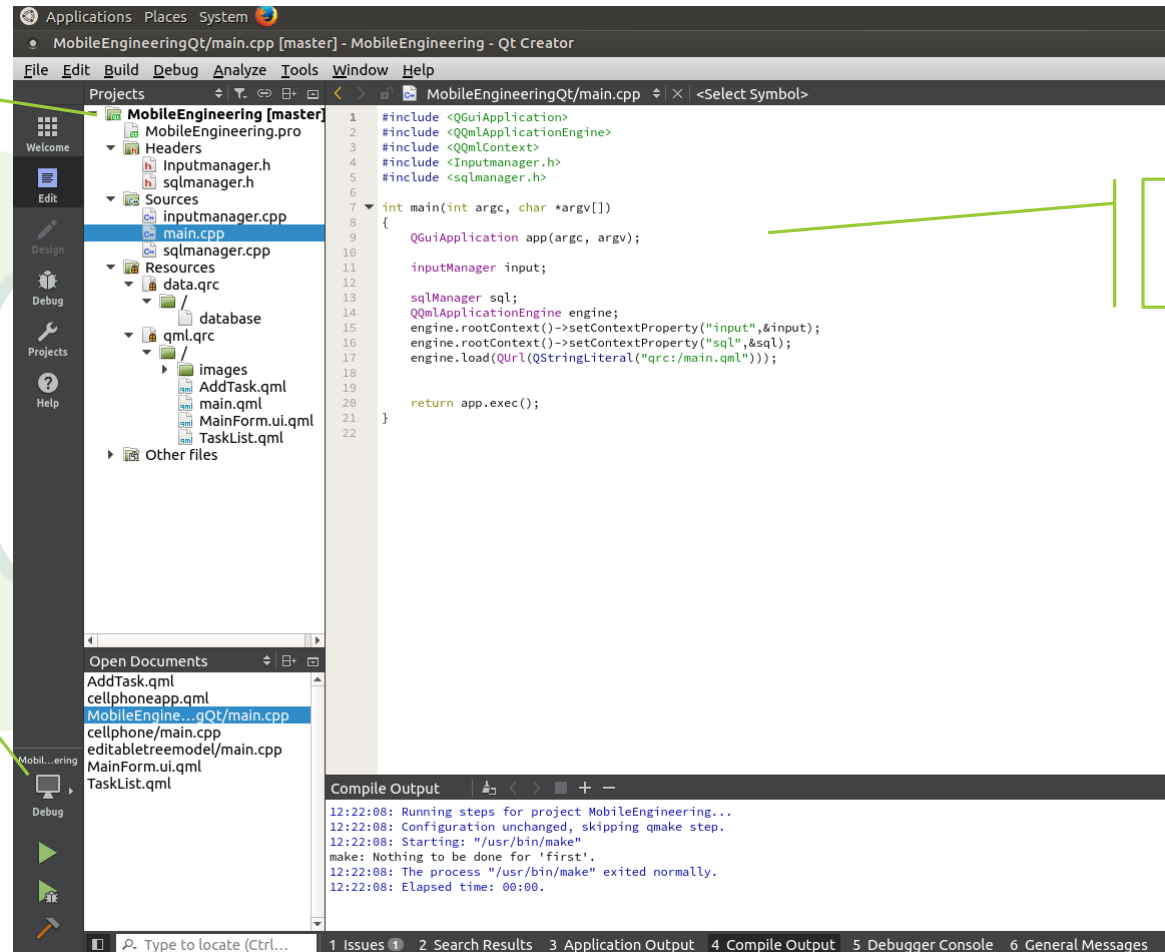


QtCreator

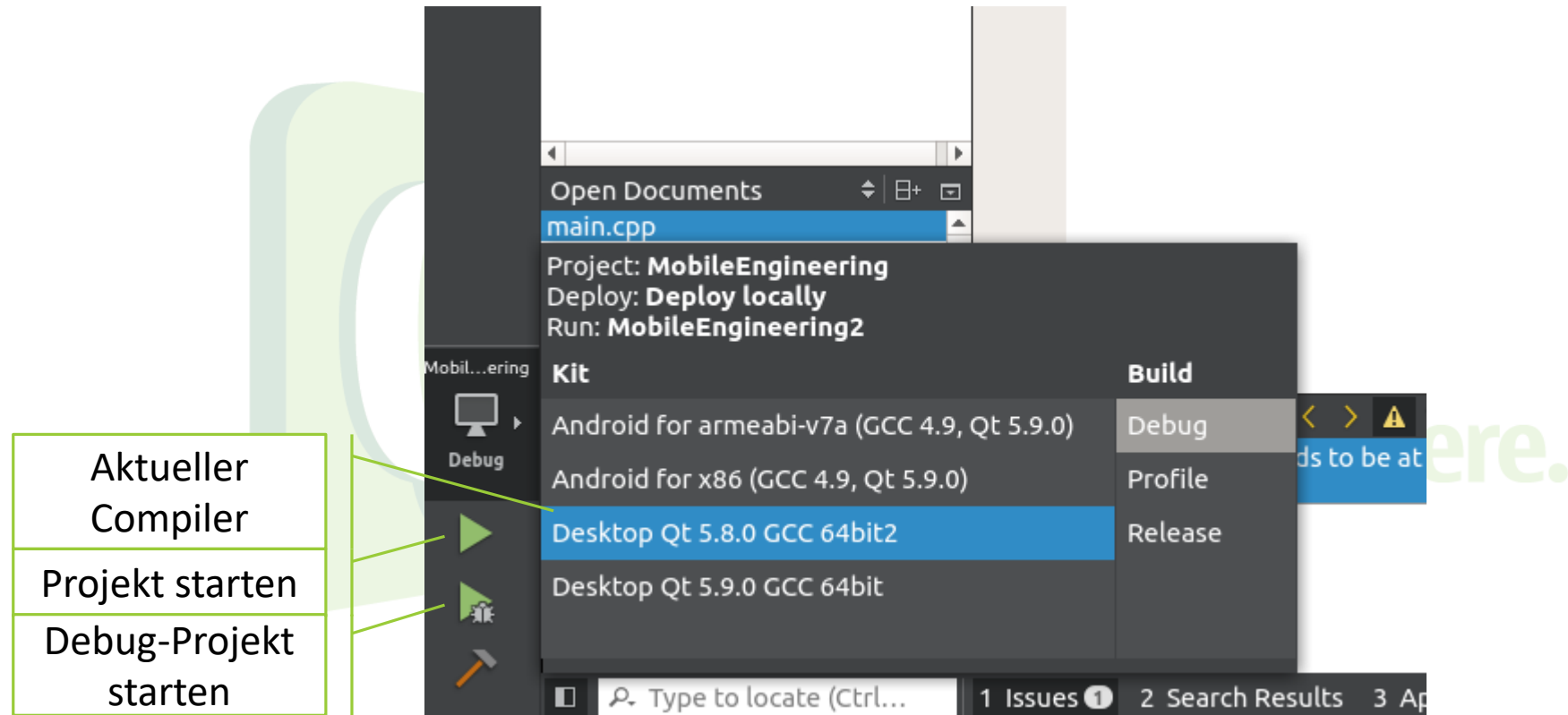
Aktives Projekt

Compiler-
auswahl

Hier coden ...

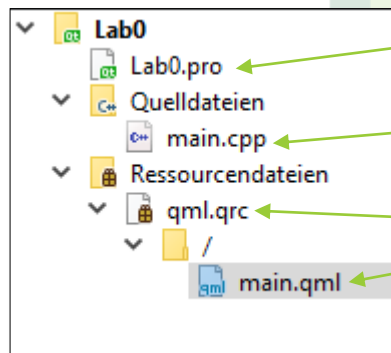


QtCreator



Aufgabe Lab 0

- Öffne das Projekt „Lab0“
 - Von der Startseite von QtCreator („Willkommen“, siehe [Abbildung 1](#)) auf „Projekt öffnen“ klicken
 - Im Ordner des Projektes „Lab0“ die Datei „Lab0.pro“ öffnen
- Folgende Projekthierarchie sollte sich öffnen:



Das ist die Projektdatei; hier wird angegeben welche Module, Unterdateien, Ressourcen Bestandteil des Projektes sind

Im Ordner Quelldateien liegen die C++-Dateien. In diesem Fall nur die Main.cpp, die den Einstiegspunkt der Anwendung definiert

„qml.qrc“ ist ein Container, qrc steht für Qt-Ressources-Container

Unter Ressourcen werden alle Dateien gepackt, die mit Bestandteil der Anwendung sein sollen. Z.B. Images, Datenbanken aber auch die QML-Dateien, die die UI definieren

Aufgabe Lab 0

- Der Inhalt der C++-Datei sieht wie folgt aus:

```
1  #include <QGuiApplication>
2  #include <QQmlApplicationEngine>
3
4  int main(int argc, char *argv[])
5  {
6      QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
7      QGuiApplication app(argc, argv);
8
9      QQmlApplicationEngine engine;
10     engine.load(QUrl(QLatin1String("qrc:/main.qml")));
11
12     return app.exec();
13 }
14
```

Imports der Anwendung

Aktivieren des High DPI Scalings (für hochauflösende Displays)

Erzeugen der Fensteranwendung

Laden der gewählten QML-Datei (bestimmt die UI)



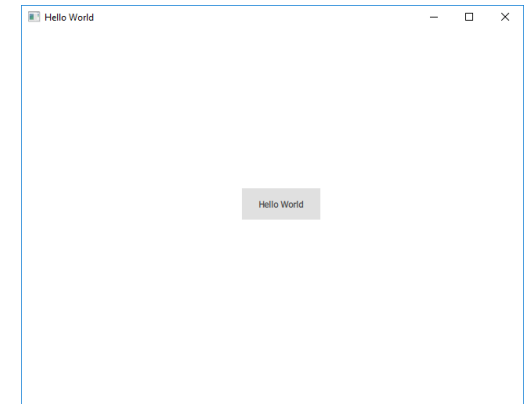
ess
mobile
y everywhere.

Aufgabe Lab 0

- Öffne die „main.qml“ mit einem Doppelklick
- Im Code-Editor erstellst du innerhalb der geschweiften Klammern des ApplicationWindow ein neues Item mit „Item{ }“. Innerhalb der geschweiften Klammern können nun die UI-Elemente erzeugt werden
- Füge einen Button („Button{ }“) hinzu. Wenn du zum Designer wechselst, sollte dort nun ein leerer Button in der Zeichenfläche zu sehen sein
 - Änderst du nun im Designer z.B. die Position oder Eigenschaften des Buttons, wird der entsprechende Code in der QML-Datei erzeugt
- Beschrifte den Button. Entweder im Designer (rechte Seite unter „Text:“) oder per QML-Editor (text: „*ButtonText*“)

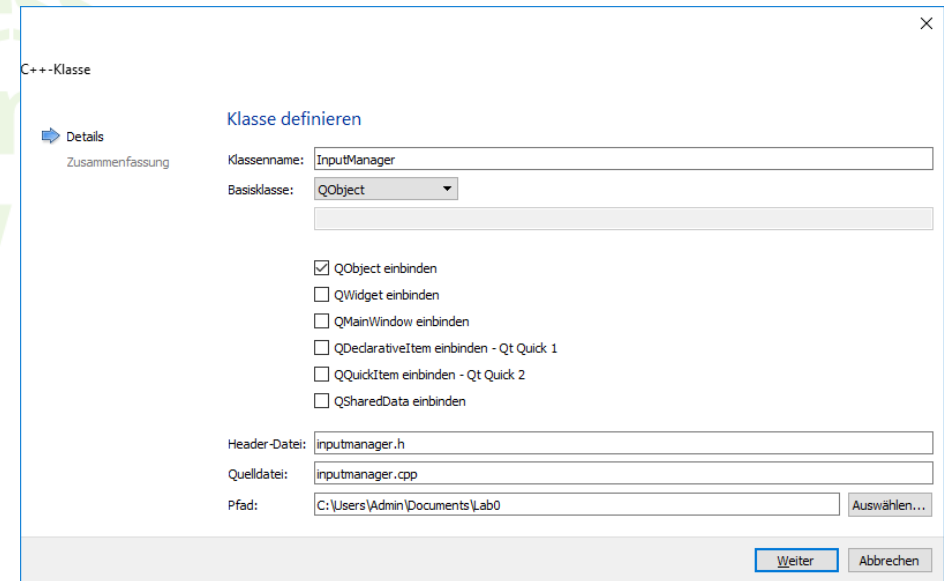
Aufgabe Lab 0

- Führe das Projekt auf dem Desktop aus. Es sollte ein weißes Fenster mit deinem Button, mit dem gewählten Text erscheinen
- Füge nun noch ein TextField neben dem Button ein. Dieses benötigt eine ID. Diese vergibst du im Designer wieder auf der rechten Seite oder im Editor mit dem Attribut „id:“
- In dem Button kann nun das von Qt integrierte Signal „clicked“ abgefangen werden. Ein Signal wird abgefragt mittels „onSignalname“, in diesem Fall : „onClicked“. In geschweiften Klammern hinter dem Signal kann nun Javascript-Code ausgeführt werden. Mache eine Konsolenausgabe, dass der Button gedrückt wurde:
`console.log(„Button “ + this.text + „ wurde gedrückt! Eingabe: “ + inputTextID.text)`
- Hieran ist folgendes zu erkennen: in einem Objekt können auf die eigenen Attribute mittels „this“ zugegriffen werden. Mittels der ID eines anderen Objektes können auf die Attribute des anderen Objektes zugegriffen werden



Aufgabe Lab 0

- Führe die Anwendung nochmals aus. Wenn der Button jetzt gedrückt wird, sollte eine Konsolenausgabe erzeugt werden:
„Button Hello World wurde gedrückt! Eingabe: Texteingabe“
- Erzeuge eine neue C++-Klasse, mittels Rechtsklick auf das Projekt
> Hinzufügen... > C++ > C++-Klasse > Auswählen
- Die Klasse soll auf das Button-Signal reagieren, daher nennen wir sie „InputManager“. Außerdem sollte die Basisklasse QObject sein. Stelle sicher, dass der Haken bei „QObject einbinden“ gesetzt ist
- Klicke auf „Weiter“ und füge sie dem Projekt „Lab0“ hinzu



Aufgabe Lab 0

- In der Header-Datei (inputmanager.h) wird nun ein Slot angelegt, hierfür fügen wir unter „public slots“ folgende Funktionsdefinition ein:
void output(QString ouputtext);
- In der C++-Datei (Inputmanager.cpp) wird die Slot-Funktion definiert. Die Funktion soll den übergebenen Text einfach in die Debug-Konsole ausgeben. Dafür muss QDebug eingebunden werden: #include <QDebug>
- Die Funktion sieht dann wie folgt aus:

```
void InputManager::output(QString ouputtext)
{
    qDebug() << ouputtext;
}
```

```
1  #ifndef INPUTMANAGER_H
2  #define INPUTMANAGER_H
3
4  #include <QObject>
5
6  class InputManager : public QObject
7  {
8      Q_OBJECT
9  public:
10     explicit InputManager(QObject *parent = 0);
11
12     signals:
13
14     public slots:
15         void output(QString ouputtext);
16 };
17
18 #endif // INPUTMANAGER_H
19
```

Aufgabe Lab 0

- Diese Funktion soll vom Button aufgerufen werden. Hierfür muss die Klasse erstmal der QML-Datei bekannt gemacht werden. Hierfür muss eine Ergänzung in der „main.cpp“ erfolgen
- Öffne die „main.cpp“ und binde den InputManager-Header (inputmanager.h) ein. Zusätzlich wird das Modul „QQmlContext“ benötigt:
`import „inputmanager.h“
import <QQmlContext>`
- Erzeuge eine Instanz des InputManagers: `InputManager inputMgr;`
- Bevor die QML-Datei geladen wird, muss die Instanz des InputManagers dem RootContext hinzugefügt werden, dies erfolgt über diesen Aufruf:
`engine.rootContext()->setContextProperty("inputManager", &inputMgr);`
- So sollte die main anschließend etwa aussehen:
- „inputManager“ ist dann die ID, über dem die Instanz des InputManagers in der QML-Datei angesprochen werden kann

```
int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;

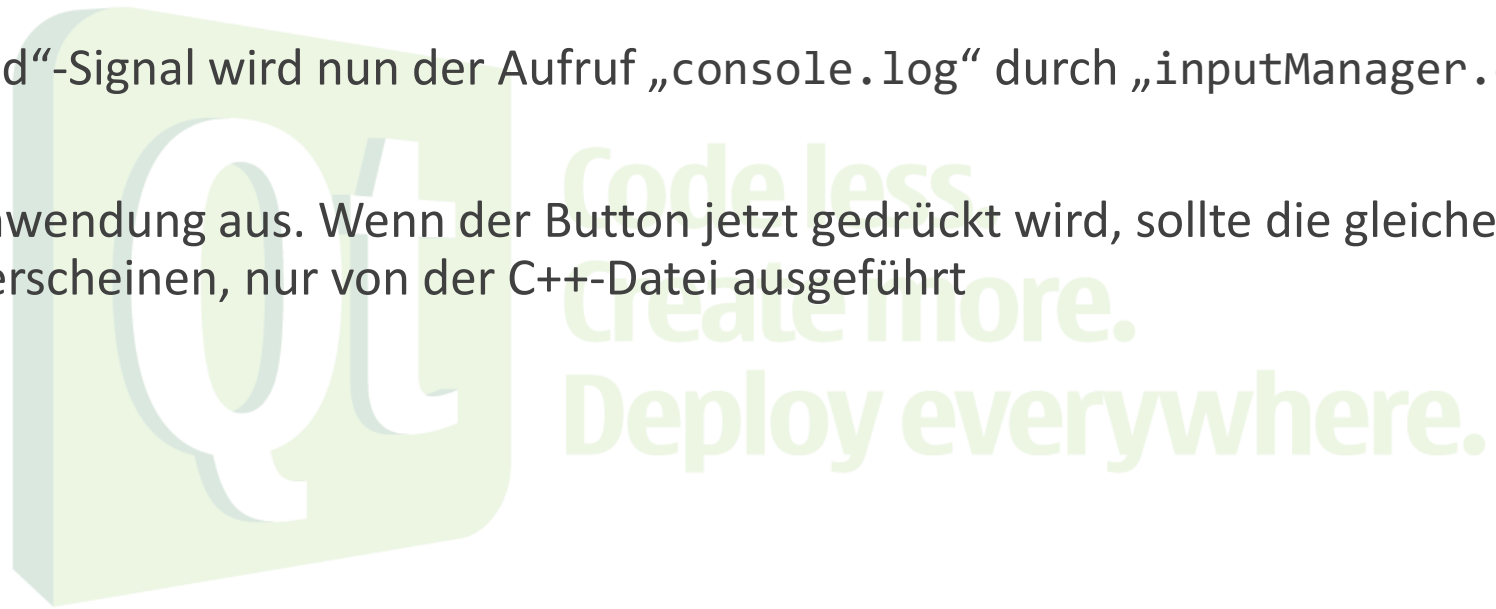
    InputManager inputMgr;
    engine.rootContext()->setContextProperty("inputManager", &inputMgr);

    engine.load(QUrl(QLatin1String("qrc:/main.qml")));

    return app.exec();
}
```

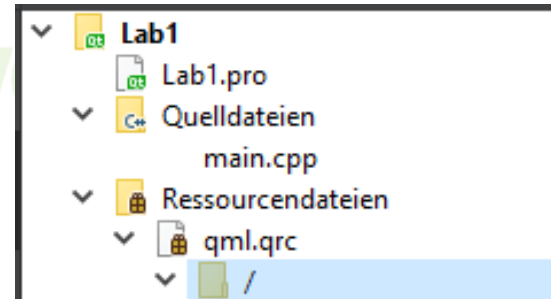
Aufgabe Lab 0

- Nun muss der Aufruf aus dem Button innerhalb der QML-Datei erfolgen. Es wird wieder in die „main.qml“ gewechselt
- Im „onClicked“-Signal wird nun der Aufruf „console.log“ durch „inputManager.output“ ersetzt
- Führe die Anwendung aus. Wenn der Button jetzt gedrückt wird, sollte die gleiche Ausgabe in der Konsole erscheinen, nur von der C++-Datei ausgeführt



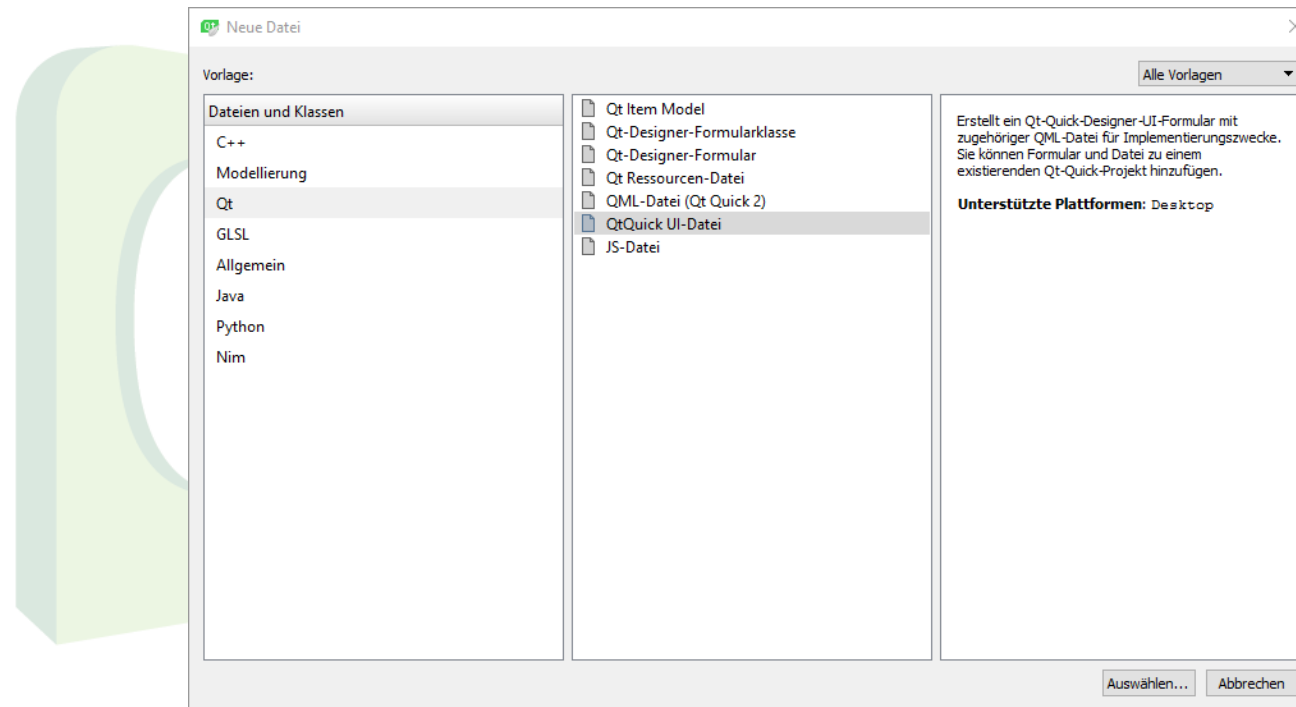
Aufgabe Lab 1a – Hello World!

- Öffne das bestehende Projekt Lab1
- Lasse die Anwendung für den Desktop kompilieren und ausführen
 - Hinweis: Der Compiler läuft durch, aber es wird keine Anwendung erscheinen
- Füge über das Kontext-Menü „Hinzufügen“ ein neues UI-Fenster hinzu
 - best practise-Tipp: Klappe in der Hierarchie „Ressourcendateien/qml.qrc“ auf und öffne das Kontextmenü auf dem „/“-Ordner mit einem Rechtsklick
 - Im Kontextmenü „Hinzufügen klicken“



Aufgabe Lab 1a – Hello World!

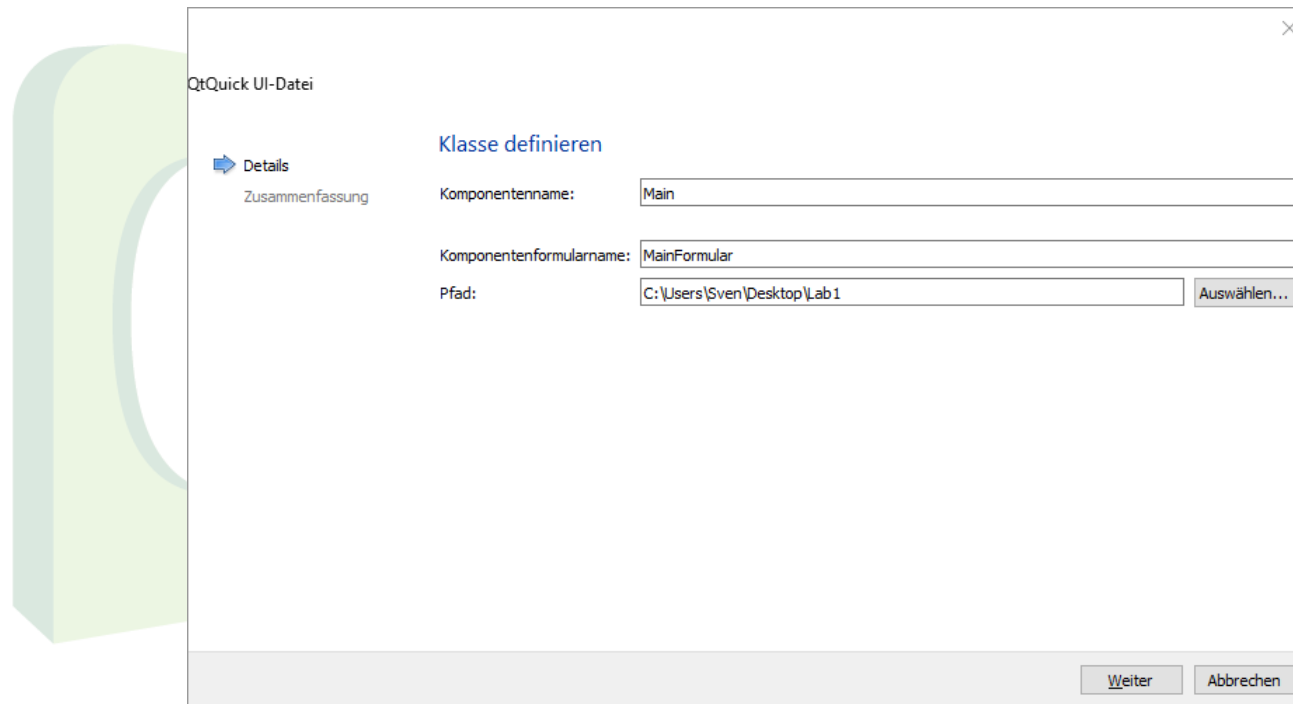
- Qt > QtQuick UI-Datei



ere.

Aufgabe Lab 1a – Hello World!

- Komponentename: Main, Komponentenformularname: MainFormular (ohne Bindestrich!!)



The screenshot shows a dialog box titled "QtQuick UI-Datei" with a close button (X) in the top right corner. On the left, there are two tabs: "Details" (selected, with a blue arrow icon) and "Zusammenfassung". The main area is titled "Klasse definieren" in blue. It contains three input fields: "Komponentenname:" with the value "Main", "Komponentenformularname:" with the value "MainFormular", and "Pfad:" with the value "C:\Users\Sven\Desktop\Lab 1". To the right of the "Pfad:" field is a button labeled "Auswählen...". At the bottom right, there are two buttons: "Weiter" and "Abbrechen".

ere.

Aufgabe Lab 1a – Hello World!

- Komponente „Main.qml“ enthält vorrangig Logik (z.B. Javascript), das Formular „MainFormular.ui.qml“ enthält die UI-Elemente (und kann kein JS)



Main.qml



MainFormular.ui.qml

- Die zugehörigen Formulare werden innerhalb der qml einmal erzeugt mit „FomularName { }“ (siehe Beispiel)

```
1  import QtQuick 2.4
2  import QtQuick.Controls 2.1
3
4  ▼ ApplicationWindow {
5      id: mainWindow
6      visible: true
7      width: 800
8      height: 400
9
10  ▼ MainFormular {
11
12      }
13  }
14
```

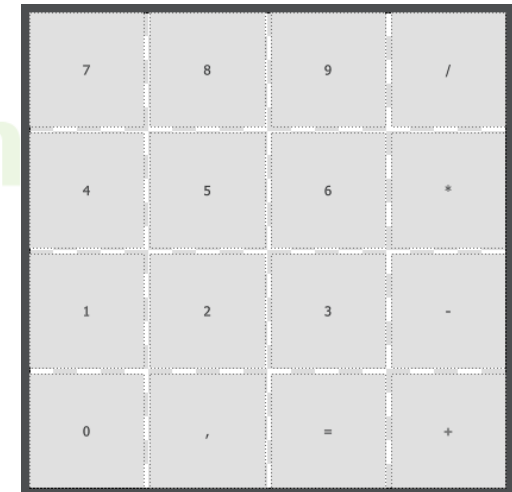
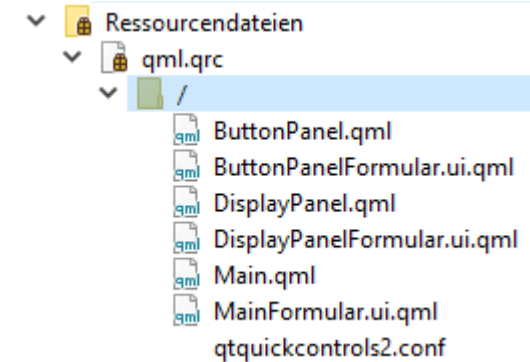

Aufgabe Lab 1a – Hello World!

- Füge in „main.qml“ ein `ApplicationWindow{}` hinzu, und setze es auf „visible: true“. Dafür führe folgende Schritte durch:
 - Es muss hierfür `QtQuick.Controls 2.1` importiert werden
 - Das `MainFormular` sollte innerhalb des `ApplicationWindow` aufgerufen werden
 - Kommentiere in der `Main.cpp` die Zeile 14 (`engine.load(...)`) ein
 - Wenn die Anwendung jetzt ausgeführt wird, startet ein weißes Fenster
- Gib dem Fenster eine Höhe und Breite und füge ein Textfeld (per Code oder Designer) ein
 - z.B. 400x800 mit „Hello World!“
 - <http://doc.qt.io/qt-5/qml-qtquick-text.html>

```
1 import QtQuick 2.4
2 import QtQuick.Controls 2.1
3
4 ApplicationWindow {
5     id:mainWindow
6     visible: true
7     width: 800
8     height: 400
9
10 MainFormular {
11
12     }
13 }
14
```


Aufgabe Lab 1b – Calculator

- Füge zwei weitere UI-Forms hinzu: ButtonPanel und DisplayPanel
- Passe die Formular-Dateien so an, dass:
 - Das Item des MainFormular das Eltern-Element ausfüllt
 - Siehe hierfür in die Doku: <http://doc.qt.io/qt-4.8/qml-item.html>, Stichwort: anchors
 - Die Items aus dem Button- und dem Display-Form 400x400px groß sind
- Dem ButtonFormular füge folgende Buttons in dieser Sortierung hinzu:
 - Tipp: Nutze GridLayout (`import QtQuick.Layouts 1.1`)
und Controls (`import QtQuick.Controls 2.1`)
 - Die Buttons sollen sich an die entsprechende Größe anpassen



Aufgabe Lab 1c – Calculator

- Das DisplayFormular soll wie folgt aussehen:
 - Auch die Textfelder sollen sich an die zur Verfügung stehende Größe anpassen
 - Dazu muss zunächst nur 2 Textfelder erzeugt werden ähnlich zu Aufgabe 1a

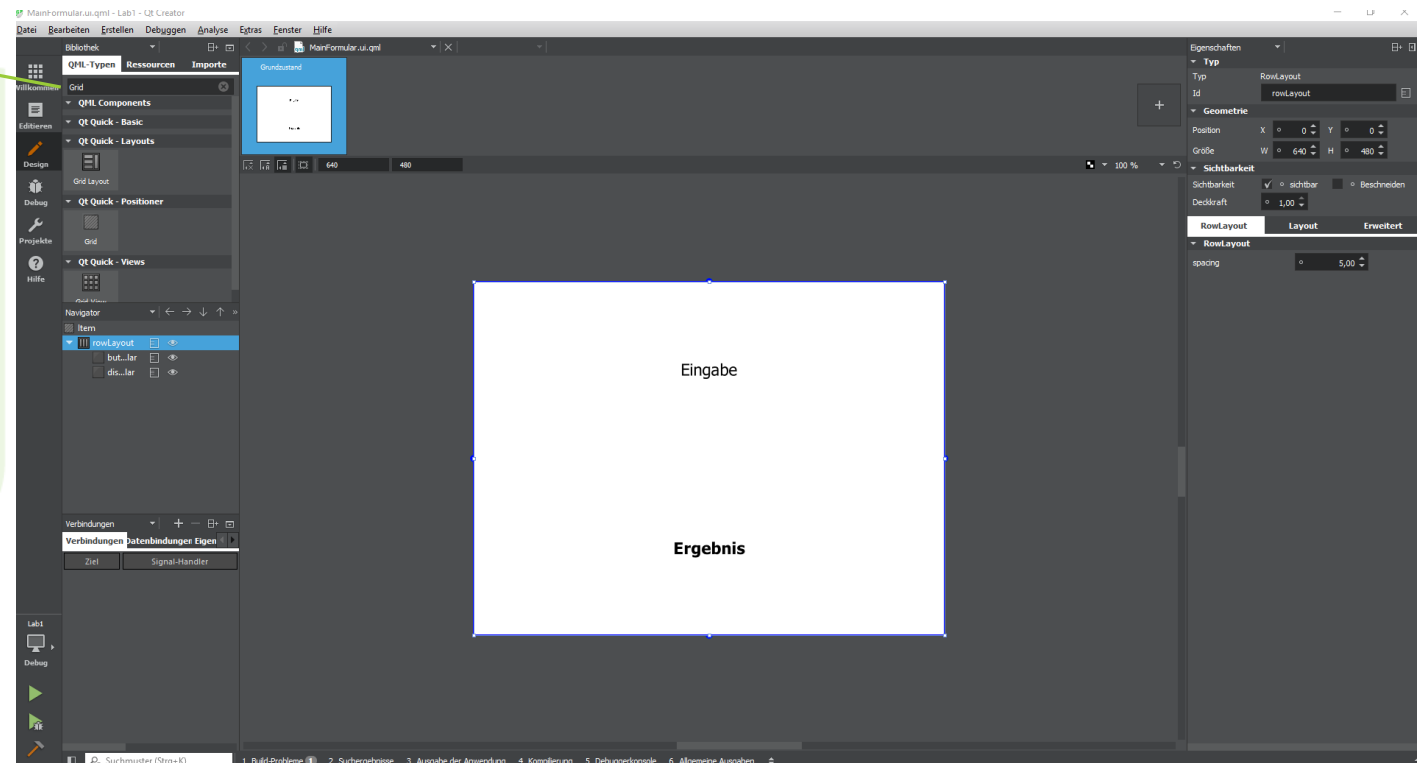


The diagram shows a rectangular box representing a calculator display. It is divided into two horizontal sections by a dotted line. The top section is labeled 'Eingabe' (Input) and the bottom section is labeled 'Ergebnis' (Result). The text is centered in each section. In the background, there is a large, stylized green 'Q' and the text 'ore. everywhere.' in a light green font.

Aufgabe Lab 1c – Calculator

- Diese Textelemente können z.B. per GridLayout angeordnet werden wenn „import QtQuick.Layouts 1.1“ angegeben ist

Suchleiste für QML-Elemente



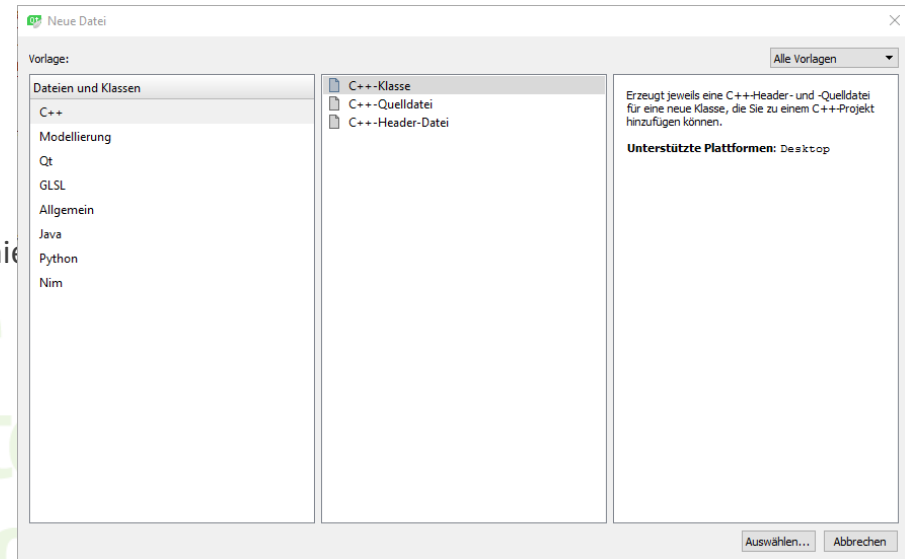
Aufgabe Lab 1c – Calculator

- Füge die QML-Dateien „DisplayPanel“ und „ButtonPanel“ mittels Layouts zum MainFormular hinzu, dass sie nebeneinander angezeigt werden
 - Wichtig: Im MainFormular erzeugst du je ein Item, dem du die Formulare (Buttons und Displays) hinzufügst
 - Ist im MainFormular „import QtQuick.Layouts 1.1“ angegeben können die Elemente in einem Layout angeordnet werden
 - Zur Erinnerung: Elemente können per „Elementname { }“ ins QML hinzugefügt werden
 - Das „Hello World!“ kannst du jetzt löschen 😊

Eingabe	7	8	9	/
	4	5	6	*
	1	2	3	-
Ergebnis	0	,	=	+

Aufgabe Lab 1d – Calculator

- Erzeuge nun eine C++-Klasse „Calculator“
 - Rechtsklick auf das Projekt -> hinzufügen
 - Bei „Klasse definieren“ „QObject Einbinden“ angeben
 - Es wird eine .h und .cpp Datei erzeugt
 - Im .h „Header“ werden die Methodennamen und Klassenfelder definiert
 - Im .cpp werden die Methoden implementiert
- Inkludiere die Klasse „QObject“ (`#include <QObject>`), lasse den Calculator von QObject erben, in die erste Zeile der Klasse wird das Präfix „Q_OBJECT“ geschrieben
- Jede C++ Klasse braucht das „Q_Object“ um Signale, Slots und Klassen Von dem Qt-Framework benutzen zu können



```
1  #ifndef CALCULATOR_H
2  #define CALCULATOR_H
3
4  #include <QObject>
5
6  class Calculator : public QObject
7  {
8      Q_OBJECT
9  public:
10     Calculator();
```

Aufgabe Lab 1d – Calculator

- Schreibe die Logik für einen Taschenrechner
- Die C++ Klasse wird von uns vorgegeben, da es ohne Wissen in Qt schwer ist nachzuvollziehen
 - Wenn ihr wollt könnt ihr es gerne selber probieren
 - Die Klasse Calculator soll Funktionen haben, die auf die verschiedenen Buttons reagiere
 - Funktion zum behandeln von Zahlen
 - Funktion um den Rechenoperator zu behandeln
 - Funktion zum Zurücksetzen der Felder nach der Berechnung
 - Funktion zum Berechnen der Werte
 - Siehe Abbildung: So kann die Headerdatei aussehen
 - Füge im Header „signals: calcDone(float calcResult);“ hinzu
 - Am Ende der Berechnungsmethode fügst du folgenden Signalaufruf hinzu:
„emit calcDone(result);“
 - Auf das Signal „calcDone“ kann nun reagiert werden
- Inkludiere die „calculator.h“ in die main.cpp und kommentiere die entsprechende Zeile ein, um den Calculator für den QML-Context freizugeben (Zeile 11)

```
1  #ifndef CALCULATOR_H
2  #define CALCULATOR_H
3
4  #include <QObject>
5
6  class Calculator : public QObject
7  {
8      Q_OBJECT
9  public:
10     Calculator();
11 public slots:
12     void setOperator(QString value);
13     void setNumber(QString value);
14     void reset();
15     void calculate();
16 private:
17     QString value1;
18     QString value2;
19     bool isValue1;
20     QString operation;
21     float result;
22 signals:
23     calcDone(float calcResult);
24 };
25
26 #endif // CALCULATOR_H
27
```

Aufgabe Lab 1d – Calculator

- Im QML kann nun auf das Signal reagiert werden
 - In Zeile 8 vom „DisplayPanel.qml“ wird auf das target: „calc“ (Wurde im main.cpp Zeile 11 so genannt) auf das Signal calcDone reagiert
 - Die Syntax dahinter ist immer „onSignalName: { //Do Javascript Stuff }“
 - Hier wird der Text des Ergebnisfeldes mit dem Rückgabewert des Signals geändert



```
1 import QtQuick 2.4
2
3
4 DisplayFormular {
5
6     Connections{
7         target:calc
8         onCalcDone: outputText.text = calcResult
9     }
10
11     Connections{
12         target:mainWindow
13         onClickChanged:{ inputText.text = inputText.text + mainWindow.click}
14     }
15
16     Connections{
17         target:mainWindow
18         onFirstClickedChanged:{
19             if(mainWindow.firstClicked){
20                 inputText.text = ""
21             }
22         }
23     }
24
25     Connections{
26         target:mainWindow
27         onFirstClickedChanged:{
28             inputText.text = ""
29         }
30     }
31 }
```


Aufgabe Lab 1e – Calculator

- Stelle in der ButtonPanelFormular.ui.qml die unterschiedlichen Buttons als Eigenschaften bereit
 - <http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html>
 - z.B. `property alias button0: buttonIdNumber0`
- Schreibe in der ButtonPanel.qml die onClicked-Methoden für die einzelnen Buttons und rufe die entsprechenden Funktionen aus der Calculator-Klasse auf
- Gebe in der DisplayPanelFormular.ui.qml die Text-Objekte als Eigenschaften frei
 - Tipp: funktioniert genau wie hier im ButtonPanelFormular weiter oben in der Aufgabe
- Schreibe in der DisplayPanel.qml die Connection zum Calculator:
 - ```
Connections{ target: calc
 onCalcDone: outputText = calcResult }
```

# Aufgabe Lab 1f – Calculator

- Mit folgenden Code können die Connections zwischen den UI-Elementen aufgebaut werden:

```
4 ▼ DisplayFormular {
5
6 ▼ Connections{
7 target:calc
8 onCalcDone: outputText.text = calcResult
9 }
10 ▼ Connections{
11 target:mainWindow
12 onClickChanged:{ inputText.text = inputText.text + mainWindow.click}
13 }
14 ▼ Connections{
15 target:mainWindow
16 onFirstClickedChanged:{
17 if(mainWindow.firstClicked){
18 inputText.text = ""
19 }
20 }
21 }
22 ▼ Connections{
23 target:mainWindow
24 onFirstClickedChanged:{
25 inputText.text = ""
26 }
27 }
28 }
29
```

```
4 ▼ ApplicationWindow {
5 id: mainWindow
6 visible: true
7
8 width: 800
9 height: 400
10
11 signal clicked(string clickButtonText)
12
13 property string click : ""
14 property bool firstClicked : true
15 property bool awake: true
16
17 function wasClicked(clicked){
18 if(awake){
19 awake = false
20 }
21
22 if(firstClicked){
23 firstClicked = false
24 }
25
26 click = clicked
27 console.log(click)
28 }
29
30 ▼ MainFormular {
31
32 }
33 }
```

# Aufgabe Lab 2 – Unterschiedliche UIs

---

- Lege eine zweite UI an, die beim Ausführen auf dem Smartphone (Hochformat) gestartet werden soll
  - Hier seid ihr völlig frei im Design, im einfachsten Fall können einfach die Formulare anstatt übereinander angezeigt werden, anstatt von nebeneinander
  - Die beiden Formulare (Display und Button) können wiederverwendet werden, die QML und CPP-Dateien müssen etwas angepasst werden
- In der Main.cpp kann mit Präprozessor-Macros das Betriebssystem unterschieden werden:
  - `Q_OS_ANDROID` ist `true`, wenn es ein Android-System ist
  - Weitere sind hier zu finden: <http://doc.qt.io/qt-5/qtglobal.html>
- Über `engine.load(QUrl(QLatin1String(„qrc://NEUEMAIN.qml“)));` lässt sich bestimmen, welche QML geladen wird

# Aufgabe Lab 3a – Sensoren

---

In dieser Aufgabe wird eine kleine App entwickelt, die die Nutzung der Sensoren innerhalb von Qt zeigt. Diese App kann gleichermaßen auf Android, wie auch auf iOS genutzt werden.

Als Beispiel wird eine kleine Kugel auf dem Bildschirm in Richtung der Kippung des Handys bewegt.

- Öffne das Projekt Lab 3
- In der Datei „Lab3.pro“ werden in der ersten Zeile die Module angegeben, die in das Qt-Projekt importiert werden. Es muss das Modul „sensors“ hinzugefügt werden
  - `QT += qml quick svg sensors`
- Die restliche Logik kann komplett in der „main.qml“ implementiert werden
- Importiere „QtSensors 5.0“ in die „main.qml“

# Aufgabe Lab 3b – Sensoren

---

- In der Datei „Lab3.pro“ werden in der ersten Zeile die Module angegeben, die in das Qt-Projekt importiert werden. Es muss das Modul „sensors“ hinzugefügt werden
  - `QT += qml quick svg sensors`
- Die restliche Logik kann komplett in der „main.qml“ implementiert werden
- Importiere „QtSensors 5.0“ in die „main.qml“
- Der Code-Block „Image{ ... }“ baut aus der eingebundenen SVG-Datei (/content/Bluebubble.svg) eine Kugel und fügt eine Animationsbewegung hinzu
- Füge unter „visible: true“ den Aufruf des Sensors hinzu, hier „Accelerometer { }“
  - Innerhalb der geschweiften Klammern wird nun die Logik geschrieben
  - Gib dem Accelerometer eine id, mit dieser kannst du die Daten später abfragen
  - Der Sensor wird mit dem Attribut „active: true“ aktiviert, außerdem muss eine Datenrate „dataRate:“ angegeben werden, z.B. mit einem Wert von 100

# Aufgabe Lab 3c – Sensor

- Die QML-Datei sollte nun wie folgt aussehen:
- Dem Accelerometer kann nun das Event „onReadingChanged: {}“ hinzugefügt werden, dieses reagiert auf alle Änderungen die von dem Sensor gemeldet werden
- Mit accelerometerID.reading.x /.y /.z werden die Werte des Sensors abgefragt
- Mit bubble.x / .y / .z kann die Position der Kugel gesetzt werden
- Weitere Informationen zum Auslesen der Accelerometer-Werte:  
<http://doc.qt.io/qt-5/qml-qtsensors-accelerometerreading.html>

```
1 import QtQuick 2.1
2 import QtQuick.Controls 1.0
3
4 import QtSensors 5.0
5
6
7 ApplicationWindow {
8 title: "Sensor Bubble"
9 id: mainWindow
10 width: 320
11 height: 480
12 visible: true
13
14 Accelerometer {
15 id: accel
16 dataRate: 100
17 active: true
18 }
19
20
21
22 Image {
23 id: bubble
24 source: "content/Bluebubble.svg"
25 smooth: true
26 property real centerX: mainWindow.width / 2
27 property real centerY: mainWindow.height / 2
28 property real bubbleCenter: bubble.width / 2
29 x: centerX - bubbleCenter
30 y: centerY - bubbleCenter
31
32 Behavior on y {
33 SmoothedAnimation {
34 easing.type: Easing.Linear
35 duration: 100
36 }
37 }
38 Behavior on x {
39 SmoothedAnimation {
40 easing.type: Easing.Linear
41 duration: 100
42 }
43 }
44 }
45 }
46
```

# Aufgabe Lab 3d – Sensoren

---

Folgende Funktionen helfen die Bewegung der Kugel zu berechnen:

```
43 ▼ function calcPitch(x,y,z) {
44 return -(Math.atan(y / Math.sqrt(x * x + z * z)) * 57.2957795);
45 }
46 ▼ function calcRoll(x,y,z) {
47 return -(Math.atan(x / Math.sqrt(y * y + z * z)) * 57.2957795);
48 }
49
```

