1.1. PySide

1.1.1. Einführung

Viele moderne Programmiersprachen besitzen ein Set an Libraries zur Erstellung von GUI-Applikationen wie Qt, Tcl/Tk oder wxWigets. PySide ist eine Qt-Toolkit für Python und ist lauffähig auf allen von Qt unterstützten Plattformen (Windows, Mac OS X und Linux).
PySide kombiniert die Vorteile von Qt und Python[1]. Ein PySide-Programmierer hat die Power von Qt zur Verfügung, kann jedoch bei der Entwicklung auf die Einfachheit von Python vertrauen - unterstützt somit auch RAD[2].

PySide ist lizensiert unter LGPL Version 2.1 und erlaubt Opensource und proprietäre Softwareentwicklung.

1.1.2. Erste GUI-Applikation

```
# Import the necessary modules required
import sys
from PySide.QtCore import Qt
from PySide.QtGui import QApplication, QLabel

# Main Function
if __name__ == '__main__':

    # Create the main application
    myApp = QApplication(sys.argv)

    # Create a Label and set its properties
    try:
        appLabel = QLabel()
        appLabel.setText("Hello, World!!!\n Look at my first app using PySide")
        appLabel.setAlignment(Qt.AlignCenter)
        appLabel.setWindowTitle("My First Application")
        appLabel.setGeometry(300, 300, 250, 175)

        # Show the Label
        appLabel.show()

        # Execute the Application and Exit
        myApp.exec_()
        sys.exit()
    except NameError:
        print("Name Developer Meeting: Error:", sys.exc_info()[1])
        pass
```

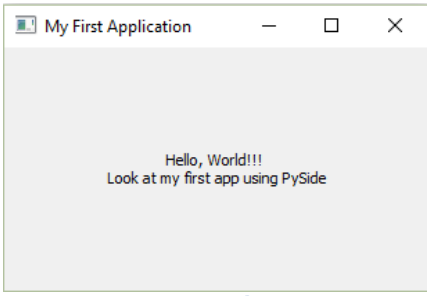

Abbildung 1: Erste GUI-Applikation mit PySide

[1] Loganathan: PySide GUI Application Development, 6ff
[2] Rapid application development

1.1.3. QApplication

Unsere Applikation ist eine Instanz der Klasse QApplication. QApplication erstellt die Main Event-loop, wo all Events vom Window-System und anderen Ressourcen bearbeitet und verteilt werden. Diese Klasse ist verantwortlich für die Initialisierung, Beendigung und das Session Management. Neben den Events wird ebenfalls das Look-and-Feel der Applikation festgelegt.

In jeder Applikation sollte nur ein QApplication-Objekt erstellt werden, auch wenn mehrere Windows vorhanden sind.
Weiters kann auf die Commandline Argumente (sys.argv) zugegriffen werden.

1.1.3.1. Umsetzung

Die Instanziierung von QApplication muss vor der Erstellung aller anderen Objekte erfolgen, da hier alle system- und applikationweiten Einstellungen erfolgen.
Im o.a. Beispiel wird ein QLabel zur Darstellung verwendet. Ähnlich wie auch in Java-Swing muss für Darstellung noch sichtbar gemacht werden. Die Qt-Module sind vollständig objektorientiert designt. So ist QLabel hat QFrame als Superklasse, welche wieder QWidget als Oberklasse besitzt.

1.1.4. QWidget

QWidget ist die Basisklasse aller UI-Objekte. Jedes QWidget ist in ein übergeordnetes QWidget eingebunden. Nicht eingebettete QWidgets werden auch Windows genannt. Ein Window besitzt ein Frame und eine Titelbar. Weiters sind auch Menubar, Toolbar und eine Statusbar möglich.

Jedes QWidget erhält alle Maus, Tastaturevents (und auch andere) vom Windowsystem. Es selbst verantwortlich für die eigene Darstellung.

Ein einfaches Window könnte folgendermaßen aussehen:

```
# Import required modules
import sys
import time
from PySide.QtGui import QApplication, QWidget

class SampleWindow(QWidget):
    """ Our main window class
    """

    # Constructor function
    def __init__(self):
        QWidget.__init__(self)
        self.setWindowTitle("Sample Window")
        self.setGeometry(300, 300, 200, 150)
        self.setMinimumHeight(100)
        self.setMinimumWidth(250)
        self.setMaximumHeight(200)
        self.setMaximumWidth(800)

if __name__ == '__main__':
```

```
# Exception Handling
try:
    myApp = QApplication(sys.argv)
    myWindow = SampleWindow()
    myWindow.show()
    time.sleep(3)
    myWindow.resize(300, 300)
    myWindow.setWindowTitle("Sample Window Resized")
    myWindow.repaint()
    myApp.exec_()
    sys.exit(0)
except NameError:
    print("Name Error:", sys.exc_info()[1])
except SystemExit:
    print("Closing Window...")
except Exception:
    print (sys.exc_info()[1])
```

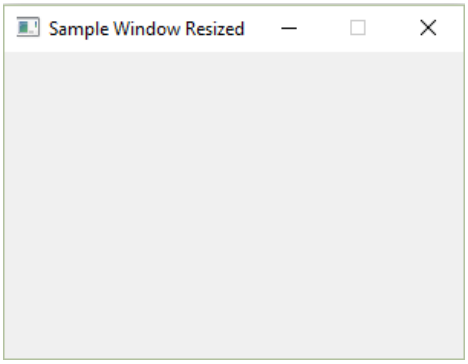

Abbildung 2: Einfaches Window mit Titelbar

In dem o.a. Beispiel wird ein Fenster vorerst in minimaler Größe erstellt, um es nach 3 Sekunden auf sein definiertes Maximum zu vergrößern.

Ein einfaches Fenster kann nun mit beliebig vielen Widgets bestückt werden:

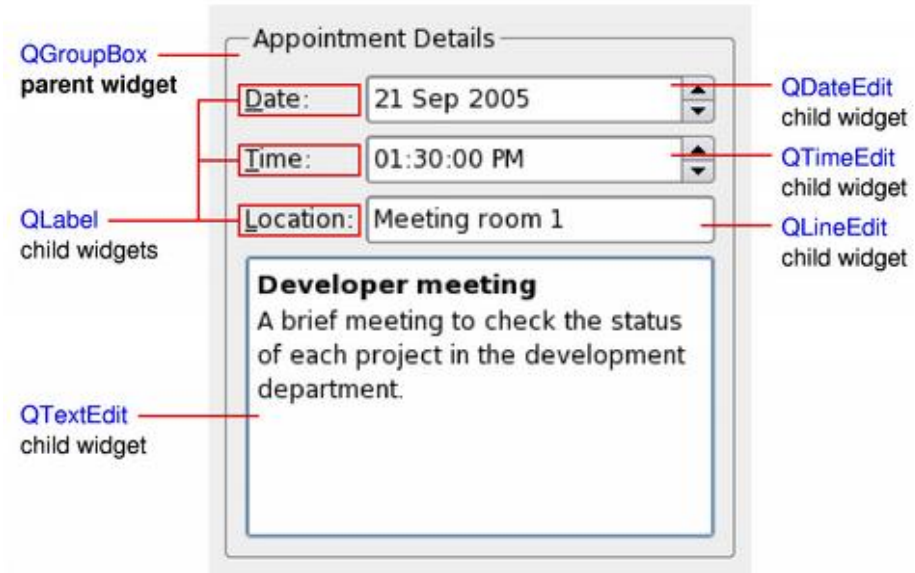

Abbildung 3: QDialog mit unterschiedlichsten Widgets

1.1.5. QLayout

Zur Strukturierung von mehreren QWidgets können verschiedene QLayouts eingesetzt werden. Die wichtigsten vordefinierten Layouts sind:

QVBoxLayout: Vertikale Anordnung
QHBoxLayout: Horizontale Anordnung
QGridLayout: Definition durch Zeile und Spalte
QFormLayout: Jede Zeile besteht aus Label und Inputfield

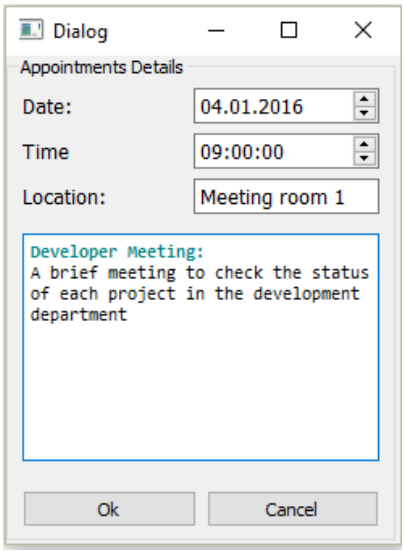

Abbildung 4: Layouts und Inputfields

Im QT-Designer bekommt man mittels Object-Inspektor einen guten Überblick über alle Widgets und Layouts:

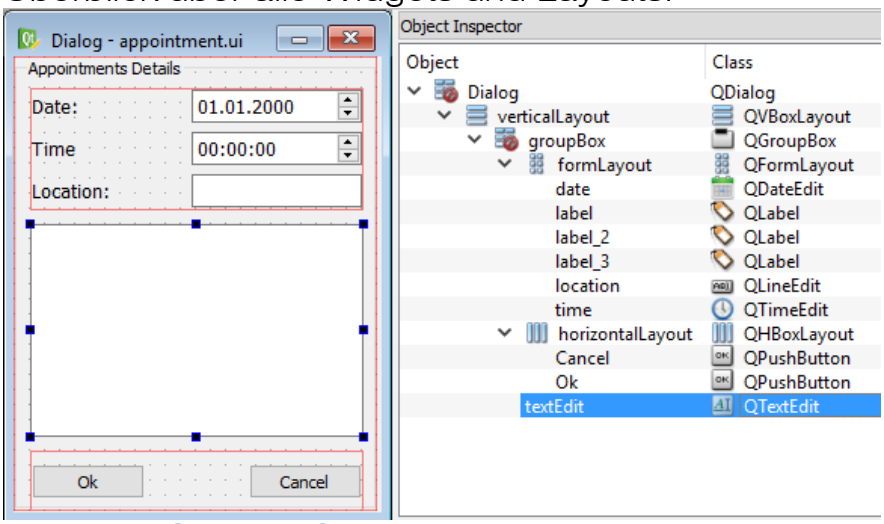

Abbildung 5: QDialog im QT-Designer

1.1.6. Signals und Slots

1.1.6.1. Einführung

Das Qt-Framework besitzt ein Eventhandling, dass dem Observer-Pattern nachempfunden ist. Der Sender (Observable) ruft bei Eintritt eines speziellen Signals (Events) bei Empfängern (Observer) eine spezielle Methode oder Callable (Slot) auf.
Auf Basis von Qt bietet PySide verschiedene Möglichkeiten zur Anbindung an Signals bzw. zur Erstellung von Slots an.

Prinzipiell können beliebig viele Slots an ein Signal gebunden werden. Ebenso kann an Slot für mehrere Signals verwendet werden.

1.1.6.2. Anbindung eines Signals

Die Standard-Variante von Qt (outdated) wäre.
Die Methode connect wird von QObject vererbt. Der Aufruf ist folgendermaßen strukturiert:

```
QObject.connect( Sender, Signal, Receiver)
```

```
# create a Timer
self.timer = QTimer(self)
# the old way
self.connect(self.timer, SIGNAL("timeout()"), self.doSomething)
```

Noch allgemeiner ist der Aufruf:

```
QObject.connect( Sender, Signal, Receiver-object, Slot)
```

```
# connect a QPushButton with QApplication.exit()
self.connect(self.myForm.Cancel, SIGNAL("clicked()"), self, SLOT("exit()"))
```

Der heute übliche Weg ist stärker an OO-Prinzipien angepasst:

```
# connect a QPushButton with self.buttonClicked()
self.myForm.Ok.clicked.connect(self.buttonClicked)
```

1.1.6.3. Der Slot

Die Standard-Variante von Qt mittels Annotation ist nur dann zwingend vorgeschrieben, wenn das Slot mit unterschiedlichen Datentypen umgehen soll:

```
# Slot for integer and string values
@Slot(int)
@Slot(str)
def saySomething(stuff):
    print (stuff)
```

Für Methoden ohne Parameter ist die Annotation redundant:

```
# it's not necessary
@Slot()
def exit(self):
    QApplication.exit()
```

1.1.6.4. Eigene Signals verwenden

PySide schlägt die Erstellung einer eigenen Klasse vor:

```
class Communicate(QObject):

    reminder = Signal()
```

Dabei muss die Klasse von QObject abgeleitet sein.

```
# create a new instance of Communicate
self.c = Communicate()
```

Die Anbindung kann analog zu vorhandenen QWidgets verwendet werden.

```
# connect the reminder to startTimer
self.c.reminder.connect(self.startTimer)
```

Ein Signal kann nun direkt emittiert werden.

```
# emit the signal
self.c.reminder.emit()
```