

**Angewandtes Programmierprojekt**

im Studiengang

Biomedizinische Informationstechnik

Thematik:

**Entwicklung eines Graphical User Interface zur Nutzung der Pythonic mmWave Toolbox.**

Eingereicht von: Oliver Jovanović, B.A., B.Sc.

E-Mail-Adresse: oliver.jovanovic001@stud.fh-dortmund.de

Matrikelnummer: 7106404

Erarbeitet im: 3. Semester

Abgabedatum: Donnerstag, 26. Januar 2023

Gutachter: Prof. Dr.-Ing. Burkhardt Igel

Inhaltsverzeichnis

[Abbildungsverzeichnis II](#_Toc125636334)

[Tabellenverzeichnis III](#_Toc125636335)

[1. Einleitung 1](#_Toc125636336)

[2. Sprint 1: Definition der GUI und Funktionalitäten 2](#_Toc125636337)

[a. Sprintplanung: Definition der GUI 2](#_Toc125636338)

[b. Sprintplanung: Definition der Funktionalitäten 2](#_Toc125636339)

[3. Sprint 2: Erstellung der GUI ohne Funktionalität 3](#_Toc125636340)

[a. Erstellung der GUI 3](#_Toc125636341)

[4. Sprint 3: GUI-Funktionalität implementieren 4](#_Toc125636342)

[a. GUI-Funktionalitäten implementieren 4](#_Toc125636343)

[5. Sprint 4: Code in MVC umschreiben 5](#_Toc125636344)

[a. Code in MVC-pattern umschreiben 5](#_Toc125636345)

[6. Sprint 5: Stringenter in MVC umschreiben und Toolbox anbinden 6](#_Toc125636346)

[a. Stringenter in MVC umschreiben 6](#_Toc125636347)

[b. Funktionalität mit Pythonic mmWave Toolbox sicherstellen 6](#_Toc125636348)

[Literaturverzeichnis 7](#_Toc125636349)

[Eidesstattliche Erklärung 8](#_Toc125636350)

[Anhang 9](#_Toc125636351)

# Abbildungsverzeichnis

[Abbildung 1: Konzeptionelles Aussehen der GUI 2](#_Toc125636417)

[Abbildung 2: Umgesetzte GUI mit Funktionalität 4](#_Toc125636418)

# Tabellenverzeichnis

[Tabelle 1: Sprintplanung und Definition Sprint 1 2](#_Toc125636422)

[Tabelle 2: Sprintplanung und Definition Sprint 2 3](#_Toc125636423)

[Tabelle 3: Sprintplanung und Definition Sprint 3 4](#_Toc125636424)

[Tabelle 4: Sprintplanung und Definition Sprint 4 5](#_Toc125636425)

[Tabelle 4: Sprintplanung und Definition Sprint 5 6](#_Toc125636426)

# Einleitung

Die hier vorliegende Arbeit dient als schriftliche Abgabe für das Fach „Angewandtes Programmierprojekt“, welche im dritten Semester an der Fachhochschule Dortmund abgeleistet wurde. Der Inhalt dieses Schriftstücks umfasst die wesentlichen Entwicklungsschritte einer graphischen Benutzeroberfläche, im weiteren als GUI (Graphical User Interface) bezeichnet. Im speziellen sind die inkrementellen Entwicklungsschritte und in Verbindung mit der agilen Methodik SCRUM beschrieben. Zu SCRUM fand ein verbindlicher Workshop im Wintersemester 2022/2023 zusammen mit dem Unternehmen Conciso GmbH statt. Der Fokus dieser Arbeit liegt auf die Entwicklung der GUI, weswegen auf die Methode SCRUM nicht weiter eingegangen und das Wissen über diese Methode als gegeben betrachtet wird.

Das Ziel dieser Arbeit ist die Bedienung der Pythonic mmWave Toolbox mittels graphischer Benutzeroberfläche. Die Entwickler der Toolbox haben die Funktionalität sowie ein Anwendungsbeispiel hier [1] beschrieben. Um die Toolbox jedoch zu nutzen haben die Entwickler ein GitHub Repository erstellt [2], auf das frei zugegriffen werden kann und in dem zuletzt vor einem Jahr Entwicklungen stattgefunden haben (stand Januar 2023). Um eine Kommunikation zwischen dem PC/Laptop und dem Radarsystem herzustellen, ist eine physische Verbindung über ein USB-Kabel sowie eine Kommunikationsverbindung notwendig. Die Kommunikationsverbindung wird wie in [2] beschrieben initiiert, indem diese mittels Kommandozeile aufgebaut wird. Die mögliche Kommandozeile sieht wie folgt aus:

* pymmw.py [-h] [-c PORT] [-d PORT] [-f HANDLER] [-n]

Für die Verbindung ist folgender Codeanteil notwendig:

* pymmw.py [-c PORT] [-d PORT]

Hierbei beschreibt -c den Kontroll-Port und -d den Daten-Port. Im hier vorliegenden Beispiel ist Windows 11 genutzt und im Fall des Autors ist der Kontroll-Port ist COM3 und der Daten-Port ist COM4.

Um die Nutzung der Pythonic mmWave Toolbox für Nutzer, die nicht mit Kommandozeilen arbeiten können zu ermöglichen, ist eine graphische Benutzeroberfläche die optimale Lösung. Zur Erstellung und der GUI ist die Skriptsprache Python in Kombination mit der Bibliothek PyQt5 genutzt worden. Im nachfolgenden ist zunächst die jeweilige *Sprintplanung* sowie das *Backlog* beschrieben. Weiter ist jeder *Sprint* umfassend beschrieben. Der Aufbau der Kapitel beinhaltet zunächst das *Sprintziel*, also das Ergebnis nach einer Woche Arbeit und die darin beinhalteten Aufgaben.

# Sprint 1: Definition der GUI und Funktionalitäten

Tabelle 1: Sprintplanung und Definition Sprint 1

|  |  |
| --- | --- |
| Sprintplanung | Backlog |
| * Definition der GUI * Definition der Funktionalitäten | * Warten auf Informationen aus dem ersten Sprint, um Backlog zu füllen. |

## Sprintplanung: Definition der GUI

Die Definition der GUI beschreibt, wie die GUI aussehen und welche Daten in der GUI verarbeitet werden sollen. Um die Beschreibung verständlich zu gestalten, ist Abbildung 1 abgebildet.

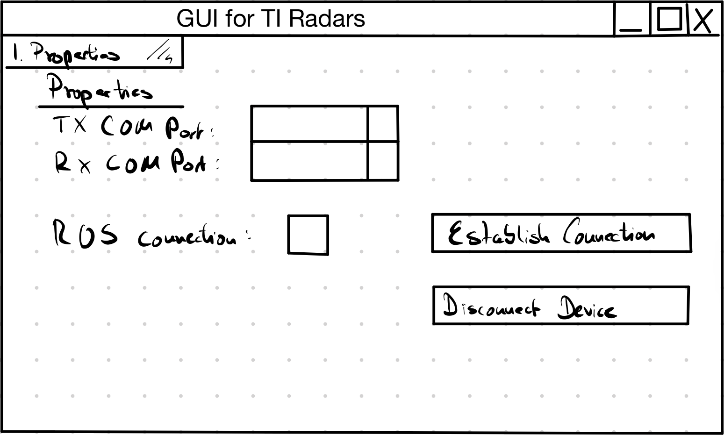


Abbildung 1: Konzeptionelles Aussehen der GUI

Wie Abbildung 1 darstellt, sind fünf Kernelemente in der GUI enthalten. Zum einen sind zwei Dropdown-Felder vorhanden, eine Checkbox und zwei Schaltflächen mit der Aufschrift *Establish Connection* sowie *Disconnect Device*. Weiter sind drei Textfelder vorhanden, welche den Inhalt bzw. die Funktionalität, die im kommenden Abschnitt näher beschrieben wird, andeuten. Diese Felder lauten *TX COM Port*, *RX COM Port* sowie *ROS connection*. Dieses Fenster stellt die Grundeigenschaften für die GUI dar und ist deshalb als *Properties* beschrieben.

## Sprintplanung: Definition der Funktionalitäten

Die Dropdown-Felder, die mit *TX COM Port* sowie *RX COM Port* beschrieben sind, werden mit seriellen Kommunikations-Ports befüllt, die anschließend durch den Anwender ausgewählt werden können. Hierzu soll der Anwender auf die Dropdown-Felder klicken und erhält anschließend eine Auswahlmöglichkeit mit den jeweils angeschlossenen Kommunikations-Ports.

Die Checkbox, die mit *ROS connection* beschrieben ist, dient zu Herstellung der Verbindung des Radars mit ROS. ROS steht für Robot Operating System. Wenn dies ausgewählt ist und anschließend auf den Knopf *Establish Connection* gedrückt wird, soll zunächst die Verbindung mit dem Radar erfolgen und anschließend mit dem ROS.

Der Knopf mit der Aufschrift *Establish Connection* stellt die Verbindung mit dem Radar und gegebenenfalls mit dem ROS her.

Der Knopf mit der Aufschrift *Disconnect Device* löst die Verbindung mit dem Radar und gegebenenfalls mit ROS.

# Sprint 2: Erstellung der GUI ohne Funktionalität

Tabelle 2: Sprintplanung und Definition Sprint 2

|  |  |
| --- | --- |
| Sprintplanung | Backlog |
| * Erstellung der GUI   + Knöpfe   + Label   + Dropdown-Menü   + Titel   + Tabs | * GUI-Funktionalität   + Button   + Checkbutton   + Dropdown-Menü   + Tabs * Funktionalität mit Pythonic mmWave Toolbox   + GUI interagiert mit der Toolbox * Funktionalität mit ROS   + Verbindung mit ROS herstellen |

## Erstellung der GUI

Wie im ersten Sprint beschrieben, benötigt die GUI verschiedene Eigenschaften, sodass sie genutzt werden kann. Da der Autor zuvor noch nie eine GUI in Python erstellt hat, nutzte dieser [3] als Leitfaden zur Erstellung von graphischen Benutzeroberflächen. Dieses Buch ist ausgewählt, da viele positive Rezensionen vorhanden sind und der Autor des Buches Jahrzehntelange Erfahrung in der Entwicklung von Python basierten graphischen Benutzeroberflächen hat.

Graphikbasierte Oberflächen werden in der Entwicklung über ein MVC-Architekturpattern implementiert. Dies bedeutet, dass die Anwendung in drei Bereiche aufgeteilt wird, dem Model, View und Controller. Das Model beinhaltet die Datenstruktur, mit der die Anwendung arbeitet. Das View veranschaulicht die Daten und Informationen beziehungsweise stellt die graphische Benutzeroberfläche dar. Abschließend der Controller. Dieser dient als direktes Bindeglied zwischen dem View und dem Model und bearbeitet die Eingaben des Anwenders. Dadurch ist die Logik von der Graphik getrennt, und somit findet eine Trennung von Verantwortlichkeiten statt [vgl. 3, p. 261].

In PyQt5 ist dies dennoch anders. Hier wird ein Konzept genutzt, das zwei Elemente besitzt, nämlich Model und View. Der dritte Bereich ist im View mit enthalten. Dadurch kann statt vom View auch vom ViewController gesprochen werden. Dies bedeutet, dass das Model die Daten oder einen Verweis auf diese und gibt einzelnen oder Bereiche von Datensätzen und die zugehörigen Metadaten oder Anzeigeanweisungen zurück. Der ViewController fordert Daten vom Model an und zeigt an, was im Widget zurückgegeben wird [vgl. 3, p. 262].

Weiter ist eine Menüschaltfläche als Titelleiste hinzugefügt, die in der Konzeptionellen Darstellung nicht enthalten ist. Diese soll die Funktionalitäten beinhalten die GUI zu verlassen, die Kommunikations-Ports zu aktualisieren sowie ein Über-Fenster, die den Entwickler der GUI vorstellt.

Weiter ist nur ein Knopf zum Verbindungsauf- und Abbau implementiert. Der zweite Knopf dient zum Schließen der graphischen Benutzeroberfläche.

In diesem Sprint sind alle Aufgaben fristgerecht erfüllt worden. Auf die Implementierung wird nicht näher eingegangen, da abschließend der Code im Anhang sowie als Datei zur Verfügung gestellt und dieser auch durch Kommentare erläutert wird.

# Sprint 3: GUI-Funktionalität implementieren

Tabelle 3: Sprintplanung und Definition Sprint 3

|  |  |
| --- | --- |
| Sprintplanung | Backlog |
| * GUI-Funktionalitäten implementieren   + Button   + Checkbutton   + Dropdown-Menü   + Tabs | * Funktionalität mit Pythonic mmWave Toolbox   + GUI interagiert mit der Toolbox * Funktionalität mit ROS   + Verbindung mit ROS herstellen |

## GUI-Funktionalitäten implementieren

Die Implementierung der Funktionalität geschieht wie in [3] beschrieben. Hierbei wird das Model-View Konzept angewandt.

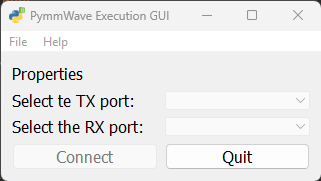


Abbildung 2: Umgesetzte GUI mit Funktionalität

Abbildung 2 stellt die umgesetzte GUI mit Funktionalität dar. Ersichtlich ist, dass zwei Tabs zur Verfügung stehen. Unter *File* findet der Anwender die Funktion *Refresh*, welche die Kommunikations-Ports aktualisiert und *Exit*, welche die GUI beendet. Unter *Help* findet der Anwender den Menüpunkt *About*, der den Entwickler der GUI vorstellt.

Die beiden Kommunikations-Ports sind ausgegraut, da keine vorhanden sind. Wenn der Anwender nun einen Radar anschließt und unter *File🡪Refresh* die Kommunikations-Ports aktualisiert werden diese angezeigt und sind auswählbar.

Der Knopf mit der Aufschrift *Connect* ist ausgegraut, wenn keine Kommunikations-Ports zur Verfügung stehen und auch wenn diese identisch sind. Wenn sie nicht identisch sind, kann der Knopf betätigt werden und ändert anschließend die Aufschrift in *Disconnect*.

Der Knopf mit der Aufschrift *Quit* führt dazu, dass eine Meldung auftaucht, die sicherstellen soll, dass der Anwender die Anwendung wirklich verlassen möchte. Falls der Anwender dies bestätigt, wird die Applikation geschlossen. Falls nicht, wird die Meldung geschlossen und der Anwender gelangt zurück zur GUI.

# Sprint 4: Code in MVC umschreiben

Tabelle 4: Sprintplanung und Definition Sprint 4

|  |  |
| --- | --- |
| Sprintplanung | Backlog |
| * Code in MVC-pattern umschreiben | * Funktionalität mit Pythonic mmWave Toolbox   + GUI interagiert mit der Toolbox |

## Code in MVC-pattern umschreiben

Die Qualität des umgesetzten Codes genügt den Stakeholdern nicht und soll deswegen in das MVC-pattern umgeschrieben werden. Dadurch fällt die Funktionalität mit ROS aus dem Backlog heraus, da hierzu nicht genug zeitliche Ressourcen zur Verfügung stehen. Dies ist mit Herrn Prof. Dr.-Ing. Andreas Becker abgesprochen und ratifiziert.

Im Anhang sind beide Codes zur Verfügung gestellt. Dadurch ist das Ausmaß des Entwicklungsprozesses sichtbar.

# Sprint 5: Stringenter in MVC umschreiben und Toolbox anbinden

Tabelle 5: Sprintplanung und Definition Sprint 5

|  |  |
| --- | --- |
| Sprintplanung | Backlog |
| * Stringenter in MVC umschreiben * Funktionalität mit Pythonic mmWave Toolbox   + GUI interagiert mit der Toolbox |  |

## Stringenter in MVC umschreiben

Der MVC-Ansatz ist ersichtlich, dennoch nicht ausreichend stringent umgesetzt. Abschließend sollen drei Klassen ersichtlich sein, Model, View sowie Controller.

* Model soll alle Daten beinhalten, die für die Verbindung mit dem IWR68434AOPEVM benötigt werden. Diese sind die beiden Kommunikations-Ports.
* View soll dem Anwender die Graphische Benutzeroberfläche darstellen und keine Logik enthalten.
* Controller soll die Logik enthalten. Somit soll der Kontroller alle Eingaben des Anwenders Handhaben und dem Model die aktualisierten Kommunikations-Ports zur Verfügung stellen, die Knopfdrücke handhaben sowie die Logik der in der Menüschaltfläche hinterlegten Knöpfe beinhalten.

Zudem besitzen nun alle Menüpunkte in der Menüschaltfläche Icons und der Menüpunkt *Help* ist unter der Menüschaltfläche *Help* hinzugefügt worden. Dieser beinhaltet eine kurze Erläuterung, wie die GUI zu verwenden ist und welche Kommunikations-Ports genutzt werden sollen.

Des Weiteren ist ein log in der finalen Version der GUI implementiert. Hierbei werden bei den wichtigen Abschnitten des Codes logs geschrieben.

## Funktionalität mit Pythonic mmWave Toolbox sicherstellen

Der zweite Punkt in diesem Sprint ist die Sicherstellung der Funktionalität mit der Pythonic mmWave Toolbox. Dies geschieht, indem der Anwender zwei unterschiedliche Kommunikations-Ports auswählt und den Knopf mit der Aufschrift *Connect* drückt. Anschließend wird folgende Kommandozeile ausgeführt:

* python pymmw.py -c {rx\_port} -d {tx\_port}

Die Kommandozeile wird erfolgreich ausgeführt und eine Verbindung wird hergestellt. Aufgrund eines Fehlers in der aktuellen Version der Pythonic mmWave Toolbox wird die Ausgabe nicht geplottet. Dies ist durch einen Thread im Repository [2] sowie durch eigene isolierte Ausführung der Kommandozeile bestätigt. Die erfassten Radardaten werden jedoch in einer Textdatei abgespeichert und sind ersichtlich. Somit ist eine Verbindung zwischen der GUI sowie der Pythonic mmWave Toolbox vollzogen und das Ziel der Arbeit erreicht.

# Literaturverzeichnis

[1] M. Constapel, M. Cimdins, and H. Hellbrück, “A Practical Toolbox for Getting Started with mmWave FMCW Radar Sensors,” 2019, doi: 10.24355/dbbs.084-201907151133-0.

[2] m6c7l, “Pythonic mmWave Toolbox for TI’s IWR Radar Sensors,” Nov. 2021. Accessed: Jan. 26 2023. [Online]. Available: [m6c7l/pymmw: Pythonic mmWave Toolbox for TI's IWR Radar Sensors (github.com)](https://github.com/m6c7l/pymmw)

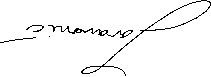
[3] J. M. Willman, *Beginning PyQt: A Hands-on Approach to GUI Programming with PyQt6,* 2nd ed. Berkeley, CA: Apress; Imprint Apress, 2022.

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit (Anzahl 1.637 Wörter) selbstständig und ohne Benutzung anderer, als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die vorliegende Arbeit wurde bisher in gleicher bzw. ähnlicher Form (im Ganzen, wie in Teilen) in keinem anderen Prüfungsverfahren als Prüfungsleistung vorgelegt und auch nicht veröffentlicht.

Dortmund, Donnerstag, 26. Januar 2023



Ort, Datum (Unterschrift)

Weitere Erklärung

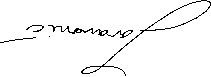
Ich erkläre mich damit einverstanden, dass die vorliegende Arbeit einer Plagiatsprüfung unterzogen wird und dass ich vor einer eventuellen Veröffentlichung der Arbeit die Zustimmung des Gutachters einholen werde.

Ich wurde darüber informiert, dass bei Verstoß gegen die eidesstattliche Erklärung die Aberkennung der Prüfungsleistung sowie ein Verfahren wegen Täuschung bzw. Betruges drohen.

Dortmund, Donnerstag, 26. Januar 2023



Ort, Datum (Unterschrift)



# Anhang

[Anhang 1: MV-Code 10](#_Toc125637342)

[Anhang 2: MVC-Code 14](#_Toc125637343)

Anhang 1: MV-Code

*"""This GUI has been created with the help of the following book:  
Beginning PyQt - A Hands-on Approach to GUI Programming with PyQt6 by Joshua M Willmann"""*# Import necessary modules  
  
import os  
import sys  
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QLabel,  
 QComboBox, QPushButton, QGridLayout, QMessageBox, QAction)  
from PyQt5.QtCore import Qt  
from PyQt5.QtGui import (QFont, QIcon)  
from serial.tools import list\_ports  
import serial  
  
def s\_ports():  
 return [port.device for port in list\_ports.comports()]  
  
  
class MainWindow(QMainWindow):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.rx\_com\_port = None  
 self.tx\_com\_port = None  
 self.initializeUI()  
 # Check if ports are the same  
 self.check\_ports()  
  
 def initializeUI(self):  
 *"""Set up the application's GUI."""* self.setMinimumSize(300, 150)  
 self.setWindowTitle("GUI for PymmWave")  
 self.setWindowIcon(QIcon("images/pyqt\_logo.png"))  
  
 self.setUpMainWindow()  
 self.createActions()  
 self.createMenu()  
 self.show()  
  
 def setUpMainWindow(self):  
 *"""Create and arrange widgets in the main window."""* # Headline  
 header\_label = QLabel("Properties")  
 header\_label.setFont(QFont("Helvetica", 14))  
 header\_label.setAlignment(Qt.AlignmentFlag.AlignLeft)  
  
 # Select Ports  
 tx\_com\_port\_label = QLabel("TX COM Port:", self)  
 tx\_com\_port\_label.setFont(QFont("Helvetica", 12))  
 self.tx\_com\_port = QComboBox()  
 self.tx\_com\_port.addItems(s\_ports())  
 self.tx\_com\_port.activated.connect(self.txPortsChoose)  
  
 rx\_com\_port\_label = QLabel("RX COM Port:", self)  
 rx\_com\_port\_label.setFont(QFont("Helvetica", 12))  
 self.rx\_com\_port = QComboBox()  
 self.rx\_com\_port.addItems(s\_ports())  
 self.rx\_com\_port.activated.connect(self.txPortsChoose)  
  
 # Connect and Disconnect Button in one  
 self.times\_pressed = 0  
 self.button = QPushButton("Connect Radar")  
 self.button.setFont(QFont("Helvetica", 12))  
 self.button.clicked.connect(self.buttonClicked)  
  
 # Close GUI button  
 self.button\_close = QPushButton("Close GUI")  
 self.button\_close.setFont(QFont("Helvetica", 12))  
 self.button\_close.clicked.connect(self.close)  
  
 # Organize the left side widgets into column 0 of the QGridLayout  
 main\_grid = QGridLayout()  
 main\_grid.addWidget(header\_label, 0, 0)  
 main\_grid.addWidget(tx\_com\_port\_label, 1, 0)  
 main\_grid.addWidget(self.tx\_com\_port, 1, 1)  
 main\_grid.addWidget(rx\_com\_port\_label, 2, 0)  
 main\_grid.addWidget(self.rx\_com\_port, 2, 1)  
 main\_grid.addWidget(self.button, 4, 0)  
 main\_grid.addWidget(self.button\_close, 4, 1)  
  
 # Set the layout for the main window  
 container = QWidget()  
 container.setLayout(main\_grid)  
 self.setCentralWidget(container)  
  
 def createActions(self):  
 *"""Create the application's menu actions."""* # Create the actions for File menu  
 self.quit\_act = QAction(QIcon("images/exit.png"), "Quit")  
 self.quit\_act.setShortcut("Ctrl+Q")  
 self.quit\_act.setStatusTip("Quit program")  
 self.quit\_act.triggered.connect(self.close)  
  
 self.read\_act = QAction("Read")  
 self.read\_act.setShortcut("Ctrl+R")  
 self.read\_act.setStatusTip("Read COM Ports in.")  
 self.read\_act.triggered.connect(self.update\_ports)  
 self.numbers\_clicked = 0  
  
 # Create actions for Help menu  
 self.about\_act = QAction("About")  
 self.about\_act.triggered.connect(self.aboutDialog)  
  
 def createMenu(self):  
 *"""Create the application's menu bar."""* # For Mac  
 self.menuBar().setNativeMenuBar(False)  
  
 # Create File menu and add actions  
 file\_menu = self.menuBar().addMenu("File")  
 file\_menu.addAction(self.quit\_act)  
 file\_menu.addAction(self.read\_act)  
  
 # Create Help menu and add actions  
 help\_menu = self.menuBar().addMenu("Help")  
 help\_menu.addAction(self.about\_act)  
  
 def buttonClicked(self):  
 # Done: Implement starting pymmWave functionality by clicking it.  
 *"""If button\_clicked is uneven, then show 'Connect Radar',  
 otherwise 'Disconnect Radar'"""* self.times\_pressed += 1  
 if self.times\_pressed % 2 != 0:  
 self.update\_ports()  
 self.button.setText("Disconnect")  
 else:  
 # Changing the current working directory  
 try:  
 tx\_port = self.tx\_com\_port.currentText()  
 rx\_port = self.rx\_com\_port.currentText()  
 self.connect\_radar(tx\_port, rx\_port)  
 os.chdir(r'C:\Users\Olive\PycharmProjects\MPKurs\Code\pymmw-master\source')  
 os.system('python pymmw.py -c', rx\_port, '-d', tx\_port)  
 self.button.setText("Disconnect Radar")  
 except:  
 self.times\_pressed -= 1  
 self.button.setDisabled(True)  
 print("No COM ports available.")  
  
 def check\_ports(self):  
 *"""Check if ports are not empty, if they are empty make them not choosable."""* available\_ports = s\_ports()  
 if not available\_ports:  
 self.button.setEnabled(False)  
 self.tx\_com\_port.setEnabled(False)  
 self.rx\_com\_port.setEnabled(False)  
 return  
 self.tx\_com\_port.setEnabled(True)  
 self.rx\_com\_port.setEnabled(True)  
 tx\_port = self.tx\_com\_port.currentText()  
 rx\_port = self.rx\_com\_port.currentText()  
 """Check if ports are the same, if true, then disable the connect button!"""  
 if tx\_port == rx\_port:  
 self.button.setEnabled(False)  
 else:  
 self.button.setEnabled(True)  
  
 def rxPortsChoose(self):  
 *"""Check if rx port is the same as tx port"""* self.check\_ports()  
  
 def txPortsChoose(self):  
 *"""Check if tx port is the same as rx port"""* self.check\_ports()  
  
 def connect\_radar(self, tx\_port, rx\_port):  
 *"""Set connection parameters for tx and rx."""* try:  
 self.ser\_tx = serial.Serial(tx\_port, 115200)  
 self.ser\_rx = serial.Serial(rx\_port, 921600)  
 except serial.SerialException as e:  
 QMessageBox.critical(self, "Error", f"failed to connect: {e}")  
  
 def aboutDialog(self):  
 *"""Display the About dialog"""* QMessageBox.about(self, "About pymmWave GUI",  
 """<p>This GUI should help you control TI Radars</p>  
 <p>Created by Oliver Jovanović</p>""")  
  
 # Has no impact!  
 def update\_ports(self):  
 *"""Update the tx and rx port while clicking on the update-button"""* self.tx\_com\_port.clear()  
 self.tx\_com\_port.addItems(s\_ports())  
 self.rx\_com\_port.clear()  
 self.rx\_com\_port.addItems(s\_ports())  
 self.check\_ports()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # print(s\_ports()) # For testing purposes.  
 app = QApplication(sys.argv)  
 window = MainWindow()  
 sys.exit(app.exec\_())

Anhang 2: MVC-Code

# Import necessary modules:  
import os  
import sys  
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QLabel,  
 QComboBox, QPushButton, QGridLayout, QMessageBox)  
from PyQt5.QtGui import (QFont, QIcon)  
from serial.tools import list\_ports  
import logging  
  
# Set up logging  
logging.basicConfig(level=logging.DEBUG, filename="application.log")  
  
  
class Model:  
 *"""With which data do we want to work?"""* def \_\_init\_\_(self):  
 self.tx\_port = None  
 self.rx\_port = None  
 self.ports = [port.device for port in list\_ports.comports()]  
  
  
class View(QMainWindow):  
 *"""How should the GUI look like?"""* def \_\_init\_\_(self, model, controller):  
 super(View, self).\_\_init\_\_()  
 self.\_model = model  
 self.tx\_port = None  
 self.rx\_port = None  
 self.initUI(controller)  
  
 def initUI(self, controller):  
 logging.debug("Initializing GUI. Available serial ports: %s", self.\_model.ports)  
 layout = QGridLayout()  
 self.setMinimumSize(320, 150)  
 self.setWindowTitle("PymmWave Execution GUI")  
 self.setWindowIcon(QIcon("images/pyqt\_logo.png"))  
  
 container = QWidget()  
 container.setLayout(layout)  
 self.setCentralWidget(container)  
  
 # Headline  
 header\_label = QLabel("Properties")  
 header\_label.setFont(QFont("Helvetica", 12))  
  
 # Ports  
 # tx port  
 tx\_port\_label = QLabel("Select te TX port:", self)  
 tx\_port\_label.setFont(QFont("Helvetica", 12))  
 self.tx\_port = QComboBox()  
 self.tx\_port.addItems(controller.\_model.ports)  
 self.tx\_port.currentTextChanged.connect(controller.check\_ports)  
  
 # rx port  
 rx\_port\_label = QLabel("Select the RX port:", self)  
 rx\_port\_label.setFont(QFont("Helvetica", 12))  
 self.rx\_port = QComboBox()  
 self.rx\_port.addItems(controller.\_model.ports)  
 self.rx\_port.currentTextChanged.connect(controller.check\_ports)  
  
 # Connect and Disconnect Button in one  
 self.times\_pressed = 0  
 self.button = QPushButton("Connect")  
 self.button.setFont(QFont("Helvetica", 12))  
 self.button.clicked.connect(controller.buttonClicked)  
  
 # Quit Button  
 self.quit\_button = QPushButton("Quit")  
 self.quit\_button.setFont(QFont("Helvetica", 12))  
 self.quit\_button.clicked.connect(controller.close)  
  
 # Add widgets to layout  
 layout.addWidget(header\_label, 0, 0)  
 layout.addWidget(tx\_port\_label, 1, 0)  
 layout.addWidget(self.tx\_port, 1, 1)  
 layout.addWidget(rx\_port\_label, 2, 0)  
 layout.addWidget(self.rx\_port, 2, 1)  
 layout.addWidget(self.button, 4, 0)  
 layout.addWidget(self.quit\_button, 4, 1)  
  
 # Create menu bar  
 menu\_bar = self.menuBar()  
  
 # Create File menu  
 file\_menu = menu\_bar.addMenu("File")  
 refresh\_action = file\_menu.addAction("Refresh", controller.update\_ports)  
 icon\_refresh = QIcon("images/refresh.png")  
 refresh\_action.setIcon(icon\_refresh)  
 file\_menu.addSeparator()  
 exit\_action = file\_menu.addAction("Exit", controller.close)  
 icon\_exit = QIcon("images/exit.png")  
 exit\_action.setIcon(icon\_exit)  
  
 # Create Help menu  
 help\_menu = menu\_bar.addMenu("Help")  
 help\_action = help\_menu.addAction("Help", controller.help)  
 icon\_help = QIcon("images/help.svg")  
 help\_action.setIcon(icon\_help)  
 help\_menu.addSeparator()  
 about\_action = help\_menu.addAction("About", controller.about)  
 icon\_about = QIcon("images/about.svg")  
 about\_action.setIcon(icon\_about)  
  
  
class Controller:  
 *"""How should the GUI behave?"""* def \_\_init\_\_(self):  
 self.\_app = QApplication(sys.argv)  
 self.\_model = Model()  
 self.\_view = View(self.\_model, self)  
 self.\_view.button.clicked.connect(self.buttonClicked)  
 self.check\_ports()  
  
 def close(self):  
 logging.debug("Closing the Application.")  
 """Prompt the user to confirm that they want to close the application."""  
 result = QMessageBox.question(self.\_view, "Confirm Exit", "Are you sure you want to exit?", QMessageBox.Yes |  
 QMessageBox.No, QMessageBox.No)  
 if result == QMessageBox.Yes:  
 self.\_app.quit()  
  
 def connect(self):  
 *"""Connect the radar"""* tx\_port = self.\_view.tx\_port.currentText()  
 rx\_port = self.\_view.rx\_port.currentText()  
 self.\_model.tx\_port = tx\_port # Should be COM3  
 self.\_model.rx\_port = rx\_port # Should be COM4  
 print(f'Connected to {tx\_port} and {rx\_port}')  
 os.chdir(r'C:\Users\Olive\PycharmProjects\MPKurs\Code\pymmw-master\source')  
 os.system('python pymmw.py -c ' + rx\_port + ' -d ' + tx\_port)  
 print("pymmwave started.")  
 print("python pymmw.py -c {tx\_port} -d {rx\_port}")  
  
 def disconnect(self):  
 print("Disconnected.")  
  
 def check\_ports(self):  
 *"""Check if ports are not empty, if they are empty make them not choosable."""* if not self.\_model.ports:  
 self.\_view.tx\_port.setEnabled(False)  
 self.\_view.rx\_port.setEnabled(False)  
 self.\_view.button.setEnabled(False)  
 else:  
 self.\_view.tx\_port.setEnabled(True)  
 self.\_view.rx\_port.setEnabled(True)  
 self.\_view.button.setEnabled(True)  
 tx\_port = self.\_view.tx\_port.currentText()  
 rx\_port = self.\_view.rx\_port.currentText()  
 """Check if ports are the same, if true, then disable the connect button!"""  
 if tx\_port == rx\_port:  
 self.\_view.button.setEnabled(False)  
 else:  
 self.\_view.button.setEnabled(True)  
  
 def update\_ports(self):  
 logging.debug("Refreshing serial ports.")  
 """Updates the list of available ports in the Model class and updates the options in the  
 QComboBox widgets of the View class accordingly."""  
 self.\_model.ports = [port.device for port in list\_ports.comports()]  
 self.\_view.tx\_port.clear()  
 self.\_view.tx\_port.addItems(self.\_model.ports)  
 self.\_view.rx\_port.clear()  
 self.\_view.rx\_port.addItems(self.\_model.ports)  
 self.check\_ports()  
  
 def buttonClicked(self):  
 *"""If button\_clicked is uneven, then show "Connect", otherwise disconnect"""* logging.debug("Connect button clicked. TX port: %s, RX port: %s", self.\_model.tx\_port, self.\_model.rx\_port)  
 if self.\_view.times\_pressed % 2 != 0:  
 self.disconnect()  
 self.button.setText("Connect")  
 else:  
 self.connect()  
 self.\_view.button.setText("Disconnect")  
 self.\_view.times\_pressed += 1  
  
 def about(self):  
 *"""Show an about-dialog."""* logging.debug("About button clicked.")  
 """Show an about-dialog."""  
 QMessageBox.about(self.\_view, "About", """<p>This GUI should help you control TI Radars</p>  
 <p>Created by Oliver Jovanović</p>""")  
  
 def help(self):  
 *"""Show a help-dialog."""* QMessageBox.about(self.\_view, "Help", "Select the tx and rx ports and click 'connect' "  
 "to establish a connection.<p> The creator of the GUI recommends: "  
 "TX = COM3 and RX = COM4.</p>")  
  
 def run(self):  
 self.\_view.show()  
 sys.exit(self.\_app.exec\_())  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 c = Controller()  
 sys.exit(c.run())