

Fachhochschule Dortmund

University of Applied Sciences and Arts

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
Tabellenverzeichnis	III
1. Einleitung.....	1
2. Sprint 1: Definition der GUI und Funktionalitäten	2
a. Sprintplanung: Definition der GUI	2
b. Sprintplanung: Definition der Funktionalitäten	2
3. Sprint 2: Erstellung der GUI ohne Funktionalität	3
a. Erstellung der GUI.....	3
4. Sprint 3: GUI-Funktionalität implementieren.....	4
a. GUI-Funktionalitäten implementieren	4
5. Sprint 4: Code in MVC umschreiben.....	5
a. Code in MVC-pattern umschreiben	5
6. Sprint 5: Stringenter in MVC umschreiben und Toolbox anbinden	6
a. Stringenter in MVC umschreiben	6
b. Funktionalität mit Pythonic mmWave Toolbox sicherstellen.....	6
Literaturverzeichnis	7
Eidesstattliche Erklärung	8
Anhang.....	9

Abbildungsverzeichnis

Abbildung 1: Konzeptionelles Aussehen der GUI.....	2
Abbildung 2: Umgesetzte GUI mit Funktionalität.....	4

Tabellenverzeichnis

Tabelle 1: Sprintplanung und Definition Sprint 1	2
Tabelle 2: Sprintplanung und Definition Sprint 2.....	3
Tabelle 3: Sprintplanung und Definition Sprint 3	4
Tabelle 4: Sprintplanung und Definition Sprint 4	5
Tabelle 4: Sprintplanung und Definition Sprint 5	6

1. 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.

2. Sprint 1: Definition der GUI und Funktionalitäten

Tabelle 1: Sprintplanung und Definition Sprint 1

Sprintplanung	Backlog
<ul style="list-style-type: none"> Definition der GUI Definition der Funktionalitäten 	<ul style="list-style-type: none"> Warten auf Informationen aus dem ersten Sprint, um Backlog zu füllen.

a. 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.

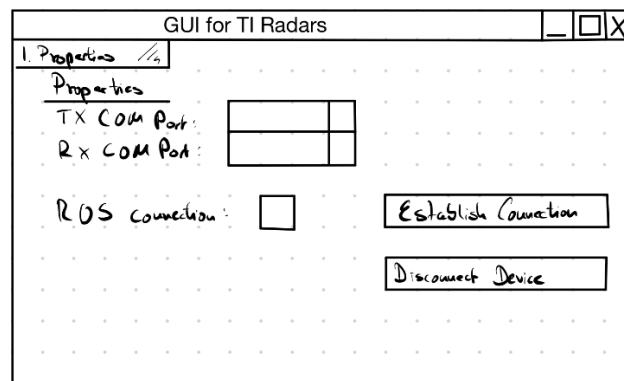


Abbildung 1: Konzeptionelles Aussehen der GUI

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.

b. 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.

3. Sprint 2: Erstellung der GUI ohne Funktionalität

Tabelle 2: Sprintplanung und Definition Sprint 2

Sprintplanung	Backlog
<ul style="list-style-type: none"> • Erstellung der GUI <ul style="list-style-type: none"> ○ Knöpfe ○ Label ○ Dropdown-Menü ○ Titel ○ Tabs 	<ul style="list-style-type: none"> • GUI-Funktionalität <ul style="list-style-type: none"> ○ Button ○ Checkbutton ○ Dropdown-Menü ○ Tabs • Funktionalität mit Pythonic mmWave Toolbox <ul style="list-style-type: none"> ○ GUI interagiert mit der Toolbox • Funktionalität mit ROS <ul style="list-style-type: none"> ○ Verbindung mit ROS herstellen

a. 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 Anzeigenweisungen 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.

4. Sprint 3: GUI-Funktionalität implementieren

Tabelle 3: Sprintplanung und Definition Sprint 3

Sprintplanung	Backlog
<ul style="list-style-type: none"> GUI-Funktionalitäten implementieren <ul style="list-style-type: none"> Button Checkbutton Dropdown-Menü Tabs 	<ul style="list-style-type: none"> Funktionalität mit Pythonic mmWave Toolbox <ul style="list-style-type: none"> GUI interagiert mit der Toolbox Funktionalität mit ROS <ul style="list-style-type: none"> Verbindung mit ROS herstellen

a. GUI-Funktionalitäten implementieren

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

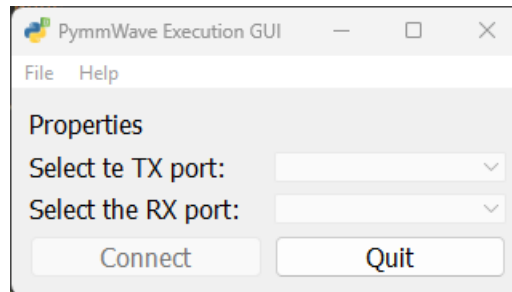


Abbildung 2: Umgesetzte GUI mit Funktionalität

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.

5. Sprint 4: Code in MVC umschreiben

Tabelle 4: Sprintplanung und Definition Sprint 4

Sprintplanung	Backlog
<ul style="list-style-type: none">• Code in MVC-pattern umschreiben	<ul style="list-style-type: none">• Funktionalität mit Pythonic mmWave Toolbox<ul style="list-style-type: none">○ GUI interagiert mit der Toolbox

a. 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.

6. Sprint 5: Stringenter in MVC umschreiben und Toolbox anbinden

Tabelle 5: Sprintplanung und Definition Sprint 5

Sprintplanung	Backlog
<ul style="list-style-type: none"> • Stringenter in MVC umschreiben • Funktionalität mit Pythonic mmWave Toolbox <ul style="list-style-type: none"> ○ GUI interagiert mit der Toolbox 	

a. 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 Menüschaftfläche hinterlegten Knöpfe beinhalten.

Zudem besitzen nun alle Menüpunkte in der Menüschaftfläche Icons und der Menüpunkt *Help* ist unter der Menüschaftfläche *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.

b. 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)

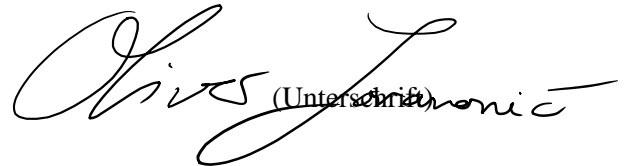
Weitere Erklärung

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
Anhang 2: MVC-Code.....	14

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 PymWave")
        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.addItem(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.addItem(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.addItem(s_ports())
    self.rx_com_port.clear()
    self.rx_com_port.addItem(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("PymmmWave 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.addItem(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.addItem(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.addItem(self._model.ports)
self._view.rx_port.clear()
self._view.rx_port.addItem(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())
```