

**Seminararbeit**

**Von C# nach Python: Software-Konzeptionierung einer  
robotergestützten Lagerverwaltung**

Analyse bestehender Software und Konzeptionierung einer integrierten Python-Anwendung mit  
kameragestützten Validierungsprozessen in der Industrie 4.0-Plattform Modellfabrik  $\mu$ Plant

Lennart Schink

Matrikelnummer:

33237484

Gutachter:

Univ.-Prof. Dr.-Ing. Andreas Kroll

Betreuer:

Dip.-Ing. Axel Dürrbaum

Tag der Abgabe:

16. Juni 2023

MRT-Nr.:

N.N



# Inhaltsverzeichnis

---

<b>Abkürzungsverzeichnis</b>	<b>XI</b>
<b>Abkürzungsverzeichnis</b>	<b>XII</b>
<b>1. Motivation und Zielsetzung</b>	<b>1</b>
<b>Symbolverzeichnis</b>	<b>1</b>
<b>Index</b>	<b>1</b>
<b>2. Softwarearchitektur der bestehender Programme</b>	<b>3</b>
2.1. Lagerverwaltung 3.0 . . . . .	3
2.1.1. Klassenstruktur des Datenmodells . . . . .	4
2.1.2. Klassenstruktur zur Steuerung des ABB Industrieboter IRB 140 . . . . .	8
2.1.3. GUI . . . . .	11
2.2. Controller . . . . .	14
2.3. RFID Server . . . . .	16
2.4. Funktionsanalyse . . . . .	19
2.4.1. Lagerverwaltung 3.0 . . . . .	19
2.4.2. Controller . . . . .	20
2.4.3. RFID Server . . . . .	20
<b>3. Konzeptionierung einer integrierten Python Anwendung</b>	<b>23</b>
3.1. PySide6 und QuickQml 2.0 . . . . .	25
3.2. GUI - Konzeptionierung . . . . .	30
3.3. Konzepte zur Datenmodellierung . . . . .	30
3.4. Konzepte für Controller- und Serviceklassen . . . . .	30
3.5. Teilautomatisierte Code Dokumentation . . . . .	30
<b>4. Analyse zur Fehlerbehandlung</b>	<b>31</b>
<b>5. Ideensammlung zu kameragestützten Validierungsprozessen in der Lagerverwaltung</b>	<b>33</b>
5.0.1. Konzepte . . . . .	33
5.0.2. Abgeleitete Anforderungen an die Kamera . . . . .	33
5.0.3. Kameraauswahl . . . . .	33
<b>6. Zusammenfassung und Ausblick</b>	<b>35</b>
<b>A. Dies ist der erste Anhang</b>	<b>XIII</b>
<b>Literaturverzeichnis</b>	<b>XV</b>



# Tabellenverzeichnis

---

2.1. Werte der Klasse ModbusBaseAddress . . . . . 10



# Abbildungsverzeichnis

---

2.1. Ansicht des Startbildschirms . . . . .	5
2.2. Klassendiagramm Datenmodells . . . . .	6
2.3. Klassendiagramm ABBRobotics Controller . . . . .	9
2.4. Klassendiagramm Modbus . . . . .	11
2.5. Klassendiagramm des Controllers . . . . .	14
2.6. Ansicht des Controller- Startbildschirm . . . . .	15
2.7. Startbildschirm des Programms RFID-Server . . . . .	17
2.8. Klassendiagramm des Programms RFID Server . . . . .	18
3.1. Beispiel: Referenzielle Integrität . . . . .	24
3.2. Model-View Konzept mit zusätzlichem Controller und Service . . . . .	29





# Listings

---

- 3.1. Beispiel einer einfachen App mittels PySide6. Zunächst wird eine Instanz der Application-Klasse und der QmlEngine erzeugt. Danach werden Objekte eines Datenmodells und eines Controllers erzeugt und als rootContext der QmlEngine registriert. Anschließend wird die QML Datei des Hauptfensters geladen, was die App startet. . . . . 27
- 3.2. Beispiel einer QML DateiDiese QML Datei erzeugt ListView QML Type, der zum Anzeigen der Daten in einem ListModel verwendet wird. Innerhalb eines Rechtecks mit farbigem Rand werden in einem RowLayout Type die Datensätze des Datenmodells gerendert und über die Funktionslogik von Signalen der QML Types und der Controllerklasse farblich markiert. . . . . 27



# Abkürzungsverzeichnis

---

Abkürzung	Bedeutung
GUI	<b>G</b> raphical <b>U</b> ser <b>I</b> nterface
UML	<b>U</b> nified <b>M</b> odeling <b>L</b> anguage
MES	<b>M</b> anufacturing <b>E</b> xecuting <b>S</b> ystem
C#	An object oriented, component oriented programming language
C++	A high level, general purpose programming language
QML	<b>Q</b> t-Project's <b>I</b> nterface <b>M</b> arkup <b>L</b> anguage
LZ	<b>L</b> agerzelle der $\mu$ Plant
FZ	<b>F</b> ertigungszelle der $\mu$ Plant
AuE	<b>A</b> bfüll- und <b>E</b> ntleer - Station der $\mu$ Plant
RFID	<b>R</b> adio <b>F</b> requency <b>I</b> Dentification
TCP/IP	<b>T</b> ransmission <b>C</b> ontrol <b>P</b> rotocoll / <b>I</b> nternet <b>P</b> rotocoll
MVC	<b>M</b> odel - <b>V</b> iew - <b>C</b> ontroller, Ein Design-Konzept für Software
URI	<b>U</b> niform <b>R</b> essource <b>I</b> dentifier, Im Qt Framework kann dies eine beliebige Ressource sein. Z.B. eine URL, ein Bild oder ein Programmteil.
QML Type	Ein QML Basiselement. Enthält alle für die Visualisierung nötigen Attribute. Eine Liste aller QML Types findet sich hier <a href="#">[1]</a>



# 1 Motivation und Zielsetzung

---

Das Institut für Mess- und Regelungstechnik an der Universität Kassel hat in den letzten Jahren eine Modellfabrik  $\mu$ Plant gebaut. Aus über 70 Einzelarbeiten ist ein modernes Industrie-4.0 Konzept geschaffen worden. Teil der  $\mu$ Plant ist ein vollautomatisiertes Lager. Das Lager besteht aus einem abgetrennten Raum, dessen Zugang über eine Tür mit einem Türschalter überwacht ist. In diesen Bereich können autonome mobile Roboter (Turtlebots) einfahren. In dem abgetrennten Bereich steht ein Industrieroboter Typ ABB IRB 140 und ein Lagerregal mit ausgewiesenen 18 Lagerplätzen. Außerdem befindet sich neben einer Andockstation für den Turtlebot ein Kommissioniertisch.

Ein pneumatischer Greifer kann Paletten, die je mit bis zu zwei Bechern bestückt werden können, zwischen dem mobilen Roboter und dem Lagerregal transportieren. Von einem PC-Arbeitsplatz aus können mittels Software die Lagerprozesse überwacht werden. Im Fehlerfall kann eingeschritten werden oder es können manuell Prozesse ausgelöst werden. Die Software ist derzeit in 3 Programme aufgeteilt: Einerseits gibt es die Lagerverwaltung 3.0 - die Hauptsoftware. Sie bildet die automatisierten Prozesse ab und verfügt über ein GUI welches u.A. den Bestand visualisiert. Daneben gibt es den Warehouse Controller, der dazu verwendet wird, manuelle Prozesse auszulösen. Zudem gibt es ein Programm „RFID-Server“ mit dem über RFID Leser der Fa. Feig Tags der Transportbehälter ausgelesen werden können.

Mit dem Wechsel des Betriebssystems von Windows 7 auf Windows 10 ist die Kompatibilität der C# Implementierung nicht mehr gegeben. Außerdem laufen Teilfunktionen des Programms nicht fehlerfrei oder tolerieren kaum Fehlbedienungen. Die Dreiteilung der Software ist im Allgemeinen auch nicht mehr erwünscht.

Diese Seminararbeit beschäftigt sich mit der Analyse der bestehenden Software: Es wird ermittelt, aus welchen Programmteilen und Funktionen die Software besteht. Aus den Erkenntnissen wird ein Konzept entwickelt, welches die Funktionen der drei Software Teile zusammenführt. Dies soll die Grundlage für eine Migration der Software nach Python schaffen.

Erkenntnisse aus der studentischen Arbeit von Sebastian Hübler aus dem Jahr 2019 [2] sollen überprüft und vertieft werden um Anforderungen an Kameras und arUco Marker zu ermitteln, die später eine automatisierte Inventur ermöglichen sollen.



# 2 Softwarearchitektur der bestehender Programme

---

Analysemethoden der Informatik für Software sind in der Regel für die verschiedenen Design-Phasen entwickelt worden. Eine von mir durchgeführte Recherche ergab, dass sich Analyse-Tools und Methoden für bestehende Software vor allem darauf fokussieren, die Performance, Speichermanagement und Benutzererfahrung zu bewerten. Die Architektur einer Software spielt dabei eine untergeordnete Rolle. Für die Neuentwicklung der bestehenden Software wird allerdings im Folgenden die Programmstruktur dargestellt und analysiert. Die Performance und der Speicherverbrauch spielen für die  $\mu$ Plant eine untergeordnete Rolle. Eine Bewertung der Programmkomponenten sollte also anhand folgender Kriterien bewertet werden:

- Ihrem Nutzen für den Anwender
- Zuverlässigkeit im Betrieb
- Umgang mit erwartbaren Fehlern

Der Programmcode sollte zudem

- Leicht lesbar und verständlich,
- Zuverlässig und robust, und
- gut erweiterbar sein.

In einem C# Projekt sind UI und Logik in getrennten Dateien implementiert. XAML Dateien sind eine angepasste Form von XML Dateien und legen somit fest, wie das GUI gerendert wird. XAML.CS Dateien enthalten die dahinterliegende Logik und Schnittstellen zu anderen Programmteilen. Am Beispiel des Startbildschirms des Programms „Lagerverwaltung 3.0“ wird der Programmaufbau geschildert. Im Weiteren Verlauf dieser Arbeit wird, wegen der Komplexität und des fehlenden Mehrwerts, darauf verzichtet. Aus gleichem Grund wird auf die detaillierte Beschreibung der Funktionsweise von grafischen Elementen ihrer Events sowie ihrer Eventhandler verzichtet.

In den erstellten Klassendiagrammen ist gut ersichtlich, welche Klassen Events nutzen. Sie erben von der Klasse `INotifyPropertyChanged`, welche die benötigten Listener dafür bereitstellt.

## 2.1. Lagerverwaltung 3.0

Wie in dem einleitenden Abschnitt angekündigt wird zunächst der grundlegende Aufbau des Programms beschrieben.

Die Datei `App.xaml` ist der Einstiegspunkt des Programms. In ihr wird ein Objekt der

---

Application-Klasse mit allen benötigten Ressourcen erzeugt. Als Start URL ist `MainWindow.xaml` angegeben.

In der Datei ist beschrieben, wie der Startbildschirm gerendert wird. Zunächst wird ein Banner gerendert, bestehend aus dem Titel des Programms, dem Logo des Instituts und der  $\mu$ Plant (Siehe Abb.2.1 Bereich „A“). Es werden außerdem alle benötigten Datenobjekte erzeugt. Sie lassen sich wie folgt einteilen:

- Objekte und Variablen, die dem Lager zugeordnet sind:
  - Ein Objekt `inventory` der Klasse `Inventory` für das Inventar mit `null` initialisiert.
  - Ein Objekt `storageMatrix` von der Klasse `PalletMatrix` erzeugt.
  - Ein Objekt `commissionMatrix` von der Klasse `PalletMatrix` erzeugt.
  - Ein Objekt `mobileRobot` von der Klasse `MobileRobot` erzeugt.
  - Außerdem eine Variable `lastCupRead` vom Datentyp `ushort` (16-Bit-Ganzzahl, vorzeichenlos) mit 0 initialisiert.
- Objekte und Variablen initialisiert, die dem ABB Controller zugeordnet sind:
  - Ein Objekt `commands` von der Klasse `controllerCommandList`.
  - Ein Objekt `controllerProperties` von der Klasse `RobotControllerProperties`.
  - Ein Objekt `controllerBase` von der Klasse `RobotControllerBas`, mit dem Initialisierungswert `null`.
  - Ein Objekt `controllerSim` von der Klasse `RobotSimulator`.

Im Constructor der Klasse `MainWindow` werden zudem der ModBus und der Roboter Controller initialisiert und gerendert (Abb.2.1 Bereiche „B“ und „C“). Die Produktliste wird aus der Datei `Produkte.db` geladen und gerendert (Abb.2.1 Bereich „D“). Daten des Lagers und des mobilen Roboters sowie die Kommissionsdaten werden aus der Datei `CommissionData.db` geladen und anschließend das Inventar gerendert (Abb.2.1 Bereich „E“ und „G“).

Im Bereich „F“ werden alle Ereignisse der Software als Text angezeigt. Das können Fehler sein aber auch Fortschritte im Programmablauf.

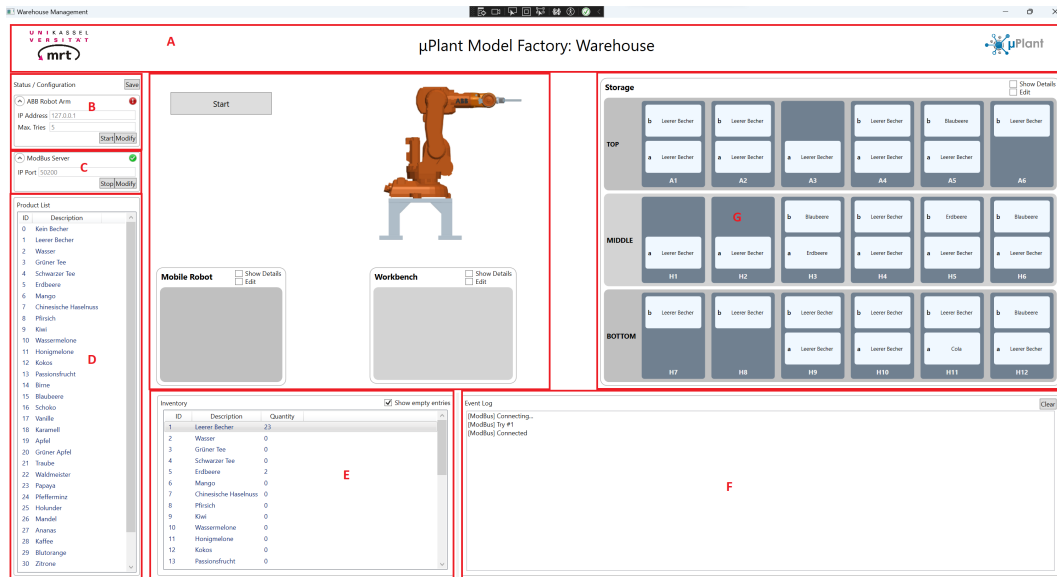
Im mittleren Bereich ist die Anordnung von Roboter, Andockstation und Kommissioniertisch symbolisiert. Der „Start“-Knopf startet den Automatikbetrieb. Wenn keine Verbindung zum Modbus hergestellt werden kann, wird dem Benutzer angeboten die Vorgänge zu simulieren. Im Klassendiagramm Abb.2.2 wird jedoch schnell deutlich, dass dieser Simulationsbetrieb nicht dazu genutzt werden kann, um etwaige Fehler zu erkennen, da dazu eine ganz andere Klasse verwendet wird. Im Bereich „Mobile Robot“ wird nach erfolgreichem Andocken das erkannte Produkt angezeigt. Dem Bediener wird hier angeboten die Daten manuell zu manipulieren oder Details auszublenden. Im Bereich „Workbench“ werden bis zu zwei Paletten mit ihrem Inhalt gerendert. Auch hier wird dem Benutzer angeboten, die Daten per Hand zu manipulieren.

### 2.1.1. Klassenstruktur des Datenmodells

Die bestehende Software dient dazu Lagerpakete vom mobilen Roboter auf die Werkbank oder ins Lager zu bewegen. Oder alle möglichen Kombinationen davon. Das Konzept der



**Abbildung 2.1.:** Startbildschirm der Anwendung, zur besseren Beschreibung sind die Bedienbereiche rot umrandet und mit Buchstaben gekennzeichnet

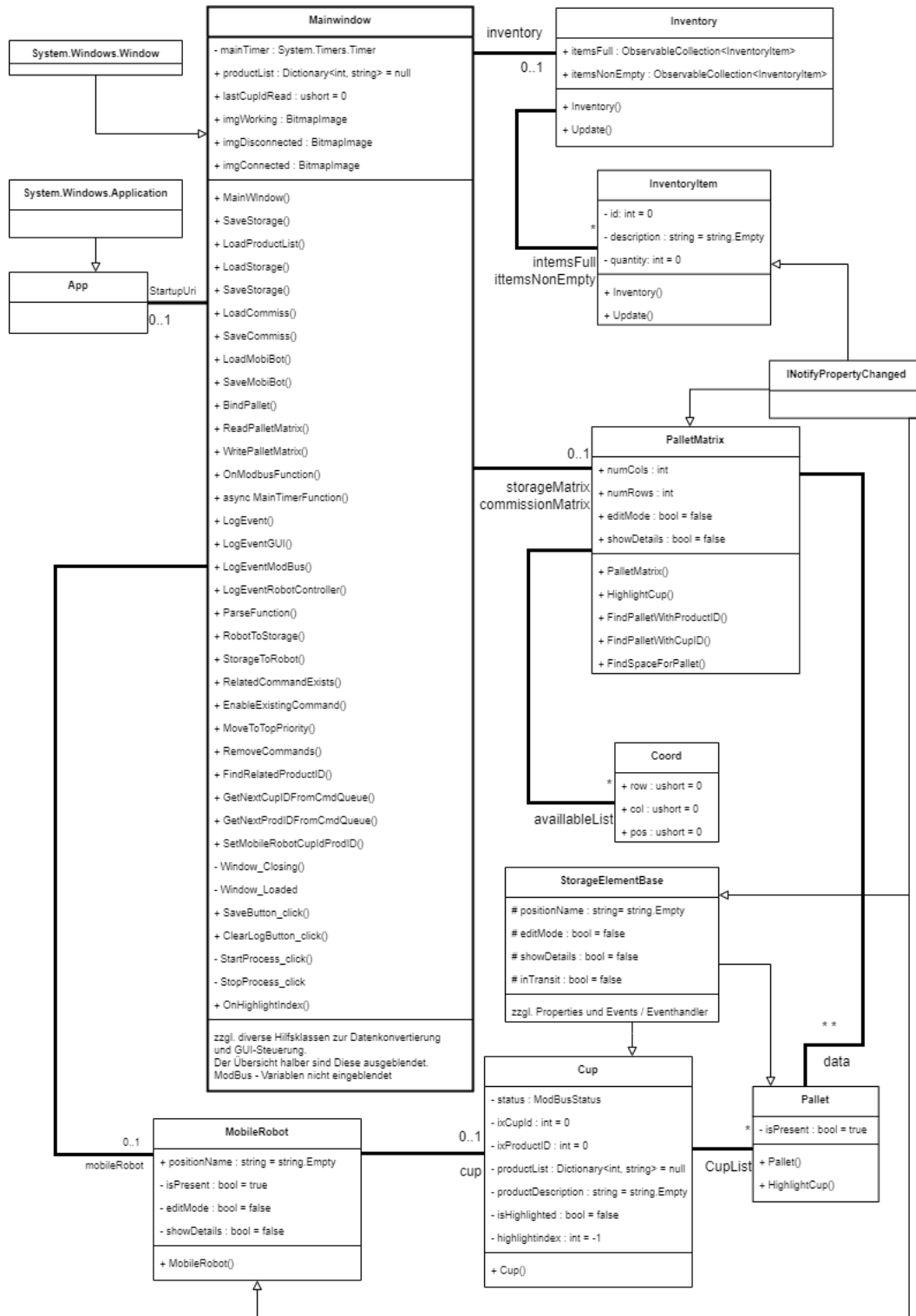


μPlant beschränkt sich dabei auf eine Palette, die so ausgeführt ist, dass der Greifer des Industrieroboters sie sicher bewegen kann. Im Wesentlichen ist es ein Quader mit zwei seitlichen Längsnuten. Eine Palette enthält zwei senkrechte Bohrungen, die es ermöglichen je einen Becher aufzunehmen. Ein Becher ist ein transparentes, zylindrisches Gefäß aus Acrylglas mit einem Absatz, der etwa mittig in der Höhe angebracht ist, sodass der Greifer die Becher einzeln bewegen kann. Es wird also immer entweder

- Eine Palette
- Ein Becher
- eine Palette mit einem oder mehreren Bechern

bewegt. Der Programmierer hat sich diese Struktur angeeignet und in der Datenmodellierung umgesetzt. In Abb.2.2 ist gut ersichtlich, dass die Klasse `StorageElementBase` an die Klassen `Cup` und `Palett` vererbt (schmale Linie mit leerem Pfeil, in Anlehnung an UML). Eigenschaften, die sowohl Palette als auch den Becher betreffen, sind in dieser Klasse implementiert. Weiterhin findet sich das Lager als eigene Klasse `Inventory` und der mobile Roboter als `MobileRobot` wieder. Die `Inventory`-Klasse ist jedoch nicht das Lager im Sinne von Abb.2.2 Bereich „G“, sondern auf die Liste in Bereich „E“.

**Abbildung 2.2.:** Klassendiagramm der MainWindow-Klasse mit vererbenden- und Datenmodell-Klassen



---

Zentrales Element ist die `MainWindow` Klasse. Sie implementiert eigentlich die Interface-Klasse `System.Windows.Window`. Der Programmierer hat sie allerdings als Daten-Hub verwendet. Wie oben geschildert werden beim Rendern des Fensters alle benötigten Datenobjekte erzeugt oder aus Dateien geladen.

- In der Klasse `Inventory` wird der Inhalt der Datei `Produkte.db` in zwei Listen geladen, sodass eine Liste mit lagernden Produkten und eine vollständige Produktliste gespeichert werden. Eine Instanz `inventory` wird zur Laufzeit erzeugt. Wenn die Dateien `Produkte.db` und `StorageData.db` zu dem Zeitpunkt nicht verfügbar ist, stürzt das Programm ab.
- In der Klasse `PalletMatrix` wird die Datei `StorageData.db` bzw. `CommissionData.db` geladen um einen zweidimensionalen Array `data` zu erzeugen. Jedes Array-Element ist ein Objekt der Klasse `Pallet` und enthält eine Liste `CupList` von Objekten der Klasse `Cup`. Diese Struktur wird dazu verwendet, um das reale Lager zu visualisieren. Zur Laufzeit werden zwei Objekte der Klasse `PalletMatrix` erzeugt:
  - `storageMatrix` bildet das Datenmodell um die Visualisierung in [Abb.2.2](#) Bereich "G" zu realisieren.
  - `commissionMatrix` bildet das Datenmodell für die Visualisierung des mobilen Roboters und der Workbench.

---

### 2.1.2. Klassenstruktur zur Steuerung des ABB Industrieroboter IRB 140

Um dem Industrieroboter ABB IRB 140 Befehle zu erteilen, muss das Programm mit der Steuerung IRC5 kommunizieren. Im Programmcode finden sich dazu wie in Abb.2.3 gezeigt, die Klassen der Bibliothek ABBRobotics. Instanzen dieser Bibliotheksbestandteile werden in der Klasse RobotController erzeugt, die von der Klasse RobotControllerBase erbt. Interessanter Weise wird in der MainWindow Klasse kein Objekt von RobotController erzeugt, sondern von der Klasse RobotControllerBase. Die Klasse ControllerCommandList erbt von einer Liste der Klasse ControllerCommand. Beim Initialisieren des Programms wird eine leere ControllerCommandList erzeugt. Die Methoden und Variablen in der Klasse implizieren, dass die Commands aus einer Datei geladen werden. Diese Methoden werden jedoch nie genutzt. Die Klasse ControllerCommand hat 5 statische Methoden. Jede von ihnen gibt ein Objekt der Klasse ControllerCommand zurück.

- StorageToWorkBench übermittelt der Steuerung den Befehl eine Palette vom Lager zum Kommissioniertisch zu transportieren.
- WorkBenchToStorage übermittelt der Steuerung den Befehl eine Palette vom Kommissioniertisch zum Lager zu transportieren.
- WorkBenchToRobot übermittelt der Steuerung den Befehl eine Palette vom Kommissioniertisch auf den mobilen Roboter zu platzieren.
- RobotToWorkBench übermittelt der Steuerung den Befehl eine Palette vom mobilen Roboter auf den Kommissioniertisch zu transportieren.
- WorkBenchToWorkBench übermittelt der Steuerung den Befehl, dass eine Palette den Platz auf dem Kommissioniertisch wechseln soll.

Als Parameter in der Methodensignatur werden die Koordinaten der Start- und Endpunkte übergeben sowie Pallet Objekte am Start- und Endpunkt. In dem ControllerCommand Objekt werden zwei Strings erzeugt, die am Beispiel StorageToWorkBench erklärt werden:

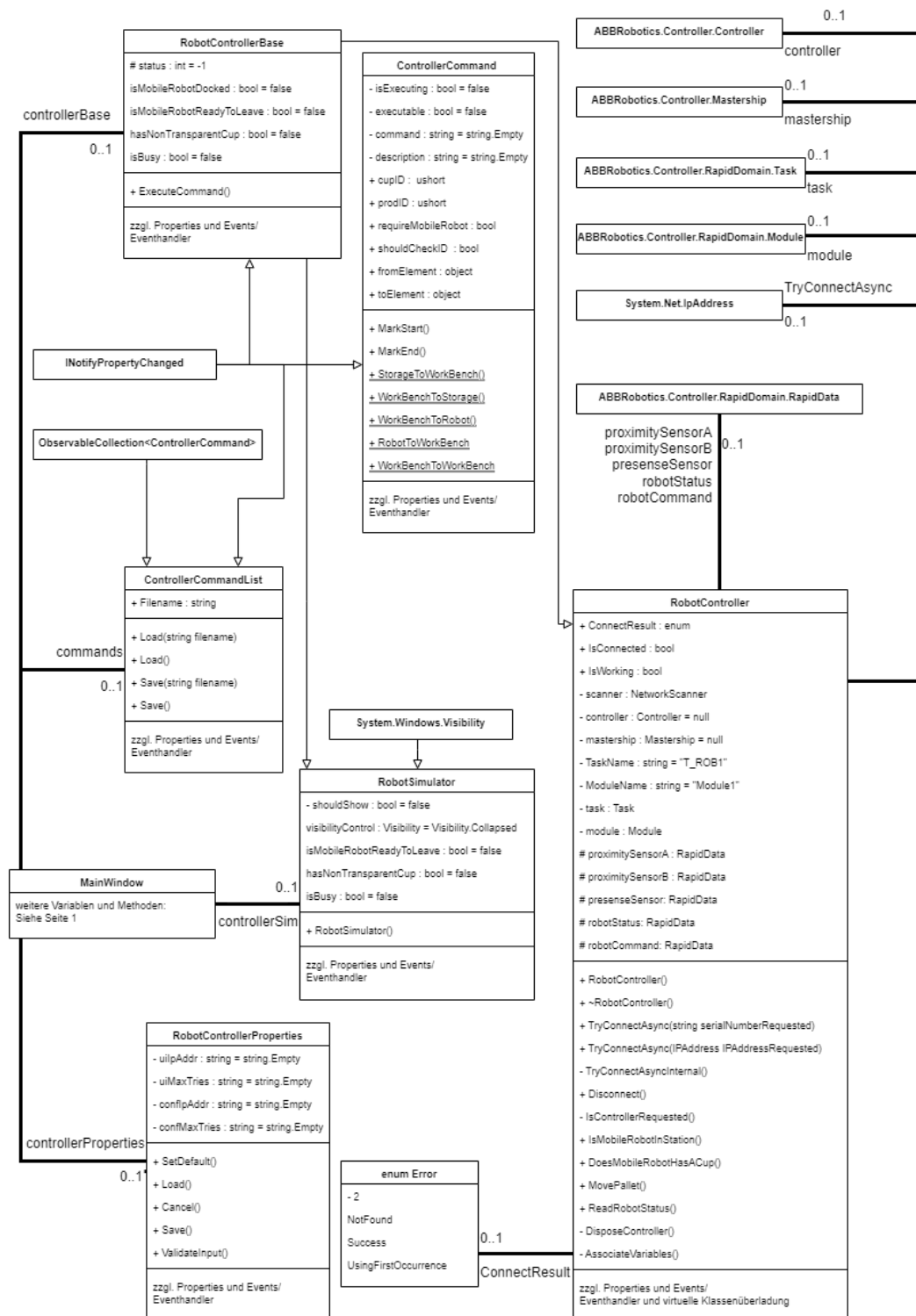
```
command = string.Format("Palette_{0}_{1}_1_{2}",  
station, position, w_col +1)
```

station ist ein Integer mit dem Wert von 3, wenn die oberste Regalreihe ausgewählt wurde, oder 2 sonst. position ist ein Integer, der sich aus der Spalte des Lagerregals errechnet. w\_col+1 ist die Angabe des Ortes am Kommissioniertischs.

```
Description = string.Format("[{0}.{1}] {2}({3}) -> {4}({5})",cupID, prodID,  
str_title_Storage, PositionName, str_title_Commissioning,PositionName)
```

Damit wird ein String erzeugt, der eine Beschreibung des Vorgangs enthält. Die ControllerCommands werden in Methoden der MainWindow Klasse erzeugt. Die Parameter der Funktions-signatur stammen aus dem Objekt commissionMatrix welche in Kapitel 2.1.1 bereits beschrieben wurde. Die MainWindow Klasse hat eine Timer-gesteuerte Funktion MainTimerFunction, die die Commands fallweise abarbeitet. Wie genau die Daten vom Modbus Server in die commissionMatrix geschrieben werden ließ sich mit meinen eher bescheidenen C# Kenntnissen nicht evaluieren.

**Abbildung 2.3.:** Klassendiagramm der MainWindow-Klasse mit Klassen aus dem ABBRobotics Controllerframework.



---

## Klassenstruktur Modbus TCP/IP

Modbus ist ein open-source Kommunikationsprotokoll welches 1979 von Modicon (heute Schneider Electric) veröffentlicht wurde. Es wird dazu verwendet Client-Server Verbindungen einzurichten und ist laut Modbus Organization de facto Standard in industriellen Herstellungsprozessen. Modbus unterstützt verschiedene Kommunikationsstrukturen. Modbus TCP/IP ist lediglich eine Variante, die 1999 entwickelt und veröffentlicht wurde. TCP/IP ist das übliche Transportprotokoll des Internets und besteht aus einer Reihe von Layer Protokollen, die einen zuverlässigen Datentransport zwischen Maschinen bereitstellen. Die Verwendung von Ethernet TCP/IP in der Fabrik ermöglicht eine echte Integration mit dem Unternehmensintranet und MES-Systemen, die die Fabrik unterstützen. Die Protokollspezifikation und Implementierungsanleitung sind auf der Seite der Modbus Organization frei verfügbar[3].

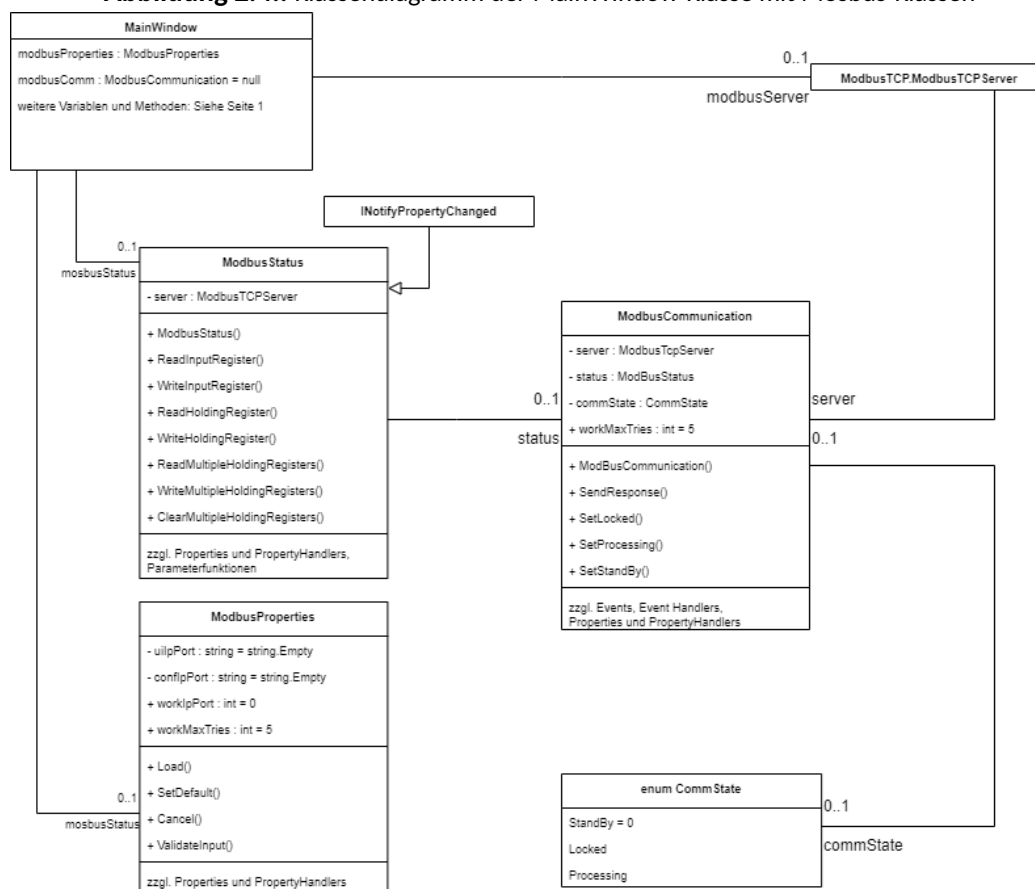
Lars Kistner [4] hat in seiner Bachelorarbeit aus dem Jahr 2016 die Kommunikationsstruktur der  $\mu$ Plant festgelegt. Wie er beschreibt, existiert ein zentraler Modbus Server, der die „Kommandos“ in die entsprechenden Modbus Adressen und/oder Funktionsregister schreibt. Diese Adressen sind in der Datei `ModbusBaseAddress.cs` niedergeschrieben und um die Adressen des mobilen Roboters ergänzt.

**Tabelle 2.1.:** Werte der Klasse `ModbusBaseAddress`

Adresse	Wert	Beschreibung
Status	1	
KeepAlive	2	
Working	3	
FunctionReady	5	
ResponseReady	6	
Done	7	
AgentLockID	8	
AgentLockRequest	9	
FunctionID	10	
FunctionParameters	11	
FunctionParametersLength	32	
ResponseID	43	
ResponseParameters	44	
ResponseParametersLength	32	
StorageCup	1024	
StorageProduct	1280	/// 1024 + 256
CommissionCup	2048	
CommissionProduct	2080	/// 2048 + 32
MobileRobotCup	2560	
MobileRobotProduct	2568	/// 2560+8
MobileRobotDocked	2576	
MobileRobotReadyToLeave	2580	

Schaut man sich das Klassendiagramm der Modbus - Implementierung 2.4 an, stellt man fest, dass es die Klasse `MainWindow` Objekte von drei verschiedenen Klassen instanziiert, die sich dem Modbus Protokoll zuordnen lassen. Sie besitzt ein Objekt der

**Abbildung 2.4.:** Klassendiagramm der MainWindow-Klasse mit Mosbus-Klassen



Klasse ModbusStatus welches ein Objekt der Klasse ModBusCommunication besitzt. modbusTCP.modbusTCPServer wird ebenfalls als Objekt in MainWindow erzeugt. Gleichzeitig wird sie aber auch in ModbusCommunication instanziiert. Die dritte Klasse ist ModbusProperties. -> Methoden analysieren und Übergang zu ABB Klassen schaffen.

CommissionMatrix Die Klasse RobotController übersetzt diese in Strings und schickt sie an die IRC5 Steuerung. Wenn diese den Befehl ausgeführt hat, meldet sie „Fertig“ zurück. Das muss anhand von Code gezeigt werden.

Im Anhang B.5 seiner Arbeit sind die Modbus-Adressen, die dem Lager zugeordnet sind, aufgelistet. Der Adressenbereich ist lückenlos zwischen 1024 und 1105. Jeder Ablageort eines Bechers (z.B. L1a für Lagerort L1, vorne) hat zwei Adressen: Eine für die Cup ID und eine für die Product ID. Das schließt den Kommissioniertisch und den mobilen Roboter mit ein.

### 2.1.3. GUI

Das GUI besteht aus einem Fenster, welches in 8 Bereiche unterteilt ist (siehe Abb.2.1). Der Bereich „A“ hat keine Bedienfunktion.

Der Bereich „B“ ermöglicht dem Benutzer die Verbindungsdaten zur Steuerung des ABB Roboter IRB140 zu konfigurieren und die Verbindung herzustellen. Die Verbindungsdaten sind voreingestellt und die Felder sind standardmäßig für die Eingabe deaktiviert. Die

---

Felder können über die Taste „Modify“ wird die Eingabe freigeschaltet. Mit der Taste „Save“ können die Änderungen gespeichert werden. Mit der „SStart“-Taste wird die Verbindung hergestellt. Die Taste verändert daraufhin ihren Text zu „Connecting...“ und bei erfolgreich hergestellter Verbindung zu „Stop“. Ein Symbolbild informiert dabei über den Verbindungsstatus.

Im Bereich „C“ kann der IP-Port des Modbus-Servers über die Taste „Modify“ angepasst werden. Nach meiner Beobachtung funktioniert der Bereich nicht wie er soll, denn die Verbindung wird direkt nach Programmstart direkt als hergestellt angezeigt. Die Verbindungstaste zum Starten steht zum Programmstart auf „Stop“.

Im Bereich „D“ befindet sich eine scrollbare Liste aller möglicher Produkte mit zwei Spalten. Es wird neben dem Produktnamen die Produkt-ID angezeigt. Wahrscheinlich wurde die Liste als Nachschlagewerk gebaut, denn es sind ansonsten keine Bedienfunktionen verfügbar.

Im Bereich „E“ ist die gleiche Liste um eine Spalte erweitert, in der die gelagerte Menge aufgeführt ist. Die Liste bildet somit eine gute Übersicht über das Inventar und ist daher auch als „Inventory“ gekennzeichnet. Mit der Checkbox „Show empty entries“ können die Produkte ohne eingelagerte Menge ausgeblendet werden.

Bereich „F“ ist ein Eventlog. Hier werden von dem Programm aus verschiedenste Mitteilungen an den Benutzer getätigt. Mit der Taste „clear“ werden alle existierenden Einträge gelöscht.

Bereich „G“ ist die Visualisierung des Lagers. Die drei Reihen „TOP“, „MIDDLE“ und „BOTTOM“ sollen die drei Regalböden des Lagerregals nachbilden. Sie sind als hellgraues Rechteck gerendert. Die Slots A1...A6, H1...H6 sowie H7...H12 sind die Plätze für je eine Palette, die wiederum Platz für je zwei Becher hat. Der Ursprung dieser Bezeichnung konnte während der Vorbereitungen auf diese Arbeit nicht geklärt werden. Das reale Pendant ist mit L1...L18 gekennzeichnet.

Das dunkelgraue Rechteck mit der Lagerort-Beschriftung symbolisiert die Palette. Die beiden weißen Rechtecke darauf symbolisieren je einen Becher. Sie sind mit „a“ für vorne und „b“ für hinten gekennzeichnet und zeigen den Produktnamen an. Ist ein Slot für einen Becher leer, wird kein entsprechendes Rechteck gerendert. Ein leerer Becher ist ein Produkt im Sinne des Programms. Ist keine Palette vorhanden, verbleibt der gesamte Bereich der Palette leer. In der oberen rechten Ecke kann der Benutzer mit der Checkbox „Show Details“ zusätzlich die Cup-ID und die Produkt-ID ein- und ausblenden. Mit der Checkbox „Edit“ kann der Benutzer alle Daten (Palette vorhanden/ nicht vorhanden, Cup-ID, Produkt-ID) als Eingabefeld sehen und ändern. Der Produktname wird dabei über die Produkt-ID referenziert. Die Änderungen werden gespeichert, indem die Checkbox einfach wieder abgewählt wird.

Im mittleren Bereich ist links der Bereich für den mobilen Roboter gerendert. Rechts ist der Bereich des Kommissioniertisches gerendert. Beide Bereiche geben dem Benutzer, wie das Lager auch, die Möglichkeit über die Checkboxes „Show Details“ und „Edit“ die Cup- bzw. Produkt-ID ein- und auszublenden oder die Daten manuell zu überschreiben. Darüber ist der Roboter IRB 140 als Symbolbild gerendert. Das Bild bietet keinerlei Bedienfunktion. Links daneben befindet sich die „Start“-Taste. Sie startet bei verbundenem Modbus den Automatikbetrieb. Ist der Modbus nicht verbunden, erscheint nach dem Klick ein Dialogfenster. Dieses fragt den Benutzer ob wegen der fehlenden Verbindung der Controller simuliert werden soll. Klickt man nun auf „Ja“ wird über dem mobilen Roboter ein weiteres Rechteck gerendert. Es enthält zwei Checkboxes „Mobile Robot is

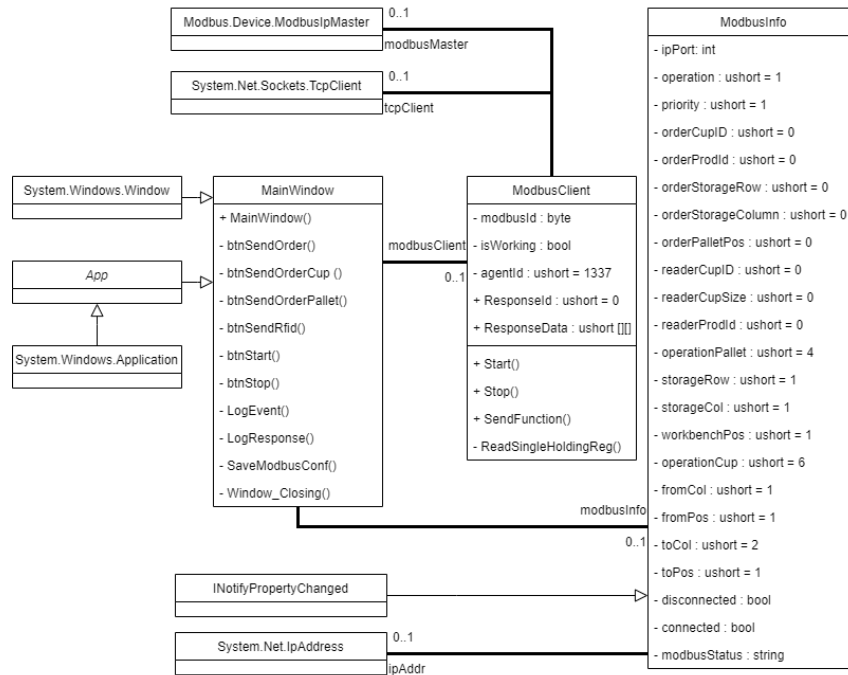


---

present “und „Robot has non transparent cup “. Erstere rendert nach dem Klick einen mobilen Roboter (schwarzes Oval) und gibt dem Benutzer die Möglichkeit wie gewohnt über die beiden Checkboxes im oberen, rechten Bereich, Daten eines Bechers und Produkts einzugeben. Klickt man nun wieder auf die Taste, die nun mit „Stop “beschriftet ist, erscheint für etwa eine Sekunde der Schriftzug „Wait. . .“ bevor das Programm wieder in den Ausgangszustand wechselt.

Weiterhin bietet das Programm keine Möglichkeit den Roboter zu steuern.

**Abbildung 2.5.:** Klassendiagramm des Warehouse-Controllers mit allen vererbenden Klassen jedoch ohne Klassen die lediglich Datentypen konvertieren.



## 2.2. Controller

Der Controller ist ein kleines Programm, welches die manuelle Bedienung der Lagerzelle ermöglicht. Wie in 2.6 ersichtlich, ist das Fenster einfach strukturiert und übersichtlich. Ganz oben können die Modbus Verbindungsdaten konfiguriert werden und ganz unten ist ein Eventlog, derüber alle Fehler und Fortschritte in Textform berichtet. Dazwischen sind über eine Registeransicht die Basisfunktionen, wie sie in den statischen Methoden aus Kapitel 2.1.2 schon beschrieben wurden. In dem Reiter kann ein Befehl konfiguriert werden und mit „Send“ wird der Befehl über die implementierten Modbus-Klassen an die Steuerung IRC5 des Industrieroboters IRB 140 gesendet.

Mit dem RFID Server Tool, darunter dargestellt, können Speicherblöcke eines RFID-Tags mit der Cup ID oder mit der Produkt ID beschrieben werden. Voraussetzung dafür ist, dass der Becher in der Andockstation des mobilen Roboters steht.

In Abb.2.5 sieht man, dass das Programm aus zwei wesentlichen Klassen neben dem MainWindow besteht. ModbusClient implementiert den Modbus Clienten zur Kommunikation mit den Modbus Servern, die in 2.1.2 beschrieben sind. Die Klasse implementiert auch das in [?] beschriebene Agentensystem. ModbusInfo dagegen hält die Daten, die zur Erzeugung der Kommandos benötigt werden.

Die Implementierung der RFID - Funktionen können anhand des Klassendiagramms nicht gezeigt werden. In Vorbereitung auf diese Arbeit habe ich nicht ausprobiert, ob das Programm diese Funktionen so erfüllt wie sie in dem GUI impliziert wird. Für die weitere Arbeit würde dies auch keinen Mehrwert bieten wie sich in Kapitel 3 noch zeigen wird.

**Abbildung 2.6.:** Ansicht des Controller Startbildschirms.

Warehouse Controller

Modbus Configuration

IP Address

127.0.0.1

IP Port

502

Disconnected

Start

Order

Basic

Pallet

Cup

Operation

Robot → Storage

Request Type

Prepare

Cup ID

0

Product ID

0

Storage Position (optional)

Any

Any

Cup Position (optional)

Any

Note: Cup ID or Product ID must be set.

Send

RFID Server

Cup ID

0

Cup Size (optional)

Any

Product ID (optional)

0

Send

Event Log

Error connecting to modbus server.

---

## 2.3. RFID Server

Dieses kleine Programm implementiert einen Modbus TCP/IP Server wie in [2.1.2](#) beschrieben. Statt mit dem ABB Controller zu kommunizieren, hält dieser Server Daten, die aus einem RFID-Tag stammen. In jeder Fertigungszelle der  $\mu$ Plant gibt es einen RFID Leser der Fa. Feig. Stationen in der  $\mu$ Plant können einen Tag am Becher beschreiben wenn sie das Produkt im Becher manipulieren. Anschließend können Stationen die einen Becher erhalten, den angekündigten Becher und das Produkt beim Eintreffen in der Station verifizieren. Die Verfahren dazu sind in [\[?\]](#) gut beschrieben. Die Daten der Tags sollen beim Schreiben und Lesen auf die Modbus Adresse des RFID Servers der jeweiligen Station geschrieben werden. Die Auftragsverwaltung der  $\mu$ Plant kann die Daten der Tags dann von den RFID Servern abrufen. Wie das funktioniert, konnte ich in Vorbereitung auf diese Arbeit nicht untersuchen.

Das GUI des RFID-Servers ist einfach gehalten und ist in [Abb.2.7](#) dargestellt. Das Hauptelement ist eine Liste. Jedes Listenelement enthält Daten zu einem RFID Modbus Server und einem Tag mit seinen Daten. Mit der Taste „Add New“ können neue, leere, Listenelemente erzeugt werden. Standardmäßig sind die Eingabefelder deaktiviert, können jedoch mit der Taste „Modify“ zum Bearbeiten freigeschaltet werden. Mit der Taste „Start“ wird die Verbindung hergestellt. Ein Feedback, ob die Verbindung erfolgreich hergestellt werden konnte ist zunächst nicht ersichtlich. Jedes Listenelement enthält eine Checkbox mit der es ausgewählt werden kann. Mit den Tasten „Select All“ und „Select None“ können alle Listenelemente an- oder abgewählt werden. Im unteren, rechten Bereich können mit den Tasten „Start Selected“ und „Stop Selected“ mehrere Server gleichzeitig gestartet und gestoppt werden. Mit der Taste „Remove Selected“ können alle ausgewählten Listenelemente gelöscht werden. Sicherheitsabfrage?!?

Die Programmstruktur kann anhand des Klassendiagramms in [2.8](#) gut erklärt werden. Eine Klasse `MainWindow` hält eine Instanz der Klasse `RFIDProducerCollection`. Diese implementiert die oben beschriebene Liste welche Objekte der Klasse `RFIDProducer` speichert. Ein `RFIDProducer` Objekt enthält ein Objekt der Klasse `ModbusClient` für die Kommunikation über TCP/IP und ein Objekt der Klasse `TagReader` für die RFID Kommunikation.




Abbildung 2.7.: Startbildschirm des Programms RFID-Server

RFID Server

Add New

^

☐ LZ

Tag Info 0 . 0 . 0   

**Name**

**Tag Reader**

IP Address

IP Port

**End Point**

IP Address

IP Port

Modbus Address

Start

Modify

▼

☐ FZ links (entleer)

Tag Info 0 . 0 . 0   

▼

☐ FZ rechts (befüll)

Tag Info 0 . 0 . 0   

▼

☐ AuE1

Tag Info 0 . 0 . 0   

▼

☐ AuE2

Tag Info 0 . 0 . 0   

Select All

Select None

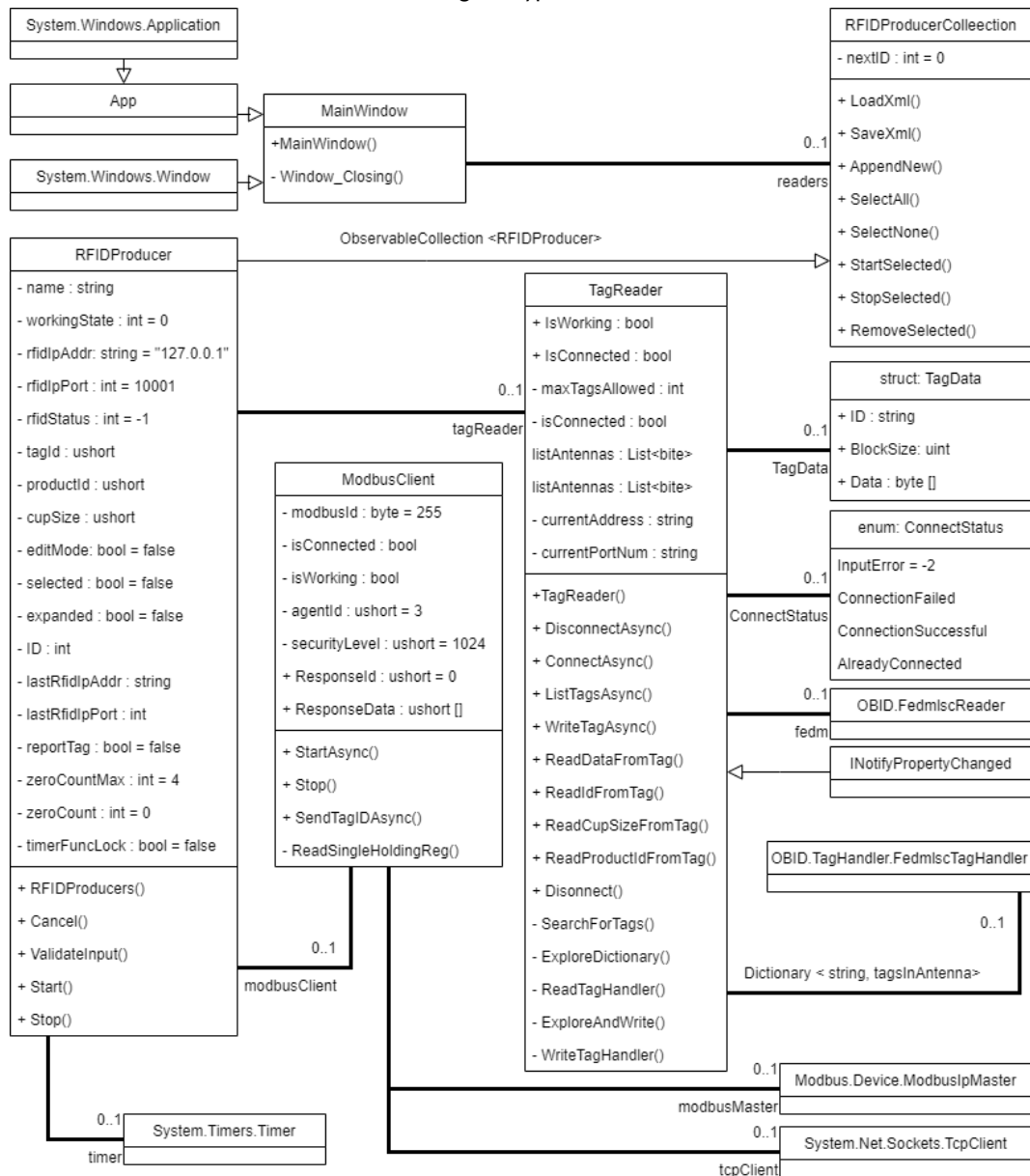
Remove Selected

Start Selected

Stop Selected

17

**Abbildung 2.8.:** Klassendiagramm des Programms RFID Server mit allen vererbenden Klassen jedoch ohne solche die lediglich Typen konvertieren.



---

## 2.4. Funktionsanalyse

In diesem Abschnitt werden die Funktionen der bisher beschriebenen Programme kurz beschrieben und daraus ein gesamter Funktionsumfang abgeleitet, der für das Kapitel 3 dienlich sein wird.

### 2.4.1. Lagerverwaltung 3.0

Dieses Programm hat genau ein Anwendungsfenster welches dem Benutzer einen umfassenden Überblick über die LZ gibt. Das Fenster hat eine Mindest- und Maximalgröße, ist dazwischen aber beliebig skalierbar. Dialoge werden genutzt, wenn Einträge gelöscht werden, Verbindungen nicht aufgebaut werden können und der Simulationsbetrieb gestartet werden kann. Felder sind standardmäßig deaktiviert und müssen über Checkboxes aktiviert werden.

#### Bedienfunktionen

Der Benutzer kann über das GUI folgende Eingriffe vornehmen:

- Die Verbindung zu einem Modbus Server konfigurieren, starten und beenden.
- Die Verbindung zur ABB IRC5 Steuerung konfigurieren, speichern, starten und beenden.
- In der Produktliste kann gescrollt werden.
- In der Inventarliste können leere Produkte ein- und ausgeblendet werden.
- Die vergangenen Einträge im Eventlogger können gelöscht werden.
- In der Lagervisualisierung können an den einzelnen Lagerorten
  - Paletten hinzugefügt oder entfernt werden
  - Becher hinzugefügt oder entfernt werden
  - Produkt ID und Cup ID der Becher konfiguriert werden
  - Durch Klick auf einen Becher werden alle Becher gleichen Produkts in der Lagervisualisierung und auch im Inventar blau hervorgehoben.
- Palette, Becher und Produkt auf dem Kommissioniertisch können konfiguriert werden
- Becher und Produkt auf dem mobilen Roboter können konfiguriert werden.
- Der Automatikbetrieb kann gestartet werden
- Wenn die Verbindung zum Modbus Server nicht hergestellt wurde kann ein Simulationsbetrieb gestartet werden
- Im Simulationsbetrieb kann zudem
  - Die Anwesenheit eines mobilen Roboters Simuliert werden
  - ausgewählt werden, dass der Becher auf dem mobilen Roboter nicht transparent ist (Funktion unklar)

---

## Informationsdarstellung

Der Benutzer wird über folgende Inhalte informiert:

- Die Verbindungseinstellungen des Modbus Servers und den Verbindungsstatus
- Die Verbindungseinstellungen des ABB-Controllers und ihren Verbindungsstatus
- Alle möglichen Produkt ID's und die zugehörigen Produktnamen
- Produkte und ihre Lagermengen im Inventarbereich.
- In der Lagervisualisierung:
  - Symbolisierung einer Palette durch dunkelgraues Rechteck, wenn eine Palette vorhanden ist
  - Symbolisierung der Becher durch weißes Rechteck, wenn ein Becher vorhanden ist
  - Produktnamen und auf Wunsch auch Produkt ID und Becher ID
- Im Eventlogger werden folgende Informationen bereit gestellt:
  - Fehler und Events der Verbindung über Modbus TCP/IP
  - Fehler und Events der Verbindung zur Steuerung IRC5 des Industrieroboters
  - Programmfortschritte und Events im Automatikbetrieb oder der Simulation

### 2.4.2. Controller

Der Benutzer kann folgende Bedienvorgänge durchführen:

- Die Verbindungseinstellungen des Modbus Servers konfigurieren und starten
- Transportbefehle können erzeugt werden:
  - Transportbefehle eines einzelnen Bechers zwischen Lager, Kommissioniertisch und Roboter
  - Transportbefehle für einer Palette zwischen Lager und Kommissioniertisch

Die beiden Befehle können einerseits direkt ausgeführt werden, andererseits können sie auch als Eingabe für die Lagerverwaltungssoftware benutzt werden.

- Ein Becher in der Andockstation kann mittels einem RFID- Lesegerät beschrieben werden.

Der Benutzer wird in diesem Programm nur über seine Eingaben informiert. Die Eingaben erfolgen wo möglich über ein Dropdown Menü

### 2.4.3. RFID Server

Der Benutzer kann folgende Bedienvorgänge durchführen:

- Einen Neuen Listeneintrag erzeugen
- Einen oder alle Listeneintrag markieren



- 
- Markierte Listeneinträge abwählen
  - ausgewählte Listeneinträge Löschen, Verbindung starten oder stoppen
  - Je Listeneintrag kann der Benutzer folgende Einstellungen vornehmen:
    - IP und Port des Tag Readers
    - IP, Port und Modbus Adresse des Endpoints
  - Ein einzelner Listeneintrag kann gestartet werden
  - Mit der Taste „Modify“ kann ein Listeneintrag zum Bearbeiten freigeschaltet werden
  - Mit der Taste „Save“ können Änderungen gespeichert werden.

Der Benutzer sieht in diesem Programme eine Liste aller eingetragenen RFID Reader, Endpoints und Modbus Adressen. Jeder Listeneintrag kann auf eine Zeile minimiert werden die lediglich den Namen des Endpunkts und die gelesenen Tag Daten anzeigt. Im ausgeklappten Zustand sind die Konfigurationen sichtbar, aber grau hinterlegt um anzuzeigen, dass die Bearbeitung gesperrt ist.



# 3

## Konzeptionierung einer integrierten Python Anwendung

---

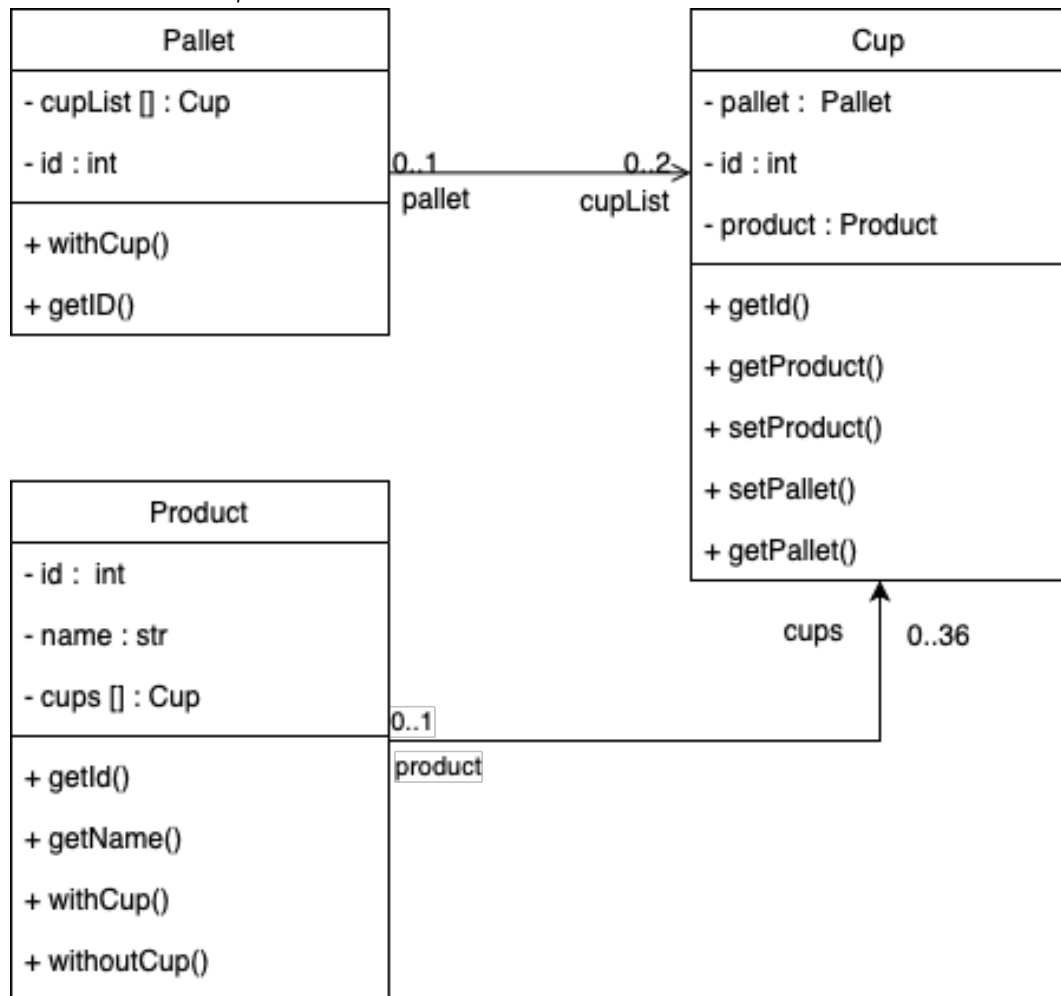
Die Basisanforderungen einer integrierten LZ-Verwaltung lassen sich aus Kapitel 2 ableiten. Hinzu kommen die später in Kapitel 5 beschriebenen Funktionen. Anhand dieser beiden Punkte kann man erkennen, dass Modularität, Erweiterbarkeit sowie Wartbarkeit wichtige Kriterien sind. Darüber hinaus müssen diese Schritte auch von Personen durchgeführt werden können, die mit der ursprünglichen Entwicklung der Software nicht involviert waren. Eine saubere Struktur der Software ist dazu unerlässlich. Die alte Software nutzte eine GUI-Klasse als Datenhub. Dadurch entstand ein unübersichtliches Klassenkonstrukt, sodass es für Außenstehende sehr schwer wurde sich in den Code einzudenken. Dadurch, dass Teile der Programmlogik in den XAML Dateien „versteckt“ wurden, wurde es beinahe unmöglich. Zum Beispiel wird anhand der Klassendiagramme deutlich, dass Daten der Modbus Adressen in die `commissionMatrix` geschrieben werden müssen, weil ausschließlich dort zu RAPID Kommandos übersetzt werden und an den ABBRobotics Controller übergeben werden. Wie genau das passiert, lässt sich anhand des C# Codes nicht nachvollziehen. Ich bin mir sicher, dass mit einer Menge Zeitaufwand dieses Rätsel gelöst werden kann, jedoch bringt es für die Konzeptionierung der neuen Software keinen Mehrwert.

Um diese Unannehmlichkeiten für die Zukunft zu vermeiden, empfiehlt es sich auf branchenübliche Design Patterns zurückzugreifen. Das grundlegendste Design Konzept ist das MVC. Dieses Konzept sieht vor, dass Daten (Model) und GUI (View) keinerlei Zugriff aufeinander erlauben. Die Schnittstelle zwischen den Daten und dem Benutzer ist ein Controller, der die Programmlogik implementiert. Für die Daten gilt, dass sie in einer Objektstruktur modelliert werden und nicht von außerhalb dieser Objekte verändert werden. Soll ein Wert in dem Datenmodell geschrieben oder verändert werden, dann muss dafür

eine öffentliche Methode existieren, die alle möglichen Fehler abfängt und referenzielle Integrität herstellt.

Als Beispiel für die referenzielle Integrität möchte ich eine Palette mit einem Becher anführen. In Abb. 3.1 sind drei Klassen, Pallet, Cup und Product, dargestellt. Die Attribute der Klassen sind `private`, was durch das „-“ angedeutet ist. Ein Objekt der Klasse Pallet kann sich also nicht direkt selbst in das entsprechende Feld eines Objekts der Klasse Cup eintragen, sondern muss dazu die entsprechende Methode `SetPallet()` aufrufen, die, durch das „+“ Symbol gekennzeichnet, `public` sind. Typischer Weise wird in diesen `get-` und `set-` Methoden das entsprechende Objekt als Parameter übergeben. Diese Methoden müssen mindestens drei wichtige Dinge erledigen. Erstens muss die Gültigkeit der übergebenen Parameter geprüft werden. Zweitens muss überprüft werden ob in dem

**Abbildung 3.1.:** Referenziellen Integrität einer Palette / Becher / Produkt - Kombination wie sie in der  $\mu$ Plant auftreten könnte.



zu schreibenden Attribut schon ein anderes Objekt vermerkt ist, wenn ja, wird dies für Schritt 3 wichtig. Drittens müssen die übergebenen Parameter verarbeitet werden: In die eigene Klasse wird die Änderung geschrieben und die Änderungen werden allen anderen betroffenen Objekten mitgeteilt. Wird zum Beispiel einem Objekt der Klasse Cup je ein Objekt der Klasse Pallet und Product übergeben, dann muss am Ende das Cup Objekt in der Liste cups der Klasse Product stehen und das Objekt der Klasse Pallet muss als Feldwert des Attributs cup das Objekt der Cup Klasse geschrieben sein. Dies wird im Allgemeinen durch den gegenseitigen Aufruf der jeweiligen get/set - Methoden sichergestellt. Für Listennamen haben sich Namen mit dem präfix with bzw. without durchgesetzt.

---

### 3.1. PySide6 und QuickQml 2.0

Laut der Qt Wiki Website [5] wurde das Qt Framework geboren als ihre Schöpfer Haavard Nord und Eric Chambe-Eng im Sommer 1990 in Norwegen an einem GUI für eine Ultraschall Datenbank arbeiten. Die Software sollte damals in C++ implementiert auf Mac, Unix und Windows laufen. Fünf Jahre später veröffentlichten Sie das erste Qt Framework unter dem Firmennamen Troll Tech. Seitdem gewann das Framework immer mehr Popularität. Im Jahr 2006 übernahm Nokia die Firma Trolltech und verkaufte das Qt Project in den Jahren 2011 und 2012 erst teilweise, dann vollständig an den Digia Konzern. Seit 2014 ist Qt als Tochterunternehmen des Digia Konzerns unter dem Namen „The Qt Company“ ein eigenständiges Unternehmen.

Das Qt Framework ist in C++ implementiert und profitiert dadurch von dem Performance-Vorteil gegenüber anderen Programmiersprachen. Die neue Software für die  $\mu$ Plant soll jedoch in Python implementiert werden. Für diese Zwecke hat Qt u.A. das Framework PySide6 veröffentlicht, welches einen Wrapper für Python Projekte bietet.

GUI's können in PySide6 im Wesentlichen auf zwei Arten erstellt werden. Eine Möglichkeit ist es, das GUI über Widgets[6] zu erstellen, die direkt im Python Code implementiert werden können. Die zweite Möglichkeit ist QtQuick [7] zu nutzen, bei dem das GUI in einer separaten QML Datei erstellt wird und mittels einer QtQmlApplicationEngine Klasse in das Programm eingebunden wird.

Für die Umsetzung eines MVC-Design Patterns empfiehlt sich die Verwendung von Qt-QuickQML. Durch die Verwendung des Frameworks wird die konsequente Trennung zwischen Interface und Datenmodell erzwungen. Veranschaulicht wird dies in Abb. 3.2. Die eigentlichen Daten (von Server oder Datei) werden in ein Model überführt. Dieses Model ist ein beliebig komplexes Klassenkonstrukt welches referenzielle Integrität aufbaut: Wird eine Stelle berechtigter Weise geändert, bleibt das Modell intakt und im Sinne seiner Logik. In Pyside6 muss dieses Datenmodell von der Basisklasse QAbstractModel abgeleitet werden. Beim Initialisierung der Anwendung wird eine Instanz dieser Klasse (oder einer abgeleiteten Klasse) erstellt und als rootcontext der QQmlApplicationEngine registriert. Somit ist das Datenmodell mit dem GUI verknüpft und kann unter seiner URI in jedem QML File angesprochen werden. In einer QML Datei wird ein entsprechendes QML Item, z.B. ListView, welches eine einfache Liste erzeugt, und dem Property model die URI des Datenmodells zugewiesen. Dadurch kennt das ListView Objekt die Indices des Datenmodells und erhält beim rendern der Liste nur die benötigten Teile der Daten. Jede weitere Aktion die durch den Benutzer auf ein Listenelement ausgelöst wird, bezieht sich auf ein Delegate des Datenmodells. Jede Änderung an dem Delegate wird zunächst gerendert und überprüft bevor das Datenmodell geändert wird. Diese Basisfunktion kommt mit dem Qt Framework an sich und müssen nicht implementiert werden. Wie jedoch das Datenmodell mit den geänderten Daten umgeht oder ob die Änderung des Datenmodells automatisch ein Update der Daten auf dem Server oder der Datei auslöst, muss vom Entwickler implementiert werden! An ein Delegate sind je nach QmlType durch das Framework Signale gebunden. Signale sind Teil des Signal/Slot Prinzips des Qt Framework [8] und stellen die Funktion eines Events dar. Durch die QmlApplicationEngine sind innerhalb eines GUIs alle rootContext Elemente ansprechbar. Wird ein Signal an beliebiger Stelle im GUI emittiert, kann es an jeder anderen Stelle als Event genutzt werden. Dadurch muss man in der Regel keine zusätzlichen Callbacks oder Lamda-Ausdrücke definieren. Außerhalb des QML Contexts, z.B. in einer Python Klasse, muss das Signal der Klasse bekannt sein, indem das Signal an die Klasse gebunden wird. Eine Funktion

---

die mit der Annotation „Slot(str)“ versehen ist, wird als Slot behandelt und kann mit dem Signal verknüpft werden, sodass diese bei Auftreten des Signals ausgeführt werden. Signale können Daten übergeben werden, die so an einen Slot übergeben werden können.

In meiner Vorbereitung auf diese Arbeit hat sich eine intuitive Vorgehensweise entwickelt, die ich für die Implementierung der Software empfehlen möchte:

Beim Initialisieren des Programms werden alle Datenmodelle (`QAbstractModel` und abgeleitete Klassen), Controller und Serviceklassen instanziiert und als `rootContext` der `QQmlApplicationEngine` gesetzt. Siehe dazu das Beispiel 3.1. Damit stehen sie dem Programmierer in jeder QML Datei zur Anwendung zur Verfügung.

In einer QML Datei können die Kontexte der `QmlEngine` nun unter ihrer URI referenziert werden. Dies ist in Beispiel 3.1 gezeigt:

In dem QML Type „`ListView`“ wird dem Property `model` die URI des `ListModel` aus 3.1 zugewiesen. Im weiteren Verlauf des Codes findet sich ein `MouseArea` QML Type. Wird in dem Bereich ein Klick mit linker Maustaste durchgeführt, wird das Signal `onClicked` emittiert. Innerhalb des `MouseArea` Codes wird definiert, dass nach einem Mausklick die Funktion `selectRow(message)` aufgerufen wird. `message` ist dabei der übergebene Parameter. Es ist schnell ersichtlich, dass das Beispiel keinerlei Importe enthält, was daran liegt, dass sowohl das Datenmodell als auch das Objekt der Controllerklasse als Resource in der `QmlEngine` registriert wurden. Im weiteren Verlauf des Codes wird eine `Connections` erzeugt, bei dem das Signal des Controllers `onRowClicked` mit einer Funktion verbunden wird. Innerhalb des `Connections` Types wird in Javascript die Funktion definiert. Zu beachten ist, dass im Controller selbst das Signal als `RowClicked` definiert wird. Innerhalb der QML Datei werden die Signale dann mit dem Präfix „on“ erfasst. Der Signalname wird als Name einer javascript - Funktion verwendet, die mit `function` gekennzeichnet ist. Der Code innerhalb des Funktionskörpers beschreibt dann die Funktionslogik in Javascript. Im Fall des Beispiels wird ein bool'sches Property umgeschaltet, wenn die `id` des Datenmodell - Delegates mit der `message` des Signals übereinstimmt.

---

**Listing 3.1:** Beispiel einer einfachen App mittels PySide6. Zunächst wird eine Instanz der Application-Klasse und der QmlEngine erzeugt. Danach werden Objekte eines Datenmodells und eines Controllers erzeugt und als rootContext der QmlEngine registriert. Anschließend wird die QML Datei des Hauptfensters geladen, was die App startet.

```
'''Create Basic Application Class and QML Engine'''
app = QtGuiApplication(sys.argv)
engine = QQmlApplicationEngine()

# Create simple ProductListModel from path to database file
listModel = ProductListModel(getProducts(PRODUCTLIST))
engine.rootContext().setContextProperty("listModel", listModel)

# create InventoryController instance
listController = ProductListController(listModel)
engine.rootContext().setContextProperty("listCon", listController)

# define load main.qml file to start application
qml_file = "../src/qml/main.qml"
engine.load(qml_file)

if not engine.rootObjects():
    sys.exit(-1)

sys.exit(app.exec())
```

**Listing 3.2:** Beispiel einer QML Datei Diese QML Datei erzeugt ListView QML Type, der zum Anzeigen der Daten in einem ListModel verwendet wird. Innerhalb eines Rechtecks mit farbigem Rand werden in einem RowLayout Type die Datensätze des Datenmodells gerendert und über die Funktionslogik von Signalen der QML Types und der Controllerklasse farblich markiert.

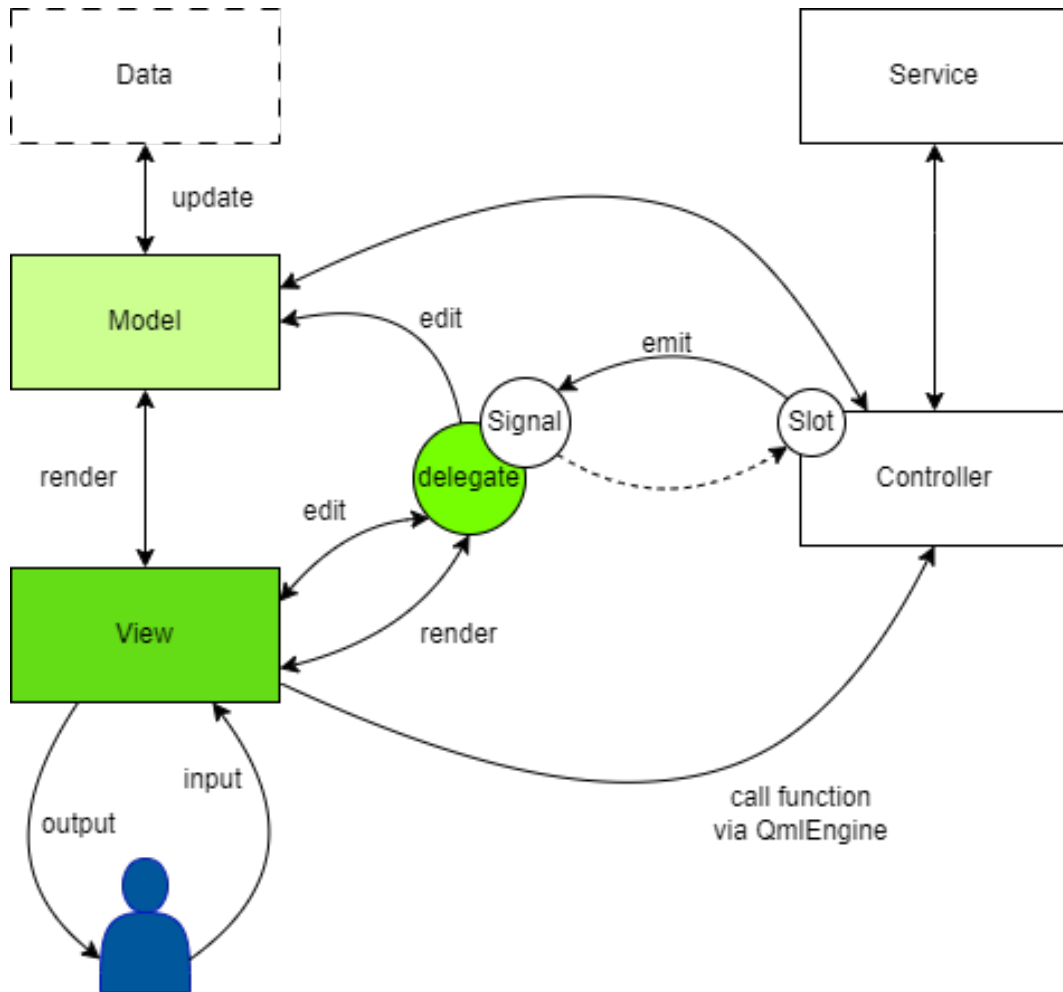
```
ListView {
    id: inventoryList
    model: listModel
    anchors.fill: parent
    anchors.margins: 10
    clip: true
    spacing: 5
    Layout.fillWidth: true
    delegate: Rectangle {
        id: rect1
        width: ListView.view.width
        height: 50
        property bool selected: false
        color: selected ? "#4FC3F7": "white"

        RowLayout {
            id: row
            anchors.fill: parent
            Text {
                id: id
                text: model.id
                font.pixelSize: 20
                verticalAlignment: Text.AlignVCenter
                Layout.fillHeight: true
                Layout.fillWidth: true
                Layout.preferredWidth: 50
            }
        }
    }
}
```





**Abbildung 3.2.:** Die Abbildung zeigt das in PySide6 verwendete Model-View Konzept, welches um einer Controllerklasse und einer Service Klasse erweitert wurde.



Mit den Beispielen 3.1 und 3.1 lässt sich sehr gut ein Schema erkennen: Die Daten sind hinter einem DatenModel geschützt und werden gleichzeitig der View zur Verfügung gestellt. Über die QML Dateien erfolgt die GUI Modellierung und Events werden mit dem Signal/Slot Prinzip behandelt. Beliebige Ressourcen können als Teil der QmlEngine registriert werden um Sie an jeder Stelle im GUI verfügbar zu machen. Das künftige Programm soll jedoch auch Programmteile aufweisen, die nicht unbedingt mit den Daten und dem GUI verknüpft sind. Das sind z.B. die Kommunikation über Modbus und dem ABB Industrieroboter oder das Übersetzen der Modbus Werte in RAPID Befehle. Diese Programmteile werden als Service Klassen bezeichnet. Wenn sich der Programmcode auf ein GUI auswirkt, wird ein Controller gebraucht, der dieses Verhalten steuert. Selbstverständlich könnten die Funktionen eines Controllers in den Service integriert werden. Dies würde aber die Wartbarkeit und Erweiterbarkeit verschlechtern. Nach dem Schema wie in 3.2 können beliebig viele Datenmodelle, GUI's, Controller und Services parallel existieren ohne sich gegenseitig zu beeinflussen.

Als Nachteil kann angeführt werden, dass sich mit zunehmendem Kontextregister der QmlEngine die Performance des Programms verschlechtern wird. Bei dem eher geringen erwarteten Funktionsumfang ist dies jedoch untergeordnet und könnte durch das dezidierte Aufteilen der Funktionen in Threads reduziert werden.

---

**3.2. GUI - Konzeptionierung**

**3.3. Konzepte zur Datenmodellierung**

**3.4. Konzepte für Controller- und Serviceklassen**

**3.5. Teilautomatisierte Code Dokumentation**

# 4 Analyse zur Fehlerbehandlung

---



# **5** Ideensammlung zu kameragestützten Validierungsprozessen in der Lagerverwaltung

---

5.0.1. Konzepte

5.0.2. Abgeleitete Anforderungen an die Kamera

5.0.3. Kameraauswahl



# 6

## Zusammenfassung und Ausblick

---

Hier wird die Arbeit zusammengefasst und ein Ausblick auf offene Fragestellungen gegeben.





# **A** Dies ist der erste Anhang

---

Hier Text einfügen.



# Literaturverzeichnis

---

- [1] „Liste aller QML Types, <https://doc.qt.io/qt-6/qtquick-qmlmodule.html>, Accessed: [15.06.2023].
- [2] S. Hübler, „BPS Tätigkeits- und Erfahrungsbericht: Objekterkennung und Verfolgung mittels Kamera in Produktionsanlagen am Beispiel der Modellfabrik µPlant, 2019.
- [3] „Modbus Organization, <http://www.modbus.org>, Accessed: [12.06.2023].
- [4] L. Kistner, „Konzeptionierung und Umsetzung einer serviceorientierten verteilten Automatisierungs-Architektur für die heterogene Modellfabrik µPlant, 2017.
- [5] „Qt History on Qt Wiki, [https://wiki.qt.io/Qt\\_History](https://wiki.qt.io/Qt_History), Accessed: [13.06.2023].
- [6] „PySide6 QtWidgets, <https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/index.html>, Accessed: [14.06.2023].
- [7] „PySide6 QtQuick, <https://doc.qt.io/qtforpython-6/PySide6/QtQuick/index.html>, Accessed: [14.06.2023].
- [8] „Signal and Slot in PySide6 Framework, <https://doc.qt.io/qtforpython-6/PySide6/QtCore/Slot.html>, Accessed: [13.06.2023].